Вы не знаете JS: Приступим!

Содержание

Вы не знаете JS: Начните и Совершенствуйтесь

Предисловие

Что нового вы изучили в последнее время?

Возможно, это был иностранный язык, например итальянский или немецкий. Или, может быть, это был графический редактор, например Photoshop. Это был кулинарный рецепт или новый способ обработки дерева, или спортивное упражнение. Я хочу чтобы Вы запомнили то чувство, когда Вы, наконец, получаете желаемое: тот момент, когда лампочка начинает гореть. Когда все, что было размыто, стало вдруг кристально ясным, когда Вы освоили настольную пилу, или поняли разницу между существительными мужского и женского рода во французском. Как ощущения? Довольно поразительно, правда?

Теперь я хочу переместить Вас немного дальше в Вашу память, прямо к моменту перед изучением нового навыка. Как Вам *это* чувство? Вероятно, немного пугающее, и может быть, немного разочаровывающее, правда? В какой то момент все мы не знали тех вещей, которые знаем сейчас, и это совершенно нормально; мы все с чего-то начинали. Изучение нового материала, это захватывающее приключение, особенно если Вы собираетесь серьезно изучить новую тему.

Я веду много занятий для начинающих программистов. Студенты, посещающие мои занятия, часто пытались самостоятельно обучаться таким вещам, как HTML или JavaScript, просто читая записи в блогах, или копируя и вставляя код, но они не смогли по-настоящему хорошо освоить материал, чтобы получить код, соответствующий их ожиданиям. И, так как они не усвоили всех тонкостей определенных тем программирования на должном уровне, они не могут написать производительный код, или отладить его, так как в действительности, они не понимают что происходит.

Я всегда верю, что процесс обучения на моих занятиях построен правильно, подразумевая, что я обучаю веб-стандартам, семантической верстке, хорошо документированному коду, и другим лучшим практикам. Я освещаю предмет со всех сторон, чтобы объяснить что и как функционирует, вместо того чтобы просто скопировать и вставить код. Когда вы стремитесь понять Ваш код, Ваши результаты улучшаются, и Вы становитесь лучше в своем деле. Код - это теперь не просто Ваша *работа*, это Ваше *ремесло*. Поэтому я люблю *Начинать и Совершенствоваться*. Кайл берет нас с собой в глубокое погружение сквозь синтаксис и терминологию чтобы дать отличное введение в язык JavaScript, не срезая углов. Эта книга не скользит по поверхности, а наоборот позволяет нам по-настоящему понять описанные концепции языка.

Потому что недостаточно уметь копировать фрагменты кода jQuery на Ваш сайт, точно также, как недостаточно научиться открывать, закрывать, и сохранять документ в Photoshop. Конечно, как только я узнала основы работы с программой, я могу создавать дизайн и делиться им. Но без настоящего знания инструментов и того, что скрывается за ними, как я могу нарисовать сетку, или разработать четкую систему типов, или оптимизировать графику для веба? То же самое относится и к JavaScript. Без понимания работы циклов, умения объявлять переменные, или понимания области видимости, мы не будем писать код лучше, чем пишем его сейчас. Мы не хотим соглашаться на что-то меньшее — это, все-таки, наше ремесло.

Чем больше JavaScript влияет на Вас, тем яснее он для Вас становится. Такие слова как замыкания, объекты, и методы, сейчас могут казаться непонятными, но эта книга поможет этим терминам стать яснее. Я хочу чтобы перед чтением этой книги Вы сохранили эти два чувства в своей памяти: чувство до, и чувство после того, как что-нибудь изучите. Это может показаться сложным, но Вы ведь взяли эту книгу чтобы начать удивительное путешествие для оттачивания своих знаний. *Начните и Совершенствуйтесь* Ваша отправная точка в понимании программирования. Наслаждайтесь свечением лампы!

Дженн Лукас
jennlukas.com, @jennlukas
Front-end консультант

Вы не знаете JS: Приступим!

Глава 1: Введение в программирование

Добро пожаловать в серию книг *Вы не знаете JS* (*You don't know JS - YDKJS*).

Книга *Приступим!* — введение в некоторые базовые концепции программирования, конечно, с намеренным уклоном в JavaScript (часто сокращаемый до JS) и еще она о том как подступиться и понять оставшиеся книги серии. Эта книга кратко проанализирует что именно вам нужно, чтобы *приступить*, особенно если вы только начинаете программировать, в том числе на JavaScript.

Эта книга начинается с объяснения базовых принципов программирования на самом высоком уровне. Она в основном предназначена для тех из вас, кто начинает *YDKJS* имея малый или не имея вовсе опыта в программировании и кто рассчитывает, что эти книги помогут встать на путь понимания программирования сквозь призму JavaScript.

Главу 1 можно представить как быстрый обзор того, что вы захотели бы изучить и возможность попрактиковаться *в программировании*. Кроме того, есть много других фантастических ресурсов по основам программирования, которые могут помочь вам изучить во всех подробностях затрагиваемые темы и я призываю вас изучить их в дополнение к этой главе.

Если вы хорошо знаете общие основы программирования, глава 2 подтолкнет вас к более близкому знакомству с духом программирования на JavaScript. Глава 2 знакомит с тем, что такое JavaScript, но, обращаю ваше внимание еще раз, это не подробное руководство — это то, за что выступают остальные книги *YDKJS*!

Если вы уже довольно комфортно чувствуете себя в JavaScript, сначала ознакомьтесь с главой 3 для беглого знакомства с тем, чего ожидать от *YDKJS*, а затем приступим!

Код

Начнем с начала.

Программа, часто упоминаемая как *исходный код* или просто *код* — это набор особых инструкций, сообщающих компьютеру какие задачи нужно сделать. Обычно код сохраняют в текстовый файл, хотя в случае JavaScript можно писать код прямо в консоли разработчика в браузере, чего мы кратко коснемся далее.

Правила допустимого формата и комбинаций операторов называются *язык программирования*, иногда их соотносят с его *синтаксисом*, аналогично английскому языку, где правила говорят вам как произносить слова и как составлять правильные предложения используя слова и знаки препинания.

Операторы

В языке программирования группа слов, чисел и операций, которые выполняют определенную задачу, называются *оператором*. В JavaScript, оператор может выглядеть так:

```
a = b * 2;
```

Символы `a` и `b` называются *переменными* (см. "Переменные"), которые примерно как обычные коробки, в которых вы можете хранить что угодно. В программах переменные хранят значения (например, число `42`), используемые программой. Представляйте их как символьную подмену для самих значений.
В противоположность им, `2` — это само значение, называемое *литеральным значением*, поскольку оно само по себе и не хранится в переменной.
Символы `=` и `*` — это *операции* (см. "Операции"), они выполняют действия со значениями и переменными, такие как присваивание и математическое умножение.
Большинство операторов в JavaScript заканчиваются точкой с запятой (`;`).
Оператор `a = b * 2;` сообщает компьютеру, грубо говоря, взять текущее значение из переменной `b`, умножить это значение на `2`, а затем сохранить результат в другую переменную, которую мы назвали `a`.
Программы — это всего лишь набор стольких операторов, сколько нужно для того, чтобы описать все шаги, необходимые для выполнения цели вашей программы.

Выражения

Операторы состоят из одного или более *выражений*. Выражение — это любая ссылка на переменную или значение или набор переменных и значений, объединенных операциями.

Например:

```
a = b * 2;
```

У этого оператора 4 выражения:

- `2` — это *выражение литерального значения*
- `b` — это *выражение переменной*, которое тут означает извлечение его текущего значения
- `b * 2` — это *арифметическое выражение*, в данном случае выполнение умножения
- `a = b * 2` — это *выражение присваивания*, в данном случае это присвоить результат выражения `b * 2`переменной `a` (подробнее о выражениях далее)

Типичное выражение, которое является законченным, называется *оператор-выражение*, например, такое как это:

```
b * 2;
```

Этот пример оператора-выражения не является типовым или полезным, и в целом не оказывает никакого эффекта на выполнение программы — он всего лишь извлекает значение `b` и умножает его на `2`, но затем ничего не делает с результатом.
Более распространенный оператор-выражение — это *оператор-выражение вызова* (см. "Функции"), поскольку весь оператор — это выражение вызова функции:

```
alert( a );
```

Выполнение программы

Так как же эти наборы программных операторов сообщают компьютеру что нужно делать? Программу нужно *выполнить*, также говорят *запуск программы*.

Операторы, подобные `a = b * 2`, понятны для разработчиков как при чтении, так и записи, но фактически в такой форме они не понятны напрямую компьютеру. Поэтому используется специальная утилита в компьютере (либо *интерпретатор*, либо *компилятор*) для перевода кода, который вы пишете, в команды, понятные компьютеру.
В некоторых языках программирования перевод команд обычно выполняется сверху вниз, строка за строкой, каждый раз когда программа запускается, что обычно называется *интерпретацией* кода.

В других языках, перевод, выполняемый заранее, называется *компиляцией* кода, поэтому когда позднее программа *запускается*, то, что запускается — это по факту уже скомпилированные инструкции компьютера, готовые к выполнению.

Обычно утверждают, что JavaScript — *интерпретируемый*, так как ваш исходный код на JavaScript обрабатывается каждый раз, когда запускается. Но это не совсем точно. Движок JavaScript на самом деле *компилирует* программу на лету и затем сразу же запускает скомпилированный код.

Примечание: Подробнее о компиляции JavaScript смотрите в первых двух главах книги *Область видимости и замыкания* этой серии.

Попробуйте сами

Эта глава проиллюстрирует каждое понятие из программирования простыми примерами кода, полностью написанными на JavaScript (очевидно!).

Нельзя не отметить, что пока вы продвигаетесь по этой главе вам может понадобиться перечитать ее несколько раз и вам следует практиковаться в каждом из понятий набирая код самостоятельно. Простейший путь сделать это — это открыть консоль в средствах разработки в ближайшем браузере (Firefox, Chrome, IE и т.п.).

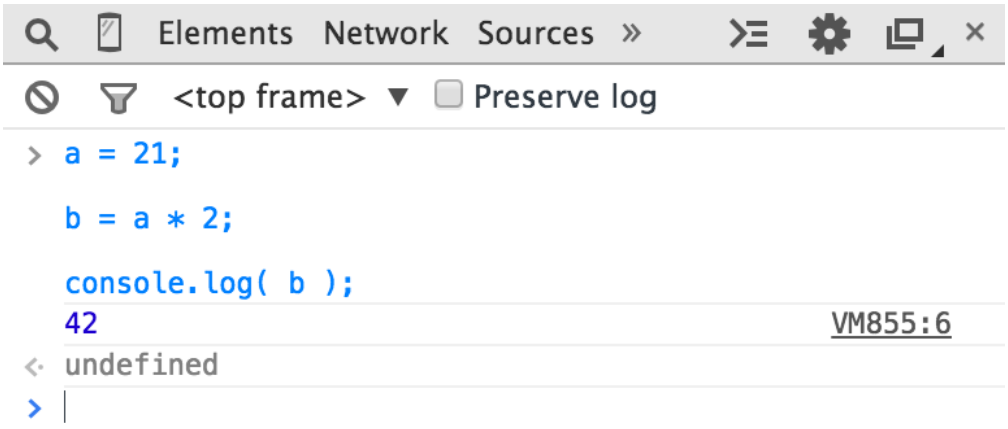
Подсказка: Обычно вы можете запустить консоль разработчика с помощью горячих клавиш или из меню. Подробнее о запуске и использовании консоли в вашем любимом браузере см. "Mastering The Developer Tools Console" (http://blog.teamtreehouse.com/mastering-developer-tools-console). Чтобы ввести несколько строк в консоли за раз, используйте `<shift> + <enter>`, чтобы переместиться на новую строку. Как только вы просто нажмете `<enter>`, консоль выполнит всё, что вы написали.
Давайте познакомимся с процессом запуска кода в консоли. Сперва я предлагаю открыть пустую вкладку в браузере. Я предпочитаю делать это набирая `about:blank` в адресной строке. Затем убедитесь, что ваша консоль разработчика, о которой мы только что упоминали, открылась.
Теперь наберите этот код и посмотрите как он выполняется:

```
a = 21;

b = a * 2;

console.log( b );
```

Набрав в консоли вышеуказанный код в браузере Chrome мы увидим что-то вроде этого:



Вперед, попробуйте! Наилучший путь обучения программированию — это начать писать код!

Вывод

В предыдущем кусочке кода мы использовали`console.log(..)`. Давайте взглянем вкратце о чем же эта строка кода.

Возможно вы это предполагали, но это и в самом деле то, как мы печатаем текст (т.е. *вывод* для пользователя) в консоли разработчика. Есть две характеристики этого оператора, которые нам следует пояснить.

Первая часть, `log( b )` указывает на вызов функции (см. "Функции"). Здесь получается, что мы передаем переменную `b` в эту функцию, которая берет значение `b` и печатает его в консоли.
Вторая часть, `console.` — это ссылка на объект, где расположена функция `log(..)`. Мы рассмотрим объекты и их свойства более детально в главе 2.
Еще один путь вывести информацию — запустить оператор `alert(..)`. Например:

```
alert( b );
```

Если вы запустите этот оператор, то заметите, что вместо вывода значения в консоль он показывает всплывающее окно с кнопкой "OK" и содержимым переменной `b`. Однако использование `console.log(..)` обычно лучше помогает кодировать и запускать программы в консоли, чем использование `alert(..)`, потому что вы можете вывести несколько значений за раз без остановки в интерфейсе браузера.
В этой книге мы будем использовать для вывода `console.log(..)`.

Ввод

Пока мы обсуждаем вывод, вы попутно могли задаться вопросом о *ввода* (т.е. о получении информации от пользователя).

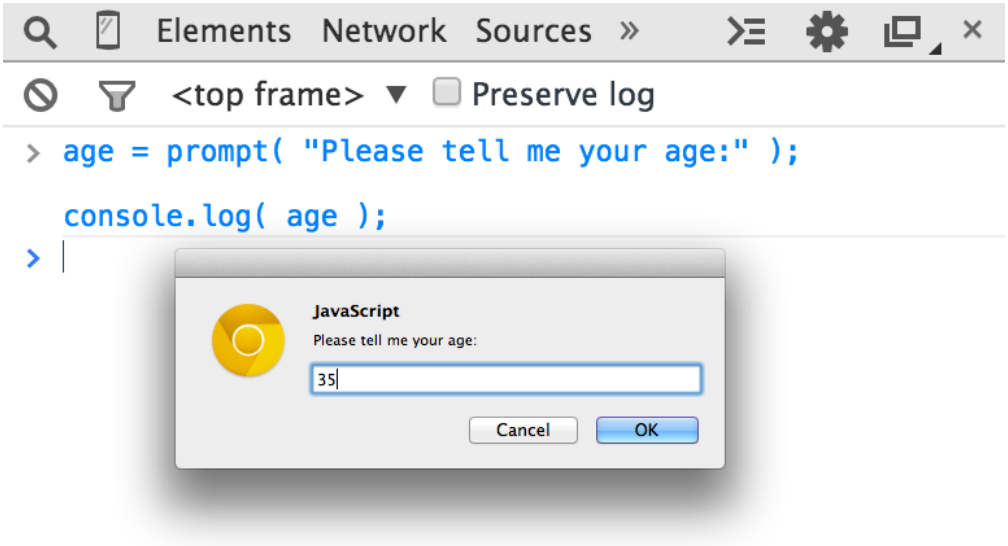
Самый распространенный путь — показать элементы формы на HTML-странице (например, строки ввода) для пользователя, чтобы он мог вводить туда данные, а затем, используя JS, считать эти значения в переменные программы.

Но есть более простой путь получать входные данные в целях обучения и демонстрации, который вы будете использовать на протяжении всей этой книги. Используйте функцию `prompt(..)`:

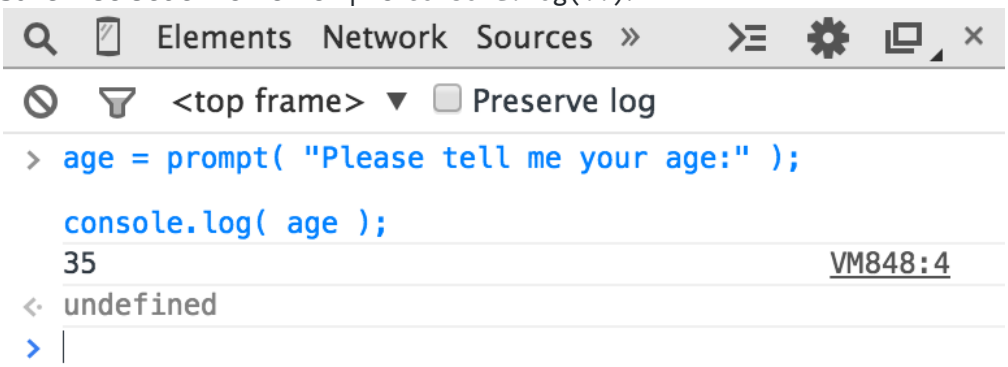
```
age = prompt( "Please tell me your age:" );

console.log( age );
```

Как вы уже могли догадаться, сообщение, которое вы передаете в `prompt(..)`, в данном случае `"Please tell me your age:"` ('"Пожалуйста сообщите мне свой возраст:"'), выводится во всплывающем окне.
Это может выглядеть примерно так:



Как только вы подтвердите ввод текста щелкнув по "OK," вы заметите, что введенное значение теперь хранится в переменной `age`, которую мы затем *выведем* с помощью `console.log(..)`:



Для упрощения, пока мы изучаем основные понятия программирования, примеры в этой книге не потребуют ввода. Зато теперь вы увидели как пользоваться `prompt(..)`. Если вы хотите проверить себя, то можете попробовать использовать ввод в порядке экспериментов с примерами.

Операции

Операции — это те действия, которые мы выполняем над переменными и значениями. Мы уже видели две операции JavaScript, `=` и `*`.
Операция `*` выполняет математическое умножение. Достаточно просто, не так ли?
Операция `=` используется для *присваивания* — сначала мы вычисляем значение с *правой стороны* (исходное значение) от `=`, а затем записываем его в переменную, которую мы указываем с *левой стороны* (переменная назначения).
Предупреждение: Такой обратный порядок для присваивания может выглядеть немного странно. Вместо `a = 42` кто-то может предпочесть поменять порядок, чтобы исходное значение было слева, а переменная назначения — справа, например `42 -> a` (это — неправильный JavaScript!). К сожалению, форма `a = 42` и похожие на нее практически полностью превалируют в современных языках

программирования. Если вам такой порядок присваивания кажется неестественным, потратьте некоторое время на привыкание к нему.
Пример:

```
a = 2;
b = a + 1;
```

Тут мы присваиваем значение `2` переменной `a`. Затем мы получаем значение переменной `a` (пока еще `2`), прибавляем к нему `1` получая в результате `3`, потом сохраняем это значение в переменной `b`.
Хоть и не являющееся технически операцией, вам необходимо ключевое слово `var` в любой программе, поскольку это основной способ, с помощью которого вы *объявляете* (т.е. *создаете*) переменные (сокращение от *var*iables) (см. "Переменные").
Вы всегда должны объявить переменную с именем до того, как начнете ее использовать. Но вам достаточно объявить переменную всего раз для каждой *области видимости* (см. "Область видимости"), а затем пользоваться ею столько раз, сколько нужно. Например:

```
var a = 20;

a = a + 1;
a = a * 2;

console.log( a );          // 42
```

Вот несколько самых базовых операций в JavaScript:

- Присваивание: `=` как в `a = 2`.
- Математические: `+` (сложение), `-` (вычитание), `*` (умножение) и `/` (деление), как в `a * 3`.
- Составное присваивание: `+=`, `-=`, `*=`, and `/=` — это составные операции, которые объединяют математическую операцию с присваиванием, как в `a += 2` (эквивалентно `a = a + 2`).
- Инкремент/Декремент: `++` (инкремент), `--` (декремент), как в `a++` (эквивалентно `a = a + 1`).
- Доступ к свойству объекта: `.` как в `console.log()`.
  Объекты — это значения, которые хранят другие значения под своими именами, называемые свойства. `obj.a`означает значение из объекта `obj` из его свойства `a`. Еще один способ доступа к свойствам — `obj["a"]`. См. главу 2.
- Равенство: `==` (нестрогое), `===` (строгое), `!=` (нестрогое неравенство), `!==` (строгое неравенство), как в `a == b`.

  См. "Значения и типы" и главу 2.

- Сравнение: `<` (меньше чем), `>` (больше чем), `<=` (меньше или нестрого равно), `>=` (больше или нестрого равно), как в `a <= b`.

  См. "Значения и типы" и главу 2.

- Логические: `&&` (и), `||` (или), как в `a || b`, которое выбирает или `a`, *или* (*or*) `b`.

Эти операции используются для создания составных условных конструкций (см. "Условные конструкции"), например: если либо `a` *либо (or)* `b` — истина.

Примечание: Для более детального рассмотрения и охвата операций, не рассмотренных здесь, см. the Mozilla Developer Network (MDN)'s "Expressions and Operators" (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators).

Значения и типы

Если вы спросите сотрудника в магазине сотовой связи сколько стоит определенный телефон и он ответит "девяносто девять, девяносто девять" (т.е., 99.99), таким образом он дает вам точную информацию о сумме денег, которую вам необходимо заплатить, чтобы купить его. Если вы хотите купить два таких телефона, вы легко сможете в уме удвоить стоимость, получив 199.98 в качестве общей стоимости.

Если тот же сотрудник возьмет другой аналогичный телефон и скажет, что он "бесплатный" (конечно, в кавычках), он не скажет вам сумму, но взамен предоставит другую форму представления ожидаемой стоимости (0.00) — слово "бесплатный".

Если затем вы спросите включено ли в комплект поставки телефона зарядное устройство, то ответ может быть только "да" или "нет".

Весьма схожим образом, когда вы указываете значения в программе, вы выбираете разные представления для этих значений в зависимости от того, что вы планируете делать с ними.

Эти разнообразные представления для значений называются *типы* в терминологии программирования. В JavaScript есть встроенные типы для каждого из этих так называемых *примитивных* значений:

- когда вам нужно работать с математикой, вам нужно `число`.
- когда вам нужно вывести значение на экран, вам нужна `строка` (один или несколько символов, слов, предложений).
- когда вам нужно принять решение в своей программе, вам нужно `логическое значение` (`true` (`истина`) или `false`(`ложь`)).

Значения, непосредственно включаемые в исходный код, называются *литералы*. `строковые` литералы заключаются в двойные кавычки `"..."` или одинарные (`'...'`) — единственная разница в них — это ваши стилистические предпочтения. Литералы `числа` и `логического зачения` пишутся как есть (т.е., `42`, `true` и т.д.).
Пример:

```
"Я - строка";
'Я - тоже строка';
```

```
42;

true;
false;
```

Кроме типов значений строка/число/логическое значение, для языков программирования привычно предоставлять такие типы как *массивы*, *объекты*, *функции* и многое другое. Мы рассмотрим детально значения и типы на протяжении этой и следующей глав.

Преобразование между типами

Если у вас есть число, но вам надо вывести его на экран, вам нужно преобразовать его значение в строку и в JavaScript такая конвертация называется "приведение (coercion)." Аналогично, если кто-то вводит серию цифр в форму на веб-странице, это строка, но если нужно потом использовать это значение для выполнения математических операций, то вам понадобится *приведение* его к числу.
JavaScript предоставляет несколько различных возможностей принудительного приведения между *типами*. Например:

```
var a = "42";
var b = Number( a );

console.log( a );       // "42"
console.log( b );       // 42
```

Использование Number(..) (встроенная функция), как было показано выше, это *явное* приведение из любого другого типа в тип число. Это выглядит достаточно очевидно.
Но каверзный вопрос заключается в том, что произойдет когда вы попытаетесь сравнить два значения разных типов, которые могут потребовать *неявного* приведения.

При сравнении строки "99.99" с числом 99.99 многие согласятся, что они равны. Но они ведь не совсем одно и то же, не так ли? Это одно и то же значение в двух разных представлениях, двух разных *типов*. Вы могли бы сказать, что они "нестрого равны," разве нет?
Чтобы помочь вам в таких стандартных ситуациях, JavaScript иногда вмешивается и *неявно* приводит значения к подходящим типам.

Поэтому если вы используете операцию нестрогого равенства == для сравнения "99.99" == 99.99, JavaScript преобразует с левой стороны "99.99" в его числовой эквивалент 99.99. После этого сравнение превращается в 99.99 == 99.99, которое конечно является истинным.
Несмотря на то, что неявное приведение было задумано, чтобы помочь вам, оно может привести в замешательство, если вы не уделили достаточно времени изучению правил, которые определяют его поведение. У большинства JS-разработчиков никогда его нет, поэтому общее отношение выражается в том, что неявное приведение сбивает с толку и вредит программам внося непредвиденные

ошибки и поэтому следует его избегать. Иногда его даже называют изъяном дизайна языка.

Однако, неявное приведение — это механизм, который *может быть изучен* и даже более того *должен быть изучен* любым, кто хочет серьезно заниматься программированием на JavaScript. Не только потому, что после изучения его правил оно не будет смущать вас, оно может в самом деле улучшить ваши программы! Усилия того стоят!

Примечание: Для получения более подробной информации о приведении см. главу 2 этой книги и главу 4 книги *Типы и синтаксис* этой серии.

Комментарии в коде

Сотрудник салона сотовой связи может набросать некоторые заметки о возможностях только что выпущенных телефонов или о новых тарифных планах, которые предлагает его компания. Эти заметки только для самого сотрудника — они не предназначены для чтения покупателями. Тем ни менее эти заметки помогают сотруднику улучшить свою работу документируя все "как" и "почему" того, что ему следует рассказать покупателям.

Один из самых важных уроков, который вы можете получить о написании кода, то, что код — не только для компьютера. Код для разработчика как каждый бит, если не больше, для компилятора.

Ваш компьютер заботится только о машинном коде, последовательности бинарных 0 и 1, которые появляются после *компиляции*. Есть почти бесконечное количество программ, которые вы могли бы написать и которые состоят из одинаковых последовательностей 0 и 1. Выбор, который вы делаете о том как написать программу, имеет значение не только для вас, но и для других членов вашей команды и даже для самого вашего будущего.

Вы должны стремиться писать программы, которые не только правильно работают, но и понятны при их изучении. Вы можете пройти долгий путь в этом направлении, к примеру выбирая понятные имена для своих переменных (см. "Переменные") и функций (см. "Функции").

Но еще одна важная часть этого процесса — это комментарии в коде. Это кусочки текста в вашей программе, которые вставляются именно для того, чтобы пояснить какие-то вещи для человека. Интерпретатор/компилятор всегда игнорирует эти комментарии.

Есть масса мнений о том, что делает код хорошо документируемым, мы не можем на самом деле определить абсолютные и универсальные правила. Но некоторые соображения и рекомендации будут весьма полезны:

- Код без комментариев не оптимален.
- Слишком много комментариев (по одному на каждую строку кода, например) возможно являются признаком плохонаписанного кода.
- Комментарии должны объяснять *почему*, а не *что*. Они могут дополнительно объяснять *как*, когда код особенно сложен.

В JavaScript есть два типа комментариев: однострочный комментарий и многострочный комментарий.

Пример:

```
// Это - однострочный комментарий

/* А это
       многострочный
             комментарий.
                    */
```

Однострочный комментарий `//` подходит если вы собираетесь разместить комментарий прямо над одиночным оператором или даже в конце строки. Всё что написано в строке после `//` интерпретируется как комментарий (и потому игнорируется компилятором) до самого конца строки. Нет никаких ограничений на то, что должно быть внутри однострочного комментария.
Пример:

```
var a = 42;             // 42 - смысл жизни
```

Многострочный комментарий `/* .. */` подходит в случае, если у вас есть несколько строк пояснений для вашего кода.
Вот типичный пример использования многострочного комментария:

```
/* Нижеприведенное значение используется, поскольку
   выяснилось, что оно отвечает
   на любой вопрос во вселенной. */
var a = 42;
```

Он может появляться в любом месте строки, даже в середине строки, поскольку есть `*/`, обозначающий его окончание. Например:

```
var a = /* произвольное значение */ 42;

console.log( a );       // 42
```

Единственное, что не может появляться в многострочном комментарии — это `*/`, так как это будет означать конец комментария.
Вы несомненно захотите начать обучение программированию начав с привычки комментировать код. На всем протяжении оставшейся части этой главы вы увидите, что я использую комментарии, чтобы пояснить код, поэтому и вы делайте также в вашей собственной практике написания кода. Поверьте, все, кто будет читать ваш код, скажут вам спасибо!

Переменные

Большинству программ нужно отслеживать то, как меняется значение на протяжении выполнения программы, проходя через различные операции, вызываемые для соответствующих задач вашей программы.

Самый простой путь сделать это в программе — это присвоить значение символьному контейнеру, называемому *переменной*, называющейся так потому, что значение в этом контейнере может *меняться* с течением времени при необходимости.

В некоторых языках программирования вы определяете переменную (контейнер), чтобы хранить определенный тип значения, такой как `число` или `строка`. *Статическая типизация*, также известная как *контроль типов*, обычно упоминается как преимущество в корректности программы, предотвращая непредусмотренные преобразования значений.
Другие языки выводят типы для значений вместо переменных. *Слабая типизация*, также известная как *динамическая типизация*, позволяет переменной хранить значения любого типа в любое время. Это обычно упоминается как преимущество в гибкости программы позволяя одной переменной представлять значение вне зависимости от того, в форме какого типа это значение может понадобиться в любой момент выполнения программы.

JavaScript использует второй подход, *динамическую типизацию*, что означает, что переменные могут хранить значения любого *типа* без какого-либо контроля *типов*.

Как уже упоминалось ранее, мы объявляем переменную используя оператор `var`, заметьте, что при этом нет больше никакой другой информации о *типе* в объявлении. Обратите внимание на эту простую программу:

```
var amount = 99.99;

amount = amount * 2;

console.log( amount );            // 199.98

// преобразует `amount` в строку и
// добавляет "$" в начало
amount = "$" + String( amount );

console.log( amount );            // "$199.98"
```

Переменная `amount` начинает свой жизненный цикл с хранения числа `99.99`, а затем хранит `числовой` результат `amount * 2`, который равен `199.98`.
Первая команда `console.log(..)` должна *неявно* привести это `числовое` значение к `строке`, чтобы вывести его в консоль.
Затем оператор `amount = "$" + String(amount)` *явно* приводит значение `199.98` к `строке` и добавляет символ `"$"` в начало. С этого момента, `amount` хранит `строковое` значение `"$199.98"`, поэтому второму

оператору `console.log(..)`не нужно выполнять никакого приведения, чтобы вывести его в консоль.
Разработчики на JavaScript отметят гибкость использования переменной `amount` для каждого из значений `99.99`, `199.98` и `"$199.98"`. Энтузиасты статической типизации предпочтут отдельную переменную, например `amountStr`, чтобы хранить окончательное представление значения `"$199.98"`, поскольку оно уже будет другого типа.
В любом случае, вы заметите, что `amount` хранит текущее значение, которое меняется по ходу выполнения программы, иллюстрируя первичную цель переменных: управление *состоянием* программы.
Другими словами, *состояние* отслеживает изменения значений при выполнении программы.

Еще одно общеупотребительное использование переменных — для централизации установки значений. Обычно это называется *константами*, когда вы объявляете переменную со значением и предполагаете, что это значение не будет меняться в течение работы программы.

Вы объявляете эти *константы*, чаще всего в начале программы, таким образом, чтобы для вас было удобно иметь всего одно место для того, чтобы поменять значение, если нужно. По соглашению, переменные в JavaScript, являющиеся константами, обычно пишутся большими буквами, с подчеркиваниями _ между словами.
Вот глупый пример:

```
var TAX_RATE = 0.08;     // 8% налог с продаж

var amount = 99.99;

amount = amount * 2;

amount = amount + (amount * TAX_RATE);

console.log( amount );                          // 215.9784
console.log( amount.toFixed( 2 ) );       // "215.98"
```

Примечание: Также как `console.log(..)` — это функция `log(..)`, доступная как свойство объекта `console`, `toFixed(..)` здесь — это функция, которая может быть доступна у `числовых` значений. `Число` в JavaScript не форматируется автоматически со знаком валюты — среда выполнения не знает ваших намерений плюс к этому не существует типа для валюты. `toFixed(..)` позволяет нам указать до скольки знаков после запятой мы хотим округлить `число` и она возвращает `строку` при необходимости.
Переменная `TAX_RATE` — всего лишь *константа* по соглашению, в этой программе нет ничего, что могло бы предотвратить ее изменение. Но если ставка налога повысится до 9%, мы все еще можем легко обновить нашу программу установив присвоенное `TAX_RATE` значение в `0.09` всего в одном месте вместо поиска всех вхождений значения `0.08` разбросанных по программе и изменения их всех.

Новейшая версия JavaScript на момент написания этих строк (обычно называемая "ES6") включает в себя новый способ объявления *констант*, используя `const` вместо `var`:

```
// согласно ES6:
const TAX_RATE = 0.08;

var amount = 99.99;

// ..
```

Константы полезны также как и переменные с неизменяемыми значениями, за исключением того, что константы также предотвращают случайное изменение где-либо после начальной установки значения. Если вы попытаетесь присвоить любое значение в `TAX_RATE` после ее объявления, ваша программа отвергнет это изменение (а в строгом (strict) режиме, прервется с ошибкой, см. "Строгий режим" в главе 2).

Кстати, такой тип "защиты" против ошибок похож на контроль типов статической типизации, так что вы в какой-то степени поймете почему статические типы в других языках могут быть привлекательными!

Примечание: Для получения более подробной информации о том, как различные значения в переменных могут использоваться в программах, см. книгу *Типы и синтаксис* этой серии.

Блоки

Сотрудник салона сотовой связи должен пройти последовательность шагов для завершения оформления покупки если вы покупаете новый телефон.

Примерно также в коде нам часто нужно группировать последовательности операторов вместе, которые мы часто называем *блоком*. В JavaScript блок определяется обрамлением одного или более операторов парой фигурных скобок `{ .. }`. Пример:

```
var amount = 99.99;

// отдельный блок
{
        amount = amount * 2;
        console.log( amount );   // 199.98
}
```

Такой вид отдельного блока `{ .. }` вполне допустим, но не часто встречается в JS-программах. Обычно блоки присоединяются к другим управляющим операторам, таким как оператор `if` (см. "Условные конструкции") или цикл (см. "Циклы"). Например:

```
var amount = 99.99;

// сумма достаточно велика?
if (amount > 10) {                      // <-- блок прикрепляется к `if`
        amount = amount * 2;
        console.log( amount );   // 199.98
}
```

Мы расскажем об операторе `if` в следующем разделе, но как вы видите блок `{ .. }` с двумя операторами присоединен к `if (amount > 10)`. Операторы внутри этого блока будут выполнены только при выполнении условия в условной конструкции.
Примечание: В отличие от многих других операторов, таких как `console.log(amount);`, блоковый оператор не требует точки с запятой (`;`) в конце оператора.

Условные конструкции

"Хотите ли вы добавить дополнительную защитную пленку в вашу покупку за $9.99?" Предупредительный сотрудник магазина попросил вас принять решение. И вам может сначала понадобиться проинспектировать текущее *состояние* вашего кошелька или банковского счета, чтобы ответить на этот вопрос. Но, очевидно, что это всего лишь простой вопрос из разряда "да или нет".

Есть довольно много способов, которыми мы можем выразить *условные конструкции* (т.е. решения) в наших программах.

Самый распространенный из них — это оператор `if`. По сути, вы говорите, "*Если* (*if*) это условие истинно, сделать следующее...". Например:

```
var bank_balance = 302.13;
var amount = 99.99;

if (amount < bank_balance) {
        console.log( "Я хочу купить этот телефон" );
}
```

Оператор `if` требует выражение между скобками `( )`, которое может быть интерпретировано либо как `истина`(`true`), либо `ложь` (`false`). В этой программе мы написали выражение `amount < bank_balance`, которое конечно же будет вычислено как `true` или `false` в зависимости от количества в переменной `bank_balance`.
Вы даже можете предоставить альтернативу если условие не будет истинным, называющуюся оператором `else`. Пример:

```
const ACCESSORY_PRICE = 9.99;

var bank_balance = 302.13;
var amount = 99.99;

amount = amount * 2;

// может ли мы позволить себе дополнительную покупку?
if ( amount < bank_balance ) {
        console.log( "Я возьму этот аксессуар!" );
        amount = amount + ACCESSORY_PRICE;
}
// иначе:
else {
        console.log( "Нет, спасибо." );
}
```

Тут если `amount < bank_balance` истинно, мы выведем `"Я возьму этот аксессуар!"` и добавим `9.99` в нашу переменную `amount`. В противном случае,

оператор `else` говорит, что мы вежливо ответим `"Нет, спасибо."` и оставим переменную `amount` без изменений.
Как мы уже обсуждали ранее в "Значения и типы", значения, которые не совпадают с ожидаемым типом, часто приводятся к этому типу.
Оператор `if` ожидает `логическое значение`, но если вы передадите что-либо отличное от `логического значения`, произойдет приведение.
JavaScript определяет список особых значений, которые считаются "ложными", так как при приведении к `логическому значению` они станут значением `false`, такие значения включают в себя `0` и `""`. Любое другое значение, не входящее в список "ложных", автоматически считается "истинным", когда приводится к `логическому значению`, оно становится равным `true`. Истинные значения включают в себя такие значения как `99.99` и `"free"`. См. "Истинный и ложный" в главе 2 для получения более детальной информации.
Условные конструкции существуют и в других формах, отличных от `if`. Например, оператор `switch` может использоваться как сокращение для последовательности операторов `if..else` (см. главу 2). Циклы (см. "Циклы") используют *условную конструкцию*, чтобы определить надо ли завершить выполнение цикла или нет.
Примечание: Детальную информацию о приведениях, которые происходят неявно в проверочных выражениях *условных конструкций*, см. главу 4 книги *Типы и синтаксис* этой серии.

Циклы

При большой посещаемости магазина есть очередь из покупателей, которым нужно поговорить с сотрудником магазина. Пока в этой очереди есть люди, сотруднику нужно продолжать обслуживать очередного покупателя.

Повторение набора действий пока не нарушится определенное условие, другими словами, повторение только пока соблюдается условие — это как раз работа для циклов. Циклы могут принимать различные формы, но все они удовлетворяют этому базовому поведению.

Цикл включает в себя проверяемое условие и блок (обычно в виде `{ .. }`). Каждый раз, когда выполняется блок в цикле, это называется *итерацией*.
Например, цикл `while` и цикл `do..while` иллюстрируют принцип повторения блока операторов до тех пора пока условие не перестанет быть равным `true`:

```
while (numOfCustomers > 0) {
        console.log( "Чем я могу вам помочь?" );

        // помощь покупателю...

        numOfCustomers = numOfCustomers - 1;
}

// против:

do {
        console.log( "Чем я могу вам помочь?" );
```

```
        // помощь покупателю...

        numOfCustomers = numOfCustomers - 1;
} while (numOfCustomers > 0);
```

Единственная разница между этими циклами — это будет ли проверяться условная конструкция до первой итерации (`while`) или после первой итерации (`do..while`).

В любом из этих циклов если условная конструкция возвратит `false`, следующая итерация не будет выполнена. Это значит, что если условие изначально будет `false`, цикл `while` никогда не будет выполнен, а цикл `do..while`выполнится только один раз.

Иногда вы используете цикл для подсчета определенного набора чисел, например от `0` до `9` (десять чисел). Это можно сделать установкой переменной в цикле итерации, например `i`, в значение `0` и увеличивая его на `1` в каждой итерации.

Предупреждение: По множеству исторических причин языки программирования почти всегда ведут подсчет в нолеподобной манере, т.е. начиная с `0` вместо `1`. Если вы не знакомы с таким типом подсчета, поначалу это может сбивать с толку. Уделите некоторое время тому, чтобы попрактиковаться в подсчете, начинающимся с `0`, чтобы освоиться в нем!

Условная конструкция проверяется на каждой итерации, как если бы был неявный оператор `if` внутри цикла.

Для выхода из цикла можно использовать JavaScript-оператор `break`. К тому же, можно обнаружить, что ужасно легко можно создать цикл, который в противном случае будет работать вечно без механизма `break`.

Проиллюстрируем:

```
var i = 0;

// цикл `while..true` будет выполняться вечно, не так ли?
while (true) {
        // прервать цикл?
        if ((i <= 9) === false) {
                break;
        }

        console.log( i );
        i = i + 1;
}
// 0 1 2 3 4 5 6 7 8 9
```

Предупреждение: Показанное выше не является практикой, которой вы должны придерживаться при реализации ваших циклов. Это представлено только в иллюстративных целях.

Если `while` (или `do..while`) может достичь цели вручную, есть еще одна синтаксическая форма, называемая цикл `for`именно для такой вот цели:

```
for (var i = 0; i <= 9; i = i + 1) {
        console.log( i );
}
// 0 1 2 3 4 5 6 7 8 9
```

Как видите, в обоих случаях условная конструкция `i <= 9` равна `true` для первых 10 итераций (`i` принимает значения от `0` до `9`) любой из форм цикла, но становится равным `false` как только `i` становится равным `10`.
У цикла `for` есть три составных части: инициализация (`var i=0`), проверка условия (`i <= 9`) и обновление значения (`i = i + 1`). Поэтому если вы собираетесь заниматься выполнением конкретного количества итераций, `for` будет более компактной и часто более легкой формой цикла для понимания и записи.
Есть другие особые формы циклов, которые предназначены для итерирования по особым значениям, таким как свойства объекта (см. главу 2), где неявная проверка условной конструкции — это все ли свойства или нет уже обработаны. Принцип "цикл работает пока не нарушится условие" соблюдается независимо от формы цикла.

Функции

Сотрудник магазина возможно не носит постоянно с собой калькулятор, чтобы учесть налоги и рассчитать окончательную стоимость покупки. Это задача, которую ему нужно определить один раз и использовать раз за разом. Преимущество в том, что у компании есть контрольно-кассовый аппарат (компьютер, планшет и т.п.), в который эти "функции" уже встроены.

Похожим образом и в вашей программе вам определенно захочется разбить задачи в коде на повторноиспользуемые части, вместо того, чтобы снова и снова однообразно повторять себя. Для реализации этого необходимо определить функцию.
Функция — обычно это именованная секция кода, которая может быть "вызвана" по имени и код внутри нее будет при этом запускаться каждый раз. Пример:

```
function printAmount() {
        console.log( amount.toFixed( 2 ) );
}

var amount = 99.99;

printAmount(); // "99.99"

amount = amount * 2;

printAmount(); // "199.98"
```

У функций могут быть аргументы (т.е. параметры) — это значения которые вы ей передаете. А также функции могут возвращать значение.

```
function printAmount(amt) {
        console.log( amt.toFixed( 2 ) );
}

function formatAmount() {
        return "$" + amount.toFixed( 2 );
}
```

```
var amount = 99.99;

printAmount( amount * 2 );                // "199.98"

amount = formatAmount();
console.log( amount );                    // "$99.99"
```

Функция `printAmount(..)` принимает параметр, который мы назвали `amt`.
Функция `formatAmount()` возвращает значение. Конечно, вы можете комбинировать параметры и возвращаемое значение в одной и той же функции.
Функции часто используются для кода, который вы планируете вызывать несколько раз, но они также полезны для организации связанных частей кода в именованные наборы, даже если вы будете вызывать их всего лишь раз.

Пример:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
        // вычисляем новую сумму с налогом
        amt = amt + (amt * TAX_RATE);

        // возвращаем новую сумму
        return amt;
}

var amount = 99.99;

amount = calculateFinalPurchaseAmount( amount );

console.log( amount.toFixed( 2 ) );               // "107.99"
```

Хотя `calculateFinalPurchaseAmount(..)` вызывается только один раз, выделение ее поведения в отдельную именованную функцию делает код, использующий ее логику (оператор `amount = calculateFinal...`), яснее. Если в функции есть несколько операторов, ее преимущества будут более очевидны.

Область видимости

Если вы попросите у продавца телефонов модель телефона, которой у магазина нет в продаже, он не сможет продать вам телефон, который вы хотите. У него есть доступ только к телефонам которые есть в наличии в магазине. Вы должны попробовать найти другой магазин, чтобы посмотреть есть ли в нем нужный вам телефон.

В программировании есть термин для этого принципа: *область видимости* (технически называемая *лексическая область видимости*). В JavaScript каждая функция получает свою собственную область видимости. Область видимости — это в основном коллекция переменных и правила доступа к этим переменным по имени. Только код внутри функции имеет доступ к переменным, *действующим в области* функции.

Имя переменной должно быть уникальным в рамках одной и той же области видимости — не может быть двух различных переменных `a`, расположенных рядом друг с другом. Но одно и тоже имя переменной `a` может появляться в разных областях видимости.

```
function one() {
        // эта `a` принадлежит только функции `one()`
        var a = 1;
        console.log( a );
}

function two() {
        // эта `a` принадлежит только функции `two()`
        var a = 2;
        console.log( a );
}

one();          // 1
two();          // 2
```

Также, область видимости может быть вложена внутрь другой области видимости, прямо как клоун на дне рождения надувает один шарик внутри другого. Если одна область вложена в другую, для кода внутри самой внутренней области доступны переменные из окружающей области.

Пример:

```
function outer() {
        var a = 1;

        function inner() {
                var b = 2;

                // здесь у нас есть доступ и к `a`, и к `b`
                console.log( a + b );    // 3
        }

        inner();

        // здесь у нас есть доступ только к  `a`
        console.log( a );                       // 1
}

outer();
```

Правила лексической области видимости говорят, что код в одной области может иметь доступ к переменным как ее самой, так и к переменным любой области снаружи этой области.

Таким образом, код внутри функции `inner()` имеет доступ к обеим переменным `a` и `b`, но у кода в `outer()` есть доступ только к `a` — у него нет доступа к `b` потому что эта переменная внутри `inner()`.
Вспомните этот код, который появлялся выше:

```
const TAX_RATE = 0.08;

function calculateFinalPurchaseAmount(amt) {
        // вычисляем новую сумму с налогом
```

```
        amt = amt + (amt * TAX_RATE);

        // возвращаем новую сумму
        return amt;
}
```

Константа (переменная) `TAX_RATE` доступна внутри функции `calculateFinalPurchaseAmount(..)`, даже несмотря на то, что мы не передавали ее внутрь, из-за лексической области видимости.

Примечание: Подробная информация о лексической области видимости есть в первых трех главах книги *Область видимости и замыкания* этой серии.

Практика

Нет абсолютно никакой адекватной замены практике при обучении программированию. Никакое, даже самое ясное, описание с моей стороны само по себе не сделает из вас программиста.

Держа это в уме, давайте попробуем попрактиковаться в некоторых принципах, которые мы изучили в этой главе. Я дам вам "требования", а вы попробуете их реализовать. Затем сверьтесь с кодом, приведенным ниже, чтобы увидеть как я их реализовал.

- Напишите программу для вычисления общей стоимости покупки телефона. Вы будете продолжать покупать телефоны (подсказка: циклы!) пока у вас не закончатся деньги на банковском счете. Вы также будете покупать аксессуары для каждого из телефонов до тех пор, пока сумма покупки не превысит ваш мысленный предел трат.
- После того, как вы посчитаете сумму покупки, прибавьте налог, затем выведите на экран вычисленную сумму покупки, правильно отформатировав ее.
- Наконец, сверьте сумму с балансом вашего банковского счета, чтобы понять можете вы себе это позволить или нет.
- Вы должны настроить некоторые константы для "ставки налога", "цены телефона", "цены аксессуара" и "предела трат" также как и переменную для вашего "баланса банковского счета".
- Вам следует определить функции для вычисления налога и для форматирования цены со знаком валюты и округлением до двух знаков после запятой.
- **Бонусная задача:** Попробуйте включить ввод данных в вашу программу, например с помощью `prompt(..)`, рассмотренную ранее в разделе "Ввод". Вы можете, например, запросить у пользователя баланс банковского счета. Развлекайтесь и будьте изобретательны!

Хорошо, вперед. Попробуйте. Не подсматривайте в мой код пока сами не попробуете!

Примечание: Так как это книга о JavaScript, очевидно что я буду решать практические упражнения на JavaScript. Но вы можете сделать это на другом языке, если чувствуете себя в нем более уверенно.

Вот мое решение для этого упражнения, написанное на JavaScript:

```
const SPENDING_THRESHOLD = 200;
const TAX_RATE = 0.08;
const PHONE_PRICE = 99.99;
const ACCESSORY_PRICE = 9.99;

var bank_balance = 303.91;
var amount = 0;

function calculateTax(amount) {
        return amount * TAX_RATE;
}

function formatAmount(amount) {
        return "$" + amount.toFixed( 2 );
}

// продолжаем покупать телефоны пока у нас остаются деньги
while (amount < bank_balance) {
        // покупаем новый телефон!
        amount = amount + PHONE_PRICE;

        // можем ли мы позволить себе аксессуар?
        if (amount < SPENDING_THRESHOLD) {
                amount = amount + ACCESSORY_PRICE;
        }
}

// не забудьте заплатить налог
amount = amount + calculateTax( amount );

console.log(
        "Ваша покупка: " + formatAmount( amount )
);
// Ваша покупка: $334.76

// можете ли вы в самом деле позволить себе эту покупку?
if (amount > bank_balance) {
        console.log(
                "Вы не можете позволить себе эту покупку. :("
        );
}
// Вы не можете позволить себе эту покупку. :(
```

Примечание: Простейший способ запустить эту JavaScript программу — набрать ее в консоли разработчика в вашем браузере.

Как у вас получилось? Не так уж сложно попробовать снова теперь, когда вы увидели мой код. И поиграть с изменением констант, чтобы увидеть как программа работает с разными значениями.

Резюме

Обучение программированию не такой уж сложный и непреодолимый процесс. Есть всего несколько базовых принципов, которые вам нужно уложить у себя в голове.

Они действуют подобно строительным блокам. Чтобы построить высокую башню, вы начинаете класть блок на блок, блок на блок. То же самое и в программировании. Вот несколько необходимых строительных блоков в программировании:

- Вам нужны *операции* для выполнения действий над значениями.
- Вам нужны значения и *типы* для выполнения различного рода действий, например, математических с `числом` или вывод со `строкой`.
- Вам нужны *переменные* для хранения данных (т.е. *состояния*) в процессе выполнения программы.
- Вам нужны *условные конструкции*, такие как оператор `if`, чтобы принимать решения.
- Вам нужны *циклы*, чтобы повторять действия пока заданное условие не прекратить быть истинным.
- Вам нужны *функции* для организации вашего кода в логические и повторноиспользуемые части программы.

Комментарии к коду — это весьма эффективный путь к написанию более читаемого кода, которые сделают вашу программу легче понимаемой, обслуживаемой и позднее исправляемой в случае проблем.

Наконец, не пренебрегайте мощью практики. Лучший путь научиться как писать код — это писать код.

Я рад, что вы теперь на верном пути к изучению написания кода! Так держать! Не забудьте ознакомиться с другими ресурсами по программированию для начинающих (книги, блоги, онлайн-тренировки и т.д.). Эта глава и эта книга — это большой старт, но они — всего лишь краткое введение.

Следующая глава рассмотрит многие принципы из этой главы, но с более специфичной для JavaScript перспективы, которая осветит многие основные темы, которые будут рассматриваться более детально на протяжении оставшихся книг серии.

Вы не знаете JS: Приступим!

Глава 2: Введение в JavaScript

В предыдущей главе я представил основные строительные блоки программирования, такие как переменные, циклы, условные операторы и функции. Конечно же, весь код, который был показан, был на JavaScript. Но в этой главе, мы хотим особенно сконцентрироваться на вещах, которые вам необходимо знать о JavaScript, чтобы усиленно изучать JS и стать JS-разработчиком.

Мы представим довольно много концепций в этой главе, которые не будут полностью рассмотрены до последующих книг *YDKJS*. Вы можете думать об этой главе как об обзоре тем, раскрытых в деталях на протяжении остальных книг серии.

Особенно это касается вас, если вы — новичок в JavaScript. Вам понадобится потратить немало времени изучая эти концепции и примеры кода, которые есть тут, много раз. Любой хороший фундамент закладывается кирпич за кирпичом, поэтому не ждите, что вы сразу же поймете их все с первого раза.

Здесь начинается ваше путешествие к серьезному изучению JavaScript.

Примечание: Как я упоминал в главе 1, вам определенно стоит попробовать весь этот код самим пока вы читаете и работаете над этой главой. Имейте в виду, что здесь есть код, который предполагает возможности, представленные в последней версии JavaScript на момент написания (обычно упоминаемой как "ES6" из-за шестой версии ECMAScript — официального названия JS спецификации). Если вы вдруг используете более старый, пред-ES6 браузер, код может не заработать. Следует использовать последние версии современных браузеров (такие как Chrome, Firefox или IE).

Значения и типы

Как мы утверждали в главе 1, в JavaScript есть типизированные значения, а не типизированные переменные. Доступны следующие встроенные типы:

- `string` (строка)
- `number` (число)
- `boolean` (логическое значение)
- `null` и `undefined` (пустое значение)
- `object` (объект)
- `symbol` (символ, новое в ES6)

JavaScript предоставляет операцию `typeof`, которая умеет оценивать значение и сообщать вам какого оно типа:

```
var a;
typeof a;                               // "undefined"

a = "hello world";
typeof a;                               // "string"

a = 42;
typeof a;                               // "number"

a = true;
typeof a;                               // "boolean"

a = null;
typeof a;                               // "object" — черт, ошибка

a = undefined;
typeof a;                               // "undefined"

a = { b: "c" };
typeof a;                               // "object"
```

Значение, возвращаемое операцией `typeof`, всегда одно из шести (семи в ES6! - тип "symbol") строковых значений. Это значит, что `typeof "abc"` вернет `"string"`, а не `string`.

Обратите внимание, как в этом коде переменная `a` хранит каждый из различных типов значений и несмотря на видимость, `typeof a` спрашивает не "тип `a`", а "тип текущего значения в `a`." Только у значений есть типы в JavaScript, переменные являются всего лишь контейнерами для этих значений.

`typeof null` — это интересный случай, так как он ошибочно возвращает `"object"`, тогда как вы ожидали бы, что он вернет `"null"`.

Предупреждение: Это давнишняя ошибка в JS, но она похоже никогда не будет исправлена. Слишком много кода в интернет полагается на нее и ее исправление соответственно повлечет за собой намного больше ошибок!

А еще, обратите внимание на `a = undefined`. Мы явно установили `a` в значение `undefined`, но это по поведению не отличается от переменной, у которой еще не установлено значение, например как тут `var a;`, в строке в начале блока кода. Переменная может получать такое состояние значения "undefined" разными

путями, включая функции, которые не возвращают значения, и использование операции `void`.

Объекты

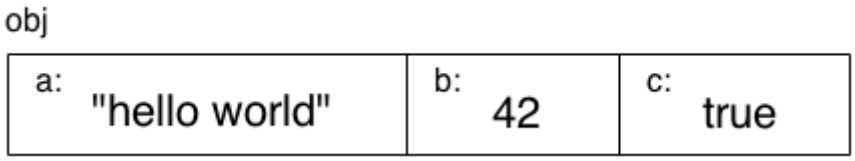
Тип `object` указывает на составное значение, где вы можете устанавливать свойства (именованные области) , которые сами хранят свои собственные значения любого типа. Это возможно одни из самых полезных типов значений во всем JavaScript.

```
var obj = {
        a: "hello world",
        b: 42,
        c: true
};

obj.a;          // "hello world"
obj.b;          // 42
obj.c;          // true

obj["a"];       // "hello world"
obj["b"];       // 42
obj["c"];       // true
```

Полезно представить значение этого `obj` визуально:



Свойства могут быть доступны либо через *точечную нотацию* (т.е., `obj.a`), либо через *скобочную нотацию* (т.е., `obj["a"]`). Точечная нотация короче и в целом легче для чтения и следовательно ей нужно отдавать предпочтение по возможности.

Скобочная нотация полезна, если у вас есть имя свойства, содержащее спецсимволы, например `obj["hello world!"]`— такие свойства часто называют *ключами*, когда к ним обращаются с помощью скобочной нотации. Нотация `[ ]`требует либо переменную (поясняется ниже), либо `строковый` *литерал* (который должен быть заключен в `" .. "` или `' .. '`). Конечно, скобочная нотация также полезна, если вы хотите получить доступ к свойству/ключу, но имя хранится в другой переменной, как здесь, например:

```
var obj = {
        a: "hello world",
        b: 42
};

var b = "a";

obj[b];                 // "hello world"
obj["b"];               // 42
```

Примечание: Более детальная информация о JavaScript `object` есть в книге *this и прототипы объектов* этой серии, особенно в главе 3.

Есть пара других типов значений, с которыми вы обычно взаимодействуете в JavaScript программах: *array* (массив) и *function* (функция). Точнее, вместо того, чтобы быть полноценными встроенными типами, о них следует думать скорее как о подтипах — особых версиях типа `object`.

Массивы

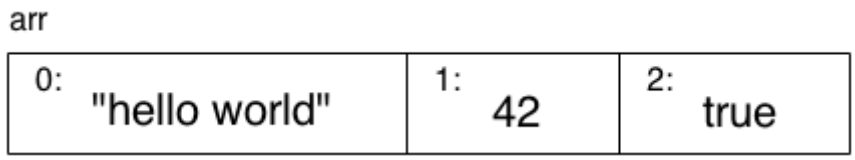
Массив — это `объект`, который хранит значения (любого типа) не отдельно в именованных свойствах/ключах, а в ячейках, доступных по числовому индексу. Например:

```
var arr = [
        "hello world",
        42,
        true
];

arr[0];                 // "hello world"
arr[1];                 // 42
arr[2];                 // true
arr.length;             // 3

typeof arr;             // "object"
```

Примечание: Языки, которые начинают счет с нуля, как и JS, используют `0` как индекс первого элемента массива.
Полезно представить `arr` визуально:



Поскольку массивы — это особые объекты (как намекает `typeof`), то у них могут быть свойства, включая автообновляемое свойство `length` (длина).
Теоретически, вы можете использовать массив как обычный объект со своими собственными свойствами или использовать `object`, но дать ему числовые свойства (`0`, `1` и т.д.) как у массива. Однако, в общем это было бы использование соответствующих типов не по назначению.
Лучший и самый естественный подход — использование массивов для значений, расположенных по числовым позициям и использовать `object` для именованных свойств.

Функции

Еще один подтип `object`, которым вы будете пользоваться во всех ваших JS программах — это функция :

```
function foo() {
        return 42;
}

foo.bar = "hello world";
```

```
typeof foo;                         // "function"
typeof foo();             // "number"
typeof foo.bar;           // "string"
```

И еще раз, функции — это подтипы `объектов`, `typeof` вернет `"function"`, что говорит о том, что `function` — основной тип и поэтому у него могут быть свойства, но обычно вы будете пользоваться свойствами функций (как например `foo.bar`) в редких случаях.

Примечание: Более детальная информация о JS значениях и их типах есть в первых двух главах книги *Типы и синтаксис* этой серии.

Методы встроенных типов

Встроенные типы и подтипы, которые мы только что обсудили, содержат логику, отраженную в свойствах и методах, которые достаточно мощны и полезны.

Например:

```
var a = "hello world";
var b = 3.14159;

a.length;                                 // 11
a.toUpperCase();          // "HELLO WORLD"
b.toFixed(4);                       // "3.1416"
```

Вопрос "Как?", возникающий о возможности вызова `a.toUpperCase()` более сложен, чем то, что этот метод существует у значения.

Коротко говоря, есть форма обертки объекта `String` (заглавная `S`), обычно называемая "родной," которая связывается с примитивным типом `string`, и именно эта обертка определяет метод `toUpperCase()` в своем прототипе.

Когда вы используете примитивное значение, такое как `"hello world"`, как `object` ссылаясь на свойство или метод (к примеру, `a.toUpperCase()` в предыдущем кусочке кода), JS автоматически "упаковывает" значение в его обертку-двойника (скрытую внутри).

Значение типа `string` может быть обернуто объектом `String`, значение типа `number` может быть обернуто объектом `Number` и `boolean` может быть обернуто объектом `Boolean`. В основном, вам не нужно беспокоиться о прямом использовании этих оберток значений — отдавайте предпочтение примитивным формам значений практически во всех случаях, а об остальном позаботится JavaScript.

Примечание: Более детальная информация о родных типах JS и "упаковке" есть в главе 3 книги *Типы и синтаксис* этой серии. Для лучшего понимания прототипов объектов см. главу 5 книги *this и прототипы объектов* этой серии.

Сравнение значений

Есть два основных типа сравнения значений, которые могут понадобится вам в JS программах: *равенство* и *неравенство*. Результатом любого сравнения является

только значение типа `boolean` (`true` или `false`), независимо от сравниваемых типов значений.

Приведение типов (coercion)

Мы немного касались приведения в главе 1, давайте здесь освежим это в памяти еще раз.

Приведение в JavaScript существует в двух формах: *явное* и *неявное*. Явное приведение, это просто то приведение, которое можно очевидным образом увидеть в коде в виде конвертации из одного типа в другой, в свою очередь неявное приведение — это когда конвертация типа может произойти в результате менее очевидных побочных эффектов других операций.

Возможно вы слышали такие мнения как "приведение - зло", опирающиеся на факт, что несомненно есть места, где приведение может привести к удивительным результатам. Вероятно ничто не вызывает расстройство у разработчиков более, чем когда язык преподносит им сюрпризы.

Приведение — не зло и не должно преподносить сюрпризы. На самом деле, большинство случаев, которые вы можете представить используя приведение типов, вполне адекватны и понятны и даже могут использоваться, чтобы *увеличить* читаемость вашего кода. Но мы не будет далее вступать в дебаты — глава 4 книги *Типы и синтаксис* этой серии книг раскроет все стороны приведения.

Вот пример *явного* приведения:

```
var a = "42";

var b = Number( a );

a;                                  // "42"
b;                                  // 42 — число!
```

А вот пример *неявного* приведения:

```
var a = "42";

var b = a * 1;   // здесь "42" неявно приводится к 42

a;                                  // "42"
b;                                  // 42 — число!
```

Истинный и ложный

В главе 1, мы кратко рассмотрели "истинную" и "ложную" природу значений: когда не-`boolean` значение приводится к `boolean`, становится ли оно `true` или `false`, соответственно?
Особый список "ложных" значений в JavaScript таков:

- `""` (пустая строка)
- `0`, `-0`, `NaN` (некорректное число)
- `null`, `undefined`
- `false`

Любое число, не входящее в этот "ложный" список — "истинно." Вот несколько примеров:

- `"hello"`
- `42`
- `true`
- `[ ]`, `[ 1, "2", 3 ]` (массивы)
- `{ }`, `{ a: 42 }` (объекты)
- `function foo() { .. }` (функции)

Важно помнить, что не-`boolean` значение следует такому приведению "истинный"/"ложный" только если оно действительно приводится к `boolean`. Это — не единственная трудность, которая может смутить вас в ситуации, когда кажется что есть приведение значения к `boolean`, когда на самом деле его нет.

Равенство

Есть четыре операции равенства: `==`, `===`, `!=` и `!==`. Формы с `!` — конечно же, симметричные версии "не равно" своих противоположностей; *не равно* не следует путать с *неравенством*.
Разница между `==` и `===` — обычно состоит в том, что `==` проверяет на равенство значений, а `===` проверяет на равенство и значений, и типов. Однако, это не точно. Подходящий способ охарактеризовать их: `==` проверяет на равенство значений с использованием приведения, а `===` проверяет на равенство не разрешая приведение. Операцию `===` часто называют "строгое равенство" по этой причине.
Посмотрите на пример неявного приведения, которое допускается нестрогим равенством `==` и не допускается строгим равенством `===`:

```
var a = "42";
var b = 42;

a == b;                  // true
a === b;         // false
```

В сравнении `a == b`, JS замечает, что типы не совпадают, поэтому он делает упорядоченный ряд шагов, чтобы привести одно или оба значения к различным типам, пока типы не совпадут, а затем уже может быть проверено простое равенство значений.
Если подумать, то есть два возможных пути, когда `a == b` может стать `true` через приведение. Либо сравнение может закончится на `42 == 42`, либо на `"42" == "42"`. Так какое же из них?
Ответ: `"42"` становится `42`, чтобы сделать сравнение `42 == 42`. В таком простом примере не так уж важно по какому пути пойдет сравнение, в конце результат будет один и тот же. Есть более сложные случаи, где важно не только каков конечный результат сравнения, но и *как* вы к нему пришли.

Сравнение `a === b` даст `false`, так как приведение не разрешено, поэтому простое сравнение значений очевидно не завершится успехом. Многие разработчики чувствуют, что операция `===` — более предсказуема, поэтому они советуют всегда использовать эту форму и держаться подальше от `==`. Думаю, такая точка зрения очень недальновидна. Я верю, что операция `==` — мощный инструмент, который помогает вашей программе, *если вы уделите время на изучение того, как это работает*.
Мы не собираемся рассматривать все скучные мельчайшие подробности того, как работает приведение в сравнениях `==`. Многие из них очень разумные, но есть несколько важных тупиковых ситуаций, с которыми надо быть осторожным. Чтобы посмотреть точные правила, можно заглянуть в раздел 11.9.3 спецификации ES5 (http://www.ecma-international.org/ecma-262/5.1/) и вы будете удивлены тем, насколько этот механизм прямолинейный, по сравнению со всей этой негативной шумихой вокруг него.
Чтобы свести целое множество деталей к нескольким простым мыслям и помочь вам узнать, использовать ли `==` или `===` в различных ситуациях, вот мои простые правила:

- Если одно из значений (т.е. сторона) в сравнении может быть значением `true` или `false`, избегайте `==` и используйте `===`.
- Если одно из значений в сравнении может быть одним из этих особых значений (`0`, `""` или `[]` — пустой массив), избегайте `==` и используйте `===`.
- Во *всех* остальных случаях, вы можете безопасно использовать `==`. Это не только безопасно, но во многих случаях это упрощает ваш код путем повышения читаемости.

Эти правила сводятся к тому, что требуют от вас критически оценивать свой код и думать о том, какого вида значения могут исходить из переменных, которые проверяются на равенство. Если вы уверены насчет значений и `==`безопасна, используйте ее! Если вы не уверены насчет значений, используйте `===`. Это просто.
Форма не-равно `!=` идет в паре с `==`, а форма `!==` — в паре с `===`. Все правила и утверждения, которые мы только что обсудили также применимы для этих сравнений на не равно.
Вам следует особо обратить внимение на правила сравнения `==` и `===`, если вы сравниваете два непримитивных значения, таких
как `object` (включая `function` и `array`). Так как эти значения на самом деле хранятся по ссылке, оба сравнения `==` и `===` просто проверят равны ли ссылки, но ничего не сделают касаемо самих значений.
Например, `массив` по умолчанию приводится к `строке` простым присоединением всех значений с запятыми (`,`) между ними. Можно было бы подумать, что эти два `массива` с одинаковым содержимым будут равны по `==`, но это не так:

```
var a = [1,2,3];
var b = [1,2,3];
var c = "1,2,3";

a == c;          // true
b == c;          // true
```

```
a == b;          // false
```

Примечание: Детальную информацию о правилах сравнения равенства == можно посмотреть в спецификации ES5 (раздел 11.9.3) а также свериться с главой 4 книги *Типы и синтаксис* этой серии; см. главу 2 для детальной информации о значениях в сравнении с ссылками.

Неравенство

Операции <, >, <= и >=, использующиеся для неравенств, упоминаются в спецификации как "относительное сравнение." Обычно они используются со значениями, сравниваемыми порядками, как числа. Легко понять, что `3 < 4`.
Но строковые значения в JavaScript тоже могут участвовать в неравенствах, используя типичные алфавитные правила (`"bar" < "foo"`).
Как насчет приведения? Тут всё похоже на правила в сравнении == (хотя и не совсем идентично!). Примечательно, что нет операций "строгого неравенства", которые бы запрещали приведение таким же путем как и "строгое равенство" ===. Пример:

```
var a = 41;
var b = "42";
var c = "43";

a < b;           // true
b < c;           // true
```

Что здесь происходит? В разделе 11.8.5 спецификации ES5 говорится, что если оба значения в сравнении < являются строками, как это было в случае с `b < c`, сравнение производится лексикографически (т.е. в алфавитном порядке, как в словаре). но если одно или оба значения не являются строкой, как в случае с `a < b`, то оба значения приводятся к числу и происходит типичное числовое сравнение.
Самое большое затруднение, в которое вы можете попасть со сравнениями между потенциально разными типами значений (помните, что нет формы "строгого неравенства"?) — это когда одно из значений не может быть превращено в корректное число, например:

```
var a = 42;
var b = "foo";

a < b;           // false
a > b;           // false
a == b;          // false
```

Подождите-ка, как это все эти три сравнения могут быть `false`? Так как значение `b` приводится к "некорректному числовому значению" `NaN` в сравнениях < и >, а спецификация говорит, что `NaN` не больше и не меньше чем любое другое значение.
Сравнение == не проходит по другой причине. `a == b` может быть некорректным если оно интерпретируется как `42 == NaN` или `"42" == "foo"` — как мы объяснили ранее, второй вариант — наш случай.

Примечание: Более детальная информация о правилах сравнения в неравенствах есть в разделе 11.8.5 спецификации ES5, также сверьтесь с главой 4 книги *Типы и синтаксис* этой серии.

Переменные

В JavaScript имена переменных (включая имена функций) должны быть корректными *идентификаторами*. Строгие и полные правила о корректных символах в идентификаторах — немного сложны, когда вы хотите использовать нестандартные символы, такие как Unicode-символы. Если вы предполагаете использовать только типичные буквенно-цифровые ASCII-символы, то правила просты.

Идентификатор должен начинаться с `a-z`, `A-Z`, `$` или `_`. Дальше он может содержать любые из этих же символов плюс цифры `0-9`.
В общем-то, те же правила, как и к идентификатору переменной, применяются и к имени свойства. Однако, определенные слова не могут использоваться как переменные, но могут как имена свойств. Эти слова называются "зарезервированными словами," и включают ключевые слова JS (`for`, `in`, `if` и т.д.), также как и `null`, `true` и `false`.
Примечание: Более детальная информация о зарезервированных словах есть в приложении А книги *Типы и синтаксис* этой серии.

Области видимости функций

Вы используете ключевое `var`, чтобы объявить переменную, которая принадлежит области видимости текущей функции или глобальной области, если находится на верхнем уровне вне любой функции.

Поднятие переменной (hoisting)

Где бы ни появлялось `var` внутри области видимости, это объявление принадлежит всей области видимости и доступно везде в ней.
Метафорически это поведение называется *поднятие (hoisting)*, когда объявление `var` концептуально "перемещается" на вершину своей объемлющей области видимости. Технически этот процесс более точно объясняется тем, как компилируется код, но сейчас опустим эти подробности.
Пример:

```
var a = 2;

foo();                                  // работает, так как определение `foo()`
                                                // "всплыло"

function foo() {
        a = 3;
```

```
        console.log( a );       // 3

        var a;                          // определение "всплыло"
                                        // наверх `foo()`
}

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

Предупреждение: Не общепринято и не так уж здраво полагаться на *поднятие* переменной, чтобы использовать переменную раньше в ее области видимости, чем появится ее объявление `var`, такое может сбить с толку. Общепринято и приемлемо использовать *всплытие* объявлений функций, что мы и делали с вызовом `foo()`, появившемся до ее объявления.

Вложенные области видимости

Когда вы объявляете переменную, она доступна везде в ее области видимости, также как и в более нижних/внутренних областях видимости. Например:

```
function foo() {
        var a = 1;

        function bar() {
                var b = 2;

                function baz() {
                        var c = 3;

                        console.log( a, b, c );  // 1 2 3
                }

                baz();
                console.log( a, b );            // 1 2
        }

        bar();
        console.log( a );                               // 1
}

foo();
```

Заметьте, что `c` не доступна внутри `bar()`, потому что она объявлена только внутри внутренней области видимости `baz()` и `b` не доступна в `foo()` по той же причине. Если вы попытаетесь получить доступ к значению переменной в области видимости, где она уже недоступна, вы получите `ReferenceError`. Если вы попытаетесь установить значение переменной, которая еще не объявлена, у вас либо закончится тем, что переменная создастся в самой верхней глобальной области видимости (плохо!), либо получите ошибку в зависимости от "строгого режима" (см. "Строгий режим"). Давайте взглянем:

```
function foo() {
        a = 1;  // `a` формально не объявлена
}

foo();
a;                      // 1 — упс, автоматическая глобальная переменная :(
```

Это очень плохая практика. Не делайте так! Всегда явно объявляйте свои переменные.

В дополнение к созданию объявлений переменных на уровне функций, ES6 *позволяет* вам объявлять переменные, принадлежащие отдельным блокам (пара `{ .. }`), используя ключевое слово `let`. Кроме некоторых едва уловимых деталей, правила области видимости будут вести себя точно также, как мы видели в функциях:

```
function foo() {
        var a = 1;

        if (a >= 1) {
                let b = 2;

                while (b < 5) {
                        let c = b * 2;
                        b++;

                        console.log( a + c );
                }
        }
}

foo();
// 5 7 9
```

Из-за использования `let` вместо `var`, `b` будет принадлежать только оператору `if` и следовательно не всей области видимости функции `foo()`. Точно также `c` принадлежит только циклу `while`. Блочная область видимости очень полезна для управления областями ваших переменных более точно, что может сделать ваш код более легким в обслуживании в долгосрочной перспективе.
Примечание: Более детальная информация об области видимости есть в книге *Область видимости и замыкания* этой серии. См. книгу *ES6 и за его пределами* этой серии, чтобы узнать больше о блочной области видимости `let`.

Условные операторы

В дополнение к оператору `if`, который мы кратко представили в главе 1, JavaScript предоставляет несколько других механизмов условных операторов, на которые нам следует взглянуть.
Иногда вы ловите себя на том, что пишете серию операторов `if..else..if` примерно как тут:

```
if (a == 2) {
        // сделать что-то
}
else if (a == 10) {
        // сделать что-то еще
}
else if (a == 42) {
        // сделать еще одну вещь
}
else {
        // резервный вариант
```

```
}
```

Эта структура работает, но она немного слишком подробна, поскольку вам нужно указать проверку для `a` в каждом случае. Вот альтернативная возможность, оператор `switch`:

```
switch (a) {
        case 2:
    // сделать что-то
                break;
        case 10:
    // сделать что-то еще
                break;
        case 42:
    // сделать еще одну вещь
                break;
        default:
    // резервный вариант
}
```

Оператор `break` важен, если вы хотите, чтобы выполнились операторы только одного `case`. Если вы опустите `break` в `case` и этот `case` подойдет или выполнится, выполнение продолжится в следующем операторе `case` независимо то того, подходит ли этот `case`. Этот так называемый "провал (fall through)" иногда полезен/желателен:

```
switch (a) {
        case 2:
        case 10:
                // какие-то крутые вещи
                break;
        case 42:
                // другие вещи
                break;
        default:
                // резерв
}
```

Здесь если `a` будет либо `2`, либо `10`, то выполнятся операторы "какие-то крутые вещи".

Еще одна форма условного оператора в JavaScript — это "условная операция", часто называемая "тернарная операция." Это примерно как более краткая форма отдельного оператора `if..else`, например:

```
var a = 42;

var b = (a > 41) ? "hello" : "world";

// эквивалентно этому:

// if (a > 41) {
//    b = "hello";
// }
// else {
//    b = "world";
// }
```

Если проверяемое выражение (здесь `a > 41`) вычисляется как `true`, результатом будет первая часть (`"hello"`), в противном случае результатом будет вторая часть (`"world"`) и затем независимо от результата он будет присвоен переменной `b`. Условная операция не обязательно должна использоваться в присваивании, но это самое распространенное ее использование.

Примечание: Более детальная информация об условиях проверки и других шаблонах для `switch` и `? :` есть в книге *Типы и синтаксис* этой серии.

Строгий режим (Strict Mode)

ES5 добавила "строгий режим" в язык, который ужесточил правила для определенных сценариев. В общем-то, эти ограничения выглядят как большее соответствие кода более безопасному и более подходящему набору рекомендаций. Также, тяготение к строгому режиму сделает ваш код более оптимизируемым движком. Строгий режим — это большая победа для кода и вам следует использовать его во всех своих программах.

Вы можете явно указать его для отдельной функции или целого файла, в зависимости от того, где вы разместите директиву строго режима:

```
function foo() {
        "use strict";

        // этот код в строгом режиме

        function bar() {
                // этот код в строгом режиме
        }
}

// этот код в нестрогом режиме
```

Сравните с:

```
"use strict";

function foo() {
        // этот код в строгом режиме

        function bar() {
                // этот код в строгом режиме
        }
}

// этот код в строгом режиме
```

Всего одно ключевое отличие (улучшение!) строго режима — запрет автоматического неявного объявления глобальных переменных из-за пропуска`var`:

```
function foo() {
        "use strict";    // включить строгий режим
        a = 1;                  // `var` missing, ReferenceError
}

foo();
```

Если вы включаете строгий режим в своем коде и получаете ошибки или код начинает вести себя ошибочно, у вас может возникнуть соблазн избегать строго режима. Но этот инстинкт — плохая идея, чтобы потворствовать этому. Если строгий режим является причиной проблем в вашей программе, почти определенно это знак того, что в вашей программе есть вещи, которые надо исправить.

Строгий режим не только способствует большей безопасности вашего кода и не только делает ваш код более оптимизируемым, но и заодно показывает будущее направление языка. Вам будет легче привыкнуть к строгому режиму сейчас, чем продолжать откладывать его в сторону — код будет сложнее потом сконвертировать!

Примечание: Более детальная информация о строгом режиме есть в главе 5 книги *Типы и синтаксис* этой серии.

Функции как значения

До сих пор, мы обсуждали функции как основной механизм *области видимости* в JavaScript. Вспомните синтаксис типичного объявления функции, указанный ниже:

```
function foo() {
        // ..
}
```

Хотя это может показаться очевидным из синтаксиса, `foo` — по сути просто переменная во внешней окружающей области видимости, у которой есть ссылка на объявляемую функцию. То есть, функция сама является значением, также как `42` или `[1,2,3]`.

Это может сперва прозвучать как странная идея, поэтому уделим время ее изучению. Вы не только можете передать значение (аргумент) *в* функцию, но и *сама функция может быть значением*, которое может быть присвоено переменным или передано, или возвращено из других функций.

В связи с этим, о значении-функции следует думать как о выражении, сродни любому другому значению или выражению.

Пример:

```
var foo = function() {
        // ..
};

var x = function bar(){
        // ..
};
```

Первое функциональное выражение, присваиваемое переменной `foo`, называется *анонимным* поскольку у него нет имени.

Второе функциональное выражение *именованное* (`bar`), несмотря на то, что является ссылкой, также присваивается переменной `x`. *Выражения с именованными функциями* как правило более предпочтительны, хотя *выражения с анонимными функциями* все еще чрезвычайно употребительны.

Более детальная информация есть в книге *Область видимости и замыкания* этой серии.

Выражения немедленно вызываемых функций (Immediately Invoked Function Expressions (IIFEs))

В предыдущем примере, ни одно из выражений с функциями не выполнялось, мы могли бы это сделать включив в код `foo()` или `x()`, например.
Есть еще один путь выполнить выражение с функцией, на который обычно ссылаются как на *immediately invoked function expression* (IIFE):

```
(function IIFE(){
        console.log( "Hello!" );
})();
// "Hello!"
```

Внешние `( .. )`, которые окружают выражение функции `(function IIFE(){ .. })` — это всего лишь нюанс грамматики JS, необходимый для предотвращения того, чтобы это выражение воспринималось как объявление обычной функции.
Последние `()` в конце выражения, строка `})();` — это то, что и выполняет выражение с функцией, указанное сразу перед ним.
Может показаться странным, но это не так уж чужеродно, как кажется на первый взгляд. Посмотрите на сходства между `foo` и `IIFE` тут:

```
function foo() { .. }

// `foo` выражение со ссылкой на функцию,
// затем `()` выполняют ее
foo();

// Выражение с функцией `IIFE`,
// затем `()` выполняют ее
(function IIFE(){ .. })();
```

Как видите, содержимое `(function IIFE(){ .. })` до ее вызова в `()` фактически такое же, как включение `foo` до его вызова после `()`. В обоих случаях ссылка на функцию выполняется с помощью `()` сразу после них.
Так как IIFE — просто функция, а функции создают *область видимости* переменных, то использование IIFE таким образом обычно происходит, чтобы объявлять переменные, которые не будут влиять на код, окружающий IIFE снаружи:

```
var a = 42;

(function IIFE(){
        var a = 10;
        console.log( a );        // 10
})();

console.log( a );                // 42
```

Функции IIFE также могут возвращать значения:

```
var x = (function IIFE(){
        return 42;
})();

x;      // 42
```

Значение `42` возвращается из выполненной `IIFE` функции, а затем присваивается в `x`.

Замыкание

Замыкание — одно из самых важных и зачастую наименее понятных концепций в JavaScript. Я не буду вдаваться в подробности сейчас, а вместо этого направляю вас в книгу *Область видимости и замыкания* этой серии. Но я хотел бы сказать несколько слов о них, чтобы вы понимали общую концепцию. Это будет одна из самых важных техник в вашем наборе навыков в JS.

Вы можете думать о замыкании как о пути "запомнить" и продолжить работу в области видимости функции (с ее переменными) даже когда функция уже закончила свою работу.

Проиллюстрируем:

```
function makeAdder(x) {
        // параметр `x` - внутренная переменная

        // внутренняя функция `add()` использует `x`, поэтому
        // у нее есть "замыкание" на нее
        function add(y) {
                return y + x;
        };

        return add;
}
```

Ссылка на внутреннюю функцию `add(..)`, которая возвращается с каждым вызовом внешней `makeAdder(..)`, умеет запоминать какое значение `x` было передано в `makeAdder(..)`. Теперь, давайте используем `makeAdder(..)`:

```
// `plusOne` получает ссылку на внутреннюю функцию `add(..)`
// с замыканием на параметре `x`
// внешней `makeAdder(..)`
var plusOne = makeAdder( 1 );

// `plusTen` получает ссылку на внутреннюю функцию `add(..)`
// с замыканием на параметре `x`
// внешней `makeAdder(..)`
var plusTen = makeAdder( 10 );

plusOne( 3 );           // 4  <-- 1 + 3
plusOne( 41 );          // 42 <-- 1 + 41

plusTen( 13 );          // 23 <-- 10 + 13
```

Теперь подробней о том, как работает этот код:

1. Когда мы вызываем `makeAdder(1)`, мы получаем обратно ссылку на ее внутреннюю `add(..)`, которая запоминает `x` как `1`. Мы назвали эту ссылку на функцию `plusOne(..)`.
2. Когда мы вызываем `makeAdder(10)`, мы получаем обратно ссылку на ее внутреннюю `add(..)`, которая запоминает `x` как `10`. Мы назвали эту ссылку на функцию `plusTen(..)`.
3. Когда мы вызываем `plusOne(3)`, она прибавляет `3` (свою внутреннюю `y`) к `1` (которая запомнена в `x`) и мы получаем в качестве результата `4`.

4. Когда мы вызываем `plusTen(13)`, она прибавляет `13` (свою внутреннюю `y`) к `10` (которая запомнена в `x`), и мы получаем в качестве результата `23`.

Не волнуйтесь, если всё это кажется странным и сбивающим с толку поначалу — это нормально! Понадобится много практики, чтобы всё это полностью понять.

Но поверьте мне, как только вы это освоите, это будет одной из самых мощных и полезных техник во всем программировании. Определенно стоит приложить усилия, чтобы ваши мозги немного покипели над замыканиями. В следующем разделе, мы немного попрактикуемся с замыканиями.

Модули

Самое распространенное использование замыкания в JavaScript — это модульный шаблон. Модули позволяют определять частные детали реализации (переменные, функции), которые скрыты от внешнего мира, а также публичное API, которое *доступно* снаружи.

Представим:

```
function User(){
        var username, password;

        function doLogin(user,pw) {
                username = user;
                password = pw;

                // сделать остальную часть работы по логину
        }

        var publicAPI = {
                login: doLogin
        };

        return publicAPI;
}

// создать экземпляр модуля `User`
var fred = User();

fred.login( "fred", "12Battery34!" );
```

Функция `User()` служит как внешняя область видимости, которая хранит переменные `username` и `password`, а также внутреннюю функцию `doLogin()`. Всё это частные внутренние детали этого модуля `User`, которые недоступны из внешнего мира.

Предупреждение: Мы не вызываем тут `new User()` намеренно, несмотря на тот факт, что это будет более естественно для большинства читателей. `User()` — просто функция, а не класс, поэтому она вызывается обычным образом. Использование `new` было бы неуместным и еще и тратой попусту ресурсов. При выполнении `User()` создается *экземпляр* модуля `User` и создается целая новая область видимости и также совершенно новая копия каждой из этих внутренних

переменных/функций. Мы присваиваем этот экземпляр в `fred`. Если мы запустим `User()` снова, мы получим новый экземпляр, целиком отдельный от `fred`. У внутренней функции `doLogin()` есть замыкание на `username` и `password`, что значит, что она сохранит свой доступ к ним даже после того, как функция `User()` завершит свое выполнение.
`publicAPI` — это объект с одним свойством/методом, `login`, который является ссылкой на внутреннюю функцию `doLogin()`. Когда вы возвращаем `publicAPI` из `User()`, он становится экземпляром, который мы назвали `fred`.
На данный момент, внешняя функция `User()` закончила выполнение. Как правило, вы думаете, что внутренние переменные, такие как `username` и `password`, при этом исчезают. Но они никуда не деваются, потому что есть замыкание в функции `login()`, хранящее их.
Вот поэтому мы можем вызвать `fred.login(..)`, что тоже самое, что вызвать внутреннюю `doLogin(..)` и у нее все еще будет доступ ко внутренним переменным `username` и `password`.
Есть большой шанс, что кратко бросив взгляд на замыкание и шаблон модуля, кое-что останется неясным. Ничего страшного! Понадобится некоторая работа, чтобы намотать всё это на ус.

Отсюда сходите почитать книгу *Область видимости и замыкания* этой серии для получения более детальных объяснений.

Идентификатор `this`

Еще одна очень часто неверно понимаемая концепция в JavaScript — это идентификатор `this`. И опять таки, есть пара глав по нему в книге *this и прототипы объектов* этой серии, поэтому здесь мы только кратко его рассмотрим.
При том что может часто казаться, что этот `this` связан с "объектно-ориентированным шаблонами," в JS `this` — это другой механизм.
Если у функции есть внутри ссылка `this`, эта ссылка `this` обычно указывает на `объект`. Но на какой `объект` она указывает зависит от того, как эта функция была вызвана.
Важно представлять, что `this` *не* ссылается на саму функцию, учитывая, что это самое распространенное неверное представление.
Вот краткая иллюстрация:

```
function foo() {
        console.log( this.bar );
}

var bar = "global";

var obj1 = {
        bar: "obj1",
        foo: foo
};

var obj2 = {
```

```
        bar: "obj2"
};

//--------

foo();                              // "global"
obj1.foo();                         // "obj1"
foo.call( obj2 );           // "obj2"
new foo();                          // undefined
```

Есть четыре правила того, как устанавливается `this` и они показаны в этих четырех последних строках кода.

1. `foo()` заканчивается установкой `this` в глобальный объект в нестрогом режиме. В строгом режиме, `this` будет `undefined` и вы получите ошибку при доступе к свойству `bar`, поэтому `"global"` — это значение для `this.bar`.
2. `obj1.foo()` устанавливает `this` в объект `obj1`.
3. `foo.call(obj2)` устанавливает `this` в объект `obj2`.
4. `new foo()` устанавливает `this` в абсолютно новый пустой объект.

Резюме: чтобы понять на что указывает `this`, вы должны проверить как вызывалась на самом деле функция. Это будет один из тех четырех путей, только что показанных, и это то, что поможет потом ответить на вопрос что будет в `this`.
Примечание: Более детальная информация о `this` есть в главах 1 и 2 книги *this и прототипы объектов* этой серии.

Прототипы

Механизм прототипов в JavaScript довольно сложен. Здесь мы только взглянем на него немного. Вам захочется потратить много времени изучая главы 4-6 книги *this и прототипы объектов* этой серии, чтобы получить детальную информацию.

Когда вы ссылаетесь на свойство объекта, то если это свойство не существует, JavaScript автоматически использует ссылку на внутренний прототип этого объекта, чтобы найти другой объект, чтобы поискать свойство там. Можете думать об этом почти как о резервном варианте когда свойство отсутствует.

Связывание ссылки на внутренний прототип от объекта к его резервному варианту происходит в момент когда объект создается. Простейший путь проиллюстрировать это — с помощью вызова встроенной функции `Object.create(..)`.
Пример:

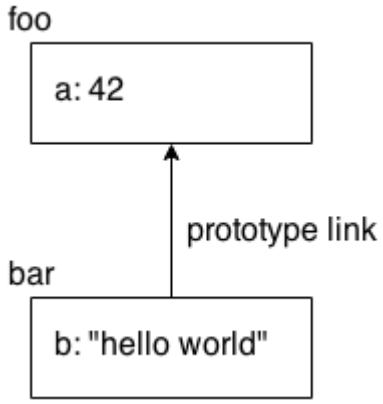
```
var foo = {
        a: 42
};

// создаем `bar` и связываем его с `foo`
var bar = Object.create( foo );

bar.b = "hello world";
```

```
bar.b;          // "hello world"
bar.a;          // 42 <-- делегируется в `foo`
```

Следующая картинка поможет визуально показать объекты `foo` и `bar` и их связь:



Свойство `a` в действительности не существует в объекте `bar`, но поскольку `bar` прототипно связан с `foo`, JavaScript автоматически прибегает к поиску `a` в объекте `foo`, где оно и находится.
Такая связь может показаться странной возможностью языка. Самый распространенный способ ею пользоваться, и я бы поспорил, что неправильно пытаться эмулировать механизм "классов" "наследование".

Но более естественный путь применения прототипов — шаблон, называемый "делегирование поведения", когда вы намеренно проектируете свои связанные объекты так, чтобы они могли *делегировать* от одного к другому части необходимого поведения.

Примечание: Более детальная информация о прототипах и делегировании поведения есть в главах 4-6 книги *this & прототипы объектов* этой серии.

Старый и новый

Некоторые из возможностей JS, которые мы уже рассмотрели, и конечно многие возможности, рассмотренные в оставшихся книгах серии, являются достаточно новыми дополнениями и не обязательно будут доступны в более старых браузерах. По факту, некоторые новейшие возможности в спецификации еще не реализованы ни в одной из стабильных версий браузеров.

Так что же вам делать со всеми этими новыми вещами? Нужно ли ждать годы или десятилетия, чтобы все старые браузеры канули в лету?

Именно так думаю многие люди об этой ситуации, но это совсем не здравый подход к JS.

Есть две основные техники, которыми можно пользоваться, чтобы "привнести" более новые возможности JavaScript в старые браузеры: полифиллинг (polyfilling) и транспиляция (transpiling).

Полифиллинг (polyfilling)

Слово "polyfill" — изобретенный термин (Реми Шарпом) (https://remysharp.com/2010/10/08/what-is-a-polyfill), используется для указания на взятие определения новой возможности и генерации кода, эквивалентного этому поведению, но с возможностью запуска в более старых окружениях JS.

Например, ES6 определяет функцию, называемую `Number.isNaN(..)` для обеспечения точной безошибочной проверки на значения `NaN`, отмечая как устаревшую исходную функцию `isNaN(..)`. Но очень легко заполифиллить эту функцию, чтобы вы могли пользоваться ею в вашем коде независимо от того, поддерживает браузер ES6 или нет.
Пример:

```
if (!Number.isNaN) {
        Number.isNaN = function isNaN(x) {
                return x !== x;
        };
}
```

Оператор `if` защищает против применения полифильного определения в браузерах с ES6, где функция уже есть. Если она еще не существует, мы определяем `Number.isNaN(..)`.
Примечание: Проверка, которую мы тут выполняем, использует преимущество причудливости значения `NaN`, которое заключается в том, что оно является единственным значением во всем языке, которое не равно самому себе. Поэтому значение `NaN` — единственное, которое может сделать условие `x !== x` истинным.
Не все новые возможностью полностью полифильны. Иногда большая часть поведения может быть сполифиллена, но еще есть пока что небольшие отступления. Вы должны быть очень, очень осторожны реализуя полифиллинг сами, следя за тем, чтобы придерживаться спецификации настолько строго, насколько возможно.

Или даже лучше используйте уже проверенный набор полифиллов, которому вы можете доверять, вроде тех, что предоставляются ES5-Shim (https://github.com/es-shims/es5-shim) и ES6-Shim (https://github.com/es-shims/es6-shim).

Транспиляция (Transpiling)

Не существует возможности полифиллить новый синтаксис, который был добавлен в язык. Новый синтаксис вызовет ошибку в старом движке JS как нераспознанный/невалидный.

Поэтому лучшим выбором будет использовать утилиту, которая конвертирует ваш более новый код в эквивалент более старого. Этот процесс обычно называют "транспиляцией", как объединение терминов трансформация и компиляция (transforming + compiling).

По большому счету, ваш исходный код написан в новом синтаксисе, но то, что вы развертываете в браузере — это транспилированный код со старым синтаксисом. Вы обычно вставляете транспилятор в ваш процесс сборки, примерно так же как linter или minifier.

Вы могли бы удивиться, а почему идете на неприятности, чтобы писать только в новом синтаксисе, чтобы потом транспилировать его в старый код? Почему бы просто не писать напрямую в старом синтаксисе?

Есть несколько важных причин, чтобы вы позаботились о транспиляции:

- Новый синтаксис, добавленный в язык, разрабатывается, чтобы заставить ваш код быть более читаемым и обслуживаемым. Старые эквиваленты часто намного более запутанны. Следует предпочитать писать с помощью более нового и ясного синтаксиса, не только для себя, но и для всех остальных членов команды разработки.
- Если вы транспилируете только для старых браузеров, но используете новый синтаксис в новейших браузерах, вы получаете преимущество оптимизации производительности браузера с помощью нового синтаксиса. Это также позволяет разработчикам браузеров делать код более приближенным к жизни для проверки их реализаций и оптимизаций.
- Использование нового синтаксиса как можно раньше позволяет ему быть протестированным более тесно в реальном мире, что обеспечивает более ранние отзывы в комитет JavaScript (TC39). Если проблемы обнаружены достаточно рано, их можно изменить/устранить до того, как эти ошибки дизайна языка станут постоянными.

Вот быстрый пример транспиляции. ES6 добавляет возможность, называемую "значения параметров по умолчанию". Это выглядит примерно так:

```
function foo(a = 2) {
        console.log( a );
}

foo();          // 2
foo( 42 );      // 42
```

Просто, правда? Еще и полезно! Но это как раз новый синтаксис, который будет считаться невалидным в до-ES6 движках. Так что же транспилятор сделает с этим кодом, чтобы заставить его работать в более старых движках?

```
function foo() {
        var a = arguments[0] !== (void 0) ? arguments[0] : 2;
        console.log( a );
}
```

Как видите, он проверяет, что значение `arguments[0]` — `void 0` (т.е. `undefined`) и если да, то предоставляет значение по умолчанию `2`, иначе он присваивает то, что было передано.

В дополнение к тому, что теперь можно использовать привлекательный синтаксис даже в старых браузерах, транспилированный код фактически делает заданное поведение яснее.

Возможно вы даже не представляли себе просто глядя на версию ES6, что `undefined` — единственное значение, которое не может быть явно задано значением по умолчанию для параметра, но транспилированный код показывает это гораздо лучше.
Последняя важная деталь, чтобы сделать акцент на транспиляторах — то, что о них следует думать как о стандартной части экосистемы и процесса разработки на JS. JS будет продолжать эволюционировать, намного быстрее, чем прежде, поэтому каждые несколько месяцев будут добавляться новый синтаксис и новые возможности.

Если вы по умолчанию используете транспилятор, вы всегда сможете переключиться на новый синтаксис, когда бы ни захотели, нежели всегда ждать годы, чтобы сегодняшние браузеры вышли из употребления.

Есть довольно много отличных транспиляторов для выбора. Вот несколько из них на момент написания этого текста:

- Babel (https://babeljs.io) (бывший 6to5): Транспилирует из ES6+ в ES5
- Traceur (https://github.com/google/traceur-compiler): Транспилирует из ES6, ES7 и далее в ES5

Не-JavaScript

На данный момент, мы рассмотрели только вещи, касающиеся самого языка JS. Реальность такова, что большая часть JS написана для запуска и взаимодействия с такими средами как браузеры. Добрая часть вещей, которые вы пишете в своем коде, строго говоря, не контролируется напрямую JavaScript. Возможно это звучит несколько странно.

Самый распространенный не-JavaScript JavaScript, с которым вы столкнетесь — это DOM API. Например:

```
var el = document.getElementById( "foo" );
```

Переменная `document` существует как глобальная переменная, когда ваш код выполняется в браузере. Она не обеспечивается ни движком JS, ни особенно не контролируется спецификацией JavaScript. Она принимает форму чего-то очень ужасного похожего на обычный JS `объект`, но не является им на самом деле. Это — специальный `объект`, часто называемый "хост-объектом."
Более того, метод `getElementById(..)` в `document` выглядит как обычная функция JS, но это всего лишь кое-как открытый интерфейс к встроенному методу, предоставляемому DOM из вашего браузера. В некоторых (нового поколения) браузерах этот слой может быть на JS, но традиционно DOM и его поведение реализовано на чем-то вроде C/C++.
Еще один пример с вводом/выводом (I/O).

Всеобщее любимое всплывающее окно `alert(..)` в пользовательском окне браузера. `alert(..)` предоставляется вашей программе на JS браузером, а не самим движком JS. Вызов, который вы делаете, отправляет сообщение во внутренности браузера и они обрабатывают отрисовку и отображение окна с сообщением.
То же происходит и с `console.log(..)` — ваш браузер предоставляет такие механизмы и подключает их к средствам разработчика.
Эта книга и вся эта серия фокусируется на языке JavaScript. Поэтому вы не увидите какого-либо подробного раскрытия деталей об этих не-JavaScript механизмах JavaScript. Как бы то ни было, вам не нужно забывать о них, поскольку они будут в каждой программе на JS, которую вы напишете!

Обзор

Первый шаг в изучении духа программирования JavaScript — получить первичное представление о его внутренних структурах, таких как значения, типы, замыкания функций, `this` и прототипы.
Конечно, каждая из этих тем заслуживает большего раскрытия, чем вы видели здесь, но именно поэтому есть главы и книги, посвященные им, на протяжении оставшихся книг серии. После того как вы почувствуете себя более уверенно в концепциях и примерах кода в этой главе, оставшиеся книги серии ждут вас, чтобы по-настоящему погрузиться и узнать язык основательно.

Последняя глава этой книги коротко подведет итоги оставшихся книг серии и других принципов, которые они раскрывают, в дополнение к тем, что мы уже изучили.

Вы не знаете JS: Приступим!

Глава 3: Введение в "Вы не знаете JS"

О чем вся эта серия? Проще говоря, она о том как взяться серьезно за задачу изучения *всех частей JavaScript*, а не только некоторого подмножества языка, которое кто-то называет "основными частями", и не только совсем минимального количества, необходимого вам, чтобы сделать свою работу.

Серьезные разработчики в других языках обычно планируют приложить усилия для изучения большей части или всего языка, на котором они в основном пишут, но разработчики на JS похоже стоят в стороне от всех в том смысле, что типично не изучают многого в языке. Это не так уж и хорошо и это не то, что нам следует продолжать позволять быть нормой.

Серия *Вы не знаете JS* (*YDKJS*) представляет разительный контраст с типичными подходами к изучению JS и непохожа практически на любые другие книги о JS, которые вы прочтете. Она требует от вас выйти из зоны комфорта и задать серьезные вопросы "почему?" для всех до единой функциональных возможностей, с которыми вы столкнетесь. Вы готовы заняться этой задачей?

Я использую эту последнюю главу, чтобы кратко подвести итог того, чего ждать от оставшихся книг серии и как наиболее эффективно приняться за постройку основы обучения JS держа в своих руках *YDKJS*.

Область действия и замыкания

Наверное одна из самых фундаментальных вещей, которые понадобятся вам, чтобы быстрее подобраться к этим терминам — это как именно создание области действия переменных на самом деле работает в JavaScript. Недостаточно иметь анекдотичные расплывчатые *представления* об области действия.

Книга *Область действия и замыкания* начинается с развенчания общего ложного представления, что JS — "интерпретируемый язык" и потому не компилируется. А вот и нет!

Движок JS компилирует ваш код прямо перед (а иногда и во время!) выполнением. Поэтому вы пойдете путем более глубокого понимания подхода компилятора к вашему коду, чтобы понять как он находит и разбирается с объявлениями переменных и функций. Попутно, мы посмотрим типичную схему JS по управлению областью действия переменных, "Всплытие (hoisting)."

Именно это критическое понимание "лексической области действия" — то, на чем мы потом будет основывать наше исследование замыкания в последней главе этой книги. Замыкание — возможно, единственное самое важное понятие во всем JS, но если вы сперва не разберетесь плотно как работает область действия, замыкание скорее всего останется вне вашего понимания.

Одно важное применение замыкания — это модульный шаблон, который мы уже кратко представили в этой книге в главе 2. Модульный шаблон, возможно, самый преобладающий шаблон организации кода во всем JavaScript; его глубокое понимание должно быть одним из самых высоких ваших приоритетов.

this и прототипы объектов

Пожалуй один из самых распространенных и устойчивых ложных фактов о JavaScript — это то, что ключевое слово `this` указывает на функцию, в которой оно появляется. Ужасное заблуждение!
Ключевое слово `this` динамически привязывается основываясь на том как выполняется функция и выясняется, что есть четыре простых правила, чтобы понять и полностью определить привязку `this`.
Тесно связан с ключевым словом `this` механизм прототипов объектов, который является цепочкой поисков свойств, похожих на то, как обнаруживаются переменные в лексической области действия. Но при погружении в прототипы есть другой большой промах с JS: идея эмуляции (замены) классов и так называемое наследование через прототипы.
К сожалению, желание привнести мышление шаблоном проектирования классов и наследования в JavaScript — это просто наихудшая вещь, которую вы могли бы сделать, несмотря на то, что синтаксис может вводить вас в заблуждение, что есть что-то подобное классам, в действительности механизм прототипов фундаментально противоположен по своему поведению.

Что является предметом спора, так это то, лучше ли проигнорировать несоответствие и притвориться, что вы реализуете "наследование", либо все-таки, что является более подходящим, изучить и принять, то, как на самом деле работает система прототипов объектов. Последнее подходяще именуется "делегированием поведения."

Это больше, чем синтаксическое предпочтение. Делегирование — это совершенно другой и более мощный шаблон проектирования, который сам по себе заменяет необходимость проектировать классы и наследование. Но эти утверждения полностью противоречат почти каждым вторым постам в блогах, книгам и конференциям по этой теме на всем протяжении существования JavaScript.

Претензии, которые я предъявляю касаемо делегирования в противовес наследованию, идут не от нелюбви к языку и его синтаксису, а из желания видеть правильное применение истинной возможности языка и свести на нет бесконечные путаницу и недовольство.

Но объяснение необходимости рассматриваемых прототипов и делегирования гораздо более запутанно, чем то, которое я тут представил. Если вы готовы переосмыслить всё, что как вы думаете вы знаете о "классах" и "наследовании" в JavaScript, я даю вам шанс "принять красную таблетку" (*Матрица*, 1999) и проверить главы 4-6 книги *this & прототипы объектов* этой серии.

Типы и синтаксис

Третья книга в этой серии в первую очередь фокусируется на разборе еще одной в высшей степени спорной теме: приведении типов. Возможно, не существует темы, вызывающей большую досаду у разработчиков на JS, чем когда вы говорите о неразберихе, окружающей неявное приведение.

По большей части, общепринятый опыт — то, что неявное приведение это "плохая часть" языка и ее следует избегать любой ценой. По факту, некоторые ушли так далеко, что называют его "дефектом" в дизайне языка. Более того, есть утилиты, чья единственная работа не делать ничего кроме проверки вашего кода и сигнализирования, если вы делаете что-то, даже мало-мальски похожее на неявное приведение.

Но действительно ли приведение — такое сбивающее с толку, такое плохое, такое коварное, что ваш код обречен с самого начала, если вы будете использовать его?

Я отвечаю - нет. После постепенного построения понимания того, как на самом деле работают типы и значения, в главах 1-3, глава 4 берет на себя этот спор и полностью объясняет как работает приведение со всеми его нюансами. Мы увидим какие нюансы приведения действительно удивляют, а какие нюансы на самом деле имеют смысл, если дать время изучить их.

Но я не предполагаю, что приведение только правильное и изучаемое, я утверждаю, что приведение — невероятно полезный и полностью недооцениваемый инструмент, который *вам следует использовать в вашем коде*. Я говорю, что приведение, когда оно используется должным образом, не только работает, но и делает ваш код лучше. Все скептики и спорщики несомненно

поднимут на смех такую позицию, но я верю, что это одно из главных направлений, чтобы улучшить вашу игру в JS.

Хотите ли вы просто продолжать следовать тому, что говорит народ или вы желаете не принимать в расчет все допущения и взглянуть на приведение свежим взглядом? Книга *Типы и синтаксис* этой серии заставит вас думать.

Асинхронность и производительность

Первые три книги этой серии фокусируются на внутренней механике языка, а четвертая книга делает небольшое отступление, чтобы рассмотреть шаблоны, вдобавок к механике языка, для управления асинхронной разработкой. Асинхронность не только критически важна для быстродействия наших приложений, она всё больше становится *тем самым* критическим фактором в легкости написания и обслуживания программы.

Сперва книга начинается с прояснения большей части путаницы с терминологией и подходом вокруг таких вещей как "async", "parallel" и "concurrent" и объясняет в деталях как применять и как не применять такие вещи в JS.

Затем мы двинемся к изучению функций обратного вызова (callback) как к основному методу обеспечения асинхронности. Но тут мы быстро заметим, что сама по себе функция обратного вызова совершенно недостаточна для современных требований к асинхронной разработке. Мы определим два главных недостатка программирования только с помощью функций обратного вызова: потеря доверия к *инверсии управления* (IoC) и нехватка заурядной целесообразности.

Чтобы решить эти два недостатка, ES6 представляет два новых механизма(и даже шаблоны): обещания и генераторы.

Обещания — это независимые от времени обертки вокруг "будущего значения (future value)", которые позволяют вам рассуждать о них и составлять их независимо от того, готово ли уже это будущее значение или еще нет. Более того, они эффективно решают проблемы доверия к IoC маршрутизируя функции обратного вызова посредством доверительного и компонуемого механизма обещаний.

Генераторы представляют новый режим выполнения для функций в JS, в соответствии с которым генератор может быть приостановлен в точках с `yield` и продолжен асинхронно позже. Возможность "приостановка-и-продолжение" позволяет синхронному, последовательно выглядящему коду в генераторе быть обработанным асинхронно за кулисами. Делая так мы устраняем путаницу с нелинейными, нелокальными переходами в функции обратного вызова и таким образом делаем наш асинхронный код синхронно выглядящим, чтобы он был более осмысленным.

Именно эта комбинация обещаний и генераторов и "превращается" в наш самый эффективный шаблон асинхронного кодирования по настоящий момент в JavaScript. Вообще-то, многое из будущих усовершенствований в асинхронности будет в ES7 и позже будет обязательно построено на этой основе. Если серьезно относиться к тому, чтобы эффективно программировать в асинхронном мире, вам понадобится хорошенько освоиться с сочетанием обещаний и генераторов.

Хоть обещания и генераторы и являются почти что явными шаблонами, которые позволяют вашим программам работать более параллельно и таким образом обрабатывать больше за меньший период, в JS есть много других аспектов оптимизации производительности, стоящих изучения.

Глава 5 затрагивает темы, такие как параллелизм программ с помощью Web Workers и параллелизм данных с помощью SIMD, а также техника низкоуровневой оптимизации, как например ASM.js. Глава 6 знакомит с оптимизацией производительности с точки зрения правильных техник оценки производительности, включая то, о каких видах производительности стоит беспокоиться, а какие проигнорировать.

Эффективное программирование на JavaScript означает написание кода, который может разрушить барьеры ограничений будучи работающим динамически в широком диапазоне браузеров и других средах. Это потребует много сложного и детального планирования и усилий с нашей стороны, чтобы перевести программу с "она работает" на "она работает хорошо".

Книга *Асинхронность и производительность* спроектирована, чтобы дать вам все инструменты и навыки, которые вам понадобятся, чтобы писать адекватный и производительный JavaScript код.

ES6 и за его пределами

Не важно насколько вы чувствуете себя владеющим JavaScript к этому моменту, правда в том, что JavaScript никогда не прекратит эволюционировать и более того, скорость эволюции быстро растет. Этот факт — это почти что образное представление духа этой серии книг, чтобы проникнуться тем, что мы никогда полностью *не узнаем* каждую часть JS, поскольку как только вы овладеете всем, появятся новые вещи, опускающие границу того, что вам нужно будет изучить.

Эта книга посвящена как краткосрочным, так и среднесрочным перспективам того, в каком направлении идет язык, не только *известные* вещи, как ES6, но и *предполагаемые* вещи за его пределами.

Кроме того, что все книги этой серии включают в себя как составную часть состояние JavaScript на момент написания, которое на полпути к внедрению ES6, основной фокус в серии все-таки больше на ES5. Теперь, мы хотим обратить наше внимание на ES6, ES7 и ...

Поскольку ES6 почти готов на момент написания этих строк, *ES6 и за его пределами* начинается с того, что делится практическими вещами из пространства ES6 в нескольких ключевых категориях, включая новый синтаксис, новые структуры данных (коллекции) и новые возможности обработки и API. Мы рассмотрим каждую из этих новых возможностей ES6 на различных уровнях детализации, включая рассмотрение деталей, которые затрагиваются в других книгах серии.

Вот некоторые захватывающие вещи в ES6, про которые вы будете с нетерпением ждать, чтобы прочесть: деструктурирование (destructuring), значения параметров по умолчанию, символы, сокращенные методы (concise methods), вычисляемые свойства, стрелочные функции (arrow functions), блочная область действия, обещания (promises), генераторы, итераторы, модули, прокси, слабосвязанные коллекции ключ-значение (weakmaps) и многое, многое другое! Ну и ну, ES6 производит огромное впечатление!

Первая часть книги — это дорожная карта по всем вещам, которые вам необходимо изучить, чтобы подготовиться к новому и улучшенному JavaScript, на котором вы будете писать и который будете исследовать в течение следующей пары лет.

Последняя часть книги фокусируется на быстром взгляде на вещи, которые мы можем ожидать в ближайшем будущем в JavaScript. Самая важная мысль здесь в том, что будет после ES6, JS похоже будет эволюционировать компонент за компонентом, а не версия за версией, что означает, что мы можем ожидать увидеть эти вещи ближайшего будущего намного скорее, чем вы могли бы представить себе.

У JavaScript блестящее будущее. Разве сейчас не самое время, чтобы начать изучать его!?

Обзор

Серия *YDKJS* посвящена утверждению, что все разработчики JS могут и должны изучить все части этого великого языка. Ни одно мнение, никакая надежда на фреймворк и ни один дедлайн проекта не должны быть извинением за то, почему вы так и не изучили и не пришли к глубокому пониманию JavaScript.

Мы возьмем каждое направление в языке и посвятим ему краткую, но очень насыщенную книгу, чтобы полностью исследовать все его части, которые как вам казалось вы знаете, но возможно не полностью.

"Вы не знаете JS" — это не критический разбор или издевательство. Это осознание того, что все мы, включая меня, находим общий язык. Изучение JavaScript — это не конечная цель, а процесс. Мы не знаем JavaScript, пока что. Но мы узнаем!

Приложение A: Благодарности!

Есть много людей, которых я хотел бы поблагодарить за помощь в том, что эта книга и вся серия появились.

Во-первых, Я должен поблагодарить мою жену Кристен Симпсон (Christen Simpson) и двух моих детей: Итана (Ethan) и Эмили (Emily), за примирение с тем, что папа всегда "зависал" на компьютере. Даже когда не пишу книги, моя одержимость JavaScript приклеивает мои глаза к экрану гораздо больше, чем следует. То время,которое я одалживал у своей семьи — это и есть причина того, что эти книги могут столь глубоко полно разъяснить JavaScript вам, читатель. Я в долгу перед моей семьей во всем.

Я был хотел поблагодарить моих редакторов в O'Reilly, а именно Симона Сэн-Лорана (Simon St.Laurent) и Брайана МакДональда (Brian MacDonald), так же как и остальную команду редакторов и специалистов по маркетингу. Работать с ними — одно удовольствие, в том числе и в том, что особенно шли навстречу во время этого эксперимента в написании, редактировании и выпуске "open source" книги.

Благодарю многих людей, кто участвовал в улучшении этой серии книг предоставляя редакторские советы и исправления, включая Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin, Kris Kowal, Rick Waldron, Jordan Harband, Benjamin Gruenbaum, Vyacheslav Egorov, David Nolen и многих других. Большое спасибо Jenn Lukas за написание предисловия для этой книги.

Спасибо бессчетному количеству людей в сообществе, включая членов комитета TC39, которые поделились столькими знаниями со всеми нами и особенно спокойно относились к моим бесконечным вопросам и исследованиям с терпением и вниманием к деталям. John-David Dalton, Juriy "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David

Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott и многие другие, я даже не могу перечислить всех.

Поскольку серия книг "Вы не знаете JS" родилась на Kickstarter, я также хочу поблагодарить всех моих (почти) 500 щедрых спонсоров, без которых эта серия книг не появилась бы:

Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Tregoing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczi Dávid, Kitt Hodsden, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel לבב-בר Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuţ Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George

Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ♥♫★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aarson Holmes, Grzegorz Ziółkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen, Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Villoslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp

Kraeutli, Mihai Păun, Sam Gharegozlou, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

Эта серия книг выпускается в виде open source, включая редактирование и выпуск. У нас есть долг благодарности перед GitHub за предоставление такого рода возможности для сообщества!

Снова спасибо всем бесчисленным людям, которых я не перечислил поименно, но кого я несмотря на это должен поблагодарить. Пусть эта серия книг будет "принадлежать" всем нам и помогать делать вклад в увеличивающиеся осведомленность и понимание языка JavaScript, к выигрышу всех настоящих и будущих участников сообщества, вносящих свой вклад.