# Список вопросов для самоподготовки JavaScript (EcmaScript 6+)

1. +Отличия var, let, const
2. +Типы данных
3. +Promise - что это? Преимущества и недостатки. Как с ним работать?
4. +Методы массивов: перечислить. Знать “что делает” тот или иной метод.
5. +Преобразования типов
6. +Null vs undefined
7. +NaN
8. Наследование через прототип
9. ?Область видимости
10. +Контекст (this)
11. +Замыкания
12. Генераторы
13. +setTimeout, setInterval, Event Loop
14. xhr vs fetch
15. +Деструктуризация объекта
16. +Spread operator
17. Загрузка js-файлов async vs defer
18. Кроссдоменные запросы (CORS, JSONP)
19. JSON vs XML

# React.js

1. React - что это? Основные принципы
2. Жизненный цикл компонента
3. Типы компоненты. Отличия и особенности
4. Система и обработка событий
5. Ref 24. Top-level API
6. Способы и виды композиции компонент
7. Структурные паттерны (+ лучшие и худшие практики). Dumb and Smart components
8. Context в React
9. Условный рендеринг
10. setState, forceUpdate
11. Выполение асинхронных действий в компоненте
12. JSX, PropTypes
13. HOC
14. Наследование vs композиция Flux & Redux
15. Flux архитектура - принципы
16. Redux. Отличия Redux от стандартной реализации Flux
17. FSA (Flux Standard Action)
18. Конфигурирование и создание store
19. React-redux (Provide, connect)
20. Action, Reducer, Container
21. Асихронные действия, middlewares
22. Структурные редьюсеры
23. Immutable.js - преимущества и недостатки
24. Selectors & Reselect.js
25. Redux-saga, концепция Общие
26. Как работает браузер? Оптимизация приложения – подходы

https://vk.com/@maxpfrontend-sobesedovanie

# Отличия var, let, const

У объявлений переменной через let есть три основных отличия от var:

1. **Область видимости переменной let – блок {...}.**

if (true) {

let apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // ошибка!

1. **Переменная let видна только после объявления.**

|  |  |
| --- | --- |
| alert(a); // undefined  var a = 5; | alert(a); // ошибка, нет такой переменной  let a = 5; |

1. **При использовании в цикле, для каждой итерации создаётся своя переменная.**

* Каждому повторению цикла соответствует своя независимая переменная let. Если внутри цикла есть вложенные объявления функций, то в замыкании каждой будет та переменная, которая была при соответствующей итерации. (Задача про [армию стрелков](https://learn.javascript.ru/task/make-army)).

**Объявление const задаёт константу, то есть переменную, которую нельзя менять:**

const apple = 5;

apple = 10; // ошибка

* В остальном объявление const полностью аналогично let.
* Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

const user = {

name: "Вася"

};

user.name = "Петя"; // допустимо

user = 5; // нельзя, будет ошибка

# Типы данных

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof {} // "object"

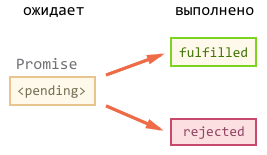
typeof null // "object" (1)

typeof function(){} // "function" (2)

* Последние две строки помечены, потому что typeof ведет себя в них по-особому. Результат typeof null == "object" – это официально признанная ошибка в языке, которая сохраняется для совместимости. На самом деле null – это не объект, а отдельный тип данных.
* Функции мы пройдём чуть позже. Пока лишь заметим, что функции не являются отдельным базовым типом в JavaScript, а подвидом объектов. Но typeof выделяет функции отдельно, возвращая для них "function". На практике это весьма удобно, так как позволяет легко определить функцию.

# Promise

* Promise (обычно их так и называют «промисы») – предоставляют удобный способ организации асинхронного кода.
* Promise – это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).



* На promise можно навешивать коллбэки двух типов:
  + onFulfilled – срабатывают, когда promise в состоянии «выполнен успешно».
  + onRejected – срабатывают, когда promise в состоянии «выполнен с ошибкой».
* Способ использования, в общих чертах, такой:

1. Код, которому надо сделать что-то асинхронно, создаёт объект promise и возвращает его.
2. Внешний код, получив promise, навешивает на него обработчики.
3. По завершении процесса асинхронный код переводит promise в состояние fulfilled (с результатом) или rejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

* Синтаксис создания Promise:

**var promise = new Promise(function(resolve, reject) {**

**})**

* + В функции можно делать любые асинхронные операции,а когда они завершатся — нужно вызвать одно из:
    - resolve(результат) при успешном выполнении
    - reject(ошибка) при ошибке
* Универсальный метод для навешивания обработчиков:

**promise.then(onFulfilled, onRejected)**

* + onFulfilled – функция, которая будет вызвана с результатом при resolve.
  + onRejected – функция, которая будет вызвана с ошибкой при reject.

Пример: В результате запуска кода выше – через 1 секунду выведется «Fulfilled: result».

‘use strict’

// Создаётся объект promise

let promise = new Promise((resolve, reject) => {

// переведёт промис в состояние fulfilled с результатом "result"

setTimeout(() => { resolve("result"); }, 1000);

});

// promise.then навешивает обработчики на успешный результат или ошибку

promise

.then(

// первая функция-обработчик - запустится при вызове resolve

// result - аргумент resolve

result => { alert("Fulfilled: " + result); },

// вторая функция - запустится при вызове reject // error - аргумент reject

error => { ("Rejected: " + error); }

);

Помнить про [чейнинг](https://learn.javascript.ru/promise#цепочки-промисов).

# Методы массивов: перечислить. Знать “что делает” тот или иной метод.

|  |  |  |  |
| --- | --- | --- | --- |
| **Команда** | **Описание** | **Код** | |
| Pop() | Удаляет последний элемент из массива и возвращает его | var fruits = ["Яблоко", "Апельсин", "Груша"];  alert( fruits.pop() ); // удалили "Груша"  alert( fruits ); // Яблоко, Апельсин | |
| Push(obj) | Добавляет элемент в конец массива:  Вызов fruits.push(...) равнозначен fruits[fruits.length] = .... | var fruits = ["Яблоко", "Апельсин"];  fruits.push("Груша");  alert( fruits ); // Яблоко, Апельсин, Груша  //или  fruits.push("Мандарин", "Персик"); // результат: ["Яблоко ", " Апельсин ", " Груша ", " Мандарин ", "Персик"] | |
| Shift() | Удаляет из массива первый элемент и возвращает его | var fruits = ["Яблоко", "Апельсин", "Груша"];  alert( fruits.shift() ); // удалили Яблоко  alert( fruits ); // Апельсин, Груша | |
| Unshift(obj) | Добавляет элемент в начало массива | var fruits = ["Апельсин", "Груша"];  fruits.unshift('Яблоко');  alert( fruits ); // Яблоко, Апельсин, Груша  //или  fruits.unshift("Ананас", "Лимон"); // результат: ["Ананас", "Лимон", "Яблоко", "Апельсин", " Груша "] | |
| split(s) | **Метод позволяет превратить строку в массив, разбив ее по разделителю s.** | | |
|  | var names = 'Маша, Петя, Марина, Василий';  var arr = names.split(', '); // ["Маша", "Петя", "Марина", "Василий"] | |
| У метода есть необязательный второй аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано – остаток массива будет отброшен. | alert( "a,b,c,d".split(',', 2) ); // a,b | |
| Вызов split с пустой строкой разобьёт по буквам. | var str = "тест";  alert( str.split('') ); // т,е,с,т | |
| join(str) | **Вызов arr.join(str) делает в точности противоположное split. Он берет массив и склеивает его в строку, используя str как разделитель.** | | |
|  | var arr = ['Маша', 'Петя', 'Марина', 'Василий'];  var str = arr.join(';');  alert( str ); // "Маша;Петя;Марина;Василий" | |
| new Array + join = Повторение строки  Код для повторения строки 3 раза. | alert( new Array(4).join("ля") ); // ляляля  /\*  Как видно, new Array(4) делает массив без элементов длины 4, который join объединяет в строку, вставляя между его элементами строку "ля".  \*/ | |
| delete | **Так как массивы являются объектами, то для удаления ключа можно воспользоваться обычным delete** | | |
| var arr = ["Я", "иду", "домой"];  delete arr[1]; // значение с индексом 1 удалено  // теперь arr = ["Я", undefined, "домой"]; | | |
| splice (index[, deleteCount, elem1, ..., elemN]) | **Удалить deleteCount элементов, начиная с номера index, а затем вставить elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.** | | |
| Удаление. | var arr = ["Я", "изучаю", "JavaScript"];  arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент  // осталось ["Я", "JavaScript"] | |
| Удаление 3ех элементов и вставка других на их место. | var arr = ["Я", "сейчас", "изучаю", "JavaScript"];  // удалить 3 первых элемента и добавить другие вместо них  arr.splice(0, 3, "Мы", "изучаем")  // теперь ["Мы", "изучаем", "JavaScript"] | |
| Splice возвращает массив из удаленных элементов. | var arr = ["Я", "сейчас", "изучаю", "JavaScript"];  // удалить 2 первых элемента  var removed = arr.splice(0, 2); // "Я", "сейчас" <-- array of removed elements | |
| Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0. | var arr = ["Я", "изучаю", "JavaScript"];  // с позиции 2, удалить 0, вставить "сложный", "язык"  arr.splice(2, 0, "сложный", "язык");  alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript" | |
| Допускается использование отрицательного номера позиции, которая в этом случае отсчитывается с конца. | var arr = [1, 2, 5]  // начиная с позиции индексом -1 (перед последним элементом)  // удалить 0 элементов,  // затем вставить числа 3 и 4  arr.splice(-1, 0, 3, 4);  alert( arr ); // результат: 1,2,3,4,5 | |
| slice (begin, end) | **Метод slice(begin, end) копирует участок массива от begin до end, не включая end. Исходный массив при этом не меняется.** | | |
|  | var arr = ["Почему", "надо", "учить", "JavaScript"];  var arr2 = arr.slice(1, 3); // элементы 1, 2 (не включая 3)  alert( arr2 ); // надо, учить | |
| Если не указать end – копирование будет до конца массива. | var arr = ["Почему", "надо", "учить", "JavaScript"];  alert( arr.slice(1) ); // взять все элементы, начиная с номера 1 | |
| Можно использовать отрицательные индексы, они отсчитываются с конца. | var arr2 = arr.slice(-2); // копировать от 2-го элемента с конца и дальше | |
| Если вообще не указать аргументов – скопируется весь массив. | var fullCopy = arr.slice(); | |
| sort() | **Метод sort() сортирует массив на месте.** | | |
| По умолчанию sort сортирует, преобразуя элементы к строке. | var arr = [ 1, 2, 15 ];  arr.sort();  alert( arr ); // 1, 15, 2 | |
| Для указания своего порядка сортировки в метод arr.sort(fn) нужно передать функцию fn от двух элементов, которая умеет сравнивать их. | function compareNumeric(a, b)  {  if (a > b) return 1;  if (a < b) return -1;  } | function compareNumeric(a, b) {  return a - b;  } |
|  | var arr = [ 1, 2, 15 ];  arr.sort(compareNumeric);  alert(arr); // 1, 2, 15  /\* передаём в sort() именно саму функцию compareNumeric, без вызова через скобки. \*/ | |
| Значения, с которыми sort вызывает функцию сравнения, можно увидеть, если вставить в неё alert: | [1, -2, 15, 2, 0, 8].sort(function(a, b) {  alert( a + " <> " + b );  }); | |
| reverse | **Метод arr.reverse() меняет порядок элементов в массиве на обратный.** | | |
|  | var arr = [1, 2, 3];  arr.reverse();  alert( arr ); // 3,2,1 | |
| concat | **Метод arr.concat(value1, value2, … valueN) создаёт новый массив, в который копируются элементы из arr, а также value1, value2, ... valueN.** | | |
|  | var arr = [1, 2];  var newArr = arr.concat(3, 4);  alert( newArr ); // 1,2,3,4 | |
| Если аргумент concat – массив, то concat добавляет элементы из него. | var arr = [1, 2];  var newArr = arr.concat([3, 4], 5); // то же самое, что arr.concat(3,4,5)  alert( newArr ); // 1,2,3,4,5 | |
| indexOf / lastIndexOf | **Метод *arr.indexOf(searchElement[, fromIndex])* возвращает номер элемента searchElement в массиве arr или -1, если его нет. Поиск начинается с номера fromIndex, если он указан. Если нет – с начала массива.**  **\*** **Метод *arr.lastIndexOf(searchElement[, fromIndex])* ищет справа-налево: с конца массива или с номера fromIndex, если он указан.** | | |
| Для поиска используется строгое сравнение ===. | var arr = [1, 0, false];  alert( arr.indexOf(0) ); // 1  alert( arr.indexOf(false) ); // 2  alert( arr.indexOf(null) ); // -1 | |
| Методы indexOf/lastIndexOf осуществляют поиск перебором. Рассмотрим задачу – есть коллекция строк, и нужно быстро проверять: есть ли в ней какой-то элемент. Массив для этого не подходит из-за медленного indexOf. Но подходит объект! | var store = {}; // объект для коллекции  var items = ["div", "a", "form"];  for (var i = 0; i < items.length; i++) {  var key = items[i]; // для каждого элемента создаём свойство  store[key] = true; // значение здесь не важно  }  /\* Теперь для проверки, есть ли ключ key, достаточно выполнить if (store[key]). \*/ | |
| object.keys(obj) | **С его помощью можно перебрать свойства объекта.** | | |
|  | var user = {  name: "Петя",  age: 30  }  var keys = Object.keys(user);  alert( keys ); // name, age | |
| forEach | **Метод *arr.forEach(callback[, thisArg])* используется для перебора массива. Он для каждого элемента массива вызывает функцию callback. Этой функции он передаёт три параметра *callback(item, i, arr).*** | | |
| * **item – очередной элемент массива.** * **i – его номер.** * **arr – массив, который перебирается.**   **Метод forEach ничего не возвращает** | var arr = ["Яблоко", "Апельсин", "Груша"];  arr.forEach(function(item, i, arr) {  alert( i + ": " + item + " (массив:" + arr + ")" );  }); | |
| filter | **Метод *arr.filter(callback[, thisArg])* используется для фильтрации массива через функцию. Он создаёт новый массив, в который войдут только те элементы arr, для которых вызов *callback(item, i, arr)* возвратит true.** | | |
|  | var arr = [1, -1, 2, -2, 3];  var positiveArr = arr.filter(function(number) {  return number > 0;  });  alert( positiveArr ); // 1,2,3 | |
| map | **Метод *arr.map(callback[, thisArg])* используется для трансформации массива. Он создаёт новый массив, который будет состоять из результатов вызова *callback(item, i, arr)* для каждого элемента arr.** | | |
|  | var names = ['HTML', 'CSS', 'JavaScript'];  var nameLengths = names.map(function(name) {  return name.length;  });  // получили массив с длинами  alert( nameLengths ); // 4,3,10 | |
| every / some | **Эти методы используются для проверки массива.**  **Метод *arr.every(callback[, thisArg])* возвращает true, если вызов callback вернёт true для каждого элемента arr.**  **Метод *arr.some(callback[, thisArg])* возвращает true, если вызов callback вернёт true для какого-нибудь элемента arr.** | | |
|  |  | var arr = [1, -1, 2, -2, 3];  function isPositive(number) {  return number > 0;  }  alert( arr.every(isPositive) ); // false, не все положительные  alert( arr.some(isPositive) ); // true, есть хоть одно положительное | |
| reduce / reduceRight | **Метод *arr.reduce(callback[, initialValue])* используется для последовательной обработки каждого элемента массива с сохранением промежуточного результата. Метод reduce используется для вычисления на основе массива какого-либо единого значения, иначе говорят «для свёртки массива». Он применяет функцию callback по очереди к каждому элементу массива слева направо, сохраняя при этом промежуточный результат.** | | |
|  | **Аргументы функции *callback(previousValue, currentItem, index, arr):***   * **previousValue – последний результат вызова функции, он же «промежуточный результат».** * **currentItem – текущий элемент массива, элементы перебираются по очереди слева-направо.** * **index – номер текущего элемента.** * **arr – обрабатываемый массив.** | var arr = [1, 2, 3, 4, 5]  // для каждого элемента массива запустить функцию,  // промежуточный результат передавать первым аргументом далее  var result = arr.reduce(function(sum, current) {  return sum + current;  }, 0);  alert( result ); // 15 | |

# Преобразования типов

Всего есть три преобразования:

* Строковое преобразование.
* Численное преобразование.
* Преобразование к логическому значению.
* Строки:
  + Сначала сравниваются первые буквы, потом вторые, и так далее, пока одна не будет больше другой.
* Сравнение разных типов
  + При сравнении значений разных типов, используется числовое преобразование.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Преобразование | **Логическое** | **Численное** | **Строковое** |
| **Тип** |  |
| **null** | | false | 0 | “null” |
| **undefined** | | false | NaN | “undefined” |
| **boolean** | true | true | 1 | “true” |
| false | false | 0 | “false” |
| **number** | 0 | false | 0 | “0” |
| 123 | true | 123 | “123” |
| NaN | false | NaN | “NaN” |
| **string** | “” | false | 0 | “” |
| “Hello” | true | NaN | “Hello” |
| “123” | true | 123\* | “123” |
| [**object**](https://learn.javascript.ru/object-conversion) | В общем случае | true | \*\*\* | [object Object]\*\* |
| [1, 2] |  | "1,2"  выводит список элементов |
| new Date | кол-во миллисекунд, прошедших с 01.01.1970 | выводит дату в виде строки |
| function() {} |  | выводит её код |

(\*) Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки "считывается" число, при ошибке результат NaN.

(\*\*) Если в объекте присутствует метод toString, который возвращает примитив, то он используется для преобразования. Метод toString не обязан возвращать именно строку.

(\*\*\*) Для численного преобразования объекта используется метод valueOf, а если его нет – то toString. Метод valueOf обязан возвращать примитивное значение, иначе его результат будет проигнорирован. При этом – не обязательно числовое.

* Функция parseInt и ее аналог parseFloat преобразуют строку символ за символом, пока это возможно.
  + alert( parseInt('12px') ); // 12
  + alert( parseInt('a123') ); // NaN
* "" + 1 + 0 = “10”
* "" - 1 + 0 = -1
* true + false = 1
* 6 / "3" = 2
* "2" \* "3" = 6
* 4 + 5 + "px" = “9px”
* "$" + 4 + 5 = “$45”
* "4" – 2 = 2
* "4px" – 2 = NaN
* 7 / 0 = “Infinity”
* " -9\n" + 5 = “ -9\n5”
* " -9\n" – 5 = -14
* 5 && 2 = 2
* 2 && 5 = 5
* 5 || 0 = 5
* 0 || 5 = 5
* null + 1 = 1
* undefined + 1 = NaN
* null == "\n0\n" = false
* +null == +"\n0\n" = true

# null и undefined

* + Значения null и undefined равны == друг другу и не равны чему бы то ни было ещё. Это жёсткое правило буквально прописано в спецификации языка.
  + При преобразовании в число null становится 0, а undefined становится NaN.
    - alert( null > 0 ); // false

alert( null == 0 ); // false

alert( null >= 0); // true

alert( undefined > 0 ); // false; undefined ->NaN

alert( undefined < 0 ); // false; undefined ->NaN

alert( undefined == 0 ); // false; undefined ->NaN

# NaN

* значение NaN по стандарту устроено так, что сравнения ==, <, >, <=, >= и даже === с ним возвращают false.
* Если математическая операция не может быть совершена, то возвращается специальное значение NaN(Not-A-Number).
  + alert( 0 / 0 ); // NaN
* Значение NaN – единственное в своем роде, которое не равно ничему, включая себя.
  + if (NaN == NaN) alert( "==" ); // Ни один вызов
  + if (NaN === NaN) alert( "===" ); // не сработает
* Значение NaN можно проверить специальной функцией isNaN(n), которая преобразует аргумент к числу и возвращает true, если получилось NaN, и false – для любого другого значения.
  + var n = 0 / 0;
  + alert( isNaN(n) ); // true
  + alert( isNaN("12") ); // false, строка преобразовалась к обычному числу 12
    - Если аргумент isNaN – не число, то он автоматически преобразуется к числу.
* Значение NaN «прилипчиво». Любая операция с NaN возвращает NaN.
  + alert( NaN + 1 ); // NaN

# Контекст. this

* Для доступа к текущему объекту из метода используется ключевое слово this.
* Значение this называется **контекстом вызова** и будет определено в момент вызова функции.
* Контекст this никак не привязан к функции, даже если она создана в объявлении объекта. Чтобы this передался, нужно вызвать функцию именно через точку (или квадратные скобки).

|  |  |
| --- | --- |
| Интересно: |  |
| 1. Обычный вызов функции в контексте объекта. 2. То же самое, скобки ни на что не влияют. 3. Здесь не просто вызов obj.method(), а более сложный вызов вида (выражение).method(). Такой вызов работает, как если бы он был разбит на две строки: 4. f = obj.go; - вычислить выражение 5. f(); - вызвать то, что получилось   При этом f() выполняется как обычная функция, без передачи this.   1. Здесь также слева от точки находится выражение, аналогичное 3. | var obj, method;  obj = {  go: function() { alert(this); }  };  obj.go(); // (1) object  (obj.go)(); // (2) object  (method = obj.go)();// (3) undefined  (obj.go || obj.stop)(); // (4) undefined  // https://learn.javascript.ru/object-methods |
| Функция, объявленная без объекта, вполне допустима: | function sayHi() {  alert( this.firstName );  } |
| Если одну и ту же функцию запускать в контексте разных объектов, она будет получать разный this: | var user = { firstName: "Вася" };  var admin = { firstName: "Админ" };  function func() { alert( this.firstName ); }  user.f = func; admin.g = func;  // this равен объекту перед точкой:  user.f(); // Вася  admin.g(); // Админ  admin['g'](); // Админ (не важно, доступ к объекту через точку или квадратные скобки) |

## Call, Apply

* Вызов func.call(context, a, b...) – то же, что обычный вызов func(a, b...), но с явно указанным this(=context).

function showFullName() {

alert( this.firstName + " " + this.lastName );

}

var user = { /\*…\*/ };

showFullName.call(user) // "Василий Петров". функция вызовется с this=user

* Вызов функции при помощи func.apply работает аналогично func.call, но принимает массив аргументов вместо списка.

func.call(context, arg1, arg2);

// идентичен вызову

func.apply(context, [arg1, arg2]);

Значение this устанавливается в зависимости от того, как вызвана функция:

* При вызове функции как метода:

obj.func(...) // this = obj

obj["func"](...)

* При обычном вызове:

func(...) // this = window (ES3) /undefined (ES5)

* В new:

new func() // this = {} (новый объект)

* Явное указание:

func.apply(context, args) // this = context (явная передача)

func.call(context, arg1, arg2, ...)

# Замыкания

Замыкание – это функция вместе со всеми внешними переменными, которые ей доступны.

«Понимать замыкания» в JavaScript означает понимать следующие вещи:

* Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment. Каждый запуск функции создает новый такой объект. На верхнем уровне им является «глобальный объект», в браузере – window.
* При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
* При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя, а затем во внешних LexicalEnvironment с места своего «рождения».

Память: Объект переменных внешней функции существует в памяти до тех пор, пока существует хоть одна внутренняя функция, ссылающаяся на него через свойство [[Scope]].

# setTimeout и setInterval

* var timerId = setTimeout(func / code, delay[, arg1, arg2...])

Метод setInterval имеет синтаксис, аналогичный setTimeout.

* var timerId = setInterval(func / code, delay[, arg1, arg2...])
  + func/code - Функция или строка кода для исполнения. Строка поддерживается для совместимости, использовать её не рекомендуется.
  + delay - Задержка в миллисекундах, 1000 миллисекунд равны 1 секунде.
  + arg1, arg2… - Аргументы, которые нужно передать функции.
* setTimeout: Исполнение функции произойдёт спустя время, указанное в параметре delay.
* setInterval запускает выполнение функции не один раз, а регулярно повторяет её через указанный интервал времени

Функции setTimeout и setInterval возвращают числовой идентификатор таймера timerId, который можно использовать для отмены действия:

var timerId = setTimeout(...);

clearTimeout(timerId);

Важная альтернатива setInterval – рекурсивный setTimeout:

|  |  |
| --- | --- |
| Вместо | Альтернатива |
| var timerId = setInterval(function() {  alert( "тик" );  }, 2000); | var timerId = setTimeout(function tick() {  alert( "тик" );  timerId = setTimeout(tick, 2000);  }, 2000); |
|  |  |
| Реальная пауза между вызовами func при setInterval меньше, чем указана в коде!  Время работы функции никак не учитывается, оно «съедает» часть интервала. | При рекурсивном setTimeout задержка всегда фиксирована и равна 100 мс.  Более гибкий метод тайминга, так как время до следующего выполнения можно запланировать по-разному, в зависимости от результатов текущего. |

# Деструктуризация

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

|  |  |  |
| --- | --- | --- |
| **Синтаксис** | **Что получилось** | **Комментарий** |
| let [firstName, lastName] = ["Илья", "Кантор"]; | firstName = “Илья”  lastName = “Кантор” |  |
| let [, , title] = "Юлий Цезарь Император Рима".split(" "); | title = “Император” | первый и второй элементы не нужны |
| let [firstName, lastName, ...rest] = "Юлий Цезарь Император Рима".split(" "); | firstName = “Юлий”  lastName = “Цезарь”  rest = [“Император” , “Рима”] | Если мы хотим получить и последующие значения массива, но не уверены в их числе – можно добавить Оператор «spread» …  Значением rest будет массив. |
| let [firstName="Гость", lastName="Анонимный"] = []; | firstName = Гость  lastName = Анонимный | значения по умолчанию |
| let options = {  title: "Меню",  width: 100,  height: 200  };  let {title, width, height} = options; | title = Меню  width = 100  height = 200 | Деструктуризация объекта |
| let options = {  title: "Меню",  width: 100,  height: 200  };  let {width: w, height: h, title} = options; | title = Меню  w = 100  h = 200 | присвоить свойство объекта в переменную с другим именем |
| let options = {  title: "Меню"  };  let {width=100, height=200, title} = options; | title = Меню  width = 100  height = 200 | значение по умолчанию |

# Методы создания массива

* var arr = [элемент1, элемент2...];
* var arr = new Array("Яблоко", "Груша", "и т.п.");
  + new Array(*элементы*, ...) создаёт массив из данных *элементов*
  + new Array(*число*), то он создает массив без элементов, но с заданной *длиной*.

# Объявление функции

|  |  |  |  |
| --- | --- | --- | --- |
| **Название** | | **Код** | **Особенность** |
| Function Declaration | | function f() {  // тело функции  } | Создается до выполнения первой строчки кода, поэтому можно вызвать до объявления. |
| Function Expression |  | var f = function(параметры) {  // тело функции  }; | Создается когда управление достигает строки с функцией. Нельзя вызывать до объявления. Можно объявить в if. |
| Анонимные функции - функциональное выражение, которое не записывается в переменную. | ask(  "sure?",  function() { alert("yes"); },  function() { alert("no"); }  ); | Если мы не собираемся вызывать функицю ещё раз, то можно просто объявить ее непосредственно там, где она нужна. |
| Named Function Expression | var f = function sayHi(...) { /\* тело функции \*/ }; | Имя функционального выражения (sayHi) имеет особый смысл. Оно доступно только изнутри самой функции (f).  Как правило, имя NFE используется для единственной цели – позволить изнутри функции вызвать саму себя. (рекурсия) |
| new Function | | var sum = new Function('a,b', ' return a+b; '); | Можно конструировать функцию, код которой неизвестен на момент написания программы, но строка с ним генерируется или подгружается динамически во время её выполнения.  Её свойство [[Scope]] ссылается не на текущий LexicalEnvironment, а на window. |

Если нет явной причины использовать Function Expression – предпочитайте Function Declaration.

«На месте» разрешено вызывать только Function Expression:

(function() {

// ...

}());

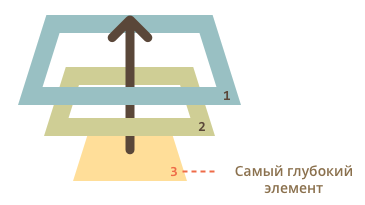
# Rest

* Если совсем простыми, то некий сервис даёт тебе доступ к своим данным, но не напрямую, а через оболочку, которая назвается REST API. Тебе даётся некий url, по которому ты можешь получить те самые данные. Как правило, при изменении параметров этого url, ты получаешь разные данные. Пример:
* Допустим ты берешь в качестве сервиса Википедию. Его специальный url, по которому ты можешь получить данные (его еще называют endpoint) выглядит так: <https://en.wikipedia.org/w/api.php>

# Всплытие элементов

* **Основной принцип всплытия:**

При наступлении события обработчики сначала срабатывают на самом вложенном элементе, затем на его родителе, затем выше и так далее, вверх по цепочке вложенности.



# Прекращение всплытия

* Всплытие идёт прямо наверх. Обычно событие будет всплывать наверх и наверх, до элемента <html>, а затем до document, а иногда даже до window, вызывая все обработчики на своем пути.
* Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие.

Для остановки всплытия нужно вызвать метод **event.stopPropagation().**

# Различие (preventDefault, stopPropagation, stopImmediatePropagation, return false из колбека обработчика)

* Если вы всего лишь хотите предотвратить действие браузера по умолчанию, то вам следует использовать **preventDefault** метод.
* Метод **stopPropagation()**, который останавливает всплытие (bubbling) события “клик” к родительским элементам.
* **stopImmediatePropagation()** останавливает не только всплытие события по родительским элементам, но также останавливает работу всех последующих обработчиков конкретного события на данном элементе.
* Возврат **false в callback**’е. Вроде бы все хорошо, но нет?
  + Первое – да, вы добились своей цели, вы предотвратили посещение ссылки (тут сработал event.preventDefault() метод), но вы также остановили распространение(propagation) события (по сути выполнили event.stopPropagation()) и теперь callback-фунция возвращает false.
  + Итак, как вы уже поняли такой способ увеличивает коэффициент кривизны кода.

# use strict

* ECMAScript 5 (ES5) добавил новые возможности и внёс в язык ряд исправлений, которые могут привести к тому, что старый код, который был написан до его появления, перестанет работать.
* Чтобы этого не случилось, решили, что по умолчанию эти опасные изменения будут выключены, и код будет работать по-старому. А для того, чтобы перевести код в режим полного соответствия современному стандарту, нужно указать специальную директиву **use strict**.

# Конструкция try…catch

* Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

try { /\* код ...\*/ }

catch (err) { /\* обработка ошибки\*/ }

* Работает она так:

1. Выполняется код внутри блока try.
2. Если в нём ошибок нет, то блок catch(err) игнорируется, то есть выполнение доходит до конца try и потом прыгает через catch.
3. Если в нём возникнет ошибка, то выполнение try на ней прерывается, и управление прыгает в начало блока catch(err).

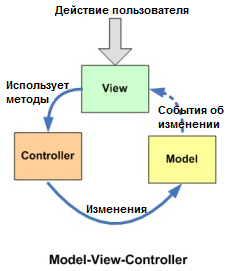
* В браузере существует специальное свойство window.onerror, если в него записать функцию, то она выполнится и получит в аргументах сообщение ошибки, текущий URL и номер строки, откуда «выпала» ошибка. Необходимо лишь позаботиться, чтобы функция была назначена заранее.

# New, Object.create()

* new X это Object.create(X.prototype) с дополнительным запуском функции constructor. (И давая constructor шанс вернуть фактический объект, который должен быть результатом выражения вместо this.)

Стрелочные функции

# MVC



React

# Типы компоненты. Отличия и особенности

## stateless и statefull

* Компоненты созданные с помощью class, называются statefull компоненты (то есть, компоненты с состоянием), а компоненты созданные с помощью функции - stateless component (то есть, компоненты без состояния). (вроде, это [заблуждение](https://medium.com/@abraztsov/паттерны-в-react-e5092c06f019).)
  + все что вы возвращаете в render методе или в return у stateless-компонента должно быть обернуто в один тэг / React.Fragment (ниже).
* У stateless компонентов нет методов жизненного цикла.
* У statefull компонента constructor нужен для привязки (bind) и задания state.

## презентационные и контейнер

## 

|  |  |
| --- | --- |
| Презентационные | Контейнеры |
|  |  |

* Выделение контейнеров – хорошая практика, даже вне зависимости от использование/не использования Redux.
* Это соответствует принципу: **разделение отвественности**, согласно которому компонент должен делать что-то одно и делать это хорошо.
* https://medium.com/@dan\_abramov/smart-and-dumb-components-7ca2f9a7c7d0

# Props

* У каждого компонента могут быть свойства. Они хранятся в this.props, и передаются компоненту как атрибуты.
* Общий вид:

const wizard = {name: Garry, surname: Potter};

<MyComponent data={wizard} eshe\_odno\_svoistvo={[1,2,3,4,5]} />

* В свойство можно передать любой javascript примитив, объект, переменную и даже выражение. Значение свойства должно быть взято в фигурные скобки.
* Значения доступны через this.props.ИМЯ\_СВОЙСТВА (в statefull-компонентах) или в первом аргументе функции (в stateless).

# Map

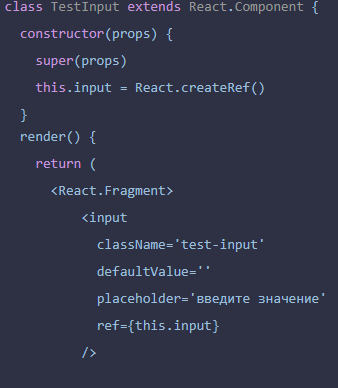
…раньше в map мы возвращали JSX-разметку. Но мы так же можем возвращать и компонент.

|  |  |
| --- | --- |
|  |  |

# Заметки

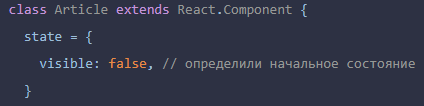
* стараемся в render держать как можно меньше кода, чтобы его было легко читать вашим коллегам.

Почему мы метод render не описываем через жирную стрелочную функцию? Потому что, это метод жизненного цикла react-компонента, и туда this "прокидывает" уже сам react.

* Главное отличие [неконтролируемого](https://reactjs.org/docs/uncontrolled-components.html) компонента от [контролируемого](https://reactjs.org/docs/forms.html#controlled-components) в том, что у него **нет обработчика изменений**, а значит нет постоянных вызовов setState и перерисовок.
  + ****Для того чтобы считать значение неконтролируемого компонента используется механизм refs.
* Для неконтролируемого компонента в момент начальной загрузки можно указывать defaultValue.
* Render может возвращать [[],примитив](https://www.youtube.com/watch?v=z8waaQ7WOoc&index=7&list=PLqHlAwsJRxANc2mFeSIRLdglGf2ZNzNBr). А также [React.Fragment](https://www.youtube.com/watch?v=GXgnR-n_yyw&list=PLqHlAwsJRxANc2mFeSIRLdglGf2ZNzNBr&index=8),
* Появились [порталы](https://www.youtube.com/watch?v=-qzRU7T4la0&index=5&list=PLqHlAwsJRxANc2mFeSIRLdglGf2ZNzNBr).

## state

* Для изменения состояния, нужно обязательно использовать метод setState, а не перезаписывать значение переменной в this.state напрямую.
* Можно так:



* Так же у setState есть возможность указать callback функцию, которая будет вызвана после того, как новое состояние "установится".
* Изменение state вызывает render компонента.
* нельзя вызывать setState внутри render: "реакт бдит".
* render - дорогостоящая операция, поэтому внимательно относитесь к тому, где вы вызываете setState, и что это за собой влечет. Банальные console.log могут вам в этом помочь.
* setState() - не изменяет this.state немедленно, а создает очередь изменений состояния. Доступ к this.state после вызова метода, потенциально может вернуть имеющееся (что равносильно - бывшее) значение.

# Prop-types

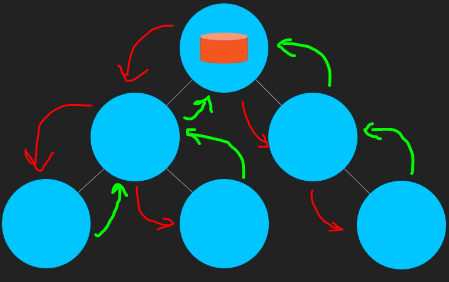
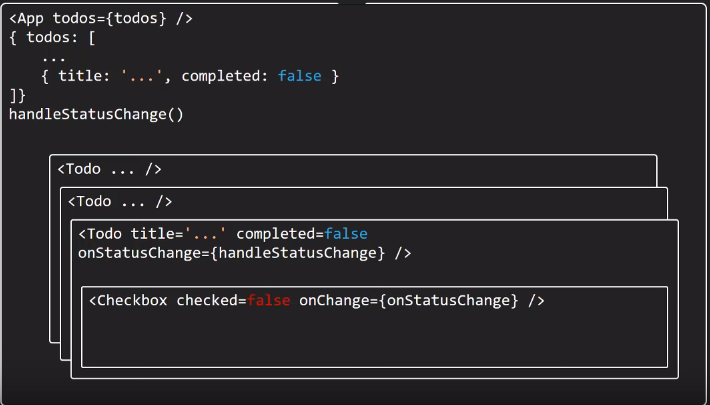
* Перед выполнением данного урока, не забывайте, что PropTypes не работает с production версией реакта. Эта фича только для разработки, так как валидация - дорогая операция.

# HOC

<https://www.youtube.com/watch?v=G2NBRu6TFPk>

# Однонаправленный поток данных

<https://www.youtube.com/watch?v=Y9bUR6IaBY4&index=3&list=PLqHlAwsJRxANc2mFeSIRLdglGf2ZNzNBr>



# Жизненный цикл компонент

* **componentWillMount** - компонент будет примонтирован. [DEPRECATED]
* **componentDidMount** - компонент примонтировался. Таймауты, ajax-запросы и взаимодействие с другими библиотеками стоит обрабатывать здесь.
* **componentWillReceiveProps** - компонент получает новые props. Этод метод не вызывается в момент первого render'a. В официальной документации очень хороший пример.
  + Обратите внимание: в этот момент, старые props доступны как this.props, а новые props доступны в виде nextProps аргумента функции. [DEPRECATED]
  + Так же, если вы вызываете setState внутри этого метода - не будет вызван дополнительный render

componentWillReceiveProps (nextProps) {

this.setState({

likesIncreasing:

nextProps.likeCount > this.props.likeCount

});

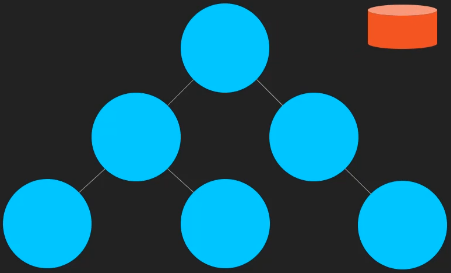
}

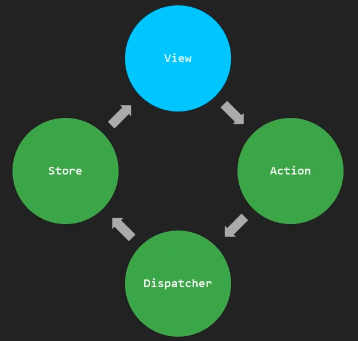
* **shouldComponentUpdate** - должен ли компонент обновиться? На самом деле, обычно реакт сам отлично разбирается. Но иногда ручное управление позволяет существенно ускорить работу в "узких местах". С этим методом нужно работать очень аккуратно.
* **componentWillUpdate** - вызывается прямо перед render, когда новые props и state получены. В этом методе нельзя вызывать setState. [DEPRECATED]
* **componentDidUpdate** - вызывается сразу после render. Не вызывается в момент первого render'а компонента.
* **componentWillUnmount** - вызывается перед тем, как компонент будет удален из DOM.

# Redux

* Для многих приложений передача данных в Однонаправленном потоке данных – является отличным решением.
* Но если приложение разрастается, то возникает слишком большая цепочка свойств сверху-вниз и событий снизу-вверх. Также трудности могут возникнуть при взаимодействии компонент из разных мест, которые напрямую не связаны. Для этого:

## Flux

* Придумали методологию Flux. Главная идея: деление состояния приложения от компонента. Теоретически, любой компонент может взаимодействовать с состоянием напрямую.



Методология flux состоит из 4-ех частей:

1. View – пользовательский интерфейс. В React – это компоненты.
2. Store – хранилище. В нем находится состояние приложения.
3. Dispatcher – диспетчер. Сообщает хранилищу о каком-либо событии. И передает необходимую информацию.
4. Action – действие/событие, которое происходит в приложении.

* Итог: компонент генерирует действие – диспетчер сообщает об этом хранилищу – хранилище изменяет состояние – данные передаются компоненту.
  + Сохраняется однонаправленный поток данных.
* Так как flux- лишь идея, то есть несколько конкретных реализаций. Например, redux.

## Функциональное программирование и reducer

В Redux применяется 2 идеи [чистых функций](https://www.youtube.com/watch?v=aqRR6zBkfj4&index=6&list=PLqHlAwsJRxANFIgAf7BO8hNYdvipLERxQ):

1. Чистые функции (pure functions);
2. Неизменность данных (immutability);

* Чистая функция – функция, зависящая ТОЛЬКО от входных параметров, а также НЕ меняющая и НЕ читающая глобальные переменные. Поэтому чистая функция должна не изменять значение, а возвращать новое.

“Строение” Redux Store: createStore принимает на вход функцию Reducer.

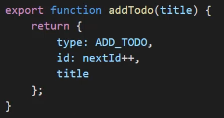
1. **Reducer (преобразователь)** – функция, которая имеет доступ к Store, и только она может его изменять. Принимает текущий state, а также action. Она описывает, КАК изменяется Store.
2. **Dispatcher** – функция, которая уведомляет хранилище о событии (action). По сути, она вызывает reducer, а также массив callback’ов, которые получены из subscribe.
3. **Action** – объект, который хранит в себе type – тип действия и payload – данные для изменения.
   1. **Action creator** – функция, принимающая параметр, а возвращающая объект-действие, который зависит от параметра. Обычно их выносят в отдельный файл.
4. **Subscribe** – хранилище при изменении состояния сообщает об этом. Subscribe принимает на вход callback и кладет его в массив функций, которые вызывает dispatcher, а возвращает unsubscribe, при вызове которого происходит отписка. Лучше подписывать в componentDidMount(), а отписываться в componentWillUnmount().

## Заметки:

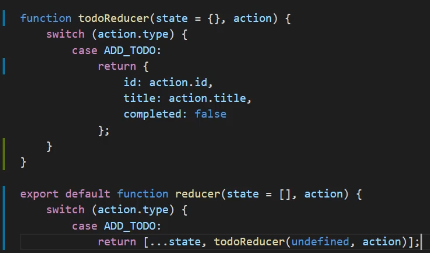
* InitialState указывается либо в reducer’e, либо в createStore.
* **Главным критерием** размещения состояния в хранилище является тот факт: используются ли данные локально лишь в одном компоненте или в нескольких.
  + Если данные используются лишь внутри компонента, лучше там их и оставить (State).
  + Если же данные используются в нескольких частях приложения, то лучше поместить их в хранилище (Store).

## Как внедрить хранилище:

* 1. Определить, какие данные необходимо поместить в хранилище, а какие в состояние компонента (см. выше).
  2. Решаем, каким образом можно будет менять состояние хранилища. Для каждого варианта создаем Action в ActionCreator.



* 1. Описываем в reducer’е случай для каждого action. Первым параметром можно указать initialState для хранилища. Вторым – action.
     1. Распространена практика Reducer Composition. Когда в Reducer’е выполняется слишком много когда, часть можно вынести в отдельную функцию.



* 1. const Store = createStore(reducer [, initialState]); передаем reducer и начальное значение. Но также его можно указать в reducer’е.

### Способы передачи хранилища компоненту

1. Через свойства:
   1. Прокинуть store в свойства
   2. Store запишем в свойства класса в конструкторе

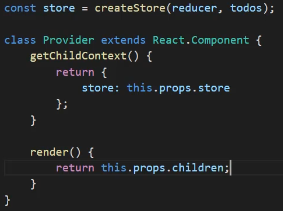
, а в stateless



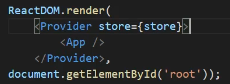
* 1. В компоненте App необходимо подписаться на обновление, ибо мы уберем state у App. Но forceUpdate() – плохая практика.
  2. Чтобы обратиться к store, можно:



1. Через контекст (нежелательно):
   1. Создадим класс



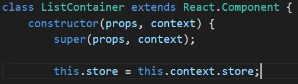
И там же:



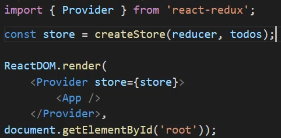
* 1. Укажем props



* 1. Укажем в конструкторе



1. Provider из react-redux + connect. Лучший вариант:
   1. Установить react-redux
   2. Обернуть компонент в Provider и передать в props



* 1. На этом этапе также необходимо указывать контекст как в 2.2 и 2.3.

## Connect

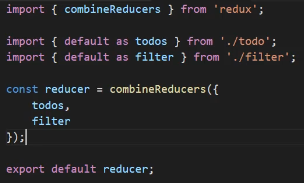
Все в этом [плейлисте](https://coursehunters.net/course/codedojo-redux), урок 26.

Если вкратце, connect упрощает container, который является “клеем” между хранилищем и презентационным компонентом.

|  |  |
| --- | --- |
| **Было** | **Стало** |
|  | Другие примеры:    Еще |

## Reducers

* Чтобы объединить reducer’ы, необходимо подключить combineReducers:



* + А import’ы – другие, более мелкие reducer’ы, вынесеные в отдельные файлы.