РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

JavaScript: элементы стиля

ПЕРЕВОД

ru_vds 22 мая 2017 в 13:11 22,6k

Оригинал: Eric Elliott

В 1920-м году вышла книга Уильяма Странка-младшего «Элементы стиля». Рекомендации из неё, касающиеся английского языка, актуальны и сегодня. Те же принципы, применённые к коду, позволяют повысить качество программ.

Надо заметить, что речь не идёт о жёстких правилах. То, о чём мы сегодня поговорим — лишь рекомендации. Даже если вы решите следовать им, вполне могут найтись веские причины для того, чтобы от них отклониться, например, если это поможет сделать код понятнее. Однако, поступая так, будьте бдительны и помните о том, что люди подвержены когнитивным искажениям. Например — выбирая между обычными и стрелочными функциями в JavaScript, тот, кто не очень хорошо знаком с последними, предпочтёт обычные функции, в силу привычки считая их понятнее, проще, удобнее.

Принципы из «Элементов стиля» не случайно живы до сих пор. Всё дело в том, что обычно их применение делает тексты лучше. Обычно автор книги оказывается прав. Отклоняться от них стоит лишь в тех случаях, когда на то есть веская причина — а не из-за прихоти или личных предпочтений.

Многие рекомендации из главы «Основные принципы композиции» применимы к программному коду:

1. Сделайте абзац минимальной частью композиции. Один абзац — одна тема.
2. Избегайте ненужных слов.
3. Используйте действительный залог.
4. Избегайте последовательностей слабо связанных предложений.
5. Слова в предложениях, связанные по смыслу друг с другом, не следует разделять другими языковыми конструкциями.
6. Используйте утвердительные высказывания.

7. Выражайте близкие по смыслу и назначению мысли в похожей форме, используя параллельные конструкции.

Почти то же самое можно сказать и о стиле кода:

1. Сделайте минимальной частью композиции функцию. Одна функция — одна задача.
2. Избегайте ненужного кода.
3. Используйте действительный залог.
4. Избегайте последовательностей слабо связанных языковых конструкций.
5. Держите в одном месте код и другие элементы программ, направленные на решение одной задачи.
6. Используйте утвердительную форму для имён переменных и при построении выражений.
7. Используйте одни и те же шаблоны для решения схожих задач.

Функция как единица композиции

> *Сущность разработки программного обеспечения — композиция. Мы создаём программы, компонуя модули, функции и структуры данных.*
>
> *Понимание процесса создания функций и того, как использовать их вместе с другими функциями — один из фундаментальных навыков программиста.*

Модуль — это коллекция функций или структур данных. Структуры данных — это то, как мы представляем состояние программы.

Однако, ничего интересного не происходит, пока дело не дойдёт до использования функций.

В JavaScript можно выделить три типа функций.

- Коммуникационные функции, то есть те, которые выполняют операции ввода-вывода.
- Процедурные функции, представляющие собой набор сгруппированных инструкций для решения некоей алгоритмической задачи.
- Функции маппинга, принимающие данные, преобразующие их, и возвращающие то, что получилось.

Функции для реализации операций ввода-выводы и неких алгоритмов обработки данных нужны практически везде, но подавляющее большинство функций, которые вам придётся использовать, будут заниматься маппингом.

Одна задача — одна функция

Если ваша функция предназначена для выполнения операций ввода-вывода, не занимайтесь в ней задачами маппинга. Если функция предназначена для маппинга, не выполняйте в ней операций ввода-вывода.

Надо сказать, что процедурные функции нарушают и правило «одна функция — одна задача», и правило, касающееся слабо связанных языковых конструкций. Однако, без таких функций не обойтись.

Идеальная функция — это простая, детерминированная, чистая функция, обладающая следующими основными свойствами:

- Одни и те же входные данные всегда дают один и тот же выход.
- При её вызове отсутствуют побочные эффекты.

Избыточный код

> *Энергичный текст лаконичен. В предложении не должно быть ненужных слов, в абзаце — ненужных предложений, по той же причине, по которой в чертеже не должно быть ненужных линий, а в механизме — лишних деталей. Это не значит, что пишущий должен использовать лишь короткие предложения, или, избегая деталей, обходиться общими описаниями. Это значит, что каждое слово должно иметь значение. [Лишние слова опущены].*
>
> *Уильям Странк-младший, «Элементы стиля»*

Лаконичный код весьма важен в деле разработки ПО, так как чем больше кода — тем больше мест, где можно допустить ошибку. *Меньший объём кода означает меньше мест, где может скрыться ошибка, что ведёт к уменьшению количества ошибок.*

Лаконичный код легче читать, так как он имеет более высокий уровень соотношения полезных данных к информационным «помехам». Читателю нужно отсеять меньше синтаксического «шума» для того, чтобы понять смысл программы. Таким образом,

меньший объём кода означает меньше синтаксического «шума», и, как результат, более чёткую передачу смысла.

Если выразиться словами из «Элементов стиля», сжатый код — это код энергичный. Вот, например, такая конструкция:

```
function secret (message) {
  return function () {
    return message;
  }
};
```

Её можно сократить до такой:

```
const secret = msg => () => msg;
```

Тем, кто знаком с минималистичным синтаксисом, характерным для стрелочных функций (они появились в ES6 в 2015-м), легче читать эту запись, а не код из первого примера. Ненужные элементы, такие, как скобки, ключевое слово `function`, инструкция `return`, здесь опущены.

В первом варианте много служебных синтаксических конструкций. Это и скобки, и ключевое слово `function`, и `return`. Они, для того, кто знаком со стрелочными функциями — не более, чем синтаксический «шум». И, в современном JavaScript, подобные конструкции существуют лишь для того, чтобы код могли читать те, кто пока недостаточно уверенно владеет ES6. Хотя, ES6 стал стандартом языка ещё в 2015-м, так что время узнать его получше

уже давно пришло.

Ненужные переменные

Иногда мы даём имя чему-то, для чего оно не очень-то и нужно. Скажем, некоей промежуточной переменной, без которой можно обойтись. Почему это вредно? Проблема тут в том, что человеческий мозг имеет ограниченные ресурсы кратковременной памяти. Встретив в тексте программы переменную, мы вынуждены запоминать её. Если имён много, наша память переполняется, при чтении периодически приходится возвращаться назад…

Именно поэтому опытные разработчики приучают себя к устранению ненужных переменных.

Например, в большинстве ситуаций следует избегать переменных, созданных лишь для хранения возвращаемого значения функции. Имя функции должно давать адекватные сведения о том, что именно функция возвратит. Рассмотрим пример:

```
const getFullName = ({firstName, lastName}) => {
  const fullName = firstName + ' ' + lastName;
  return fullName;
};
```

Избавившись от ненужного, код можно переписать так:

```
const getFullName = ({firstName, lastName}) => (
  firstName + ' ' + lastName
);
```

Ещё один распространённый подход к сокращению числа переменных заключается в применении композиции функций и так называемой «бесточечной нотации».

Бесточечная нотация — это способ объявления функций без упоминания аргументов, которыми оперируют эти функции. Обычными способами применения такого подхода являются каррирование и композиция функций.

Вот пример каррирования:

```
const add2 = a => b => a + b;
// теперь мы можем объявить бесточечную версию inc(),
// которая позволяет добавить 1 к любому числу.
const inc = add2(1);
inc(3); // 4
```

Взгляните на объявление функции `inc()`. Обратите внимание на то, что здесь нет ни ключевого слова `function`, ни синтаксических элементов, характерных для объявления стрелочных функций. Нет здесь и описания параметров функции, так как функция не использует их. Вместо этого она возвращает другую функцию, которая знает, что делать с переданными ей аргументами.

Взглянем на пример, в котором используется композиция функций. Композиция функций — это применение функции к результатам, возвращаемым другой функцией. Осознаёте вы это или нет, но вы применяете композицию функций постоянно.

Например, когда пользуетесь цепочками вызовов методов вроде `.map()` и `promise.then()`. Если обратиться к наиболее общей форме записи композиции функций, то получится такая конструкция: `f(g(x))`. В математике это обычно записывают как `f ∘ g`, что читается как «применение функции `f` к результату функции `g`».

Задействуя композицию двух функций, вы избавляетесь от необходимости создавать переменную для хранения промежуточного значения между вызовами функций.

Посмотрим, как этот приём позволяет писать более чистый код:

```
const g = n => n + 1;
const f = n => n * 2;
// С использованием точечной нотации:
const incThenDoublePoints = n => {
  const incremented = g(n);
  return f(incremented);
};
incThenDoublePoints(20); // 42
// compose2 — Принимает две функции и возвращает их композицию
const compose2 = (f, g) => x => f(g(x));
// В бесточечной нотации:
const incThenDoublePointFree = compose2(f, g);
incThenDoublePointFree(20); // 42
```

То же самое можно сделать с любой функцией.

Функтором называют объект, реализующий функцию маппинга. Например, в JS это массивы (`Array.map()`) или промисы (`promise.then()`). Напишем ещё одну версию функции `compose2`, используя цепочку вызовов функций маппинга для

целей композиции функций:

```
const compose2 = (f, g) => x => [x].map(g).map(f).pop();
const incThenDoublePointFree = compose2(f, g);
incThenDoublePointFree(20); // 42
```

Практически то же самое вы делаете всякий раз, используя цепочки вызовов в промисах.

По сути, каждая библиотека функционального программирования реализует минимум два способа композиции функций. Это функция `compose()`, которая применяет функции справа налево, и `pipe()`, которая применяет функции слева направо.

Например, в Lodash такие функции называются, соответственно, `compose()` и `flow()`. Когда я применяю эту библиотеку, то пользуюсь функцией `flow()` так:

```
import pipe from 'lodash/fp/flow';
pipe(g, f)(20); // 42
```

Однако, такой функционал можно реализовать и самостоятельно, без библиотек:

```
const pipe = (...fns) => x => fns.reduce((acc, fn) => fn(acc), x);
pipe(g, f)(20); // 42
```

Если вышеописанное кажется вам чем то очень уж заумным, и вы не знаете, как бы вы всем этим воспользовались, поразмыслите

вот над чем:

Сущность разработки программного обеспечения — композиция. Мы создаём программы, компонуя небольшие модули, функции и структуры данных.

Понимание инструментов для композиции функций и объектов так же важно для программиста, как для строителя — умение управляться с дрелью и монтажным пистолетом. А использование императивного кода для объединения функций и неоправданное применение переменных для хранения промежуточных результатов напоминает сборку мебели с помощью клейкой ленты.

В итоге предлагаем вам запомнить следующее:

- Если есть возможность выразить некую идею в меньшем объёме кода, не меняя и не запутывая её смысла — так и поступите.
- То же самое касается и переменных. Если есть такая возможность, и это не нанесёт ущерб логике и понятности программы — чем меньше будет переменных — тем лучше.

Действительный залог

> *Действительный залог обычно означает более ясное и живое выражение мысли, нежели страдательный.*
>
> *Уильям Странк-младший, «Элементы стиля»*

Давайте программным конструкциям настолько ясные и чёткие имена, насколько это возможно:

- `myFunction.wasCalled()` лучше, чем `myFunction.hasBeenCalled()`
- `createUser()` лучше, чем `User.create()`
- `notify()` лучше, чем `Notifier.doNotification()`

Называйте функции-предикаты и логические переменные так, будто они — это вопросы, допускающие ответ «да» или «нет»:

- `isActive(user)` лучше, чем `getActiveStatus(user)`
- `isFirstRun = false;` лучше, чем `firstRun = false;`

Используйте глагольные формы в именах функций:

- `increment()` лучше, чем `plusOne()`
- `unzip()` лучше, чем `filesFromZip()`
- `filter(fn, array)` лучше, чем `matchingItemsFromArray(fn, array)`

Обработчики событий

Именование обработчиков событий и методов жизненного цикла является исключением из правила использования глаголов в именах функций, так как они применяются как квалификаторы. Они показывают, не «что» делать, а «когда». Именовать их следует,

придерживаясь такой схемы: «<когда выполнять действие>, <глагол>».

- `element.onClick(handleClick)` лучше, чем `element.click(handleClick)`
- `component.onDragStart(handleDragStart)` лучше, чем `component.startDrag(handleDragStart)`

Имена обработчиков событий из списка, которые признаны неудачными, выглядят так, как будто мы хотим вызвать событие, а не отреагировать на него.

Методы жизненного цикла

Взгляните на следующие варианты методов жизненного цикла гипотетического компонента, которые созданы для вызова функции-обработчика перед обновлением этого компонента:

- `componentWillBeUpdated(doSomething)`
- `componentWillUpdate(doSomething)`
- `beforeUpdate(doSomething)`

В первом примере мы используем страдательный залог («будет обновлён», а не «обновит»). Такое название избыточно, оно не яснее других вариантов.

Второй пример выглядит лучше, но смысл этого метода жизненного цикла заключается в вызове обработчика. Имя `componentWillUpdate(handler)` читается так, будто компонент

собирается воздействовать на обработчик, обновить его, что не выражает истинного значения этой программной конструкции. Мы имеем в виду следующее: «Прежде чем компонент обновится, вызови обработчик». Имя `beforeComponentUpdate()` выражает наше намерение яснее всего.

Мы можем и дальше пойти по пути упрощения. Так как речь идёт о методах объекта, при их вызове будет упомянут и сам объект. Это значит, что добавление имени объекта к имени метода избыточно. Подумайте о том, как будет выглядеть следующая конструкция, если вызвать метод, обращаясь к компоненту: `component.componentWillUpdate()`. Это будет читаться так же, как: «У Васи Васи будут на обед котлеты». Двойное упоминание имени объекта избыточно. В результате, получается следующее: `component.beforeUpdate(doSomething)` лучше, чем `component.beforeComponentUpdate(doSomething)`.

Функциональные примеси — это функции, которые добавляют свойства и методы к объектам. Такие функции вызывают друг за другом в конвейере, напоминающем сборочную линию на заводе. Каждая функция принимает на входе `instance`, объект, и что-то к нему добавляет, прежде чем передать следующей функции в конвейере.

Я предпочитаю именовать такие функции, используя прилагательные. Для того, чтобы подобрать подходящее слово, можно воспользоваться суффиксами «ing» и «able». Вот примеры:

- const duck = composeMixins(flying, quacking);

- const box = composeMixins(iterable, mappable);

Последовательности слабо связанных языковых конструкций

… череда высказываний скоро становится однообразной и скучной.

Уильям Странк-младший, «Элементы стиля».

Разработчики наполняют функции последовательностями языковых конструкций. Эти конструкции задуманы так, чтобы выполнялись они одна за другой, по сути, являясь примером череды слабо связанных высказываний. Подобный подход, когда в некоем блоке программы собрано слишком много таких вызовов, ведёт к появлению так называемого «спагетти-кода».

Кроме того, наборы вызовов часто повторяются во множестве схожих форм. При этом каждый из повторяющихся блоков вполне может немного отличаться от других, а часто такие отличия возникают совершенно неожиданно. Например, основные потребности некоего компонента пользовательского интерфейса соответствуют потребностям практически всех подобных компонентов. Реализовать то, что нужно всем этим компонентам, можно, основываясь на различных стадиях их жизненного цикла, разбив реализацию на несколько функций.

Рассмотрим такую последовательность вызовов:

```
const drawUserProfile = ({ userId }) => {
  const userData = loadUserData(userId);
  const dataToDisplay = calculateDisplayData(userData);
  renderProfileData(dataToDisplay);
};
```

Эта функция выполняет три разных дела: загрузку данных, построение, на основе того, что было загружено, модели данных элемента интерфейса, и вывод элемента на страницу.

В большинстве современных библиотек для разработки интерфейсов каждую из вышеописанных задач решают отдельно от других, скажем, с помощью выделенной функции. Разделяя эти задачи, мы можем без особых проблем комбинировать функции, достигая нужного результата в различных ситуациях.

При таком подходе мы могли бы полностью заменить, скажем, функцию вывода компонента, и это не повлияло бы на другие части программы. В React, например, имеется множество подсистем рендеринга, предназначенных для разных платформ и разных сценариев использования библиотеки. Вот далеко не полный их список: ReactNative для нативных iOS и Android-приложений, AFrame для WebVR, ReactDOM/Server для отрисовки компонентов на стороне сервера.

Ещё одна проблема с вышеописанной функцией заключается в том, что она не позволяет подготовить модель элемента интерфейса и вывести её на страницу, не загрузив сначала исходные данные. Что если эти данные уже загружены? В конечном итоге, если подобная функция, совмещающая в себе

несколько операций, вызывается несколько раз, это приводит к выполнению ненужных действий.

Разделение операций, кроме того, открывает дорогу к их независимому тестированию. В процессе написания кода я постоянно запускаю модульные тесты для того, чтобы сразу же оценивать влияние на приложение вносимых в него изменений. Однако, если, как в нашем примере, объединить код рендеринга элемента управления с кодом загрузки исходных данных, не получится просто передать какие-нибудь условные данные функции вывода элемента для тестовых целей. Тут придётся тестировать всё — и загрузку, и подготовку, и вывод данных. Это, если проверить надо лишь что-то одно, приведёт к неоправданным затратам времени: данные, например, надо загрузить по сети, обработать, вывести в браузер… Для получения результатов тестирования придётся ждать дольше, чем при проверке отдельного компонента. Разделение функций позволит тестировать их отдельно от других частей приложения.

В нашем примере уже есть три отдельных функции, вызовы которых вполне можно поместить в разные методы жизненного цикла компонента. Например, загрузить исходные данные можно при подключении компонента, обработку этих данных и вывод компонента на экран можно выполнить в ответ на событие, связанное с обновлением состояния элемента интерфейса.

Применение вышеописанных принципов приводит к появлению программного обеспечения с более чётко определёнными сферами ответственности отдельных его компонентов. Каждый из

компонентов может повторно использовать одни и те же структуры данных и обработчики событий жизненного цикла, в результате мы не выполняем по много раз действия, которые достаточно выполнить лишь однажды.

Хранение кода и других элементов программ, направленных на решение одной задачи

Многие фреймворки и шаблоны предусматривают организацию файлов программ по их типу. Если речь идёт о простом проекте, вроде маленького калькулятора, или ToDo-приложения, проблем такой подход не вызовет, но в более крупных разработках лучше группировать файлы в соответствии с функционалом приложения, который они реализуют.

Вот, например, два варианта иерархии файлов для ToDo-приложения. Первый вариант представляет группировку файлов по типу:

```
.
├── components
│   ├── todos
│   └── user
├── reducers
│   ├── todos
│   └── user
└── tests
    ├── todos
    └── user
```

Второй — группировку по логическому принципу:

```
.
├── todos
│   ├── component
│   ├── reducer
│   └── test
└── user
    ├── component
    ├── reducer
    └── test
```

Группировка файлов по принципу реализуемого ими функционала позволяет, при необходимости внесения изменений в какую-то часть приложения, не переходить постоянно из папки в папку в поисках нужных файлов.

В итоге, рекомендуем группировать файлы, основываясь на том, какой функционал приложения они реализуют.

Использование утвердительной формы для имён переменных и при построении выражений

> *Делайте ясные утверждения. Избегайте вялого, бесцветного, нерешительного, уклончивого языка. Не используйте слово как средство отрицания, в антитезе, или как способ уклониться от темы.*
>
> *Уильям Странк-младший, «Элементы стиля»*

Перейдём сразу к примерам имён переменных:

- `isFlying` лучше, чем `isNotFlying`
- `late` лучше, чем `notOnTime`

Условный оператор

Такая конструкция:

```
if (err) return reject(err);
// делаем что-нибудь...
```

… лучше такой:

```
if (!err) {
  // ... делаем что-нибудь
} else {
  return reject(err);
}
```

Тернарный оператор

Так:

```
{
  [Symbol.iterator]: iterator ? iterator : defaultIterator
}
```

…лучше, чем так:

```
{
  [Symbol.iterator]: (!iterator) ? defaultIterator : iterator
}
```

Об отрицательных высказываниях

Иногда логическая переменная интересует нас лишь в ситуациях, когда её значение ложно. Использование для такой переменной имени в утвердительной форме приведёт к тому, что при её проверке придётся применять оператор логического отрицания, `!`. В подобных случаях лучше давать переменным чёткие отрицательные имена. Слово «not» в имени переменной и оператор `!` в операциях сравнения приводят к появлению размытых формулировок. Рассмотрим несколько примеров.

`if (missingValue)` лучше, чем `if (!hasValue)`

`if (anonymous)` лучше, чем `if (!user)`

`if (isEmpty(thing))` лучше, чем `if (notDefined(thing))`.

Аргументы функций, принимающих значения null и undefined

Не создавайте функции, при вызове которых необходимо передавать `undefined` или `null` вместо необязательных параметров. В подобных ситуациях лучше всего пользоваться объектом с именованными параметрами:

```
const createEvent = ({
  title = 'Untitled',
  timeStamp = Date.now(),
  description = ''
}) => ({ title, description, timeStamp });
// позже...
const birthdayParty = createEvent({
  title: 'Birthday Party',
  description: 'Best party ever!'
});
```

…лучше, чем:

```
const createEvent = (
  title = 'Untitled',
  timeStamp = Date.now(),
  description = ''
) => ({ title, description, timeStamp });
// позже...
const birthdayParty = createEvent(
  'Birthday Party',
  undefined, // Этого можно было избежать
  'Best party ever!'
);
```

Шаблоны и решение схожих задач

> *… параллельное построение требует внешней схожести фрагментов текста, имеющих сходное содержание и назначение. Подобие формы позволяет читателю легче распознавать сходство содержимого.*
>
> *Уильям Странк-младший, «Элементы стиля»*

При создании приложений программисту часто надо решать очень похожие друг на друга задачи. Кода, который может повторяться, обычно гораздо больше, чем совершенно уникального. Как результат, в ходе работы приходится постоянно делать одно и то же. Хорошо здесь то, что это даёт возможность для обобщения похожего кода и создания абстракций. Для этого достаточно выявить одинаковые части кода, выделить их и использовать везде, где они нужны. А в ходе разработки обращать внимание лишь на уникальные для того или иного фрагмента приложения

конструкции. На самом деле, именно этой цели служат различные библиотеки и фреймворки.

Вот, например, компоненты пользовательского интерфейса.И десяти лет не прошло с тех пор, когда обычным делом было валить в одну кучу обновления интерфейса с помощью jQuery, логику приложения и организацию его взаимодействия с внешним миром. Позже к программистам начало приходить понимание, что в клиентских веб-приложениях вполне можно использовать MVC, и они начали отделять модели от логики обновления интерфейса.

В итоге веб-приложения стали строить, применяя компонентный подход, что позволило декларативно моделировать компоненты, используя нечто вроде шаблонов, созданных с помощью HTML или JSX.

Всё это привело к использованию одинаковой логики обновления пользовательского интерфейса для всех компонентов, что гораздо лучше, чем уникальный императивный код.

Тем, кто знаком с компонентами, очень легко понять, как они работают, даже если речь идёт о незнакомом им приложении. А именно, знающему человеку сразу понятно, что есть некая декларативная разметка, описывающая элементы пользовательского интерфейса, обработчики событий для управления поведением компонента, и события жизненного цикла, к которым привязывают функции обратного вызова, вызываемые тогда, когда это нужно.

Когда мы используем одинаковые шаблоны для решения сходных задач, любой, кто знаком с шаблонами, сможет быстро понять, что именно делает код.

Выводы: код должен быть простым, но не упрощённым

Стандарт ES6 был принят в 2015-м, но и сегодня, через два года, многие разработчики избегают новых возможностей. Они стремятся писать код, который, по их мнению, легче читать, лишь потому, что им так привычнее. Среди таких новых возможностей — оператор ..., стрелочные функции, неявный возврат. Уход от новых технологий ради привычных, но устаревших — большая ошибка. Знакомство с новым происходит через практику, а после того, как возможности ES6 становятся привычными, совершенно очевидным становится их преимущество перед альтернативами из ES5. Сжатый код проще в сравнении с перегруженной синтаксическими конструкциями альтернативой.

Итак, код должен быть простым, но не упрощённым. Принимая это во внимание, компактный код обладает следующими преимуществами:

- Он менее подвержен ошибкам.
- Его легче отлаживать.

Если, при таком подходе, подумать об ошибках, то получается следующее:

- Их долго и дорого исправлять.
- Одни ошибки ведут к появлению других.
- Ошибки тормозят процесс работы над основным функционалом программных проектов.

Если учесть вышесказанное, то компактный код обладает ещё и следующими полезными свойствами:

- Его легче писать.
- Его легче читать.
- Его легче поддерживать.

Новые синтаксические конструкции, продвинутые методики, вроде каррирования и композиции функций, дают программисту преимущества, позволяя писать более качественный код. В изучение всего этого стоит вложить время и силы. Отказ от нового, возможно, прикрытый заботой о тех, кто будет читать код и может ничего не понять, приводит к написанию программ, которые похожи на сюсюканье взрослого с малышом, едва научившимся ходить.

Совершенно естественно допускать, что читателю кода ничего не известно о реализации тех или иных механизмов, но не надо считать его туповатым или не знающим языка.

Выражение мыслей в коде должно быть ясным, но не примитивно упрощённым. Это и вредно, и ведёт к пустой трате времени. Советуем каждому, кто всё ещё не пишет на ES6, подумать о том, чтобы, через практику, разобраться с новыми возможностями

языка, обогатить собственный «словарь программиста» и сделать свои программные тексты лаконичнее и понятнее. И, конечно, надеемся, что идеи из книги Уильяма Странка-младшего, применённые к JS, помогут вам улучшить ваш код.

Уважаемые читатели! А какими возможностями ES6 пользуетесь вы?

Похожие публикации

MVC на чистом JavaScript

28

ПЕРЕВОД

ru_vds • 21 июля 2017 в 15:00

Путь к трансдьюсерам на чистом JavaScript

12

ПЕРЕВОД

ru_vds • 26 мая 2017 в 14:08

JavaScript-тренды, на которые стоит обратить внимание в 2017-м

87

ПЕРЕВОД

ru_vds • 9 января 2017 в 14:26

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни

33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018

70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ

40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

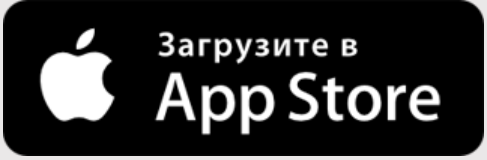
8

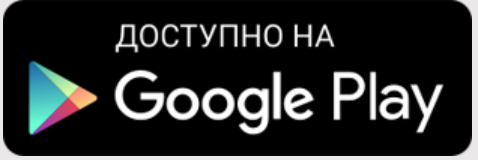
marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32





Полная версия