O'REILLY®

«Вряд ли вы сможете найти более глубокий или вдумчивый анализ ES6».

Ангус Кролл, автор книги «If Hemingway Wrote JavaScript»

КАЙЛ СИМПСОН

ES6 & НЕ ТОЛЬКО

ВЫ НЕ ЗНАЕТЕ JS

ББК 32.988.02-018
УДК 004.738.5
С37

Симпсон К.

С37 ES6 и не только. — СПб.: Питер, 2018. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02445-7

Даже если у вас уже есть опыт работы с JavaScript, скорее всего, язык вы в полной мере не знаете. Особое внимание в этой книге уделяется новым функциям, появившимся в Ecmascript 6 (ES6) — последней версии стандарта JavaScript.

ES6 повествует о тонкостях языка, малознакомых большинству работающих на JavaScript программистов. Вооружившись этими знаниями, вы достигнете подлинного мастерства; выучите новый синтаксис; научитесь корректно использовать итераторы, генераторы, модули и классы; сможете более эффективно работать с данными; познакомитесь с новыми API, например Array, Object, Math, Number и String; расширите функционал программ с помощью мета-программирования.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1491904244 англ.

ISBN 978-5-496-02445-7

Оглавление

1 ES: современность и будущее

Для чтения этой книги вы должны хорошо владеть языком JavaScript вплоть до последнего (на момент написания книги) стандарта, который называется ES5 (точнее, ES5.1), поскольку мы с вами будем рассматривать новый стандарт ES6, попутно пытаясь понять, какие перспективы ждут JS.

Если вы не очень уверены в своих знаниях JavaScript, рекомендую предварительно ознакомиться с предшествующими книгами серии *You Don't Know JS*.

- *Up & Going*: Вы только начинаете изучать программирование и JS? Перед вами карта, которая поможет вам в путешествии по новой области знаний.
- *Scope & Closures*: Известно ли вам, что в основе лексического контекста JS лежит семантика компилятора (а не интерпретатора)? Можете ли вы объяснить, каким образом замыкания являются прямым результатом лексической области видимости и функций как значений?
- *this & Object Prototypes*: Можете ли вы назвать четыре варианта значения ключевого слова `this` в зависимости от контекста вызова? Приходилось ли вам путаться в псевдоклас-

сах JS, вместо того чтобы воспользоваться более простым шаблоном проектирования behavior delegation? А слышали ли вы когда-нибудь про объекты, связанные с другими объектами (OLOO)?

- *Types & Grammar*: Знакомы ли вы со встроенными типами в JS и, что более важно, знаете ли способы корректного и безопасного приведения типов? Насколько уверенно вы разбираетесь в нюансах грамматики и синтаксиса этого языка?
- *Async & Performance*: Вы все еще используете обратные вызовы для управления асинхронными действиями? А можете ли вы объяснить, что такое объект promise и как он позволяет избежать ситуации, когда каждая фоновая операция возвращает свой результат (или ошибку) в обратном вызове? Знаете ли вы, как с помощью генераторов улучшить читабельность асинхронного кода? Наконец, известно ли вам, что представляет собой полноценная оптимизация JS-программ и отдельных операций?

Если вы уже прочитали все эти книги и освоили рассматриваемые там темы, значит, пришло время погрузиться в эволюцию языка JS и исследовать перемены, которые ждут нас как в ближайшее время, так и в отдаленной перспективе.

В отличие от предыдущего стандарта, ES6 нельзя назвать еще одним скромным набором добавленных к языку API. Он принес с собой множество новых синтаксических форм, и к некоторым из них, вполне возможно, будет не так-то просто привыкнуть. Появились также новые структуры и новые вспомогательные модули API для различных типов данных.

ES6 — это шаг далеко вперед. Даже если вы считаете, что хорошо знаете JS стандарта ES5, вы столкнетесь с множеством незнакомых вещей, так что будьте готовы! В книге рассмотрены все основные нововведения ES6, без которых невозможно войти в курс дела, а также дан краткий обзор планируемых функций — о них имеет смысл знать уже сейчас.

Весь приведенный в книге код рассчитан на среду исполнения ES6+. На момент написания этих строк уровень поддержки ES6 в браузерах и в JS-средах (таких, как Node.js) несколько разнился, так что вы можете обнаружить, что полученный вами результат отличается от описанного.

Поддержка версий

Стандарт JavaScript официально называется ECMAScript (или сокращенно ES), и до недавнего времени все его версии обозначались только целыми числами. ES1 и ES2 не получили известности и массовой реализации. Первой широко распространившейся основой для JavaScript стал ES3 — стандарт этого языка для браузеров Internet Explorer с 6-й по 8-ю версию и для мобильных браузеров Android 2.x. По политическим причинам, о которых я умолчу, злополучная версия ES4 так и не увидела света.

В 2009 году был официально завершен ES5 (ES5.1 появился в 2011-м), получивший распространение в качестве стандарта для множества современных браузеров, таких как Firefox, Chrome, Opera, Safari и др.

Следующая версия JS (появление которой было перенесено с 2013-го сначала на 2014-й, а затем на 2015 год) в обсуждениях фигурировала под очевидным именем ES6. Но позднее стали поступать предложения перейти к схеме именования, основанной на годе выхода очередной версии, например ES2016 (она же ES7), которая будет закончена до конца 2016 года. Согласны с таким подходом далеко не все, но есть вероятность, что стандарт ES6 станет известен пользователям под названием ES2015. А появление версии ES2016 станет свидетельством окончательного перехода на новую схему именования.

Кроме того, было отмечено, что скорость эволюции JS превышает одну версию в год. Как только в обсуждениях стандарта возникает новая идея, разработчики браузеров предлагают прототипы нового

функционала, а программисты-первопроходцы принимаются экспериментировать с кодом.

Обычно задолго до официального одобрения новый функционал становится стандартом де-факто благодаря ранним прототипам движка и инструментария. Соответственно, имеет смысл рассматривать будущие версии JS как связанные с появлением нового функционала, а не с произвольным набором основных особенностей (как делается сейчас) или с годом (как планируется).

В этом случае номер версии перестает иметь ту важность, которой обладал раньше, а JavaScript превращается в живой, постоянно меняющийся стандарт. И лучше не говорить о коде как о «написанном в соответствии с таким-то стандартом», а рассматривать его в зависимости от поддерживаемых функциональных особенностей.

Транскомпиляция

Быстрая эволюция функционала ставит серьезную проблему перед разработчиками, желающими использовать новые возможности в ситуации, когда их сайты или приложения работают в более старых браузерах, не поддерживающих нововведения.

По всей видимости, ES5 не прижился во многих местах, потому что в основном базу кода не приводили в соответствие с новым стандартом до тех пор, пока не прекратилась поддержка большинства, если не всех, предшествующих платформ. В результате многие разработчики только недавно начали пользоваться такими вещами, как, к примеру, строгий режим, появившийся в ES5 более пяти лет назад.

Подобные многолетние промедления повсеместно считаются вредными для будущего экосистемы JS. Люди, занимающиеся развитием языка, мечтают, чтобы разработчики начинали создавать код с учетом новых функциональных особенностей и шаблонов, сразу

же после того, как будет утверждена спецификация, и браузеры смогут все это реализовывать.

Как же разрешить противоречие? Здесь на помощь приходят специальные инструменты, в частности техника *транскомпиляции*[1]. Грубо говоря, вы посредством специального инструмента преобразуете код ES6 в эквивалент (или нечто близкое к таковому), работающий в окружениях ES5.

В качестве примера возьмем сокращенные определения свойства (см. раздел «Расширения объектных литералов» в главе 2). Вот как это делается в ES6:

```
var    foo = [1,2,3];
var    obj = {
       foo    // означает `foo: foo`
};

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

А вот каким образом (примерно) он транскомпилируется:

```
var    foo = [1,2,3];

var    obj = {
       foo: foo
};

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

Такое небольшое, но удобное преобразование позволяет в случае одинаковых имен сократить объявление объектного литерала `foo: foo` до `foo`. Действия транскомпилятора в этом случае представляют собой встроенный рабочий процесс, аналогичный линтингу, минификации и другим подобным операциям.

[1] Transpiling — термин образован от transformation (преобразование) и compiling (компиляция) (*англ.*). — *Примеч. пер.*

Библиотеки Shim (полизаполнения)

Далеко не всем новым функциональным особенностям ES6 требуется транскомпилятор. *Полизаполнения* (polyfills), которые также называют библиотеками Shim, представляют собой шаблоны для определения поведений из новой среды для более старых сред. В синтаксисе полизаполнения недопустимы, но для различных API их вполне можно использовать.

Давайте рассмотрим новый метод `Object.is(..)`, предназначенный для проверки строгой эквивалентности двух значений, но без подробных исключений, которые есть у оператора `===` для значений `NaN` и `-0`. Полизаполнение для метода `Object.is(..)` создается очень просто:

```
if (!Object.is) {
    Object.is = function(v1, v2) {
        // проверка для значения `-0`
        if (v1 === 0 && v2 === 0) {
            return 1 / v1 === 1 / v2;
        }
        // проверка для значения `NaN`
        if (v1 !== v1) {
            return v2 !== v2;
        }
        // все остальное
        return v1 === v2;
    };
}
```

Обратите внимание на внешнее граничное условие оператора `if`, охватывающее полизаполнение. Это важная деталь, означающая, что резервное поведение данный фрагмент кода включает только в более старых контекстах, где рассматриваемый API еще не определен; необходимости переписывать существующий API практически никогда не возникает.

Есть замечательная коллекция ES6 Shim (https://github.com/paulmillr/es6-shim/), которую стоит включать во все новые JS-проекты.

Предполагается, что JS ждет непрерывное развитие и что поддержка в браузерах новых функций будет реализовываться постепенно по мере их появления, а не большими фрагментами. Так что самая лучшая стратегия сохранения актуальности — это добавление в базу кода полизаполнений, включение транскомпиляции в процесс сборки и постоянная готовность самого разработчика к изменениям.

Те же, кто мыслит консервативно и откладывает использование нового функционала, пока не исчезнут все работающие без него браузеры, всегда будут плестись далеко позади. Их обойдут стороной все инновации, позволяющие сделать написание кода на JavaScript более результативным, рациональным и надежным.

Подводим итоги

На момент написания книги стандарт ES6 (кто-то называет его ES2015) только появился, поэтому вам предстоит многому научиться.

Однако куда важнее перестроить свое мировоззрение в соответствии с новым вариантом развития языка JavaScript. Обыкновение годами ждать официальных документов, одобряющих смену стандарта, должно остаться в прошлом.

Теперь новые функциональные особенности JavaScript сразу же после своего появления реализуются в браузерах, и только от вас зависит, начнете вы пользоваться ими немедленно или же продолжите действовать неэффективно в попытках запрыгнуть в уходящий поезд.

Неважно, какие еще формы примет JavaScript, — теперь это будет происходить быстрее, чем когда-либо в прошлом. Транскомпиляторы и полизаполнения — вот инструменты, которые позволят вам все время оставаться на переднем крае развития языка.

Вы должны принять новую реальность JavaScript, где разработчикам настоятельно рекомендуется перейти от выжидания к активной позиции. А начнется все с изучения ES6.

2 Синтаксис

Если у вас есть хоть какой-то опыт написания программ на JS, скорее всего, вы достаточно хорошо знакомы с его синтаксисом. У него немало своих особенностей, но несмотря на это он достаточно рационален и несложен, а кроме того, имеет множество аналогий в других языках.

Стандарт ES6 добавляет ряд новых синтаксических форм, к которым вам нужно будет привыкнуть. О них я расскажу в этой главе, чтобы дать представление о том, с чем вам предстоит работать.

На момент написания этой книги некоторые составляющие нового функционала уже были полноценно реализованы в браузерах (Firefox, Chrome и др.), другие — реализованы лишь частично, а какие-то — вообще пока недоступны нигде. Соответственно, если вы будете использовать приведенные в книге примеры как есть, результаты могут оказаться непредсказуемыми, так что лучше вам прибегнуть к транскомпиляторам, умеющим работать с большей частью новых функциональных особенностей.

Например, существует замечательная, легкая в использовании «песочница» ES6Fiddle (http://www.es6fiddle.net/) для запуска ES6-кода прямо в браузере, а также онлайновая REPL-среда для транскомпилятора Babel (http://babeljs.io/repl/).

Объявления на уровне блоков кода

Вы, наверное, знаете, что областью видимости переменной в JavaScript в основном всегда была функция. Предпочтительным способом задания видимости переменной в определенном блоке кода, кроме обычного объявления функции, является немедленно вызываемая функция (IIFE). Например:

```
var a = 2;

(function IIFE(){
    var a = 3;
    console.log( a );    // 3
})();

console.log( a );        // 2
```

Оператор let

Впрочем, теперь у нас есть возможность создать объявление, связанное с произвольным блоком, которое вполне логично называется *блочной областью видимости* (block scoping). Для этого достаточно будет пары фигурных скобок `{ .. }`. Вместо оператора `var`, объявляющего область видимости переменных внутри функции, в которую он вложен (или глобальную область видимости, если он находится на верхнем уровне), мы воспользуемся оператором `let`:

```
var a = 2;

{
    let a = 3;
    console.log( a );    // 3
}

console.log( a );        // 2
```

Использование отдельных блоков `{ .. }` — не очень распространенная практика в JS, но это всегда работает. Если вы имели дело

с языками, в которых возможна область видимости на уровне блоков, вы легко распознаете данный шаблон.

На мой взгляд, это самый лучший способ создания переменных с блочной областью видимости. Оператор `let` всегда должен находиться в верхней части блока, причем желательно, чтобы он был один, вне зависимости от количества объявляемых переменных.

Если говорить о стилистике, то я считаю, что правильнее помещать оператор `let` на одну строчку с открывающейся фигурной скобкой `{`, как бы подчеркивая, что единственное назначение блока — это определение области видимости переменных.

```
{ let a = 2, b, c;
    // ..
}
```

Я понимаю, что написанное мной выглядит странно и, скорее всего, противоречит рекомендациям, которые даются в других книгах по ES6. Но у меня есть на это веские причины.

Существует еще одна экспериментальная (не входящая в стандарт) форма объявления при помощи оператора `let`, называемая let-блоком. Она выглядит следующим образом:

```
let (a = 2, b, c) {
    // ..
}
```

Я называю эту форму *явным* объявлением области видимости внутри блока. Имитирующая оператор `var` форма объявления переменной `let ..` более *неявна* — она как бы захватывает все содержимое фигурных скобок `{ .. }`. Как правило, разработчики считают *явные* механизмы более предпочтительными, и мне кажется, что в данном случае имеет смысл придерживаться этого подхода.

Если сравнивать два предыдущих фрагмента кода, в глаза бросается их сходство, и, с моей точки зрения, оба они стилистически

могут рассматриваться как *явное* объявление блочной области видимости. К сожалению, наиболее *явная* форма `let (..) { .. }` в стандарт ES6 не вошла. Существует вероятность, что она появится в последующих версиях ES6, пока же предыдущий пример — это лучшее, чем вы можете воспользоваться.

Подчеркнуть *неявную* природу объявления переменных с помощью оператора `let ..` можно так, как показано ниже:

```
let a = 2;

if (a > 1) {
    let b = a * 3;
    console.log( b );        // 6

    for (let i = a; i <= b; i++) {
        let j = i + 10;
        console.log( j );
    }
    // 12 13 14 15 16

    let c = a + b;
    console.log( c );        // 8
}
```

Попробуйте, не глядя на предыдущие фрагменты кода, ответить, какие переменные существуют только внутри оператора `if`, а какие — только внутри цикла `for`.

Правильный ответ: у `if` это переменные `b` и `c`, а у `for` — `i` и `j`.

Долго ли вы размышляли? И не показался ли вам странным тот факт, что переменная `i` не попала в область видимости оператора `if`? Причина заминки с ответом и возникших вопросов (или, как я обычно говорю, «ментальной нагрузки») появилась потому, что механизм работы оператора `let`, во-первых, непривычный для вас, а во-вторых, *неявный*.

Кстати, помещать объявление `let c = ..` так низко — небезопасно. Если переменные, объявленные с помощью оператора `var`, связываются со всей областью видимости функции вне зависимости

от места их расположения, то использование `let` связывает переменную с областью видимости блока, но инициализация произойдет только после объявления этой переменной. Попытки обращения к ней до инициализации вызовут ошибку, в то время как в случае оператора `var` порядок действий значения не имеет (если не брать во внимание стилистический аспект).

Смотрите:

```
{
    console.log( a );        // значение не определено
    console.log( b );        // ReferenceError!

    var a;
    let b;
}
```

Ошибка `ReferenceError`, генерируемая при попытке слишком рано обратиться к переменной, которая ниже объявляется при помощи оператора `let`, технически называется ошибкой временной мертвой зоны (TDZ — temporal dead zone). Это значит, что вы хотите получить доступ к переменной, которая уже объявлена, но еще не инициализирована. Такая ситуация — не единственная, где появляется ошибка TDZ. Работая с JS стандарта ES6, вы еще не раз с ней столкнетесь. Кроме того, имейте в виду, что «инициализация» возможна и без явного присвоения значения, в частности достаточно выражения `let b;`. Переменная, которой во время объявления ничего не присваивается, по умолчанию получает значение `undefined`. Соответственно, выражение `let b;` эквивалентно выражению `let b = undefined;`. При этом вне зависимости от того, явным или неявным образом было присвоено значение, доступ к переменной `b` появится только после выполнения оператора `let b`.

Есть и еще один подводный камень: с переменными, вызывающими ошибку мертвой зоны, оператор `typeof` ведет себя не так, как с необъявленными (или объявленными) переменными. Например:

```
{
    // переменная `a` не объявлена
    if (typeof a === "undefined") {
        console.log( "cool" );
    }
    // переменная `b` объявлена, но находится в мертвой зоне
    if (typeof b === "undefined") {        // ReferenceError!
        // ..
    }

    // ..

    let b;
}
```

Переменная `a` не была объявлена, так что единственный безопасный способ проверить, существует она или нет, — использовать оператор `typeof`. При этом запись `typeof b` приводит к появлению ошибки TDZ, так как строчка `let b`, объявляющая переменную, располагается ниже.

Думаю, теперь вы понимаете, почему я настоятельно рекомендую выполнять все объявления с помощью оператора `let` в самой верхней строчке блока. Только так можно избежать преждевременного обращения к переменным. Кроме того, при таком подходе сразу видно, какие переменные содержит тот или иной блок.

При этом поведение самого блока (операторов `if`, циклов `while` и т. п.) вовсе не обязательно должно совпадать с поведением области видимости.

Такая ясность, для достижения которой достаточно следовать кое-каким принципам, избавит вас от головной боли на стадии рефакторинга и от неожиданных проблем в будущем.

Более подробно оператор `let` и блочная область видимости рассматриваются в главе 3 книги *Scope & Closures*.

Оператор let и цикл for

Единственный случай, когда я советую отказаться от *явной* формы объявления блочной области видимости с помощью оператора `let`, — это его вставка в заголовок цикла `for`. Данный нюанс кому-то может показаться незначительным, но я считаю, что здесь имеет место важная функциональная особенность ES6.

Рассмотрим пример:

```
var funcs = [];

for (let i = 0; i < 5; i++) {
    funcs.push( function(){
        console.log( i );
    } );
}

funcs[3]();        // 3
```

Оператор `let i` в заголовке цикла `for` объявляет переменную `i` на каждой его итерации. Это означает, что создаваемые внутри цикла на каждой итерации замыкания переменные ведут себя именно так, как ожидается.

Если в том же самом фрагменте вставить в заголовок цикла `for` выражение `var i`, вы получите значение 5 вместо 3, так как в этом случае замыканию будет подвергаться только переменная `i` во внешней области видимости, а не новая `i` для каждой итерации.

Аналогичный результат дает более развернутый вариант кода:

```
var funcs = [];

for (var i = 0; i < 5; i++) {
    let j = i;
    funcs.push( function(){
        console.log( j );
    } );
}

funcs[3](); // 3
```

Здесь мы принудительно создаем на каждой итерации новую переменную `j`, а затем замыкание действует, как и в предыдущем случае. Мне больше нравится предыдущий подход с формой `for (let .. ) ..`, дающий дополнительные возможности. Кто-то, вероятно, скажет, что он в некотором смысле неявен, но, с моей точки зрения, он достаточно явный и весьма полезный.

Аналогичным образом оператор `let` работает с циклами `for..in` и `for..of` (см. раздел «Цикл for..of» этой главы).

Объявления с оператором const

Существует еще одна возможность объявления с блочной областью видимости. Ее нам дает создающий *константы* оператор `const`.

Что такое константа? Это переменная, которая после присвоения начального значения доступна только для чтения. Например:

```
{
    const a = 2;
    console.log( a );      // 2

    a = 3;                 // TypeError!
}
```

Вы не можете менять значение такой переменной после того, как задали его во время объявления. Объявление при помощи оператора `const` всегда должно сопровождаться инициализацией. Если вам требуется *константа* со значением `undefined`, напишите `const a = undefined`.

В случае констант ограничение накладывается не на их значение, а на операцию присваивания. Другими словами, значение оказывается неизменяемым не потому, что речь идет о константе, а из-за невозможности присваивания другого. Если значение комплексное, например, в случае массива или объекта, оно допускает модификацию: