Майкл С. Миковски
Джош К. Пауэлл

Разработка

одностраничных веб-приложений

Предисловие
Грегори Д. Бенсона

MANNING

УДК 004.738.5:004.45JavaScript
ББК 32.973.202-018.2
М59

Майкл С. Миковски, Джош К. Пауэлл
М59 Разработка одностраничных веб-приложений / пер. с анг. Слинкина А. А. – М.: ДМК Пресс, 2014. – 512 с.: ил.

ISBN 978-5-97060-072-6

Если ваш сайт представляет собой набор дергающихся страниц, связанных ссылками, то вы отстали от жизни. Следующей ступенью вашей карьеры должны стать одностраничные приложения (SPA). В таком приложении отрисовка пользовательского интерфейса и бизнес-логика перенесены в браузер, а взаимодействие с сервером сводится к синхронизации данных. Пользователь работает с таким сайтом, как с персональным приложением на рабочем столе, что гораздо удобнее и приятнее. Однако разрабатывать, сопровождать и тестировать SPA нелегко.

В этой книге показано как организуется командная разработка передовых SPA —проектирование, тестирование, сопровождение и развитие — с применением JavaScript на всех уровнях и без привязки к какому-то конкретному каркасу.

Попутно вы отточите навыки работы с HTML5, CSS3 и JavaScript и узнаете об использовании JavaScript не только в браузере, но также на сервере и в базе данных.

УДК 004.738.5:004.45JavaScript
ББК 32.973.202-018.2

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-61729-075-6 (анг.)
ISBN 978-5-97060-072-6 (рус.)

© 2014 by Manning Publications Co.
© Оформление, перевод, ДМК Пресс, 2014

Содержание

Предисловие ... 13

Вступление ... 15

Благодарности ... 16

Об этой книге ... 19

Об иллюстрации на обложке ... 25

Часть I. Введение в SPA ... 26

Глава 1. Наше первое одностраничное приложение ... 28

1.1. Определение, немного истории и несколько слов о предмете книги ... 29
 1.1.1. Немного истории ... 29
 1.1.2. Почему SPA на JavaScript появились так поздно? ... 30
 1.1.3. Предмет книги ... 34
1.2. Создаем наше первое SPA ... 36
 1.2.1. Постановка задачи ... 36
 1.2.2. Общая структура файла ... 37
 1.2.3. Настройка инструментов разработчика в Chrome ... 38
 1.2.4. Разработка HTML и CSS ... 39
 1.2.5. Добавление JavaScript-кода ... 40
 1.2.6. Изучение приложения с помощью инструментов разработчика в Chrome ... 46
1.3. Чем хорошо написанное SPA удобно пользователям ... 49
1.4. Резюме ... 51

Глава 2. Новое знакомство с JavaScript ... 53

2.1. Область видимости переменной ... 55
2.2. Поднятие переменных ... 58
2.3. Еще о поднятии переменных и объекте контекста выполнения ... 60
 2.3.1. Поднятие ... 60
 2.3.2. Контекст выполнения и объект контекста выполнения ... 62
2.4. Цепочка областей видимости ... 66
2.5. Объекты в JavaScript и цепочка прототипов ... 69
 2.5.1. Цепочка прототипов ... 73

2.6. Функции – более пристальный взгляд 78
2.6.1. Функции и анонимные функции 78
2.6.2. Самовыполняющиеся анонимные функции 79
2.6.3. Паттерн модуля – привнесение в JavaScript закрытых переменных 82
2.6.4. Замыкания 88
2.7. Резюме 92

Часть II. Клиентская часть одностраничного приложения 94

Глава 3. Разработка оболочки 96
3.1. Знакомимся с Shell 96
3.2. Организация файлов и пространств имен 98
3.2.1. Создание дерева файлов 98
3.2.2. HTML-файл приложения 100
3.2.3. Создание корневого пространства имен CSS 101
3.2.4. Создание корневого пространства имен JavaScript 103
3.3. Создание функциональных контейнеров 104
3.3.1. Стратегия 105
3.3.2. HTML-код модуля Shell 105
3.3.3. CSS-стили модуля Shell 106
3.4. Отрисовка функциональных контейнеров 109
3.4.1. Преобразование HTML в JavaScript-код 110
3.4.2. Добавление HTML-шаблона в JavaScript-код 111
3.4.3. Создание таблицы стилей для Shell 113
3.4.4. Настройка приложения для использования Shell 115
3.5. Управление функциональными контейнерами 116
3.5.1. Метод сворачивания и раскрытия окна чата 117
3.5.2. Добавление обработчика события щелчка мышью по окну чата 119
3.6. Управление состоянием приложения 124
3.6.1. Какого поведения ожидает пользователь браузера? 124
3.6.2. Стратегия работы с элементами управления историей 125
3.6.3. Изменение якоря при возникновении события истории 126
3.6.4. Использование якоря для управления состоянием приложения 128
3.7. Резюме 135

Глава 4. Добавление функциональных модулей 137
4.1. Стратегия функциональных модулей 138
4.1.1. Сравнение со сторонними модулями 139
4.1.2. Функциональные модули и паттерн «фрактальный MVC» 141
4.2. Подготовка файлов функционального модуля 144
4.2.1. Планируем структуру каталогов и файлов 145
4.2.2. Создание файлов 146
4.2.3. Что мы соорудили 152
4.3. Проектирование API модуля 153
4.3.1. Паттерн якорного интерфейса 153
4.3.2. API конфигурирования модуля Chat 155
4.3.3. API инициализации модуля Chat 156
4.3.4. Метод setSliderPosition из API модуля Chat 157
4.3.5. Каскадное конфигурирование и инициализация 158
4.4. Реализация API функционального модуля 160
4.4.1. Таблицы стилей 160
4.4.2. Модификация модуля Chat 166
4.4.3. Модификация модуля Shell 172
4.4.4. Прослеживание выполнения 179
4.5. Добавление часто используемых методов 181
4.5.1. Метод removeSlider 181
4.5.2. Метод handleResize 183
4.6. Резюме 188

Глава 5. Построение модели 189
5.1. Что такое модель 189
5.1.1. Что мы собираемся сделать 190
5.1.2. Что делает модель 192
5.1.3. Чего модель не делает 193
5.2. Подготовка файлов модели, и не только 194
5.2.1. Планируем структуру каталогов и файлов 194
5.2.2. Создание файлов 196
5.2.3. Использование унифицированной библиотеки ввода 202
5.3. Проектирование объекта people 202
5.3.1. Проектирование объекта person 203
5.3.2. Проектирование API объекта people 205
5.3.3. Документирование API объекта people 209
5.4. Реализация объекта people 210
5.4.1. Создание подставного списка людей 212

5.4.2. Начало реализации объекта people ... 213
5.4.3. Завершение работы над объектом people ... 218
5.4.4. Тестирование API объекта people ... 225
5.5. Реализация аутентификации и завершения сеанса в Shell ... 228
5.5.1. Проектирование пользовательского интерфейса аутентификации ... 229
5.5.2. Модификация JavaScript-кода модуля Shell ... 229
5.5.4. Тестирование аутентификации и завершения сеанса в пользовательском интерфейсе ... 233
5.6. Резюме ... 234

Глава 6. Завершение модулей Model и Data ... 235
6.1. Проектирование объекта chat ... 235
6.1.1. Проектирование методов и событий ... 236
6.1.2. Документирование API объекта chat ... 239
6.2. Реализация объекта chat ... 240
6.2.1. Начинаем с метода join ... 240
6.2.2. Модификация модуля Fake для поддержки метода chat.join ... 243
6.2.3. Тестирование метода chat.join ... 246
6.2.4. Добавление средств работы с сообщениями в объект chat ... 247
6.2.5. Модификация модуля Fake для имитации работы с сообщениями ... 252
6.2.6. Тестирование работы с сообщениями в чате ... 254
6.3. Добавление поддержки аватаров в модель ... 256
6.3.1. Добавление поддержки аватаров в объект chat ... 256
6.3.2. Модификация модуля Fake для имитации аватаров ... 258
6.3.3. Тестирование поддержки аватаров ... 259
6.3.4. Разработка через тестирование ... 260
6.4. Завершение функционального модуля Chat ... 262
6.4.1. Модификация JavaScript-кода модуля Chat ... 263
6.4.2. Модификация таблиц стилей ... 271
6.4.3. Тестирование пользовательского интерфейса чата ... 276
6.5. Разработка функционального модуля Avatar ... 277
6.5.1. JavaScript-код модуля Avatar ... 278
6.5.2. Создание таблицы стилей для модуля Avatar ... 284
6.5.3. Модификация модуля Shell и головного HTML-документа ... 285
6.5.4. Тестирование функционального модуля Avatar ... 286

6.6. Привязка к данным и jQuery 287
6.7. Разработка модуля Data 288
6.8. Резюме 291

Часть III. Сервер SPA 292

Глава 7. Веб-сервер 294
7.1. Роль сервера 294
7.1.1. Аутентификация и авторизация 294
7.1.2. Валидация 295
7.1.3. Сохранение и синхронизация данных 296
7.2. Node.js 297
7.2.1. Почему именно Node.js? 297
7.2.2. Приложение «Hello World» для Node.js 298
7.2.3. Установка и использование Connect 302
7.2.4. Добавление промежуточного уровня Connect 304
7.2.5. Установка и использование Express 305
7.2.6. Добавление промежуточного уровня в Express-приложение 308
7.2.7. Окружения в Express 309
7.2.8. Обслуживание статических файлов с помощью Express 310
7.3. Более сложная маршрутизация 312
7.3.1. CRUD-маршруты для управления пользователями 312
7.3.2. Обобщенная маршрутизация для операций CRUD 319
7.3.3. Перенос маршрутизации в отдельный модуль Node.js 322
7.4. Добавление аутентификации и авторизации 327
7.4.1. Базовая аутентификация 327
7.5. Веб-сокеты и Socket.IO 329
7.5.1. Простой пример применения Socket.IO 329
7.5.2. Socket.IO и сервер обмена сообщениями 333
7.5.3. Обновление JavaScript-кода с помощью Socket.IO 334
7.6. Резюме 338

Глава 8. Серверная база данных 339
8.1. Роль базы данных 339
8.1.1. Выбор хранилища данных 340
8.1.2. Исключение преобразования данных 340
8.1.3. Помещайте логику туда, где она нужнее 341
8.2. Введение в MongoDB 342
8.2.1. Документоориентированное хранилище 343

8.2.2. Динамическая структура документа 343
8.2.3. Начало работы с MongoDB 345
8.3. Драйвер MongoDB 347
8.3.1. Подготовка файлов проекта 347
8.3.2. Установка и подключение MongoDB 348
8.3.3. Использование методов CRUD в MongoDB 350
8.3.4. Добавление операций CRUD в серверное приложение 353
8.4. Валидация данных, поступивших от клиента 357
8.4.1. Проверка типа объекта 357
8.4.2. Проверка объекта 360
8.5. Создание отдельного модуля CRUD 368
8.5.1. Подготовка структуры файлов 369
8.5.2. Перенос операций CRUD в отдельный модуль 372
8.6. Реализация модуля chat 378
8.6.1. Начало модуля Chat 379
8.6.2. Создание обработчика сообщения adduser 382
8.6.3. Создание обработчика сообщения updatechat 386
8.6.4. Создание обработчиков отключения 388
8.6.5. Создание обработчика сообщения updateavatar 390
8.7. Резюме 393

Глава 9. Подготовка SPA к промышленной эксплуатации 395

9.1. Поисковая оптимизация SPA 396
9.1.1. Как Google индексирует SPA 396
9.2. Облачные и сторонние службы 400
9.2.1. Анализ работы сайта 400
9.2.2. Протоколирование ошибок на стороне клиента 403
9.2.3. Сети доставки содержимого 406
9.3. Кэширование и отключение кэширования 406
9.3.1. Варианты кэширования 407
9.3.2. Веб-хранилище 408
9.3.3. HTTP-кэширование 410
9.3.4. Кэширование на сервере 414
9.3.5. Кэширование запросов к базе данных 420
9.4. Резюме 421

Приложение А. Стандарт кодирования на JavaScript 424

А.1. Зачем нам стандарт кодирования? 424

А.2. Форматирование кода и комментарии ... 425
А.2.1. Форматирование кода с учетом удобства чтения ... 426
А.2.2. Комментарии как средство пояснения и документирования ... 434
А.3. Именование переменных ... 437
А.3.1. Сокращение и повышение качества комментариев за счет соглашений об именовании ... 437
А.3.2. Рекомендации по именованию ... 439
А.3.3. Практическое применение рекомендаций ... 447
А.4. Объявление и присваивание переменным ... 448
А.5. Функции ... 450
А.6. Пространства имен ... 453
А.7. Имена и структура дерева файлов ... 454
А.8. Синтаксис ... 455
А.8.1. Метки ... 456
А.8.2. Предложения ... 456
А.8.3. Прочие замечания о синтаксисе ... 459
А.9. Валидация кода ... 460
А.9.1. Установка JSLint ... 460
А.9.2. Настройка JSLint ... 461
А.9.3. Использование JSLint ... 462
А.10. Шаблон модуля ... 463
А.11. Резюме ... 465

Приложение Б. Тестирование SPA ... 467

Б.1. Режимы тестирования ... 468
Б.2. Выбор каркаса тестирования ... 472
Б.3. Настройка nodeunit ... 473
Б.4. Создание комплекта тестов ... 474
Б.4.1. Инструктируем Node.js загрузить наши модули ... 475
Б.4.2. Подготовка одного теста в nodeunit ... 478
Б.4.3. Создание первого настоящего теста ... 479
Б.4.4. План событий и тестов ... 480
Б.4.5. Создание комплекта тестов ... 483
Б.5. Адаптация модулей SPA для тестирования ... 496
Б.6. Резюме ... 499

Предметный указатель ... 501

Глава 2

Новое знакомство с JavaScript

В этой главе:
- ✧ Область видимости переменных, поднятие переменных и объект контекста выполнения.
- ✧ Что такое цепочка областей видимости переменной и как ей можно воспользоваться.
- ✧ Создание объектов в JavaScript с помощью прототипов.
- ✧ Самовыполняющиеся анонимные функции.
- ✧ Модули и закрытые переменные.
- ✧ Замыкания – сочетание приятного с полезным.

В этой главе мы дадим обзор уникальных особенностей JavaScript, которые необходимо понимать для написания сколько-нибудь серьезного естественного одностраничного приложения на JavaScript. В листинге 2.1 приведен фрагмент кода, разработанного в главе 1, иллюстрирующий рассматриваемые концепции. Если вы ясно понимаете, *как* и *почему* они используются, то можете вообще пропустить эту главу или, просмотрев ее по диагонали, перейти к главе 3, где мы начнем работать над SPA.

Прорабатывая материал дома, вы можете копировать код из листингов, приведенных в этой главе, на консоль, входящую в состав инструментов разработчика в Chrome, и, нажав **Enter**, смотреть, как он выполняетя. Мы настоятельно рекомендуем не пренебрегать подобной возможностью.

Листинг 2.1 ❖ JavaScript-код приложения

```
...
  var spa = (function ( $ ) {
    // Переменные в области видимости модуля
    var
      configMap = {
        extended_height  : 434,
        extended_title   : 'Click to retract',
        retracted_height : 16,
        retracted_title  : 'Click to extend',
        template_html    : '<div class="spa-slider"></div>'
```

← Самовыполняющиеся анонимные функции, паттерн модуля.

← Прототипическое наследование, понятие переменных, область видимости переменной.

```
      },
      $chatSlider,
      toggleSlider, onClickSlider, initModule;
...

    // Открытый метод
    initModule = function ( $container ) {   ◄── Анонимные функции, паттерн модуля, замыкания.

      $container.html( configMap.template_html );
      $chatSlider = $container.find( '.spa-slider' );

      $chatSlider
        .attr( 'title', configMap.retracted_title )
        .click( onClickSlider );

      return true;
    };

    return { initModule : initModule };   ◄── Паттерн модуля, цепочка контекстов.

  }( jQuery ));   ◄── Самовыполняющиеся анонимные функции.

...
```

Стандарты кодирования и синтаксис JavaScript

Новичкам синтаксис JavaScript может показаться странным. Прежде чем идти дальше, важно понять, что такое блоки объявления переменных и литеральные объекты. Если вы уже с ними знакомы, можете пропустить эту врезку. Полный перечень важных, с нашей точки зрения, особенностей синтаксиса JavaScript и стандартов хорошего кодирования см. в приложении А.

Блоки объявления переменных

```
var spa = "Hello world!";
```

Переменные в JavaScript объявляются после ключевого слова `var`. Переменная может содержать данные любого типа: массивы, целые, с плавающей точкой и т. д. Тип переменной не указывается, поэтому JavaScript считается *слаботипизированным* языком. Даже после присваивания переменной значения ее тип может быть изменен в результате последующего присваивания значения другого типа, поэтому язык также считается *динамическим*.

В JavaScript объявления переменных и операторы присваивания можно сцеплять, поместив их после ключевого слова `var` и разделив запятыми:

```
var book, shopping_cart,
    spa = "Hello world!",
    purchase_book = true,
    tell_friends = true,
    give_5_star_rating_on_amazon = true,
    leave_mean_comment = false;
```

Существует несколько мнений о том, какой формат блока объявления переменных считать лучшим. Мы предпочитаем размещать в начале объявления неинициализированных переменных, а вслед за ними – объявления вместе с определениями. Мы также предпочитаем помещать запятые в конце строк, как показано выше, но не настаиваем на этом с пеной у рта, тем более что интерпретатору JavaScript это безразлично.

Литеральные объекты

Литеральный объект – это объект, определенный в виде списка атрибутов, перечисленных через запятую и заключенных в фигурные скобки. Значение атрибута отделяется от имени двоеточием, а не знаком равенства. Литеральные объекты могут также содержать массивы, представленные в виде списка элементов, перечисленных через запятую и заключенных в квадратные скобки. Для определения методов в качестве значения одного из атрибутов задается функция.

```
var spa = {
  title: "Single Page Web Applications",//атрибут
  authors: [ "Mike Mikowski", "Josh Powell" ], //массив
  buy_now: function () { //функция
    console.log( "Book is purchased" );
  }
}
```

Литеральные объекты и блоки объявления переменных используются в этой книге сплошь и рядом.

2.1. Область видимости переменной

Начать обсуждение удобно с вопроса о поведении переменных и о том, когда переменная находится или не находится в области видимости.

Областью видимости переменной в JavaScript является функция. Переменные могут быть глобальными или локальными. *Глобальная* переменная видна в любой точке программы, *локальная* – только там, где объявлена. Повторим еще раз: единственным блоком, определяющим область видимости переменной, в JavaScript является функция. Глобальные переменные определены вне любой функции, локальные – внутри функции. Просто, не правда ли?

По-другому эту ситуацию можно описать, уподобив функцию тюрьме, а определенные в ней переменные – заключенным. Как тюрьма является местом содержания заключенных и не позволяет им выходить за территорию, так и функция содержит переменные и не выпускает их наружу. Это иллюстрируется в следующем фрагменте кода.

```
var regular_joe = 'Я глобальная!';

function prison() {
```

```
  var prisoner = 'Я локальная!';
}

prison();
console.log( regular_joe ); ◄——— Выводится «Я глобальная!».
console.log( prisoner );    ◄——— Выводится «Error: prisoner is not defined».
```

JavaScript 1.7, 1.8, 1.9+ и область видимости блока

В версии JavaScript 1.7 появился новый конструктор переменной с областью видимости блока – предложение `let`. К сожалению, хотя для версий JavaScript 1.7, 1.8, 1.9 существуют стандарты, даже версия 1.7 поддержана в браузерах несогласованно. И до тех пор, пока браузеры не станут совместимыми в части обновления JavaScript, мы будем считать, что версии JavaScript, начиная с 1.7, не существуют. Тем не менее посмотрим, как это должно работать:

```
let (prisoner = 'Я в тюрьме!') {
  console.log( prisoner ); ◄——— Выводится «Я в тюрьме!».
}
console.log( prisoner );   ◄——— Выводится «Error: prisoner isn't defined».
```

Чтобы использовать JavaScript 1.7, укажите номер версии в атрибуте `type` тега `script`:

```
<script type="application/javascript;version=1.7">
```

Это лишь мимолетное знакомство с JavaScript 1.7+; существует еще много изменений и дополнительных возможностей.

Ах, если бы все было так просто. Первое, на что натыкаешься, изучая правила видимости в JavaScript, – возможность объявить глобальную переменную внутри функции, просто опустив объявление `var`, как показано на рис. 2.1. А, как в любом языке программирования, глобальные переменные – почти всегда Плохая Идея.

```
function prison () {
  prisoner_1 = 'Я сбежала!';
  var prisoner_2 = 'Я заперта!';
}

prison();
console.log( prisoner_1 ); ◄——— Выводится «Я сбежала!».

console.log( prisoner_2 ); ◄——— Выводится «Error: prisoner_2 is not defined».
```

Ничего хорошего в этом нет – заключенные не должны сбегать. Еще одно место, где часто встречается этот подводный камень, – объявление счетчика цикла `for` без ключевого слова `var`. Рассмотрим следующие два определения функции `prison`:

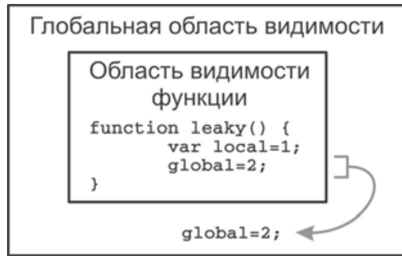


Рис. 2.1 ❖ Забыв поставить ключевое слово `var` при объявлении локальной переменной, вы создадите глобальную переменную

```
// неправильно
function prison () {
  for( i = 0; i < 10; i++ ) {
    //...
  }
}
prison();
console.log( i ); // i равно 10
delete window.i;

// допустимо
function prison () {
  for( var i = 0; i < 10; i++ ) {
    //...
  }
}
prison();
console.log( i ); // i не определена

// наилучшее решение
function prison () {
  var i;
  for ( i = 0; i < 10; i++ ) {
    // ...
  }
}
prison();
console.log( i ); // i не определена
```

Последняя версия нравится нам больше, потому что, объявляя переменную в начале функции, мы сразу понимаем, какова ее область видимости. Видя объявление переменной в инициализаторе цикла `for`, человек может ошибочно подумать, что областью ее видимости

является только сам цикл, как в некоторых других языках программирования.

Следуя этой логике, мы рекомендуем помещать все объявления и большую часть присваиваний в начало той функции, где они находятся, чтобы область видимости переменной не вызывала сомнений:

```
function prison() {
  var prisoner = 'Я локальная!',
      warden   = 'Я тоже локальная!',
      guards   = 'И я локальная!'
  ;
}
```

Перечисляя объявления локальных переменных через запятую в одном предложении `var`, мы делаем их хорошо видными и, что важнее, уменьшаем шанс по ошибке создать глобальную переменную вместо локальной. Также обратите внимание на аккуратное выравнивание строк и на то, что точка с запятой в конце воспринимается как естественное завершение блока объявления переменных. Об этом и других способах форматирования JavaScript-кода с целью повысить удобочитаемость и понятность мы будем говорить в приложении А «Стандарты кодирования на JavaScript». С методом объявления локальных переменных связана еще одна особенность JavaScript – поднятие переменных (variable hoisting). Рассмотрим ее.

```
function hoisted() {                    function hoisted() {
      console.log(v);                         var v;
      var v=1;                                console.log(v);
}                                             v=1;
                                        }
```

Рис. 2.2 ❖ Объявления переменных в JavaScript «поднимаются» в начало объемлющей функции, но инициализация производится там, где переменная встретилась. Интерпретатор JavaScript на самом деле не переписывает код, объявление перемещается при каждом вызове функции

2.2. Поднятие переменных

Любое объявление переменной в JavaScript *поднимается* в начало области видимости объемлющей функции, при этом переменной присваивается значение `undefined`. Получается, что переменная, объявленная в любом месте функции, существует во всем коде этой функции, но остается неопределенной, пока ей не будет присвоено значение (рис. 2.2).

```
function prison () {
  console.log(prisoner);   ← Выводится «prisoner is undefined».
    var prisoner = 'Теперь я определена!';

  console.log(prisoner);   ← Выводится «Теперь я определена!».
}
prison();
```

Сравните код на этом рисунке с попыткой обратиться к переменной, не объявленной ни локально, ни глобально. Это приведет к ошибке времени выполнения, и интерпретатор JavaScript остановится на соответствующем предложении:

```
function prison () {

  console.log(prisoner);   ← Выводится «Error: prisoner is not defined», и интерпретатор JavaScript прекращает выполнение программы.

}
prison();
```

Поскольку объявления переменных все равно поднимаются в начало области видимости функции, мы рекомендуем объявлять все переменные в начале функции, предпочтительно в одном предложении `var`. Это согласуется с поведением JavaScript и помогает избежать путаницы, проиллюстрированной на рисунке выше.

```
function prison () {
  console.log(prisoner);   ← Выводится «undefined».
  var prisoner, warden, guards;

  console.log(prisoner);   ← Выводится «undefined».
  prisoner = 'prisoner присвоено значение';

  console.log(prisoner);   ← Выводится «prisoner присвоено значение».
}
prison();
```

В сочетании области видимости и механизм поднятия переменных иногда приводят к неожиданному поведению. Взгляните на следующий код:

```
var regular_joe = 'Regular Joe';   ← regular_joe определена в глобальной области видимости.
function prison () {
  console.log(regular_joe);   ← Глобальная переменная regular_joe печатается внутри функции prison, выводится 'Regular Joe'.
}
prison();
```

Когда внутри функции `prison` значение переменной `regular_joe` передается функции `console.log()`, интерпретатор JavaScript сначала

проверяет, объявлена ли `regular_joe` в локальной области видимости. Поскольку это не так, интерпретатор далее просматривает глобальную область видимости, обнаруживает, что там переменная определена, и возвращает ее значение. Это называется *проходом по цепочке областей видимости*. Но что, если переменная объявлена также в локальной области видимости?

```
var regular_joe = 'regular_joe присвоено значение';
function prison () {
  console.log(regular_joe);
  var regular_joe;
}
prison();
```

Выводится «undefined». Объявление `regular_joe` поднимается в начало функции, и это поднятое объявление проверяется еще до поиска `regular_joe` в глобальной области видимости.

Противоречит интуиции? Странно? Что ж, пройдем вслед за JavaScript по всему пути поднятия объявлений.

2.3. Еще о поднятии переменных и объекте контекста выполнения

Считается, что любой разработчик на JavaScript, если хочет быть успешным, обязан понимать рассмотренные до сих пор концепции. Но сделаем еще один шаг и заглянем под капот: вы станете одним из немногих, кто понимает, как на самом деле работает JavaScript. Начнем с одной из самых «магических» особенностей JavaScript: поднятия переменных и функций.

2.3.1. Поднятие

Как всегда бывает с магией, фокус вызывает чуть ли не разочарование, когда его секрет раскрыт. А секрет в том, что интерпретатор JavaScript, входя в область видимости, делает два прохода по коду. На первом проходе инициализируются переменные, а на втором выполняется код. Просто, я знаю – и не понимаю, почему это обычно объясняют другими словами. Но посмотрим внимательнее, что интерпретатор делает на первом проходе, потому что это имеет некоторые любопытные последствия.

На первом проходе интерпретатор JavaScript просматривает код и делает три вещи:

1. Объявляет и инициализирует аргументы функций.
2. Объявляет локальные переменные, в том числе анонимные функции, присвоенные локальной переменной, но не инициализирует их.

3. Объявляет и инициализирует функции.

Листинг 2.2 ❖ Первый проход

```
function myFunction( arg1, arg2 ) {      ← 1 Объявляет и инициализирует аргументы функций.
  var local_var = 'foo',                 ← 2 Объявляет локальные переменные, в том числе анонимные функции, присвоенные локальной переменной, но не инициализирует их.
    a_function = function () {
      console.log( 'a function' );
    };

  function inner () {                    ← 3 Объявляет и инициализирует функции.
    console.log('inner');
  }
}
myFunction( 1,2 );
```

На первом проходе локальным переменным *не* присваиваются значения, потому что значение может вычисляться в коде, а код на первом проходе не исполняется. Аргументам же значения присваиваются, потому что код, в котором они вычисляются, уже выполнен до передачи аргумента функции.

Убедиться в том, что значения аргументов присваиваются на первом проходе, можно, проведя сравнение с кодом из предыдущего раздела, где демонстрировалось поднятие переменных.

Листинг 2.3 ❖ Переменные не определены до момента объявления

```
var regular_joe = 'regular_joe присвоено значение';
function prison () {
  console.log(regular_joe);   ← Выводится «undefined». Объявление regular_joe поднимается в начало функции, и это поднятое объявление проверяется еще до поиска regular_joe в глобальной области видимости.
  var regular_joe;
}
prison();
```

Переменная `regular_joe` равна `undefined`, потому что она объявлена в функции `prison`, но если `regular_joe` также передается в качестве аргумента, то еще до объявления она имеет значение.

Листинг 2.4 ❖ Переменные имеют значение до момента объявления

```
var regular_joe = 'regular_joe присвоено значение';
function prison ( regular_joe ) {
  console.log(regular_joe);   ← Выводится «аргумент regular_joe». Аргументам присваиваются значения на первом проходе. Если не знать о двух проходах интерпретатора JavaScript, то кажется, что аргумент regular_joe должен быть перезаписан в результате поднятия объявления локальной переменной regular_joe.
  var regular_joe;

  console.log(regular_joe);   ← Выводится «аргумент regular_joe». Сюрприз! Поскольку regular_joe было присвоено значение как аргументу, то при объявлении локальной переменной с тем же именем оно не перезаписывается. Это объявление излишне.
}
prison( 'аргумент regular_joe');
```

Если у вас от всего этого голова идет кругом, то это нормально. Мы объяснили, что интерпретатор JavaScript делает два прохода по

исполняемой функции и что на первом проходе он сохраняет переменные, но мы еще не видели, *как* переменные хранятся. Знание о том, как интерпретатор хранит переменные, наверное, устранит все оставшиеся недоразумения. Переменные хранятся как атрибуты так называемого *объекта контекста выполнения*.

2.3.2. Контекст выполнения и объект контекста выполнения

С каждым вызовом функции связывается новый контекст выполнения. Контекст выполнения – это концепция, а не объект, точнее концепция выполняемой функции. Можно провести аналогию с атлетом в беговом или прыжковом контексте. Мы могли бы сказать «бегущий атлет», а не «атлет в беговом контексте», и точно так же можно было бы говорить о выполняемой функции, но технический жаргон устроен иначе, и мы говорим *контекст выполнения*.

Контекст выполнения знает обо всем, что происходит во время выполнения функции. Он отличается от объявления функции, потому что в объявлении говорится о том, что *будет* происходить во время выполнения функции, а контекст – это и *есть* ее выполнение.

Все переменные и функции, определенные внутри функции, считаются частью контекста ее выполнения. Контекст выполнения – это часть того, что имеют в виду программисты, говоря об *области видимости* функции (scope). Говорят, что переменная «находится в области видимости», если она доступна в текущем контексте выполнения. По-другому то же самое можно выразить, сказав, что переменная доступна, когда функция выполняется.

Переменные и функции, являющиеся частью контекста выполнения, хранятся в *объекте контекста выполнения* – реализации контекста выполнения в стандарте ECMA. Объект контекста выполнения – это объект внутри интерпретатора JavaScript, а не переменная, доступная непосредственно написанной на JavaScript программе. Но к ней несложно получить косвенный доступ, потому что всякий раз, используя какую-то переменную, вы обращаетесь к атрибуту объекта контекста выполнения.

Выше мы сказали, что интерпретатор JavaScript делает два прохода по контексту выполнения – для объявления и инициализации переменных. Но где эти переменные хранятся? А вот как раз в объекте контекста выполнения – в виде его атрибутов. Пример хранения переменных приведен в табл. 2.1.

Таблица 2.1. Объект контекста выполнения

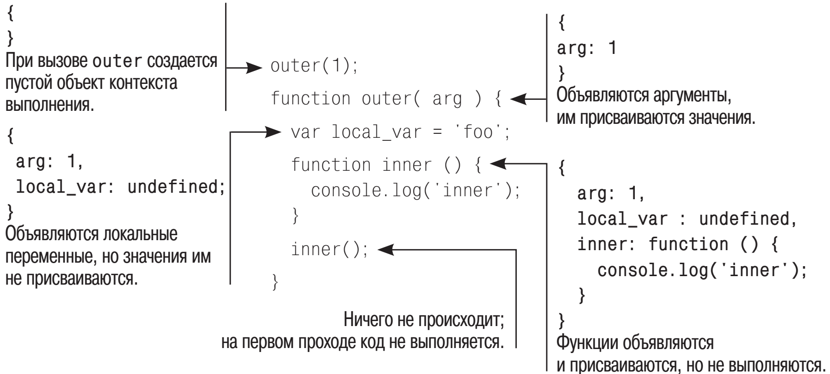
Код	Объект контекста выполнения

`var example_variable = "example",`
`    another_example = "another";`
	`{`
	`  example_variable: "example",`
	`  another_example: "another"`
	`};`

Возможно, вы никогда не слышали об объекте контекста выполнения. Эту тему нечасто обсуждают в сообществе веб-разработчиков, наверное, потому что объект контекста выполнения скрыт глубоко в недрах реализации JavaScript и напрямую программисту недоступен.

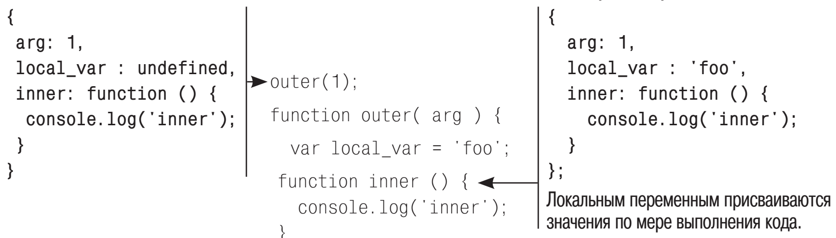
Без понимания того, что такое объект контекста выполнения, не понять материала этой главы, поэтому пройдем по всему жизненному циклу этого объекта и создающего его JavaScript-кода.

Листинг 2.5 ❖ Объект контекста выполнения – первый проход



Теперь, когда аргументы и функции объявлены и инициализированы, а локальные переменные только объявлены, начинается второй проход, на котором выполняется JavaScript-код, а локальным переменным присваиваются значения.

Листинг 2.6 ❖ Объект контекста выполнения – второй проход



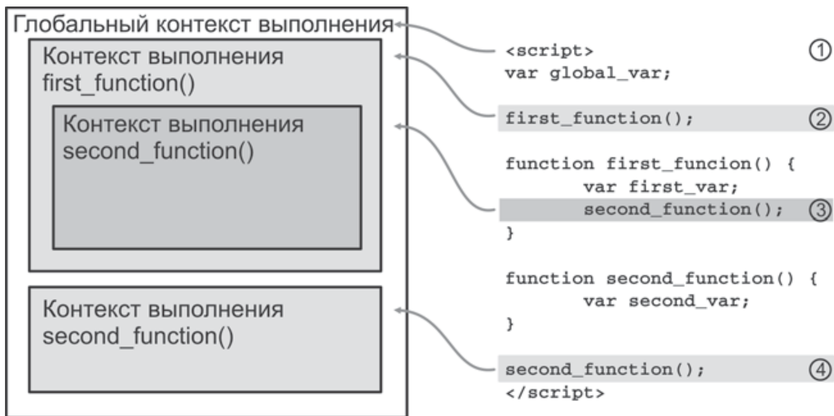


Рис. 2.3 ❖ Вызов функции создает контекст выполнения

```
inner();  <--  {
}                arg: 1,
                 local_var : 'foo',
                 inner: function () {
                   console.log('inner');
                 }
               }
```

Атрибуты, представляющие переменные в этом объекте контекста выполнения, остаются такими же, но при вызове функции `inner` внутри него создается новый объект контекста выполнения.

Количество уровней вложенности может быть гораздо больше, так как каждый вызов функции внутри контекста выполнения создает новый контекст выполнения, вложенный в текущий. Что, опять голова кругом? Тогда пора нарисовать картинку. См. рис. 2.3.

1. Все, что находится внутри тега `<script>`, образует глобальный контекст выполнения.
2. Вызов функции `first_function` создает новый контекст выполнения внутри глобального. Во время выполнения `first_function` имеет доступ к переменным в том контексте выполнения, в котором была вызвана. В данном случае `first_function` имеет доступ к переменным, определенным в глобальном контексте выполнения, и к локальным переменным, определенным внутри `first_function`. Говорят, что эти переменные находятся *в области видимости*.
3. При вызове функции `second_function` создается новый контекст выполнения внутри контекста выполнения `first_function`.

Функция `second_function` имеет доступ к переменным из контекста выполнения `first_function`, потому что она в нем и вызвана. Но `second_function` также имеет доступ к переменным из глобального контекста выполнения и к локальным переменным, определенным внутри `second_function`. Говорят, что эти переменные находятся *в области видимости*.

4. Функция `second_function` вызывается снова, на этот раз в глобальном контексте выполнения. Теперь она не имеет доступа к переменным в контексте выполнения `first_function`, потому что вызывалась не в этом контексте. Проще говоря, при втором вызове `second_function` не имеет доступа к переменным, определенным внутри `first_function`, потому что не вызывалась из `first_function`.

Этот контекст выполнения `second_function` также не имеет доступа к переменным, созданным при предыдущем вызове `second_function`, потому что эти вызовы произведены в разных контекстах выполнения. Иначе говоря, при вызове функции мы не имеем доступа к локальным переменным, созданным во время ее предыдущего вызова. Говорят, что эти недоступные переменные находятся *вне области видимости*.

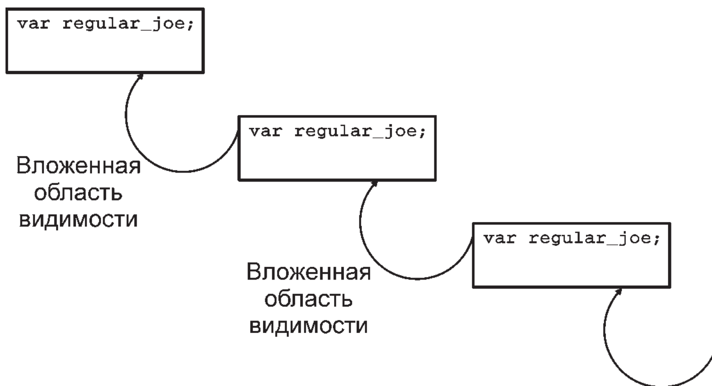


Рис. 2.4 ❖ Во время выполнения интерпретатор JavaScript просматривает иерархию областей видимости, пытаясь разрешить имена переменных

Порядок, в котором интерпретатор JavaScript просматривает объекты контекста выполнения при доступе к переменным, находящимся «в области видимости», называется *цепочкой областей видимости*.

Вместе с *цепочкой прототипов* эта цепочка определяет порядок, в котором JavaScript находит переменные и их атрибуты. Эти концепции мы обсудим в следующих разделах.

2.4. Цепочка областей видимости

До сих пор при обсуждении областей видимости переменных мы ограничивались *глобальными* и *локальными* переменными. Для начала это неплохо, но понятие области видимости не такое плоское, как следует из обсуждения вложенных контекстов выполнения в предыдущем разделе. Область видимости переменной правильнее представлять себе как цепочку, показанную на рис. 2.4. При поиске определения переменной интерпретатор JavaScript сначала просматривает локальный объект контекста выполнения. Если в нем определение не найдено, то интерпретатор переходит вверх по цепочке областей видимости к контексту выполнения, в котором был создан текущий контекст, и ищет определение переменной там. Так происходит до тех пор, пока определение не будет найдено или не будет достигнута глобальная область видимости.

Модифицируем предыдущий пример, чтобы проиллюстрировать концепцию цепочки областей видимости. Код в листинге 2.7 напечатает такие сообщения:

```
Я здесь, чтобы спасти положение!
regular_joe присвоено значение
undefined
```

Листинг 2.7 ❖ Пример цепочки областей видимости – переменная regular_joe определена в каждой области видимости

```
var regular_joe = 'Я здесь, чтобы спасти положение!';   ◄── regular_joe присвоено значение
                                                             в глобальном контексте.
// печатается 'Я здесь, чтобы спасти положение!'
console.log(regular_joe);   ◄── Вызывающая область видимости: глобальная. Ближайшее соответствие в цепочке областей видимости: глобальная переменная regular_joe.
function supermax(){
  var regular_joe = 'regular_joe присвоено значение';

  // печатается  'regular_joe присвоено значение'
  console.log(regular_joe);   ◄── Вызывающая область видимости: глобальная -> supermax(). Ближайшее соответствие в цепочке областей видимости: переменная regular_joe, определенная внутри supermax().

  function prison () {
    var regular_joe;
    console.log(regular_joe);   ◄── Вызывающая область видимости: глобальная -> supermax() -> prison(). Ближайшее соответствие в цепочке областей видимости: переменная regular_joe, определенная внутри prison().
  }

  // печатается 'undefined'
```

```
  prison();
}
supermax();
```

Во время выполнения интерпретатор JavaScript просматривает цепочку областей видимости, пытаясь разрешить имена переменных. Просмотр начинается с текущей области видимости, а затем продолжается вверх по цепочке до области видимости верхнего уровня, каковой является объект `window` (в браузерах) или `global` (в Node.js). Как только будет найдено первое соответствие, поиск прекращается. Это означает, что переменные в более глубоко вложенных областях видимости могут скрывать те, что определены в областях, более близких к глобальной, так как обнаруживаются раньше. Хорошо это или плохо, зависит от того, ожидаете вы такого поведения или нет. В реальной программе следует стремиться к тому, чтобы имена переменных были по возможности уникальны; рассматриваемый код, в котором одно и то же имя вводится в трех вложенных областях видимости, вряд ли можно считать удачным, он приведен лишь для иллюстрации идеи.

В листинге выше значение переменной `regular_joe` запрашивается в трех областях видимости.

1. В последней строке главной программы вызов `console.log (regular_joe)` производится в глобальной области видимости. Интерпретатор JavaScript начинает с поиска свойства `regular_joe` в объекте глобального контекста выполнения. Такое свойство там есть, оно имеет значение `Я здесь, чтобы спасти положение!`, которое и используется.
2. В последней строке функции `supermax` также находится вызов `console.log(regular_joe)`. Он производится в контексте выполнения `supermax`. Интерпретатор JavaScript начинает с поиска свойства `regular_joe` в объекте контекста выполнения `supermax`. Такое свойство там есть, оно имеет значение `regular_joe присвоено значение`, которое и используется.
3. И в последней строке функции `prison` мы также видим вызов `console.log(regular_joe)`. Он производится в контексте выполнения `prison`. Интерпретатор JavaScript начинает с поиска свойства `regular_joe` в объекте контекста выполнения `prison`. Такое свойство там есть, оно имеет значение `undefined`, которое и используется.

В данном примере переменная `regular_joe` определена во всех трех областях видимости. В следующей версии (листинг 2.8) мы определим

эту переменную только в глобальной области видимости. Теперь программа три раза печатает строку «Я здесь, чтобы спасти положение!».

Листинг 2.8 ❖ Пример цепочки областей видимости – переменная regular_joe определена только в одной области видимости

```
var regular_joe = 'I am here to save the day!';
// печатается 'Я здесь, чтобы спасти положение!'
console.log(regular_joe);
function supermax(){

  // печатается 'Я здесь, чтобы спасти положение!'
  console.log(regular_joe);

  function prison () {
    console.log(regular_joe);
  }

  // печатается 'Я здесь, чтобы спасти положение!'
  prison();
}
// печатается 'Я здесь, чтобы спасти положение!'. Дважды.
supermax();
```

Пояснения к листингу:

- `var regular_joe = ...` — `regular_joe` присвоено значение в глобальном контексте.
- `console.log(regular_joe);` (глобальный) — Вызывающая область видимости: глобальная. В ней переменная и будет найдена.
- `console.log(regular_joe);` (в `supermax`) — Вызывающая область видимости: глобальная -> `supermax()`. Ближайшее соответствие в цепочке областей видимости: глобальная переменная `regular_joe`.
- `console.log(regular_joe);` (в `prison`) — Вызывающая область видимости: глобальная -> `supermax()` -> `prison()`. Ближайшее соответствие в цепочке областей видимости: глобальная переменная `regular_joe`.

Важно помнить, что запрошенная переменная может быть найдена в любом месте цепочки областей видимости. Контроль над тем, откуда берутся значения, лежит на нас, и если мы не будем этого понимать, то в коде воцарится мучительный хаос. Стандарты кодирования на JavaScript, приведенные в приложении А, рекомендуют ряд способов справиться с этой проблемой, и мы будем ими пользоваться по ходу дела.

Глобальные переменные и объект window

То, что мы привыкли называть *глобальными* переменными, – на самом деле свойства объекта верхнего уровня, представляющего среду выполнения. В браузере таковым является объект `window`; в Node.js – объект `global`, и область видимости переменных работает по-разному.

Объект `window` содержит много свойств, которые сами по себе содержат объекты, методы (`onload`, `onresize`, `alert`, `close`...), элементы DOM (`document`, `frames`...), а также другие переменные. Доступ ко всем этим свойствам осуществляется с помощью синтаксиса `window.property`.

```
window.onload = function(){
 window.alert('window loaded');
}
```

Объект верхнего уровня в Node.js называется `global`. Поскольку Node.js – веб-сервер, а не браузер, множество доступных функций и свойств существенно отличается.

Когда интерпретатор JavaScript, работающий в браузере, проверяет существование *глобальной* переменной, он просматривает объект `window`.