Шаблоны программирования и проектирования высококачественных приложений

JavaScript Шаблоны

Символ®

O'REILLY® YAHOO! PRESS

Стоян Стефанов

JavaScript Patterns

Stoyan Stefanov

O'REILLY®

JavaScript
Шаблоны

Стоян Стефанов

Санкт-Петербург — Москва
2011

Стоян Стефанов

JavaScript. Шаблоны

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Корректор	*О. Макарова*
Верстка	*К. Чубаров*

Стефанов С.

JavaScript. Шаблоны. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 272 с., ил.

ISBN 978-5-93286-208-7

Разработчики серверных, клиентских или настольных приложений на JavaScript нередко сталкиваются с проблемами, связанными с объектами, функциями, наследованием и другими особенностями этого языка. Какие же приемы разработки приложений на JavaScript являются наиболее удачными? Данная книга дает ответ на этот вопрос, предлагая большое количество различных шаблонов программирования на JavaScript, таких как «единственный объект» (singleton), «фабрика» (factory), «декоратор» (decorator) и другие. Можно сказать, что они являются не только методами решения наиболее типичных задач разработки ПО, но и заготовками решений для целых категорий таких задач.

Использование шаблонов при программировании на языке JavaScript имеет свои особенности. Некоторые из них, разработанные с позиций языков со строгим контролем типов, таких как C++ и Java, не могут непосредственно применяться в языках с динамической типизацией, таких как JavaScript. Для таких шаблонов в языке JavaScript имеются более простые альтернативы.

Написанная экспертом по языку JavaScript Стояном Стефановым – ведущим специалистом компании Yahoo! и создателем инструмента оптимизации производительности веб-страниц YSlow 2.0, – книга включает практические советы по реализации каждого из рассматриваемых шаблонов с примерами программного кода. Автор также приводит антишаблоны – приемы программирования, которых следует по возможности избегать.

ISBN 978-5-93286-208-7
ISBN 978-0-596-80675-0 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 380-5007, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 23.03.2011. Формат 70×100 1/16. Печать офсетная.
Объем 17 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Моим девочкам,
Еве, Златине и Натали

Оглавление

Предисловие

Шаблоны программирования представляют собой методы решения наиболее типичных задач веб-разработки. Более того, можно сказать, что они являются заготовками решений для целых категорий таких задач.

Шаблоны позволяют разбить проблему на фрагменты и сосредоточиться на отдельных ее частях, применять опыт, накопленный предыдущими поколениями программистов, а также помогают разработчикам взаимодействовать друг с другом благодаря использованию общей терминологии. По этим причинам очень важно уметь выделять шаблоны программирования и знать, как с ними работать.

Целевая аудитория

Эта книга не для начинающих – она предназначена для профессиональных разработчиков и программистов, желающих поднять свои навыки владения JavaScript на новый уровень.

Основы программирования на JavaScript (такие как циклы, условные инструкции и замыкания) вообще не рассматриваются в этой книге. Если вам потребуется освежить свои знания по определенным темам, обращайтесь к списку дополнительной литературы.

В то же время некоторые темы (такие как создание объектов или особенности объектов среды выполнения) могут показаться слишком простыми для этой книги; однако они рассматриваются с точки зрения шаблонов программирования и, на мой взгляд, имеют важное значение для овладения возможностями языка.

Если вы ищете наиболее эффективные и действенные шаблоны, которые помогут вам писать более удобочитаемый и надежный программный код на JavaScript, то эта книга для вас.

Типографские соглашения

В этой книге приняты следующие соглашения:

Курсив

Применяется для выделения новых терминов, адресов **URL и электронной почты**, имен файлов и каталогов, а также типов файлов.

`Моноширинный шрифт`

Применяется для представления листингов программ, а также для выделения в обычном тексте программных элементов, таких как имена переменных, функций, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный жирный`

Используется для выделения команд или текста, который должен быть введен пользователем.

`Моноширинный курсив`

Обозначает элементы в программном коде, которые должны быть замещены конкретными значениями или значениями, определяемыми контекстом.

Так выделяются советы, предложения или примечания общего характера.

Так выделяются предупреждения или предостережения.

Использование программного кода примеров

Данная книга призвана оказать вам помощь в решении ваших задач. Вообще вы можете свободно использовать примеры программного кода из этой книги в своих приложениях и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам обращаться за разрешением не нужно. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам *необходимо* получить разрешение от издательства O'Reilly. Для цитирования данной книги или примеров из нее при ответе на вопросы получение разрешения не требуется. При включении существенных объемов программного кода примеров из этой книги в вашу документацию вам *необходимо* получить разрешение издательства.

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN. Например: «JavaScript Patterns, by Stoyan Stefanov (O'Reilly). Copyright 2010 Yahoo!, Inc., 9780596806750».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online – это виртуальная библиотека, которая позволяет легко и быстро находить ответы на вопросы среди более чем 7500 технических и справочных изданий и видеороликов.

Подписавшись на услугу, вы сможете загружать любые страницы из книг и просматривать любые видеоролики из нашей библиотеки. Читать книги на своих мобильных устройствах и сотовых телефонах. Получать доступ к новинкам еще до того, как они выйдут из печати. Читать рукописи, находящиеся в работе и посылать свои отзывы авторам. Копировать и вставлять отрывки программного кода, определять свои предпочтения, загружать отдельные главы, устанавливать закладки на ключевые разделы, оставлять примечания, печатать страницы и пользоваться массой других преимуществ, позволяющих экономить ваше время.

Благодаря усилиям O'Reilly Media данная книга также доступна через услугу Safari Books Online. Чтобы получить полный доступ к электронной версии этой книги, а также книг с похожими темами издательства O'Reilly и других издательств, подпишитесь бесплатно по адресу *http://my.safaribooksonline.com*.

Как с нами связаться

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в Соединенных Штатах Америки или в Канаде)
707-829-0515 (международный)
707-829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

http://www.oreilly.com/catalog/9780596806750/

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, Центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Благодарности

Я бесконечно признателен моим рецензентам, делившимся со мной своей энергией и своими знаниями, позволившими улучшить эту книгу. Их блоги и сообщения в твиттере неизменно вызывают у меня глубокий интерес и являются неиссякающим источником тонких наблюдений, замечательных идей и шаблонов программирования.

- Дмитрий Сошников (Dmitry Soshnikov) (*http://dmitrysoshnikov.com*, *@DmitrySoshnikov*)
- Андреа Джиммарчи (Andrea Giammarchi) (*http://webreflection.blogspot.com*, *@WebReflection*)
- Асен Божилов (Asen Bozhilov) (*http://asenbozhilov.com*, *@abozhilov*)
- Юрий Зайцев (Juriy Zaytsev) (*http://perfectionkills.com*, *@kangax*)
- Райан Гров (Ryan Grove) (*http://wonko.com*, *@yaypie*)
- Николас Закас (Nicholas Zakas) (*http://nczonline.net*, *@slicknet*)
- Реми Шарп (Remy Sharp) (*http://remysharp.com*, *@rem*)
- Ильян Пейчев (Iliyan Peychev)

Авторство

Некоторые из шаблонов, представленных в книге, были созданы автором на основе его личного опыта и изучения популярных библиотек JavaScript, таких как jQuery и YUI. Но большая часть шаблонов была выработана и описана сообществом программистов на JavaScript. То есть данная книга является результатом коллективного труда многих разработчиков. Чтобы не прерывать рассказ хронологией выделения шаблонов и перечислением авторов, список ссылок и дополнительной литературы был вынесен на сайт книги *http://www.jspatterns.com/book/reading/*.

Если вы найдете интересную и оригинальную статью, которую я не указал в списке ссылок, примите мои искренние извинения и сообщите мне адрес, чтобы я смог добавить его в список на сайте *http://jspatterns.com*.

Дополнительная литература

Эта книга не для начинающих, и в ней вы не найдете описание базовых особенностей языка, таких как циклы и условные инструкции. Если у вас есть потребность в пособии по JavaScript, можно порекомендовать следующие книги:

- «Object-Oriented JavaScript» Стоян Стефанов (Stoyan Stefanov) (Packt Publishing)

- «JavaScript: The Definitive Guide», Дэвид Флэнаган (David Flanagan) (O'Reilly)[1]
- «JavaScript: The Good Parts», Дуглас Крокфорд (Douglas Crockford) (O'Reilly)
- «Pro JavaScript Design Patterns», Росс Гермес (Ross Hermes) и Дастин Диаз (Dustin Diaz) (Apress)
- «High Performance JavaScript», Николас Закас (Nicholas Zakas) (O'Reilly)[2]
- «Professional JavaScript for Web Developers», Николас Закас (Nicholas Zakas) (Wrox)

Об авторе

Стоян Стефанов (Stoyan Stefanov) – веб-разработчик компании Yahoo!, автор книги «Object-Oriented JavaScript», консультант книг «Even Faster Web Sites» и «High Performance JavaScript», технический редактор книг «JavaScript: The Good Parts» и «PHP Mashups». Он постоянно выступает на конференциях, рассказывая о JavaScript, PHP и других технологиях веб-разработки, и ведет свой блог (*http://www.phpied.com*). Стоян является автором smush.it – инструмента оптимизации графических изображений и создателем YSlow 2.0 – инструмента оптимизации производительности Yahoo.

[1] Дэвид Флэнаган «JavaScript. Подробное руководство», 5-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

[2] Николас Закас «JavaScript. Оптимизация производительности». – Пер. с англ. – СПб.: Символ-Плюс, готовится к выпуску в III квартале 2011.

1

Введение

JavaScript – это язык Всемирной паутины. Первоначально он представлял собой средство манипулирования отдельными типами элементов веб-страниц (такими как изображения и поля форм), но с тех пор этот язык значительно вырос. В настоящее время помимо создания клиентских сценариев, выполняемых броузером, JavaScript можно использовать для разработки программ под широкий спектр разнообразных платформ. На этом языке можно писать серверные сценарии (используя .NET или Node.js), обычные приложения (способные работать в любых операционных системах) и расширения для приложений (например, для Firefox или Photoshop), приложения для мобильных устройств и сценарии командной строки.

JavaScript отличается от многих других языков программирования. В нем отсутствуют классы, а функции являются обычными объектами, которые используются для решения самых разных задач. При его появлении многие разработчики считали его недостаточно совершенным, но за последние годы это убеждение изменилось на диаметрально противоположное. Интересно отметить, что некоторые языки программирования, например Java и PHP, стали заимствовать определенные особенности JavaScript, например замыкания и анонимные функции, которыми разработчики на JavaScript располагают уже давно.

Динамичность языка JavaScript позволит вам сделать его похожим на язык программирования, в котором вы чувствуете себя кофортно. Но более правильным было бы изучить особенности JavaScript и научиться использовать его отличия.

Шаблоны

В общем случае слово «шаблон» – это «повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования

в рамках некоторого часто возникающего контекста» (*http://ru.wikipedia.org/wiki/Pattern*).

В разработке программного обеспечения шаблоном называется решение типичной задачи. Шаблон – это необязательно фрагмент программного кода, который можно скопировать и вставить в программу. Чаще всего это эффективный прием, удачная абстракция или пример решения целого класса задач.

Важно уметь выделять шаблоны, потому что:

- Они помогают писать более эффективный программный код, используя наработанные приемы и не изобретая велосипед.
- Они предоставляют дополнительный уровень абстракции – в каждый конкретный момент мы способны держать в уме лишь определенный объем информации, а шаблоны позволяют при решении сложных задач не погружаться в детали реализации, но обращаться с ними как с завершенными строительными компонентами (шаблонами).
- Они упрощают дискуссию между разработчиками, зачастую удаленными друг от друга и не имеющими возможности встретиться лично. Простое упоминание некоторого приема программирования позволяет легко убедиться, что мы говорим об одном и том же. Например, намного проще сказать (и подумать) «немедленно вызываемая функция» (immediate function), чем «функция, определение которой заключается в круглые скобки, вслед за которыми указывается еще одна пара круглых скобок, чтобы обеспечить немедленный вызов функции сразу после ее определения».

В этой книге рассматриваются следующие типы шаблонов:

- Шаблоны проектирования
- Шаблоны кодирования
- Антишаблоны

Термин *шаблоны проектирования* впервые был определен «бандой четырех» (по количеству авторов) в книге, вышедшей в далеком 1994 году под названием «Design Patterns: Elements of Reusable Object-Oriented Software»[1]. Примерами шаблонов проектирования могут служить такие шаблоны, как singleton (одиночка), factory (фабрика), decorator (декоратор), observer (наблюдатель) и так далее. Одна из особенностей использования шаблонов проектирования при программировании на языке JavaScript состоит в том, что, несмотря на их независимость от языка программирования, большая часть шаблонов проектирования разрабатывалась тем не менее с позиций языков со строгим контролем

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования». – Пер. с англ. – Питер, 2001.

типов, таких как C++ и Java. Вследствие этого некоторые шаблоны не могут непосредственно применяться в языках с динамической типизацией, таких как JavaScript. Некоторые шаблоны помогают обойти ограничения, накладываемые языками со строгим контролем типов и с механизмом наследования на основе классов. Для таких шаблонов в языке JavaScript могут иметься более простые альтернативы. Реализации нескольких шаблонов проектирования на языке JavaScript рассматриваются в главе 7 этой книги.

Шаблоны кодирования, представляющие для нас гораздо больший интерес, – это шаблоны, характерные для JavaScript и учитывающие уникальные особенности этого языка, такие как различные способы использования функций. Шаблоны кодирования на JavaScript являются основной темой этой книги.

В этой книге вы периодически будете сталкиваться с *антишаблонами*. Название «антишаблон» имеет несколько негативный или даже обидный оттенок, но вы не должны придавать ему такое значение. Антишаблон – это не ошибка, а некоторый широко распространенный прием, который скорее порождает проблемы, чем решает их. Антишаблоны будут явно отмечаться комментариями в программном коде.

JavaScript: концепции

Давайте коротко пробежимся по некоторым важным концепциям, которые лежат в основе последующих глав.

JavaScript — объектно-ориентированный язык

JavaScript – это объектно-ориентированный язык программирования, что нередко удивляет разработчиков, которые прежде слышали о нем, но не использовали в своей работе. Любые элементы, которые вы видите в сценариях на JavaScript, наверняка являются объектами. В этом языке программирования существует всего пять элементарных типов данных, не являющихся объектами: числа, строки, логические значения, `null` и `undefined`, при этом первые три типа имеют соответствующее объектно-ориентированное представление в виде простых оберток (подробнее о них рассказывается в следующей главе). Числа, строки и логические значения могут быть легко преобразованы в объекты либо явно – программистом, либо неявно – интерпретатором JavaScript.

Функции также являются объектами. Они могут иметь свои свойства и методы.

Самой простой операцией в любом языке программирования является определение переменной. Однако в JavaScript, определяя переменную, вы уже имеете дело с объектом. Во-первых, переменная автоматически становится *свойством* внутреннего объекта, так называемого объекта активации (Activation Object) (или свойством глобального объекта, если определяется глобальная переменная). Во-вторых, эта переменная

фактически является объектом, потому что она обладает собственными свойствами (или *атрибутами*), которые определяют доступность переменной для изменения, удаления или использования в качестве последовательности в циклах `for-in`. Эти атрибуты явно не определены в стандарте ECMAScript 3, но в версии 5 предложены специальные методы для управления ими.

Так что же такое объекты? Они предусматривают так много возможностей, что на первый взгляд должны быть чем-то особенным. В действительности они чрезвычайно просты. *Объект* – это всего лишь коллекция именованных свойств, список пар ключ-значение (во многом идентичный ассоциативным массивам в других языках программирования). Некоторые свойства объектов могут быть функциями (объектами функций), которые в этом случае называются *методами*.

Еще одна особенность создаваемых вами объектов состоит в том, что они остаются доступными для изменения в любой момент. (Впрочем, стандарт ECMAScript 5 определяет API, предотвращающий возможность изменения.) Вы можете добавлять, удалять и изменять свойства объектов. Для тех, кого волнуют проблемы безопасности и доступа, возникающие в связи с этим, в книге приводятся шаблоны, решающие эти проблемы.

И наконец, следует помнить, что существуют две основные разновидности объектов:

Собственные объекты языка

Определяются стандартом ECMAScript.

Объекты окружения

Определяются средой выполнения (например, броузером).

Собственные объекты языка могут быть разделены на *встроенные* (например, `Array`, `Date`) и *пользовательские* (`var o = {};`).

Объектами окружения являются такие объекты, как `window` и все объекты DOM. Если вам интересно узнать, используете ли вы объекты окружения, попробуйте выполнить свой сценарий в иной среде, отличной от броузера. Если он работает, следовательно, вы наверняка используете только собственные объекты языка.

В JavaScript отсутствуют классы

Это утверждение постоянно будет встречаться на протяжении всей книги: в языке JavaScript отсутствуют классы. Эта особенность будет в диковинку опытным программистам, обладающим опытом работы с другими языками программирования, и им понадобится определенная практика, чтобы «отучиться» от использования классов и привыкнуть иметь дело только с объектами.

Отсутствие классов позволяет писать более короткие программы – нет необходимости добавлять определение класса, чтобы создать объект. Взгляните, как выглядит создание объекта в языке Java:

```
// Создание объекта в языке Java
HelloOO hello_oo = new HelloOO();
```

Необходимость написать одно и то же три раза выглядит излишеством, когда требуется создать множество простых объектов. А чаще всего мы хотим, чтобы наши объекты были именно простыми.

В JavaScript сначала создается пустой объект, а потом к нему добавляются желаемые свойства и методы. Объекты конструируются за счет добавления к ним свойств элементарных типов, функций или других объектов. «Пустой» объект в действительности не совсем пустой; он обладает несколькими встроенными свойствами, но не имеет «собственных» свойств. Подробнее об этом мы поговорим в следующей главе.

В книге, написанной «бандой четырех», говорится: «предпочтение отдается приему составления объектов, а не наследованию». Это означает, что создание объектов из компонентов, имеющихся под рукой, является более предпочтительным, чем создание длинных иерархий наследования родитель–потомок. Следовать этому совету в JavaScript очень легко, потому что в языке отсутствуют классы, а конструирование объектов – это, собственно, как раз то, что вы так или иначе делаете.

Прототипы

В JavaScript присутствует механизм наследования, хотя это всего лишь способ обеспечить многократное использование программного кода. (Многократному использованию посвящена целая глава в этой книге.) Наследование можно реализовать несколькими способами, которые обычно опираются на использование прототипов. *Прототип* – это объект (что совершенно неудивительно), а каждая создаваемая вами функция получает свойство `prototype`, ссылающееся на новый пустой объект. Этот объект практически идентичен тому, который создается литеральной формой или с помощью конструктора `Object()`, за исключением того, что свойство `constructor` этого объекта ссылается на созданную вами функцию, а не на встроенный конструктор `Object()`. Вы можете добавлять новые члены к этому пустому объекту и затем получать другие объекты, наследующие свойства и методы этого объекта и использующие их для своих нужд.

Мы еще вернемся к проблеме наследования, а пока просто имейте в виду, что прототип – это объект (а не класс и не какая-то другая специальная конструкция) и каждая функция имеет свойство `prototype`.

Среда выполнения

Чтобы запустить программу на JavaScript, необходимо иметь среду выполнения. Естественной средой обитания программ на JavaScript являет-

ся броузер, но эта среда не единственная. Значительная часть шаблонов, представленных в этой книге, касается ядра JavaScript (ECMAScript), то есть они могут применяться в любой среде выполнения.

Исключениями являются:

- Шаблоны, применяемые в броузерах, которые описаны в главе 8.
- Некоторые другие примеры, иллюстрирующие практическое применение шаблонов.

Каждая среда выполнения может определять собственные *объекты окружения*, которые не определены в стандарте ECMAScript и могут проявлять незаданное и неожидаемое поведение.

ECMAScript 5

Ядро языка программирования JavaScript (за исключением DOM, BOM и дополнительных объектов окружения) опирается на стандарт ECMAScript, или ES. Версия 3 стандарта была официально принята в 1999 году и в настоящее время реализована в броузерах. Работа над версией 4 была остановлена, а в декабре 2009 года, спустя 10 лет после утверждения предыдущей версии, была одобрена версия 5.

Версия 5 определяет несколько новых встроенных объектов, методов и свойств, но наиболее важным новшеством является введение так называемого *строгого* (*strict*) режима, в котором запрещается использование некоторых особенностей языка, что делает программы более простыми и более устойчивыми к ошибкам. Например использование инструкции `with`, необходимость которой вызывала споры на протяжении нескольких лет. Теперь, согласно ES5, эта инструкция будет вызывать ошибку при выполнении в строгом режиме, а в нестрогом режиме будет выполняться как обычно. Строгий режим включается простой строкой, которая игнорируется прежними реализациями языка. Это означает, что попытка включить строгий режим не будет порождать ошибку в броузерах, не поддерживающих такую возможность.

В каждой отдельно взятой области видимости (в объявлении функции, в глобальной области видимости или в начале строки, которая передается функции `eval()`) теперь можно перейти в строгий режим, как показано ниже:

```
function my() {
    "use strict";
    // остальная часть функции...
}
```

Это означает, что в теле функции будет доступен лишь ограниченный набор возможностей языка. Старые версии броузеров будут воспринимать эту строку как обычное строковое значение, которое не присваи-

вается никакой переменной, а потому никак не используется и не вызывает ошибок.

В дальнейшем предполагается, что строгий режим станет единственно возможным. С этой точки зрения ES5 является переходной версией – разработчикам предлагается писать программный код, который будет действовать в строгом режиме, но никто не принуждает их к этому.

В этой книге не рассматриваются шаблоны, имеющие отношение к характерным особенностям, привнесенным стандартом ES5, потому что к моменту написания этих строк ни один из современных броузеров не реализовал требования ES5. Однако примеры, которые приводятся в этой книге, предполагают возможность перехода на новый стандарт:

- Их выполнение не будет приводить к ошибкам в строгом режиме
- В них не используются устаревшие конструкции, такие как `arguments.callee`
- Примеры используют шаблоны ES3, для которых в ES5 предусматриваются встроенные эквиваленты, такие как `Object.create()`

JSLint

JavaScript – это интерпретируемый язык программирования, для которого не предусматривается статическая проверка типов на этапе компиляции. Вследствие этого вполне возможно развернуть программу с ошибкой, вызванной обычной опечаткой, не заметив этого. Чтобы не допустить подобных ситуаций, можно воспользоваться инструментом JSLint (*http://jslint.com*).

JSLint – это инструмент проверки качества программного кода на JavaScript, созданный Дугласом Крокфордом (Douglas Crockford), который проверит ваш программный код и предупредит обо всех потенциальных проблемах. Обязательно проверяйте свои сценарии с помощью JSLint. Этот инструмент «будет задевать ваше самолюбие», как предупреждает его создатель, но только на первых порах. Благодаря JSLint вы будете быстро учиться на собственных ошибках и постепенно выработаете в себе основные привычки профессионального программиста на JavaScript. Отсутствие предупреждений после проверки с помощью JSLint поднимет вашу *уверенность* в программном коде, так как вы будете знать, что второпях не допустили элементарных оплошностей или синтаксических ошибок.

Начиная со следующей главы, вам часто будут встречаться упоминания о JSLint. Все примеры программного кода в книге успешно прошли проверку инструментом JSLint (с текущими, на момент написания этих строк, настройками по умолчанию), за исключением нескольких из них, которые явно отмечены как антишаблоны.

Настройки JSLint по умолчанию предполагают, что проверяемый программный код написан для работы в строгом режиме.

Консоль

На протяжении всей книги мы будем использовать объект `console`. Этот объект не является частью языка – он определяется средой выполнения и присутствует в большинстве современных броузеров. В Firefox, например, этот объект входит в состав расширения Firebug. Консоль в Firebug имеет графический интерфейс, позволяющий быстро вводить и проверять небольшие фрагменты программного кода на JavaScript или экспериментировать с текущей загруженной страницей (рис. 1.1). Кроме того, Firebug – это очень удобный инструмент для исследований. Аналогичная функциональность присутствует в броузерах, реализованных на основе движка WebKit (Safari и Chrome), как часть комплекта инструментов Web Inspector, и в IE, начиная с версии 8, как часть комплекта инструментов Developer Tools.

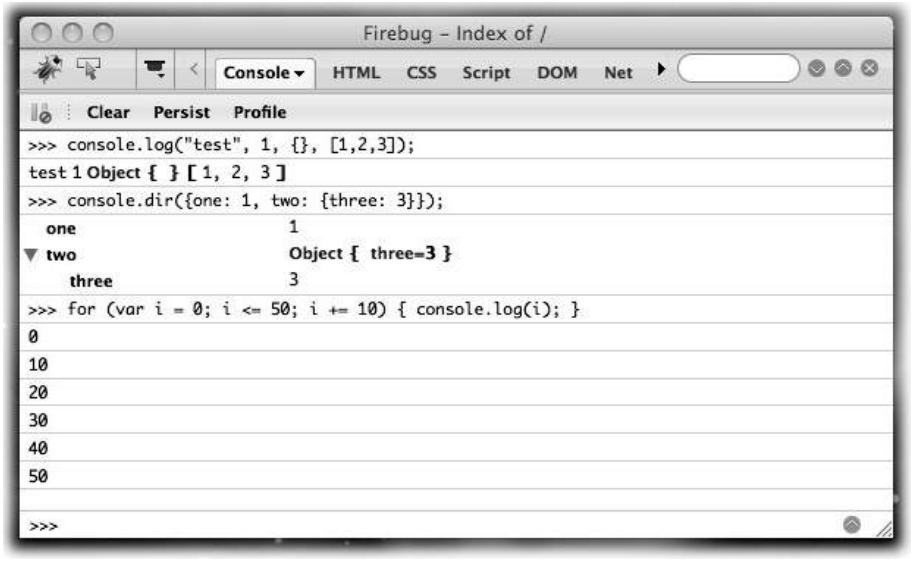

Рис. 1.1. *Использование консоли в броузере Firebug*

В большинстве примеров вместо обновления страницы или вызова функции `alert()` используется объект `console`, который предоставляет простой и ненавязчивый способ вывода информации.

Мы будем часто использовать метод `log()`, который выводит все переданные ему параметры, а иногда метод `dir()`, который перечисляет пе-

реданные ему объекты и выводит значения их свойств. Ниже приводится пример использования этих методов:

```
console.log("test", 1, {}, [1,2,3]);
console.dir({one: 1, two: {three: 3}});
```

При работе в консоли не требуется использовать метод `console.log()` — вызов метода можно просто опустить. Чтобы не загромождать текст, в некоторых примерах именно так и делается, если предполагается, что они должны проверяться в консоли:

```
window.name === window['name']; // true
```

Эта инструкция эквивалентна следующей:

```
console.log(window.name === window['name']);
```

и выведет в консоли слово `true`.

2

Основы

В этой главе рассматриваются основные приемы, шаблоны и способы оформления высококачественного программного кода на JavaScript, такие как отказ от глобальных переменных, объявление переменных в единственной инструкции var, вынесение вычисления переменной длины из заголовка цикла, следование соглашениям по оформлению программного кода и многие другие. Кроме того, в этой главе приводятся некоторые рекомендации, напрямую не связанные с программным кодом и имеющие отношение к самому процессу создания сценариев, включая написание документации, проведение сравнительных исследований и проверку сценариев с помощью JSLint. Эти приемы и рекомендации помогут вам писать более качественный, более понятный и более простой в сопровождении программный код – код, которым можно будет гордиться (и который можно будет читать) спустя месяцы и годы.

Создание простого в сопровождении программного кода

Ошибки в программах достаточно сложно отыскивать и исправлять. И эта сложность возрастает с течением времени, особенно если ошибки вкрадываются в уже выпущенный программный продукт. Проще всего ошибки исправлять в процессе работы, сразу же после их обнаружения, то есть пока ваши мысли крутятся вокруг проблемы, которую вы пытаетесь решить. Сделать это будет намного сложнее, когда вы перейдете к решению другой задачи и забудете все, что связано с данным конкретным фрагментом программного кода. При последующем возвращении к программному коду вам потребуется:

- Некоторое время, чтобы повторно изучить и понять проблему
- Некоторое время, чтобы понять программный код, который решает эту проблему

Еще одна проблема, характерная для крупных проектов и компаний, состоит в том, что ошибку приходится исправлять уже другому программисту, а не тому, который ее допустил, а натыкается на нее обычно еще кто-то третий. Поэтому очень важно сократить время, необходимое на то, чтобы понять, как действует программный код, написанный некоторое время тому назад вами или другим разработчиком. Это чрезвычайно важно не только с экономической, но и с психологической точки зрения, потому что все мы предпочитаем создавать что-то новое и захватывающее, а не тратить часы и дни, оживляя старую плохо работающую программу.

Другой жизненный факт, связанный с разработкой ПО, заключается в том, что большая часть времени обычно тратится на *чтение* программного кода, а не на его *написание*. Иногда, когда разработчик с головой уходит в решение проблемы, он может за один день написать значительный объем кода. Этот код наверняка будет исправно работать на данном этапе, но по мере разработки приложения могут возникать ситуации, когда придется вернуться к этому программному коду, внимательно просмотреть его и изменить, если это потребуется. Например:

- При обнаружении ошибок
- При добавлении в приложение новых особенностей
- При необходимости обеспечить работоспособность в другой среде выполнения (например, в новом броузере, появившемся на рынке)
- Когда потребуется изменить реализацию
- Когда необходимо переписать код заново, или перенести реализацию на другую архитектуру, или даже переложить ее на другой язык

В результате для внесения изменений в программный код, который был написан за несколько человеко-часов, может потребоваться затратить несколько человеко-недель, – на его чтение. Именно поэтому так важно для успеха приложения писать простой в сопровождении программный код.

Простой в сопровождении код предполагает соответствие следующим критериям:

- Удобочитаемость
- Непротиворечивость
- Предсказуемость
- Оформление, как если бы он был написан одним человеком
- Наличие документации

Все примеры, которые приводятся в оставшейся части главы, демонстрируют реализацию этих требований при программировании на JavaScript.

Минимизация количества глобальных переменных

Для управления областями видимости в JavaScript используются функции. Переменная, объявленная внутри функции, является *локальной* по отношению к этой функции и недоступна за ее пределами. С другой стороны, переменные, объявленные или просто используемые за пределами функций, являются *глобальными*.

В любой среде выполнения JavaScript существует *глобальный объект*, доступный при использовании ссылки `this` за пределами функций. Любая создаваемая вами глобальная переменная становится свойством глобального объекта. Для удобства глобальный объект в броузерах имеет дополнительное свойство `window`, которое (обычно) ссылается на сам глобальный объект. Следующий фрагмент демонстрирует, как можно создать глобальную переменную и обратиться к ней в броузере:

```
myglobal = "hello";               // антишаблон
console.log(myglobal);            // "hello"
console.log(window.myglobal);     // "hello"
console.log(window["myglobal"]); // "hello"
console.log(this.myglobal);       // "hello"
```

Проблемы, связанные с глобальными переменными

Проблема, связанная с глобальными переменными, порождается тем, что они доступны всему приложению на JavaScript или всей веб-странице. Все эти переменные находятся в одном глобальном пространстве имен, вследствие чего всегда есть вероятность конфликта имен – когда две различные части приложения определяют глобальные переменные с одинаковыми именами, но для разных целей.

Кроме того, в веб-страницы часто включается программный код, написанный сторонними разработчиками, например:

- Сторонние библиотеки JavaScript
- Сценарии рекламодателя
- Сторонние сценарии для отслеживания пользователей и сбора статистической информации
- Различные виджеты, эмблемы и кнопки

Предположим, что один из сторонних сценариев определяет глобальную переменную с именем, например, `result`. Далее, в одной из своих функций вы также определяете глобальную переменную с именем `result`. В результате сто́ит функции изменить значение переменной `result`, и сторонний сценарий может перестать работать.

Поэтому так важно сохранять добрососедские отношения с другими сценариями, присутствующими в странице, и использовать как можно меньше глобальных переменных. Далее в этой книге вы познакомитесь

с некоторыми стратегиями уменьшения количества глобальных переменных, такими как шаблон создания отдельных пространств имен или немедленно вызываемые функции, но самым действенным шаблоном, позволяющим уменьшить количество глобальных переменных, является повсеместное использование объявлений `var`.

Легкость непреднамеренного создания глобальных переменных в JavaScript обусловлена двумя особенностями этого языка. Во-первых, в нем допускается использовать переменные без их объявления. И во-вторых, в этом языке имеется понятие *подразумеваемых глобальных переменных*, когда любая необъявленная переменная превращается в свойство глобального объекта (и становится доступной, как любая другая объявленная глобальная переменная). Взгляните на следующий пример:

```
function sum(x, y) {
    // антишаблон: подразумеваемая глобальная переменная
    result = x + y;
    return result;
}
```

В этом фрагменте используется необъявленная переменная `result`. Эта функция прекрасно работает, но после ее вызова в глобальном пространстве имен появится переменная `result`, что впоследствии может стать источником ошибок.

Чтобы избежать подобных проблем, всегда объявляйте переменные с помощью инструкции `var`, как показано в улучшенной версии функции `sum()`:

```
function sum(x, y) {
    var result = x + y;
    return result;
}
```

Еще один антишаблон, применение которого приводит к созданию подразумеваемых глобальных переменных, – это объединение нескольких операций присваивания в одной инструкции `var`. Следующий фрагмент создаст локальную переменную `a`, но переменная `b` будет глобальной, что, вероятно, не совсем совпадает с намерениями программиста:

```
// антишаблон, не используйте его
function foo() {
    var a = b = 0;
    // ...
}
```

Если вам интересно разобраться, объясню, что это происходит из-за того, что выражения присваивания выполняются справа налево. Сначала выполняется выражение `b = 0`, а в данном случае `b` – это необъявленная переменная. Это выражение возвращает значение `0`, которое присваивается новой локальной переменной, объявленной инструкци-

ей var a. Другими словами, данная инструкция эквивалентна следующей:

```
var a = (b = 0);
```

Если бы переменные уже были объявлены, то объединение операций присваивания в одну инструкцию не привело бы к созданию глобальной переменной. Например:

```
function foo() {
    var a, b;
    // ...
    a = b = 0; // обе переменные - локальные
}
```

Переносимость – еще одна причина избегать глобальных переменных. Когда необходимо, чтобы программный код выполнялся в различных окружениях, глобальные переменные становятся небезопасными, потому что можно по неосторожности затереть какой-нибудь объект окружения, отсутствующий в среде выполнения, для которой первоначально создавался сценарий (из-за чего вы могли считать имя, выбранное для переменной, безопасным), но имеющийся в каких-то других окружениях.

Побочные эффекты, возникающие в отсутствие объявления var

Между подразумеваемыми глобальными переменными и глобальными переменными, объявленными явно, существует одно тонкое отличие – возможность использования оператора delete для удаления переменных:

- Глобальные переменные, созданные явно с помощью инструкции var (в программе, за пределами какой-либо функции), не могут быть удалены.
- Подразумеваемые переменные, созданные без помощи инструкции var (независимо от того, были они созданы в пределах функции или нет), могут быть удалены.

Следующий фрагмент демонстрирует, что подразумеваемые глобальные переменные технически не являются настоящими переменными, но являются свойствами глобального объекта. Свойства могут удаляться с помощью оператора delete, тогда как переменные – нет:

```
// определение трех глобальных переменных
var global_var = 1;
global_novar = 2;         // антишаблон
(function () {
    global_fromfunc = 3; // антишаблон
}());
```

```
// попытка удаления
delete global_var;      // false
delete global_novar;    // true
delete global_fromfunc; // true

// проверка удаления
typeof global_var;      // "number"
typeof global_novar;    // "undefined"
typeof global_fromfunc; // "undefined"
```

В строгом режиме, предусматриваемом стандартом ES5, попытка присвоить значение необъявленной переменной (такой как переменные в двух антишаблонах выше) приведет к появлению ошибки.

Доступ к глобальному объекту

В броузерах глобальный объект доступен в любой части сценария как свойство с именем `window` (если только вы не предприняли каких-либо неожиданных действий, например, объявив локальную переменную с именем `window`). Однако в других окружениях это удобное свойство может называться иначе (или вообще быть недоступным для программиста). Если вам необходимо обратиться к глобальному объекту без использования жестко определенного идентификатора `window`, можно выполнить следующие операции на любом уровне вложенности в области видимости функции:

```
var global = (function () {
    return this;
}());
```

Этот прием позволит получить доступ к глобальному объекту в любом месте, потому что внутри функций, которые вызываются как функции (то есть не как конструкторы с оператором `new`), ссылка `this` всегда указывает на глобальный объект. Однако этот прием не может использоваться в строгом режиме, предусматриваемом стандартом ECMAScript 5, поэтому вам необходимо будет задействовать иной шаблон, если потребуется обеспечить работоспособность сценария в строгом режиме. Например, если вы разрабатываете библиотеку, можно обернуть программный код библиотеки немедленно вызываемой функцией (этот шаблон рассматривается в главе 4), а из глобальной области видимости передавать этой функции ссылку `this` в виде параметра.

Шаблон единственной инструкции var

Прием использования единственной инструкции `var` в начале функций – достаточно полезный шаблон, чтобы взять его на вооружение. Его применение дает следующие преимущества:

- Все локальные переменные, необходимые функции, объявляются в одном месте, что упрощает их поиск

- Предотвращает логические ошибки, когда переменная используется до того, как она будет объявлена (подробнее об этом рассказывается в разделе «Проблемы с разбросанными переменными» ниже)
- Помогает не забывать объявлять переменные и позволяет уменьшить количество глобальных переменных
- Уменьшает объем программного кода (который придется вводить и передавать по сети)

Шаблон единственной инструкции `var` выглядит следующим образом:

```
function func() {
    var a = 1,
        b = 2,
        sum = a + b,
        myobject = {},
        i,
        j;

    // тело функции...
}
```

Вы используете единственную инструкцию `var` и объявляете в ней сразу несколько переменных, разделяя их запятыми. При этом допускается одновременно *инициализировать* переменные начальными значениями. Это поможет предотвратить логические ошибки (все неинициализированные переменные по умолчанию получают значение `undefined`), а также повысит удобочитаемость программного кода. Когда в дальнейшем вы вновь вернетесь к этому коду, вы быстро сможете разобраться, для чего предназначена каждая переменная, опираясь на начальные значения. Например, является та или иная переменная объектом или целым числом.

Кроме того, во время объявления переменной можно предусмотреть выполнение фактических операций, таких как вычисление суммы `sum = a + b` в предыдущем примере. Другой пример: работая с деревом DOM (Document Object Model – объектная модель документа), можно одновременно с объявлением локальных переменных присваивать им ссылки на элементы, как показано ниже:

```
function updateElement() {
    var el = document.getElementById("result"),
        style = el.style;

    // Выполнение операций над переменными el и style...
}
```

Подъем: проблемы с разбросанными переменными

В языке JavaScript допускается вставлять несколько инструкций `var` в функции, и все они будут действовать, как если бы объявления пере-

менных находились в начале функции. Такое поведение называется *подъемом* (*hoisting*) и может приводить к логическим ошибкам, когда переменная сначала используется, а потом объявляется ниже в этой же функции. В языке JavaScript переменная считается объявленной, если она используется в той же области видимости (в той же функции), где находится объявление `var`, даже если это объявление располагается ниже. Взгляните на следующий пример:

```
// антишаблон
myname = "global";          // глобальная переменная
function func() {
    alert(myname);          // "undefined"
    var myname = "local";
    alert(myname);          // "local"
}
func();
```

Вы могли бы предположить, что первый вызов `alert()` выведет запрос «global», а второй – «local». Это вполне разумное предположение, потому что к моменту первого вызова `alert()` переменная `myname` еще не была объявлена, и поэтому функция должна была бы, вероятно, «увидеть» глобальную переменную `myname`. Но на самом деле все совсем не так. Первый вызов `alert()` выведет текст «undefined», потому что переменная `myname` считается объявленной как локальная переменная функции. (Даже несмотря на то, что объявление находится ниже.) Все объявления переменных как бы поднимаются в начало функции. Таким образом, чтобы избежать подобных накладок, лучше помещать объявления всех переменных, которые предполагается использовать, в начало функции.

Предыдущий фрагмент действует так, как если бы он был реализован, как показано ниже:

```
myname = "global";          // глобальная переменная
function func() {
    var myname;             // то же, что и -> var myname = undefined;
    alert(myname);          // "undefined"
    myname = "local";
    alert(myname);          // "local"
}
func();
```

Для полноты картины следует отметить, что фактическая реализация немного сложнее. Программный код обрабатывается интерпретатором в два этапа, причем объявления переменных, функций и формальных параметров обрабатываются на первом этапе, то есть на этапе синтаксического анализа и установки контекста. На втором этапе, этапе выполнения программного кода, производится создание функций-выражений и неквалифицированных идентификаторов (необъявленных переменных). Однако в утилитарных целях вполне можно пользоваться кон-

цепцией *подъема* (*hoisting*), которая хотя и не определяется стандартом ECMAScript, но часто используется для описания этого поведения.

Циклы for

Циклы `for` используются для выполнения итераций по элементам массивов или объектов, напоминающих массивы, таких как `arguments` и `HTMLCollection`. Ниже приводится типичный шаблон использования цикла `for`:

```
// не самый оптимальный способ использования цикла
for (var i = 0; i < myarray.length; i++) {
    // выполнить какие-либо операции над myarray[i]
}
```

Недостаток этого шаблона состоит в том, что в каждой итерации выясняется длина массива. Это может уменьшить скорость работы цикла, особенно если `myarray` является не массивом, а объектом `HTMLCollection`.

`HTMLCollection` – это объект, возвращаемый методами DOM, такими как:

- `document.getElementsByName()`
- `document.getElementsByClassName()`
- `document.getElementsByTagName()`

Существует еще несколько объектов `HTMLCollection`, которые появились до введения стандарта DOM и по-прежнему используются. В их число (наряду с другими) входят:

`document.images`

Все элементы IMG на странице.

`document.links`

Все элементы A.

`document.forms`

Все формы.

`document.forms[0].elements`

Все поля из первой формы на странице.

Проблема при работе с такими коллекциями состоит в том, что при обращении к ним выполняются запросы к документу (к HTML-странице). Это означает, что при каждом определении длины коллекции выполняется запрос к механизму DOM, а операции с деревом DOM, как правило, являются довольно дорогостоящими.

Именно поэтому при работе с циклами `for` лучше определять длину массива (или коллекции) заранее, как показано в следующем примере:

```
for (var i = 0, max = myarray.length; i < max; i++) {
    // выполнить какие-либо операции над myarray[i]
}
```

При таком подходе значение свойства `length` будет извлекаться всего один раз за все время работы цикла.

Прием предварительного определения длины при выполнении итераций по объектам `HTMLCollections` дает прирост в скорости во всех броузерах – от двух (Safari 3) до 190 раз (IE7). (Более подробно об этом рассказывается в книге «High Performance JavaScript»[1], написанной Николасом Закасом (Nicholas Zakas) [O'Reilly].)

Обратите внимание, что в случаях, когда вы предполагаете явно изменять коллекцию в цикле (например, добавлять дополнительные элементы DOM), вероятно, лучше будет определять длину коллекции в каждой итерации, а не один раз.

Следуя шаблону единственной инструкции `var`, объявление `var` можно вынести за пределы цикла, как показано ниже:

```
function looper() {
    var i = 0,
        max,
        myarray = [];

    // ...

    for (i = 0, max = myarray.length; i < max; i++) {
        // выполнить какие-либо операции над myarray[i]
    }
}
```

Достоинство такого подхода заключается в его последовательности – вы придерживаетесь шаблона с единственной инструкцией `var`. Недостаток состоит в том, что подобные циклы сложнее копировать и перемещать в процессе рефакторинга программного кода. Например, при копировании цикла из одной функции в другую вам также придется позаботиться об объявлении переменных `i` и `max` в новой функции (и, возможно, удалить их из оригинальной функции, если там они больше не используются).

И последний штрих к циклам `for` – замена выражения `i++` одним из следующих:

```
i = i + 1
i += 1
```

JSLint предложит вам сделать это. Причина заключается в том, что операторы `++` и `--` проявляют «излишнюю хитрость». Если вы не согласс-

[1] Закас Н. «JavaScript. Оптимизация производительности». – Пер. с англ. – СПб.: Символ-Плюс. Выход книги ожидается в III квартале 2011 г.

ны с такой заменой, установите в настройках JSLint параметр `plusplus` в значение `false`. (По умолчанию он принимает значение `true`.) Далее в этой книге будет использоваться последний из предложенных шаблонов: `i += 1`.

Ниже представлены два других немного более оптимизированных шаблона оформления циклов `for`, в которых:

- Используется на одну переменную меньше (отсутствует переменная `max`).
- Счет итераций ведется в обратном порядке – от максимального значения к нулю, что обычно дает дополнительный прирост в скорости из-за того, что операция сравнения с числом `0` немного эффективнее, чем операция сравнения с длиной массива или любым другим числом, отличным от `0`.

Первый шаблон цикла:

```
var i, myarray = [];

for (i = myarray.length; i--;) {
    // выполнить какие-либо операции над myarray[i]
}
```

И второй шаблон, основанный на использовании цикла `while`:

```
var myarray = [],
    i = myarray.length;

while (i--) {
    // выполнить какие-либо операции над myarray[i]
}
```

Прирост скорости, который дают эти незначительные улучшения, будет заметен только при выполнении операций, где требуется наивысшая производительность. Кроме того, JSLint будет выдавать предупреждения для каждого встреченного выражения `i--`.

Циклы for-in

Циклы `for-in` должны использоваться для обхода только тех объектов, которые не являются массивами. Обход с применением циклов `for-in` также называется *перечислением*.

С технической точки зрения ничто не мешает использовать цикл `for-in` для обхода элементов массива (потому что массивы в JavaScript также являются объектами), но делать так не рекомендуется. Это может привести к логическим ошибкам, если объект массива был расширен дополнительными функциональными возможностями. Кроме того, циклы `for-in` не гарантируют какой-то определенный порядок обхода свойств. Поэтому для обхода массивов рекомендуется использовать циклы `for`, а циклы `for-in` использовать для обхода свойств объектов.

При обходе в цикле свойств объекта важно использовать метод `hasOwnProperty()`, чтобы отфильтровать свойства, которые были унаследованы от прототипа.

Взгляните на следующий пример:

```
// объект
var man = {
    hands: 2,
    legs: 2,
    heads: 1
};

// где-то далее в сценарии ко всем объектам
// добавляется дополнительный метод
if (typeof Object.prototype.clone === "undefined") {
    Object.prototype.clone = function () {};
}
```

В этом примере мы создали простой объект с именем `man`, определив его с помощью литерала объекта. Где-то в тексте сценария, до или после создания объекта `man`, прототип объекта `Object` был расширен дополнительным методом с именем `clone()`. Цепочка наследования прототипа постоянно проверяется интерпретатором, а это означает, что все объекты автоматически получают доступ к новому методу. Чтобы избежать обнаружения метода `clone()` в процессе перечисления свойств объекта `man`, необходимо вызывать метод `hasOwnProperty()`, чтобы отфильтровать свойства прототипа. Отказ от фильтрации приведет к тому, что функция `clone()` окажется в числе перечисленных свойств, что в большинстве случаев может вызвать нежелательное поведение:

```
// 1.
// цикл for-in
for (var i in man) {
    if (man.hasOwnProperty(i)) { // фильтрация
        console.log(i, ":", man[i]);
    }
}
/*
результаты в консоли
hands : 2
legs : 2
heads : 1
*/

// 2.
// антишаблон:
// цикл for-in без проверки с помощью hasOwnProperty()
for (var i in man) {
    console.log(i, ":", man[i]);
}
```

```
/*
результаты в консоли
hands : 2
legs : 2
heads : 1
clone: function()
*/
```

Другой шаблон с использованием метода `hasOwnProperty()` – вызов метода относительно `Object.prototype`, как показано ниже:

```
for (var i in man) {
    if (Object.prototype.hasOwnProperty.call(man, i)) { // фильтрация
        console.log(i, ":", man[i]);
    }
}
```

Преимущество этого способа состоит в том, что он позволяет избежать конфликтов имен в случае, если объект `man` переопределит метод `hasOwnProperty`. Кроме того, можно предотвратить поиск метода в длинном списке свойств объекта `Object`, сохранив ссылку на него в локальной переменной:

```
var i,
    hasOwn = Object.prototype.hasOwnProperty;
for (i in man) {
    if (hasOwn.call(man, i)) { // фильтрация
        console.log(i, ":", man[i]);
    }
}
```

Строго говоря, отказ от использования метода `hasOwnProperty()` не является ошибкой. В зависимости от решаемой вами задачи вы можете опустить его вызов и тем самым немного ускорить выполнение цикла. Но если вы не уверены в содержимом объекта (и в том, какие свойства присутствуют в цепочке наследования прототипа), безопаснее будет добавить проверку `hasOwnProperty()`.

Имеется возможность использовать иной вариант оформления программного кода (который не проходит проверку с помощью JSLint) – опустить фигурную скобку перед началом тела цикла и поместить условную инструкцию `if` в одну строку с инструкцией `for`. В таком виде инструкция цикла читается почти как законченное предложение («для каждого элемента, который является собственным свойством X объекта, выполнить некоторые операции»). Кроме того, при таком оформлении получается меньше отступов:

```
// Внимание: не проходит проверку в JSLint
var i,
```

```
    hasOwn = Object.prototype.hasOwnProperty;
for (i in man) if (hasOwn.call(man, i)) { // фильтрация
    console.log(i, ":", man[i]);
}
```

Расширение встроенных прототипов (в том числе нежелательное)

Расширение свойств `prototype` функций-конструкторов — это довольно мощный способ расширения функциональных возможностей, но иногда он может оказаться слишком мощным.

Весьма заманчиво было бы расширить возможности прототипов встроенных конструкторов, таких как `Object()`, `Array()` или `Function()`, но это может отрицательно сказаться на удобстве сопровождения, потому что такой способ делает программный код менее предсказуемым. Другие разработчики, использующие ваш код, наверняка предполагают, что встроенные методы JavaScript будут действовать как обычно, и не ожидают столкнуться с вашими расширениями. Кроме того, свойства, добавляемые к прототипу, могут обнаруживаться в циклах, не использующих метод `hasOwnProperty()`, что чревато ошибками.

Поэтому будет лучше отказаться от идеи расширения встроенных прототипов. Исключение из правил можно сделать только при соблюдении всех следующих условий:

1. Если ожидается, что данная функциональность будет предусмотрена в будущих версиях ECMAScript или реализована в JavaScript в виде встроенных методов. Например, вы можете добавить методы, предусмотренные стандартом ECMAScript 5, пока они не появятся в броузерах. В этом случае вы просто заранее определите удобные методы.
2. Вы проверили, что ваше свойство или метод не реализовано где-либо в другом месте и не является частью реализации JavaScript в одном из броузеров, поддерживаемых вами.
3. Вы описали это расширение в документации и сообщили об изменениях всем участникам проекта.

Если все три условия соблюдены, вы можете добавить свое расширение в прототип, используя следующий шаблон:

```
if (typeof Object.protoype.myMethod !== "function") {
    Object.protoype.myMethod = function () {
        // реализация...
    };
}
```

Шаблон switch

Применяя следующий шаблон, вы сможете улучшить удобочитаемость и надежность инструкций `switch` в своих сценариях:

```
var inspect_me = 0,
    result = '';

switch (inspect_me) {
case 0:
    result = "zero";
    break;
case 1:
    result = "one";
    break;
default:
    result = "unknown";
}
```

В этом простом примере демонстрируется использование следующих соглашений по оформлению:

- Каждая инструкция `case` выравнивается по инструкции `switch` (исключение из правил оформления отступов в фигурных скобках).
- Программный код внутри каждой инструкции `case` оформляется с дополнительным отступом.
- Каждая инструкция `case` завершается явно с помощью инструкции `break;`.
- Старайтесь не пользоваться приемом «проваливания» в следующую инструкцию `case` (когда преднамеренно опускается инструкция `break`). Применяйте его, только если вы абсолютно уверены, что это наилучшее решение, при этом обязательно описывайте такие инструкции `case` в комментариях, потому что те, кто будет читать ваш код, могут воспринять отсутствие инструкции `break` как ошибку.
- Заканчивайте инструкцию `switch` веткой `default:`, чтобы гарантировать получение нормального результата даже в случае отсутствия совпадений с какой-либо из инструкций `case`.

Избегайте неявного приведения типов

При сравнивании переменных интерпретатор JavaScript неявно выполняет приведение типов переменных. Именно поэтому такие операции сравнения, как `false == 0` или `"" == 0`, возвращают `true`.

Чтобы избежать путаницы, вызванной неявным приведением типов, всегда используйте операторы `===` и `!==`, которые сравнивают и значения, и типы выражений, участвующих в операции:

```
var zero = 0;
if (zero === false) {
    // тело этой инструкции не будет выполнено,
    // потому что zero - это 0, а не false
}

// антишаблон
if (zero == false) {
    // тело этой инструкции будет выполнено
}
```

Существует другая точка зрения, согласно которой в некоторых ситуациях, когда достаточно использовать оператор `==`, оператор `===` может оказаться избыточным. Например, когда вы используете метод `typeof`, вы знаете, что он возвращает строку, поэтому в данном случае нет смысла использовать более строгий оператор сравнения. Однако JSLint требует использовать строгие версии операторов сравнения – они придают непротиворечивость программному коду и упрощают его чтение. (Не приходится думать: «данный оператор `==` использован намеренно или это упущение программиста?»)

Не используйте eval()

Если вы намереваетесь использовать функцию `eval()` в своем сценарии, вспомните мантру «`eval()` – это зло». Эта функция принимает произвольную строку и выполняет ее как программный код на JavaScript. Если заранее известно, какой программный код потребуется выполнить (то есть он не определяется во время выполнения), то нет смысла использовать функцию `eval()`. Если же программный код генерируется динамически во время выполнения сценария, зачастую есть лучший способ достичь тех же целей без применения функции `eval()`. Например, для доступа к динамическим свойствам лучше и проще использовать форму записи с квадратными скобками:

```
// антишаблон
var property = "name";
alert(eval("obj." + property));

// предпочтительный способ
var property = "name";
alert(obj[property]);
```

Кроме того, при использовании `eval()` вы ставите под удар безопасность приложения, потому что в этом случае появляется риск выполнить программный код (например, полученный из сети), измененный злоумышленником. Подобный антишаблон часто применяется для обработки ответов в формате JSON, полученных по запросам Ajax. В таких ситуациях для анализа ответов в формате JSON лучше и безопаснее использовать встроенные методы, предоставляемые броузером. Для

анализа ответов в броузерах, не имеющих собственного метода JSON.parse(), можно воспользоваться библиотекой, которая доступна на сайте *JSON.org*.

Также важно помнить, что передача строк функциям setInterval(), setTimeout() и конструктору Function() аналогична вызову функции eval(), поэтому такого их применения желательно избегать. За кулисами интерпретатор JavaScript вынужден будет проанализировать передаваемую строку и выполнить ее как программный код:

```
// антишаблон
setTimeout("myFunc()", 1000);
setTimeout("myFunc(1, 2, 3)", 1000);

// предпочтительный способ
setTimeout(myFunc, 1000);
setTimeout(function () {
    myFunc(1, 2, 3);
}, 1000);
```

Вызов конструктора new Function() по своему действию напоминает функцию eval(), и потому использовать его следует с особой осторожностью. Это очень мощный конструктор, но зачастую он применяется не совсем правильно. Если вам действительно необходимо задействовать функцию eval(), подумайте о возможности ее замены вызовом конструктора new Function(). Такой прием имеет небольшое преимущество, так как оцениваемый программный код, переданный в вызов new Function(), выполняется в локальной области видимости функции, поэтому все переменные, объявленные в этом коде с помощью инструкции var, не станут глобальными автоматически. Другой способ предотвратить автоматическое создание глобальных переменных состоит в том, чтобы обернуть вызов eval() немедленно вызываемой функцией (подробнее о немедленно вызываемых функциях рассказывается в главе 4).

Взгляните на следующий пример. Здесь только переменная un станет глобальной:

```
console.log(typeof un);                          // "undefined"
console.log(typeof deux);                        // "undefined"
console.log(typeof trois);                       // "undefined"

var jsstring = "var un = 1; console.log(un);";
eval(jsstring);                                  // выведет "1"

jsstring = "var deux = 2; console.log(deux);";
new Function(jsstring)();                        // выведет "2"

jsstring = "var trois = 3; console.log(trois);";
(function () {
    eval(jsstring);
```

```
}());                                          // выведет "3"

console.log(typeof un);                        // "number"
console.log(typeof deux);                      // "undefined"
console.log(typeof trois);                     // "undefined"
```

Еще одно отличие `eval()` от конструктора `Function` состоит в том, что функция `eval()` способна вторгаться в объемлющие области видимости, тогда как конструктор `Function` чуть более ограничен в своих возможностях. Независимо от того, где вызывается конструктор `Function`, ему будет доступна только глобальная область видимости. Поэтому он не сможет повредить локальные переменные. В следующем примере функция `eval()` может обратиться к переменной во внешней области видимости и изменить ее, тогда как конструктор `Function` лишен такой возможности (обратите также внимание, что вызов `Function()` идентичен вызову `new Function()`):

```
(function () {
    var local = 1;
    eval("local = 3; console.log(local)");    // выведет 3
    console.log(local);                       // выведет 3
}());

(function () {
    var local = 1;
    Function("console.log(typeof local);")(); // выведет undefined
}());
```

Преобразование строки в число с помощью parseInt()

Функция `parseInt()` позволяет получить число из строки. Во втором параметре функция принимает основание системы счисления, но этот параметр часто опускается, хотя делать это нежелательно. Проблемы возникают, если строка начинается с символа `0`, например, это часть даты, введенной в поле формы. Согласно ECMAScript 3, строки, начинающиеся с символа `0`, интерпретируются как восьмеричные числа (в системе счисления с основанием 8). Однако это требование изменилось в ES5. Чтобы избежать несогласованности и получения неожиданных результатов, всегда указывайте параметр, определяющий систему счисления:

```
var month = "06",
    year = "09";
month = parseInt(month, 10);
year = parseInt(year, 10);
```

Если в этом примере опустить второй параметр, то вызов `parseInt(year)` вернет значение `0`, потому что строка «09» будет воспринята как представление восьмеричного числа (как если бы была вызвана функция

parseInt(year, 8)), а цифра 9 не является допустимой в восьмеричной системе счисления.

Ниже приводятся другие варианты преобразования строки в число:

```
+"08"         // результат: 8
Number("08") // 8
```

Эти альтернативы часто выполняются быстрее, чем функция parseInt(), потому что parseInt(), как следует из ее имени, не просто преобразует строку, а анализирует ее. Однако если вы ожидаете, что входная строка может иметь вид «08 hello», то в таких случаях функция parseInt() будет возвращать число, а другие варианты преобразования будут терпеть неудачу и возвращать значение NaN.

Соглашения по оформлению программного кода

В практике очень важно определить соглашения по оформлению программного кода и неукоснительно следовать им – они обеспечивают единство стиля программного кода, делают его *предсказуемым* и более простым для чтения и понимания. Новый разработчик, присоединяющийся к коллективу, сможет ознакомиться с соглашениями и быстрее влиться в общую работу, получив ключ к пониманию программного кода, написанного другими членами коллектива.

На встречах и в списках рассылки не раз разгорались жаркие споры по поводу отдельных аспектов оформления программного кода (например, как оформлять отступы – символами табуляции или пробелами?). Поэтому, если вы предлагаете утвердить некоторые соглашения по оформлению программного кода в своей организации, готовьтесь столкнуться с сопротивлением и услышать противоположные, не менее сильные аргументы. Помните, что важнее выработать соглашения, любые соглашения, и следовать им, чем настаивать на конкретных деталях самих соглашений.

Отступы

Программный код без отступов невозможно читать. Хуже этого может быть только программный код, где отступы используются непоследовательно, – наличие отступов вроде бы говорит о следовании соглашениям, но в таком коде вас могут ждать неприятные сюрпризы. Важно точно определить порядок оформления отступов.

Некоторые разработчики предпочитают оформлять отступы символами табуляции, потому что любой сможет настроить свой текстовый редактор так, чтобы отступы отображались в нем в соответствии с личными предпочтениями. Некоторые предпочитают пробелы – обычно четыре пробела. Неважно, какой способ выберете вы, важно, чтобы все члены коллектива следовали одному и тому же соглашению. В этой

книге, например, используются отступы по четыре пробела. Точно такая же величина отступа принята по умолчанию в JSLint.

Какие фрагменты должны оформляться с отступами? Здесь все просто – все, что находится внутри фигурных скобок. То есть тела функций, циклов (`do`, `while`, `for`, `for-in`), условных инструкций `if`, инструкций выбора `switch` и определения свойств объектов при записи в форме литерала. Ниже приводится несколько примеров употребления отступов:

```
function outer(a, b) {
    var c = 1,
        d = 2,
        inner;
    if (a > b) {
        inner = function () {
            return {
                r: c - d
            };
        };
    } else {
        inner = function () {
            return {
                r: c + d
            };
        };
    }
    return inner;
}
```

Фигурные скобки

Всегда используйте фигурные скобки, даже когда они являются необязательными. С технической точки зрения, если тело инструкции `if` или `for` состоит из единственной инструкции, фигурные скобки можно опустить, но тем не менее желательно, чтобы вы всегда использовали их. Это делает программный код более целостным и упрощает внесение изменений в будущем.

Представьте, что имеется цикл `for`, тело которого состоит из единственной инструкции. Вы могли бы опустить фигурные скобки, и это не было бы ошибкой:

```
// плохой пример для подражания
for (var i = 0; i < 10; i += 1)
    alert(i);
```

Но что если позднее вам потребуется добавить еще одну строку в тело цикла?

```
// плохой пример для подражания
for (var i = 0; i < 10; i += 1)
```

```
    alert(i);
    alert(i + " is " + (i % 2 ? "odd" : "even"));
```

Второй вызов `alert()` находится за пределами цикла, хотя наличие отступа может обмануть вас. Поэтому лучше использовать фигурные скобки всегда, даже когда тело составной инструкции состоит из единственной строки:

```
// лучше
for (var i = 0; i < 10; i += 1) {
    alert(i);
}
```

То же относится и к условным инструкциям `if`:

```
// плохо
if (true)
    alert(1);
else
    alert(2);

// лучше
if (true) {
    alert(1);
} else {
    alert(2);
}
```

Местоположение открывающей скобки

Каждый разработчик также имеет свое мнение о том, где должна находиться открывающая фигурная скобка – в той же строке, что и основная инструкция, или в следующей.

```
if (true) {
    alert("It's TRUE!");
}
```

Или:

```
if (true)
{
    alert("It's TRUE!");
}
```

В данном конкретном примере местоположение открывающей скобки – вопрос личных предпочтений, но в некоторых ситуациях программа может проявлять различное поведение в зависимости от того, где находится открывающая фигурная скобка. Такое отличие обусловлено действием *механизма подстановки точки с запятой* – в языке JavaScript допускается опускать символ точки с запятой в конце строки; она будет добавлена автоматически. Такое поведение интерпретатора может

вызывать проблемы, когда функция, например, должна вернуть литерал объекта, а открывающая фигурная скобка находится в следующей строке:

```
// Внимание: функция возвращает неожиданное значение
function func() {
    return
    {
        name: “Batman”
    };
}
```

Если вы полагаете, что эта функция вернет объект со свойством `name`, то будете удивлены. Из-за того что в конце инструкции `return` подразумевается точка с запятой, функция вернет значение `undefined`. Интерпретатор воспримет предыдущий фрагмент, как если бы он был записан так:

```
// Внимание: функция возвращает неожиданное значение
function func() {
    return undefined;
    // следующий код никогда не будет выполнен...
    {
        name: “Batman”
    };
}
```

Поэтому всегда используйте фигурные скобки и всегда помещайте открывающую скобку в конец строки с предыдущей инструкцией:

```
function func() {
    return {
        name: “Batman”
    };
}
```

Что касается точек с запятой: всегда используйте точки с запятой, как и фигурные скобки, даже там, где они подразумеваются интерпретатором JavaScript. Это не только дисциплинирует и обеспечивает более строгий подход к программированию, но и помогает разрешать неоднозначности, подобные продемонстрированным выше.

Пробелы

Правильное использование пробелов также может повысить удобочитаемость и целостность программного кода. При письме вы добавляете пробелы после точек и запятых. В программном коде на JavaScript вы можете следовать той же логике и добавлять пробелы после выражений-перечней (эквивалентных письму через запятую) и завершающих ин-

струкции точек с запятой (которые можно сравнить с точкой в предложениях, выражающих «законченную мысль»).

Желательно вставлять дополнительные пробелы в следующих местах:

- После точек с запятой, отделяющих части инструкции `for`, например: `for (var i = 0; i < 10; i += 1) {...}`
- При инициализации нескольких переменных (`i` и `max`) в цикле `for`: `for (var i = 0, max = 10; i < max; i += 1) {...}`
- После запятых, отделяющих элементы массива: `var a = [1, 2, 3];`
- После запятых, отделяющих описания свойств объекта, и после двоеточий, отделяющих имена свойств от их значений: `var o = {a: 1, b: 2};`
- После запятых, отделяющих аргументы функции: `myFunc(a, b, c)`
- Перед фигурными скобками в объявлениях функций: `function myFunc() {}`
- После ключевого слова `function` в определениях анонимных функций: `var myFunc = function () {};`

Было бы совсем неплохо отделять пробелами операторы от операндов, то есть добавлять пробелы до и после `+`, `-`, `*`, `=`, `<`, `>`, `<=`, `>=`, `===`, `!==`, `&&`, `||`, `+=` и так далее:

```
// последовательное использование пробелов
// делает программный код более простым для чтения,
// позволяя ему "дышать"
var d = 0,
    a = b + 1;
if (a && b && c) {
    d = a % c;
    a += d;
}

// антишаблон
// отсутствие или непоследовательное использование пробелов
// усложняет восприятие программного кода
var d= 0,
    a =b+1;
if (a&& b&&c) {
    d=a %c;
    a+= d;
}
```

И последнее замечание, касающееся пробелов: употребляйте пробелы вместе с фигурными скобками:

- Перед открывающими фигурными скобками ({) в функциях, в инструкциях `if-else`, `switch`, в циклах и в литералах объектов.
- Между закрывающей фигурной скобкой (}) и инструкциями `else` и `while`.

Веским аргументом против использования пробелов может быть увеличение размера файла, однако эта проблема решается за счет применения приема сжатия (обсуждается ниже в этой главе).

Многие часто упускают из виду такой важный аспект, влияющий на удобочитаемость программного кода, как пустые строки. Вы можете использовать их для визуального отделения блоков программного кода, что подобно разбивке на абзацы в обычных книгах.

Соглашения по именованию

Еще один способ сделать программный код более предсказуемым и простым в сопровождении заключается в использовании соглашений по именованию, то есть соглашений о выборе имен для переменных и функций.

Ниже приводятся некоторые предложения по соглашениям об именовании, которые вы могли бы или принять в том виде, в каком они даются здесь, или доработать в соответствии со своими предпочтениями. Еще раз напомню, что наличие соглашений и следование им намного важнее конкретных деталей, содержащихся в соглашениях.

Заглавные символы в именах конструкторов

В языке JavaScript отсутствуют классы, но в нем имеются функции-конструкторы, которые вызываются вместе с оператором `new`:

```
var adam = new Person();
```

Поскольку конструкторы – это тоже функции, было бы неплохо иметь подсказку, которая позволила бы, глядя на имя функции, сразу сказать, действует она как конструктор или как обычная функция.

Такой подсказкой служит первый заглавный символ в именах конструкторов. Если первым в имени функции или метода используется символ нижнего регистра, это говорит о том, что данная функция не может использоваться совместно с оператором `new`:

```
function MyConstructor() {...}
function myFunction() {...}
```

В следующей главе приводятся несколько шаблонов, дающих возможность программно наделить свои конструкторы чертами поведения конструкторов; даже простое следование предлагаемому соглашению окажется полезным, по крайней мере для тех, кто будет читать исходные тексты.

Выделение слов

Когда имя переменной или функции составляется из нескольких слов, было бы неплохо выработать соглашение, позволяющее выделить на-

чало каждого слова. Обычно для этого используется соглашение о *смене регистра символов* (*camel case*). В соответствии с этим соглашением символы слов вводятся в нижнем регистре, а первые символы каждого слова – в верхнем.

Для имен конструкторов можно применить запись *с первым заглавным символом* (в верхнем регистре), например `MyConstructor()`. А для имен обычных функций и методов – запись имени *с первым строчным символом* (в нижнем регистре), например `myFunction()`, `calculateArea()` и `getFirstName()`.

А какие имеются соглашения по именованию переменных, не являющихся функциями? Разработчики обычно используют запись имен переменных с первым строчным символом (в нижнем регистре), но с таким же успехом можно записывать все символы в именах переменных строчными символами, разделяя слова символом подчеркивания, например `first_name`, `favorite_bands` и `old_company_name`. Такая форма записи позволяет визуально отделять функции от всех остальных идентификаторов – переменных простых типов и объектов.

В именах методов и свойств стандарт ECMAScript использует соглашение о смене регистра, однако имена, состоящие из нескольких слов, встречаются довольно редко (свойства `lastIndex` и `ignoreCase` объектов регулярных выражений).

Другие шаблоны именования

Иногда соглашения об именовании используются разработчиками для имитации тех или иных особенностей языка.

Например, в языке JavaScript отсутствует возможность объявлять константы (хотя существует несколько встроенных констант, таких как `Number.MAX_VALUE`), поэтому разработчики выработали соглашение записывать имена переменных, значения которых не должны изменяться в течение всего времени работы сценария, только заглавными символами, например:

```
// драгоценные константы, не изменяйте их
var PI = 3.14,
    MAX_WIDTH = 800;
```

Существует еще одно соглашение об использовании имен, состоящих исключительно из заглавных символов, – для обозначения глобальных переменных. Следование этому соглашению способствует уменьшению количества глобальных переменных и делает их легко заметными в программном коде.

Другой пример использования соглашений для имитации функциональных возможностей – соглашение об именовании частных членов. В языке JavaScript имеется возможность реализовать истинное сокрытие, но иногда разработчики считают, что проще использовать началь-

ный символ подчеркивания в именах, чтобы обозначить частный метод или свойство. Взгляните на следующий пример:

```
var person = {
    getName: function () {
        return this._getFirst() + ' ' + this._getLast();
    },
    _getFirst: function () {
        // ...
    },
    _getLast: function () {
        // ...
    }
};
```

В этом примере подразумевается, что `getName()` – это общедоступный метод, являющийся частью официального API, а методы `_getFirst()` и `_getLast()` – частные. Они также являются обычными общедоступными методами, но использование начального символа подчеркивания предупреждает пользователей объекта `person`, что наличие этих методов не гарантируется в следующих версиях и они не должны использоваться непосредственно. Имейте в виду, что JSLint будет выдавать предупреждения для имен, начинающихся с символа подчеркивания, если не установить параметр настройки `nomen: false`.

Ниже приводятся некоторые вариации соглашения `_private`:

- Для обозначения частных имен использовать завершающий символ подчеркивания, как в именах `name_` и `getElements_()`
- Для обозначения защищенных имен использовать один символ подчеркивания (например `_protected`), а для обозначения частных имен – два (например `__private`)
- В броузере Firefox доступны некоторые внутренние переменные, не являющиеся частью языка. В именах этих переменных используются по два символа подчеркивания в начале и в конце, например `__proto__` и `__parent__`

Комментарии

Вы обязательно должны комментировать свой программный код, даже если вы уверены, что никто другой никогда не увидит его. Часто, углубившись в решение проблемы, вы полагаете очевидным то, что делает тот или иной фрагмент сценария, но спустя всего неделю вы уже с трудом можете вспомнить, как действует данный фрагмент.

Не нужно перегибать палку и комментировать действительно очевидные вещи: каждую переменную или каждую строку кода. Но вам определенно стоит добавлять описания функций, их аргументов и возвращаемых значений, а также всех интересных или необычных алгоритмов

и приемов. Рассматривайте комментарии как подсказки для тех, кому придется читать ваш код в будущем, – они должны иметь возможность понять, что делает ваш код, опираясь лишь на комментарии и имена функций и свойств. Если у вас в сценарии имеется, к примеру, пять или шесть строк программного кода, который решает определенную задачу, читатель сможет пропустить эти строки, не вникая в детали, если вы напишете однострочный комментарий, описывающий *назначение* этого фрагмента. Нет никаких жестких и точных правил, определяющих порядок комментирования программного кода; описание некоторых фрагментов (таких как регулярные выражения) в действительности может быть длиннее, чем сам программный код.

Самое важное, но и самое сложное – заставить себя обновлять комментарии, потому что устаревшие комментарии могут вводить в заблуждение, что еще хуже, чем полное отсутствие комментариев.

Комментарии, как вы увидите в следующем разделе, способны даже помочь в создании документации, генерируемой автоматически.

Документирование API

Большинство разработчиков считают создание документации весьма скучным и неблагодарным делом. Но в действительности это может быть и не так. Документация с описанием прикладного интерфейса (API) может быть сгенерирована из комментариев в программном коде. Благодаря такому способу вы сможете получить документацию, фактически не писав ее. Многие программисты находят эту идею просто изумительной, потому что создание удобочитаемого справочника из набора определенных ключевых слов и специальных «команд» во многом напоминает программирование.

Первоначально такой подход к созданию документации с описанием API зародился в мире Java, где имеется утилита с именем javadoc, распространяемая в составе Java SDK (Software Developer Kit – набор инструментальных средств разработчика). Эта идея была перенесена во многие другие языки программирования. И для JavaScript имеется два замечательных инструмента, оба распространяются бесплатно и с открытыми исходными текстами: JSDoc Toolkit (*http://code.google.com/p/jsdoc-toolkit/*) и YUIDoc (*http://yuilibrary.com/projects/yuidoc*).

Процесс создания документации с описанием API включает в себя:

- Создание специально отформатированных блоков кода
- Запуск инструмента, выполняющего анализ программного кода и комментариев
- Вывод результатов работы инструмента, обычно в формате HTML

Специальный синтаксис, который вам придется изучить, насчитывает около десятка тегов и имеет следующий вид:

```
/**
 * @tag value
 */
```

Например, допустим, что имеется функция с именем `reverse()`, которая переворачивает строку. Она принимает строковый параметр и возвращает другую строку. Описание этой функции могло бы выглядеть так:

```
/**
 * Reverse a string
 *
 * @param {String} input String to reverse
 * @return {String} The reversed string
 */
var reverse = function (input) {
    // ...
    return output;
};
```

В этом примере тег `@param` используется для описания входных параметров, а тег `@return` – для описания возвращаемых значений. Инструмент, генерирующий документацию, проанализирует эти теги и после этого создаст набор хорошо отформатированных документов в формате HTML.

Пример использования YUIDoc

Первоначально инструмент YUIDoc создавался с целью создания документации для библиотеки YUI (Yahoo! User Interface – пользовательский интерфейс Yahoo!), но он с успехом может использоваться в любых других проектах. Инструмент определяет несколько соглашений, которым вы должны следовать при его использовании, касающихся, например, понятий модулей и классов. (Несмотря на то, что в JavaScript отсутствует понятие класса.)

Давайте рассмотрим законченный пример создания документации с помощью YUIDoc.

На рис. 2.1 демонстрируется, как будет выглядеть документация, которую вы получите в конце. Фактически у вас есть возможность скорректировать шаблон HTML, чтобы привести его в соответствие с требованиями вашего проекта и изменить оформление страниц.

Демонстрационную реализацию этого примера вы найдете по адресу *http://jspatterns.com/book/2/*.

В этом примере все приложение находится в единственном файле (*app.js*) с единственным модулем (`myapp`) в нем. Подробнее о модулях рассказывается в последующих главах, а пока будем считать модулем тег в комментарии, который распознается инструментом YUIDoc.

Рис. 2.1. *Документация, сгенерированная инструментом YUIDoc*

Содержимое файла *app.js* начинается со следующего комментария:

```
/**
 * My JavaScript application
 *
 * @module myapp
 */
```

Затем определяется пустой объект, который будет играть роль пространства имен:

```
var MYAPP = {};
```

Далее определяется объект `math_stuff`, обладающий двумя методами: `sum()` и `multi()`:

```
/**
 * A math utility
 * @namespace MYAPP
 * @class math_stuff
 */
MYAPP.math_stuff = {
```

```
    /**
     * Sums two numbers
     *
     * @method sum
     * @param {Number} a First number
     * @param {Number} b The second number
     * @return {Number} The sum of the two inputs
     */
    sum: function (a, b) {
        return a + b;
    },

    /**
     * Multiplies two numbers
     *
     * @method multi
     * @param {Number} a First number
     * @param {Number} b The second number
     * @return {Number} The two inputs multiplied
     */
    multi: function (a, b) {
        return a * b;
    }
};
```

На этом завершается определение первого «класса». Обратите внимание на выделенные теги:

`@namespace`

Глобальная ссылка на ваш объект.

`@class`

Не совсем корректное имя (в языке JavaScript нет классов), используемое для обозначения объекта или функции-конструктора.

`@method`

Объявляет метод объекта и определяет его имя.

`@param`

Перечисляет аргументы, принимаемые функцией. Типы параметров определяются в фигурных скобках, за которыми следуют имена и описания параметров.

`@return`

Напоминает тег `@param`, но описывает значение, возвращаемое методом, которое не имеет имени.

В качестве второго «класса» используется функция-конструктор, к прототипу которой добавляются дополнительные методы. Это сделано лишь для того, чтобы вы могли понять, как описывать объекты, создаваемые различными способами:

```
/**
 * Constructs Person objects
 * @class Person
 * @constructor
 * @namespace MYAPP
 * @param {String} first First name
 * @param {String} last Last name
 */
MYAPP.Person = function (first, last) {
    /**
     * Name of the person
     * @property first_name
     * @type String
     */
    this.first_name = first;
    /**
     * Last (family) name of the person
     * @property last_name
     * @type String
     */
    this.last_name = last;
};

/**
 * Returns the name of the person object
 *
 * @method getName
 * @return {String} The name of the person
 */
MYAPP.Person.prototype.getName = function () {
    return this.first_name + ' ' + this.last_name;
};
```

Как выглядит сгенерированная документация с описанием конструктора `Person`, можно видеть на рис. 2.1. В листинге были выделены следующие элементы:

- `@constructor` указывает, что этот «класс» в действительности является функцией-конструктором
- `@property` и `@type` описывают свойства объекта

Система YUIDoc нечувствительна к языку программирования – она анализирует только блоки комментариев и не обращает внимания на программный код JavaScript. Недостатком такого подхода является необходимость явно указывать имена свойств, параметров и методов в комментариях, например `@property first_name`. Преимущество же заключается в том, что привыкнув к этому инструменту, вы сможете применять его для создания документации к программам на любом языке программирования.

Пишите так, чтобы можно было читать

Создание комментариев с описанием API – это не только ленивый способ создания справочной документации; он служит еще одной цели – повышению качества программного кода, заставляя вас еще раз просмотреть свой код.

Любой писатель или редактор скажет вам, что этап редактирования имеет важное значение: это, возможно, самый важный этап в подготовке книги или статьи. Создание рукописи – это лишь первый шаг. Рукопись представляет некоторую информацию читателю, но зачастую не самым ясным и простым способом.

То же относится и к программному коду. Когда вы садитесь и решаете какую-то проблему, это решение является всего лишь первым, черновым вариантом. Реализованное решение генерирует желаемый результат, но насколько это решение оптимально? Насколько легко будет читать, понимать, сопровождать и обновлять программный код? Вернувшись к программному коду – желательно спустя некоторое время, – вы практически наверняка обнаружите участки, которые можно было бы улучшить – изменить код так, чтобы его было проще читать, повысить эффективность и так далее. Это, по сути, и есть этап редактирования, который может помочь вам в достижении вашей цели – создание высококачественного программного кода. Однако очень часто мы работаем в условиях жестко ограниченного времени («есть проблема и решить ее надо вчера»), и, как правило, у нас не остается времени на редактирование. Именно поэтому создание документации с описанием API дает дополнительную возможность редактирования.

Часто при создании блоков комментариев с описанием приходится вновь погружаться в проблему. Иногда при таком повторном погружении становится очевидным, например, что вот этот третий параметр метода бывает необходим гораздо чаще, чем второй, а второй параметр практически всегда получает значение по умолчанию `true`, поэтому есть смысл изменить интерфейс метода, поменяв параметры местами.

Фраза «писать так, чтобы можно было читать» означает, что вы должны писать программный код или, по крайней мере, API, помня о том, что кто-то будет его читать. Одна только мысль об этом заставит вас редактировать код и находить более удачные решения стоящих перед вами проблем.

Говоря о первых, черновых вариантах решения, стоит заметить, что часто они пишутся, «чтобы потом выбросить их». На первый взгляд эта идея кажется абсурдной, но она обретает особый смысл, особенно если вы работаете над критически важным приложением (от которого могут зависеть человеческие жизни). Суть идеи состоит в том, что первое найденное решение должно быть отброшено и необходимо приступить

к поиску другого решения, с нуля. Первое решение может быть вполне рабочим, но это не более чем черновик, один из примеров решения проблемы. Второе решение всегда получается лучше, потому что к началу его разработки вы глубже понимаете проблему. При создании второго решения вы не должны допускать копирования кусков из первого решения; это не позволит вам пойти наиболее простым путем и согласиться на неидеальное решение.

Оценка коллегами

Другой способ улучшить программный код – попросить коллег оценить его. Такого рода оценка, которая может быть формализована и стандартизована и может выполняться даже с привлечением специализированных инструментов, вполне способна стать частью общего процесса разработки. Ни нехватка времени, ни отсутствие специализированных инструментов не должны быть препятствием. Вы можете просто попросить коллегу взглянуть на ваш код или продемонстрировать работу кода, сопровождая показ комментариями.

Так же как написание любой документации, оценка коллег заставит вас писать более ясный программный код просто потому, что вы будете знать, что кому-то придется прочитать его и понять, что он делает.

Оценка коллегами рекомендуется не только потому, что она способствует улучшению программного кода, но и потому, что в процессе такой оценки происходит обмен знаниями и изучается опыт и индивидуальные подходы других разработчиков.

Если вы работаете в одиночку и рядом с вами нет коллег, которые могли бы оценить ваш код, это также не должно быть препятствием. Вы всегда можете открыть хотя бы какую-то часть исходных текстов или просто выложить в своем блоге наиболее интересующие вас фрагменты, и тогда их смогут оценить люди во всем мире.

Еще один отличный способ получить оценку коллег – настроить свою систему управления версиями (CVS, Subversion или Git), чтобы она отправляла коллегам извещения по электронной почте при сохранении в репозитории изменений в программном коде. Большая часть этих электронных писем так и останутся непрочитанными, но вполне может статься, что кто-то из ваших коллег решит оторваться от работы и взглянуть на код, который вы только что отправили в репозиторий.

Сжатие... при подготовке к эксплуатации

Сжатие – это процесс удаления пробелов, комментариев и других несущественных фрагментов из программного кода на JavaScript с целью уменьшить размеры файлов, которые необходимо будет передавать по сети от сервера к броузеру. Обычно такое сжатие выполняется с привле-

чением специализированных инструментов (компрессоров), таких как Yahoo! YUICompressor или Google Closure Compiler, и помогает уменьшить время загрузки страниц. Очень важно выполнять процедуру сжатия сценариев перед передачей их в эксплуатацию, потому что она обеспечивает существенную экономию, зачастую уменьшая размеры файлов наполовину.

Ниже приводится пример программного кода, прошедшего процедуру сжатия (это часть утилиты Event из библиотеки YUI2):

```
YAHOO.util.CustomEvent=function(D,C,B,A){this.type=D;thisscope=C||window;this.
silent=B;this.signature=A||YAHOO.util.CustomEvent.LIST;this.subscribers=[];if
(!this.silent){}var E="_YUICEOnSubscribe";if(D!==E){this.subscribeEvent=new
YAHOO.util.CustomEvent(E,this,true);}...
```

Помимо удаления пробелов, символов перевода строки и комментариев, компрессоры также дают переменным более короткие имена (но только если это допустимо), как, например, имена параметров `D`, `C`, `B`, `A` в предыдущем фрагменте. Компрессоры могут переименовывать только локальные переменные, потому что переименование глобальных переменных может привести к нарушениям в работе сценария. Именно поэтому рекомендуется использовать локальные переменные везде, где только возможно. Если внутри функции вы используете глобальную переменную, например ссылку на элемент DOM, не раз и не два, есть смысл присвоить ее значение локальной переменной. Помимо экономии пространства и уменьшения времени загрузки, это увеличит скорость поиска переменной по ее имени в процессе интерпретации, благодаря чему сжатый программный код будет выполняться немного быстрее.

Следует также отметить, что Google Closure Compiler может попытаться сжать имена глобальных переменных (при работе в «расширенном» режиме), что довольно рискованно и потребует от вас особого внимания в обмен на более высокую степень сжатия.

Сжатие программного кода перед передачей в эксплуатацию имеет большое значение, потому что помогает увеличить производительность страниц, но эту работу должны выполнять компрессоры. Будет большой ошибкой пытаться писать сценарии уже в сжатом виде. Вы всегда должны использовать описательные имена переменных, добавлять пробелы, отступы, комментарии и так далее. Код, который вы пишете, должен легко читаться (людьми), чтобы тот, кто будет сопровождать его, смог быстро разобраться в нем, а сжатие следует доверить компрессору (машине).

Запуск JSLint

Инструмент JSLint уже был представлен в предыдущей главе и не раз упоминался в этой. Сейчас вы наверняка согласитесь, что проверка программного кода с его помощью – хороший шаблон программирования.

Что же делает JSLint в процессе выполнения? Он отыскивает нарушения некоторых шаблонов программирования, обсуждавшихся в этой главе (шаблон единственного объявления `var`, использование основания системы счисления в функции `parseInt()`, повсеместное использование фигурных скобок), и многие другие, включая следующие:

- Программный код, который никогда не выполняется
- Обращения к переменным до их объявления
- Использование недопустимых символов UTF
- Использование `void`, `with` или `eval`
- Некорректное экранирование символов в регулярных выражениях

Инструмент JSLint написан на JavaScript (и наверняка сам прошел бы проверку с помощью JSLint). Самое интересное, что он доступен в двух вариантах – как веб-инструмент и как самостоятельная утилита для многих платформ и интерпретаторов JavaScript. Вы можете загрузить его и запускать локально на любой из платформ, где имеется WSH (Windows Scripting Host – среда выполнения сценариев для Windows – компонент, входящий в состав всех операционных систем Windows), JSC (JavaScriptCore, компонент операционной системы Mac OSX) или Rhino (интерпретатор JavaScript, выпущенный компанией Mozilla).

Было бы просто отлично загрузить JSLint и интегрировать его в свой текстовый редактор, чтобы выработать в себе привычку запускать его перед каждым сохранением файла. (Неплохо было бы назначить для запуска JSLint комбинацию клавиш.)

В заключение

В этой главе мы узнали, что значит писать простой в сопровождении программный код. Эта тема имеет большое значение не только для успеха проекта, но также для душевного спокойствия и благополучия разработчиков и всех, кто их окружает. Затем мы поговорили о некоторых основных приемах и шаблонах, таких как:

- Снижение количества глобальных переменных, в идеале – не более одной на приложение.
- Использование единственного объявления `var` в функциях, что позволяет одним взглядом охватить все переменные и предотвращает появление неожиданностей, вызванных особенностями механизма подъема переменных.
- Циклы `for`, циклы `for-in`, инструкции `switch`, «`eval()` – это зло», нежелательность расширения прототипов.
- Следование соглашениям по оформлению программного кода (последовательное использование пробелов и отступов; использование

фигурных скобок и точек с запятой даже там, где они являются необязательными) и соглашениям по именованию (конструкторов, функций и переменных).

Кроме того, в этой главе мы рассмотрели некоторые дополнительные приемы, не связанные с программным кодом, а имеющие отношение скорее к процессу разработки в целом, – добавление комментариев, создание документации с описанием API, оценка программного кода коллегами, недопустимость попыток писать сжатый программный код в ущерб удобочитаемости и постоянная проверка программного кода с помощью JSLint.

3

Литералы и конструкторы

Возможность определения объектов в форме литералов, имеющаяся в языке JavaScript, обеспечивает более краткий, более выразительный и менее подверженный ошибкам способ записи определений объектов. В этой главе рассматриваются литералы объектов, массивов и регулярных выражений, а также поясняется, почему эта форма определения предпочтительнее использования эквивалентных встроенных функций-конструкторов, таких как `Object()` и `Array()`. Дополнительно будет представлен формат **JSON с целью продемонстрировать, как можно использовать** литералы массивов и объектов для определения формата передачи данных. Кроме того, в этой главе будут рассматриваться собственные конструкторы и принудительное использование оператора `new` с целью обеспечить корректное поведение конструкторов.

В дополнение к главной идее главы (которая состоит в том, чтобы побудить вас использовать литералы вместо конструкторов) мы рассмотрим встроенные конструкторы-обертки `Number()`, `String()` и `Boolean()` и сравним их с элементарными числовыми, строковыми и логическими значениями. В заключение кратко обсуждается использование различных встроенных конструкторов `Error()`.

Литералы объектов

Объекты **JavaScript можно представлять себе как хеш-таблицы, содержащие** пары ключ-значение (напоминающие конструкции, которые в других языках называются «ассоциативными массивами»). Значениями могут быть данные простых типов или другие объекты – в обоих случаях они называются *свойствами*. Кроме того, значениями могут быть функции, и в этом случае они называются *методами*.

Объекты, которые вы создаете сами (другими словами, *объекты языка, определяемые пользователем*), постоянно доступны для изменения. Многие из свойств таких объектов также доступны для изменения. Вы можете для начала создать пустой объект, а затем постепенно добавлять в него новую функциональность. Форма определения объектов в виде литералов идеально подходит для создания таких постепенно расширяемых объектов.

Взгляните на следующий пример:

```
// сначала создается пустой объект
var dog = {};

// затем к нему добавляется одно свойство
dog.name = "Benji";

// а потом метод
dog.getName = function () {
    return dog.name;
};
```

В предыдущем примере мы начали создавать объект с чистого листа, создав пустой объект. Затем мы добавили к нему свойство и метод. На любом этапе выполнения программы вы можете:

- Изменять свойства и методы, например:

```
dog.getName = function () {
    // переопределить метод так, чтобы он возвращал
    // одно и то же значение
    return "Fido";
};
```

- Удалять свойства и методы:

```
delete dog.name;
```

- Добавлять дополнительные свойства и методы:

```
dog.say = function () {
    return "Woof!";
};
dog.fleas = true;
```

Начинать с создания пустого объекта не обязательно. Литералы позволяют добавлять необходимую функциональность еще на этапе создания, как показано в следующем примере.

```
var dog = {
    name: "Benji",
    getName: function () {
        return this.name;
    }
};
```

В этой книге вам еще не раз будет встречаться фраза «пустой объект». Важно понимать, что это делается для упрощения описания, и в действительности в JavaScript нет пустых объектов. Даже простейший объект {} уже имеет свойства и методы, унаследованные от `Object.prototype`. Под «пустым» мы будем подразумевать объект, не имеющий других свойств, кроме унаследованных.

Синтаксис литералов объектов

Если прежде вы не использовали форму определения объектов в виде литералов, первое время она будет казаться немного странной. Но чем чаще вы будете применять ее, тем больше она будет вам нравиться. Ниже перечислены основные правила синтаксиса литералов объектов:

- Определение объекта заключается в фигурные скобки { и }.
- Определения свойств и методов внутри объекта разделяются запятыми. После последней пары имя-значение допускается добавлять последнюю запятую, однако она будет вызывать ошибку в IE, поэтому не используйте ее.
- Имена свойств отделяются от значений двоеточиями.
- При присваивании литерала объекта переменной не забудьте точку с запятой после закрывающей скобки }.

Создание объектов с помощью конструкторов

В языке JavaScript отсутствуют классы, и это обстоятельство обеспечивает значительную гибкость, так как не требуется заранее знать что-либо о своем объекте – вам не нужен класс, который будет играть роль «кальки». Но в JavaScript существуют функции-конструкторы, для вызова которых используется синтаксис, заимствованный из других объектно-ориентированных языков, таких как Java.

Вы можете создавать объекты, используя свои собственные функции-конструкторы или встроенные конструкторы, такие как `Object()`, `Date()`, `String()` и так далее.

В следующем примере демонстрируются два эквивалентных способа создания двух идентичных объектов:

```
// первый способ -- с помощью литерала
var car = {goes: "far"};

// другой способ -- с помощью встроенного конструктора
// внимание: это антишаблон
var car = new Object();
car.goes = "far";
```

В этом примере сразу бросается в глаза одно из преимуществ литералов – более краткая форма записи. Еще одна причина, по которой литералы являются *более предпочтительным способом создания объектов*, состоит в том, что такая форма записи лишний раз подчеркивает, что объекты – это всего лишь хеши, доступные для изменения, а не нечто особенное, которое требуется «выпекать» по определенному «рецепту» (на основе класса).

Другое преимущество использования литералов перед конструктором `Object()` заключается в отсутствии необходимости разрешения имен в разных областях видимости. Поскольку в локальной области видимости может быть создан локальный конструктор с тем же именем, интерпретатору придется просматривать всю цепочку областей видимости от места вызова `Object()`, пока не будет найден глобальный конструктор `Object()`.

Недостатки конструктора Object

Нет никаких причин использовать конструктор `new Object()`, когда есть возможность создавать объекты в виде литералов, но вы можете столкнуться с устаревшим программным кодом, написанным другими разработчиками, поэтому вы должны знать об одной «особенности» этого конструктора (и познакомиться с еще одной причиной против его использования). Рассматриваемая особенность заключается в том, что конструктор `Object()` принимает параметр и в зависимости от его значения может делегировать создание объекта другому встроенному конструктору, вернув в результате объект не того типа, который вы ожидаете.

Ниже приводятся несколько примеров передачи числа, строки и логического значения в вызов `new Object()`. В результате в каждом случае возвращаются объекты, созданные различными конструкторами:

```
// Внимание: все эти примеры являются антишаблонами

// пустой объект
var o = new Object();
console.log(o.constructor === Object);   // true

// объект-число
var o = new Object(1);
console.log(o.constructor === Number);   // true
console.log(o.toFixed(2));               // "1.00"

// объект-строка
var o = new Object("I am a string");
console.log(o.constructor === String);   // true

// обычные объекты не имеют метода substring()
// зато этот метод имеется у объектов-строк
console.log(typeof o.substring);         // "function"
```

```
// логический объект
var o = new Object(true);
console.log(o.constructor === Boolean);  // true
```

Такое поведение конструктора `Object()` может приводить к неожиданным результатам, когда значение, передаваемое ему, генерируется динамически и тип его заранее неизвестен. Еще раз призываю вас не использовать конструктор `new Object()`. Применяйте более простую и надежную форму записи объектов в виде литералов.

Собственные функции-конструкторы

Кроме использования литералов и встроенных функций-конструкторов объекты можно создавать с помощью собственных функций-конструкторов, как демонстрируется в следующем примере:

```
var adam = new Person("Adam");
adam.say();      // "I am Adam"
```

Этот новый шаблон очень напоминает создание объектов в языке Java с использованием класса `Person`. Синтаксис очень похож, но в действительности в языке JavaScript отсутствуют классы, и `Person` – это всего лишь функция.

Ниже приводится одна из возможных реализаций функции-конструктора `Person`.

```
var Person = function (name) {
    this.name = name;
    this.say = function () {
        return "I am " + this.name;
    };
};
```

При вызове функции-конструктора с оператором `new` внутри функции происходит следующее:

- Создается пустой объект, наследующий свойства и методы прототипа функции, и ссылка на него сохраняется в переменной `this`.
- Добавление новых свойств и методов в объект осуществляется с помощью ссылки `this`.
- В конце функция неявно возвращает объект, на который ссылается переменная `this` (если явно не возвращается никакой другой объект).

Вот как примерно выглядит то, что происходит за кулисами:

```
var Person = function (name) {

    // создается пустой объект
    // с использованием литерала
    // var this = {};
```

```
    // добавляются свойства и методы
    this.name = name;
    this.say = function () {
        return "I am " + this.name;
    };

    // return this;
};
```

Для простоты примера метод `say()` был добавлен к ссылке `this`. В результате при каждом вызове конструктора `new Person()` в памяти будет создаваться новая функция. Совершенно очевидно, что это неэкономный подход к расходованию памяти, потому что реализация метода `say()` не изменяется от одного экземпляра к другому. Эффективнее было бы добавить метод к прототипу функции `Person`:

```
Person.prototype.say = function () {
    return "I am " + this.name;
};
```

Подробнее о прототипах мы поговорим в следующих главах, а пока просто запомните, что члены, общие для всех экземпляров, такие как методы, следует добавлять к прототипу.

Существует еще кое-что заслуживающее упоминания для полноты картины. Выше было сказано, что внутри конструктора, за кулисами, выполняется такая операция:

```
// var this = {};
```

Это не совсем так, потому что «пустой» объект в действительности не является пустым – он наследует свойства и методы от прототипа функции `Person`. То есть точнее было бы эту операцию представить так:

```
// var this = Object.create(Person.prototype);
```

Поближе с методом `Object.create()` мы познакомимся далее в этой книге.

Значения, возвращаемые конструкторами

При вызове с оператором `new` функция-конструктор всегда возвращает объект. По умолчанию это объект, на который указывает ссылка `this`. Если внутри конструктора к нему не добавляются никакие свойства, возвращается «пустой» объект («пустой», если не считать свойства и методы, унаследованные от прототипа конструктора).

Конструкторы неявно возвращают значение `this`, даже если в них отсутствует инструкция `return`. Однако за программистом сохраняется возможность вернуть любой другой объект по своему выбору. В следующем примере создается и возвращается новый объект, на который ссылается переменная `that`.

```
var Objectmaker = function () {

    // это свойство `name` будет проигнорировано,
    // потому что конструктор
    // возвращает совсем другой объект
    this.name = "This is it";

    // создать и вернуть новый объект
    var that = {};
    that.name = "And that's that";
    return that;
};

// проверка
var o = new Objectmaker();
console.log(o.name); // "And that's that"
```

Как видите, всегда есть возможность вернуть из конструктора любой объект при условии, что это действительно объект. Попытка вернуть что-то другое, не являющееся объектом (например, строку или логическое значение `false`), не будет рассматриваться как ошибка, но она будет проигнорирована, и конструктор вернет объект, на который указывает ссылка `this`.

Шаблоны принудительного использования new

Как уже говорилось выше, конструкторы – это обычные функции, которые вызываются с использованием оператора `new`. Что произойдет, если забыть добавить оператор `new` при вызове конструктора? Это не будет рассматриваться интерпретатором как ошибка, но может привести к логической ошибке и к неожиданным результатам. Если вызвать конструктор без оператора `new`, ссылка `this` внутри конструктора будет указывать на глобальный объект. (В броузерах ссылка `this` будет указывать на объект `window`.)

Если в конструкторе создается какое-либо новое свойство, такое как `this.member`, то при вызове конструктора без оператора `new` будет создано новое свойство `member` глобального объекта, к которому можно будет обратиться так: `window.member` или просто `member`. Такое поведение конструктора является крайне нежелательным, потому что вы всегда должны бороться за чистоту глобального пространства имен.

```
// конструктор
function Waffle() {
    this.tastes = "yummy";
}

// новый объект
var good_morning = new Waffle();
```

```
console.log(typeof good_morning); // "object"
console.log(good_morning.tastes); // "yummy"

// антишаблон:
// пропущен оператор `new`
var good_morning = Waffle();
console.log(typeof good_morning); // "undefined"
console.log(window.tastes);       // "yummy"
```

Такое нежелательное поведение устранено в стандарте ECMAScript 5, и в строгом режиме ссылка `this` больше не указывает на глобальный объект. Но если реализация **ES5 недоступна, вы все равно можете предпринять** определенные действия, благодаря которым функция-конструктор всегда будет действовать как конструктор, даже при вызове без оператора `new`.

Соглашения по именованию

Самый простой вариант – использование соглашений по именованию, предложенных в предыдущей главе, в соответствии с которыми имена конструкторов начинаются с заглавного символа (`MyConstructor`), а имена «обычных» функций и методов – со строчного символа (`myFunction`).

Использование ссылки that

Следовать соглашениям по именованию, конечно, полезно, однако соглашения – это просто подсказка, а не обеспечение корректного поведения. Ниже описывается шаблон, который гарантирует, что ваши конструкторы всегда будут вести себя как конструкторы. Суть состоит в том, чтобы вместо ссылки `this` добавлять новые члены к ссылке `that` и возвращать ее явно.

```
function Waffle() {
    var that = {};
    that.tastes = "yummy";
    return that;
}
```

При создании простых объектов вам даже не потребуется локальная переменная, как `that` в нашем примере, – можно просто вернуть объект, определенный как литерал:

```
function Waffle() {
    return {
        tastes: "yummy"
    };
}
```

Любая из представленных выше реализаций `Waffle()` всегда будет возвращать объект независимо от того, как вызывается эта функция:

```
var first = new Waffle(),
    second = Waffle();
console.log(first.tastes);  // "yummy"
console.log(second.tastes); // "yummy"
```

Проблема такого подхода заключается в том, что будет утрачена связь с прототипом, поэтому все члены, добавленные к прототипу функции `Waffle()`, окажутся недоступными для объектов.

Обратите внимание, что имя переменной `that` выбрано лишь для удобства – оно не является частью языка. С таким же успехом можно использовать любое другое имя. В число наиболее употребительных имен входят `self` и `me`.

Конструкторы, вызывающие сами себя

Чтобы избавиться от недостатка, присущего предыдущему шаблону, и обеспечить доступность свойств прототипа в экземплярах объекта, можно воспользоваться следующим решением. В конструкторе можно проверить, является ли ссылка `this` экземпляром конструктора, и если не является, вызвать конструктор еще раз, но уже с оператором `new`:

```
function Waffle() {

    if (!(this instanceof Waffle)) {
        return new Waffle();
    }

    this.tastes = "yummy";

}
Waffle.prototype.wantAnother = true;

// проверка вызовов
var first = new Waffle(),
    second = Waffle();

console.log(first.tastes);       // "yummy"
console.log(second.tastes);      // "yummy"

console.log(first.wantAnother);  // true
console.log(second.wantAnother); // true
```

Другой распространенный способ проверки экземпляра состоит в том, чтобы сравнить не с конкретным именем конструктора, а со свойством `arguments.callee`.

```
if (!(this instanceof arguments.callee)) {
    return new arguments.callee();
}
```

В этом шаблоне используется то обстоятельство, что внутри каждой функции создается объект с именем `arguments`, содержащий все параметры, переданные функции при вызове. А объект `arguments` имеет свойство `callee`, которое ссылается на вызываемую функцию. Тем не менее имейте в виду, что свойство `arguments.callee` недоступно в строгом режиме ES5, поэтому лучше воздержаться от его использования в будущем, а также удалять все обращения к этому свойству, которые встретятся вам в существующих сценариях.

Литералы массивов

Массивы, как и многое другое в языке JavaScript, являются объектами. Массивы могут создаваться с помощью встроенного конструктора `Array()`, но точно так же они могут создаваться с использованием литералов. И подобно литералам объектов, литералы массивов в использовании проще и предпочтительнее.

Ниже демонстрируется создание двух массивов с одним и тем же набором элементов двумя разными способами – с помощью конструктора `Array()` и с помощью литерала.

```
// массив из трех элементов
// внимание: антишаблон
var a = new Array("itsy", "bitsy", "spider");

// точно такой же массив
var a = ["itsy", "bitsy", "spider"];

console.log(typeof a); // "object", потому что массивы - это объекты
console.log(a.constructor === Array); // true
```

Синтаксис литералов массивов

Синтаксис литералов массивов очень прост: это список элементов, разделенных запятыми, заключенный в квадратные скобки. Элементами массивов могут быть значения любых типов, включая объекты и другие массивы.

Синтаксис литералов массивов прозрачен и элегантен. В конце концов, массив – это всего лишь список значений, нумерация которых начинается с нуля. Нет никакой необходимости усложнять процедуру создания массивов (и писать более громоздкий код) использованием конструктора и оператора `new`.

Странности конструктора Array

Еще одна причина, по которой следует стараться избегать пользоваться `new Array()`, – наличие ловушек, которые этот конструктор подготовил для вас.

Когда конструктору `Array()` передается единственное число, оно не становится первым элементом массива. Вместо этого число интерпретируется как размер массива. То есть вызов `new Array(3)` создаст массив с длиной, равной 3, но без элементов. Если попытаться обратиться к любому из элементов такого массива, вы получите значение `undefined`, потому что элементы фактически отсутствуют. Следующий пример демонстрирует различия в поведении массивов, созданных с помощью литерала и конструктора с единственным значением.

```
// массив с одним элементом
var a = [3];
console.log(a.length);    // 1
console.log(a[0]);        // 3

// массив с тремя элементами
var a = new Array(3);
console.log(a.length);    // 3
console.log(typeof a[0]); // "undefined"
```

Уже такое поведение конструктора неожиданно, но ситуация еще более ухудшается, если передать конструктору `new Array()` не целое число, а число с плавающей точкой. Такой вызов конструктора завершится ошибкой, потому что число с плавающей точкой не может использоваться в качестве длины массива:

```
// использование литерала массива
var a = [3.14];
console.log(a[0]); // 3.14

var a = new Array(3.14); // RangeError: недопустимая длина массива
console.log(typeof a);   // "undefined"
```

Чтобы избежать появления подобных ошибок при создании массивов во время выполнения, лучше использовать литералы массивов.

Несмотря на свои недостатки, конструктор `Array()` может оказаться полезным, например для повторения строк. Следующая инструкция вернет строку, содержащую 255 пробелов (почему не 256, подумайте сами):

```
var white = new Array(256).join(' ');
```

Проверка массивов

При применении к массивам оператор `typeof` возвращает строку «object».

```
console.log(typeof [1, 2]); // "object"
```

Хотя такое поведение оператора в чем-то оправданно (массивы – это объекты), но несет немного информации. Часто бывает необходимо точно

знать, является ли некоторое значение массивом. Иногда можно встретить код, проверяющий наличие свойства `length` или других методов, которыми обладают массивы, таких как `slice()`. Но все эти проверки ненадежны, потому что нет ничего, что препятствовало бы другим объектам, которые не являются массивами, иметь свойства и методы с теми же именами. Иногда для проверки используется конструкция `instanceof Array`, но она дает ошибочные результаты в некоторых версиях IE.

В ECMAScript 5 определяется новый метод `Array.isArray()`, который возвращает `true`, если его аргумент является массивом. Например:

```
Array.isArray([]); // true

// попытка обмануть проверку
// с помощью объекта, похожего на массив
Array.isArray({
    length: 1,
    "0": 1,
    slice: function () {}
});                // false
```

Если этот новый метод недоступен в вашем окружении, проверку можно выполнить с помощью метода `Object.prototype.toString()`. Если вызвать метод `call()` функции `toString` в контексте массива, он должен вернуть строку «[object Array]». В контексте объекта этот метод должен вернуть строку «[object Object]». Таким образом, вы можете реализовать собственную проверку, как показано ниже:

```
if (typeof Array.isArray === "undefined") {
    Array.isArray = function (arg) {
        return Object.prototype.toString.call(arg) === "[object Array]";
    };
}
```

JSON

Теперь, когда вы познакомились с литералами массивов и объектов, обсуждавшимися выше, рассмотрим формат JSON обмена данными, название которого является аббревиатурой от слов JavaScript Object Notation (форма записи объектов JavaScript). Его легко и удобно использовать во многих языках программирования и особенно в JavaScript.

Фактически в формате JSON для вас нет ничего нового. Он представляет собой комбинацию форм записи литералов массивов и объектов. Ниже приводится пример строки в формате JSON:

```
{"name": "value", "some": [1, 2, 3]}
```

Единственное, что отличает синтаксис формата JSON от синтаксиса литералов объектов, – это необходимость заключать имена свойств в кавычки. В литералах объектов кавычки необходимо употреблять только

в том случае, если имена свойств не являются допустимыми идентификаторами, например, включают пробелы: {"first name": "Dave"}.

В строки JSON не допускается включать литералы функций и регулярных выражений.

Обработка данных в формате JSON

Как уже упоминалось в предыдущей главе, из-за проблем, связанных с безопасностью, не рекомендуется вслепую преобразовывать строки JSON в объекты с помощью функции `eval()`. Для этого лучше использовать метод `JSON.parse()`, который стал частью языка, начиная с реализации стандарта ES5, и предоставляется реализациями JavaScript в современных броузерах. В устаревших реализациях для доступа к объекту JSON и его методам можно использовать библиотеку JSON.org (*http://www.json.org/json2.js*).

```
// входная строка в формате JSON
var jstr = '{"mykey": "my value"}';

// антишаблон
var data = eval('(' + jstr + ')');

// предпочтительный способ
var data = JSON.parse(jstr);

console.log(data.mykey); // "my value"
```

Если вы уже используете некоторую библиотеку JavaScript, весьма вероятно, что она включает поддержку формата JSON, поэтому вам может не потребоваться дополнительная библиотека JSON.org. Например, библиотека YUI3 включает такую поддержку:

```
// входная строка в формате JSON
var jstr = '{"mykey": "my value"}';

// проанализировать строку и преобразовать ее в объект
// с помощью объекта YUI
YUI().use('json-parse', function (Y) {
    var data = Y.JSON.parse(jstr);
    console.log(data.mykey); // "my value"
});
```

В библиотеке jQuery имеется метод `parseJSON()`:

```
// входная строка в формате JSON
var jstr = '{"mykey": "my value"}';

var data = jQuery.parseJSON(jstr);
console.log(data.mykey); // "my value"
```

Метод JSON.stringify() выполняет операцию, противоположную методу JSON.parse(). Он принимает произвольный объект или массив (или значение простого типа) и преобразует его в строку JSON.

```
var dog = {
    name: "Fido",
    dob: new Date(),
    legs: [1, 2, 3, 4]
};

var jsonstr = JSON.stringify(dog);

// в результате будет получена строка jsonstr:
// {"name":"Fido","dob":"2010-04-11T22:36:22.436Z","legs":[1,2,3,4]}
```

Литералы регулярных выражений

Регулярные выражения в JavaScript также являются объектами, и у вас есть два способа их создания:

- С помощью конструктора new RegExp()
- С помощью литералов регулярных выражений

Следующий фрагмент демонстрирует два способа создания регулярного выражения, соответствующего символу обратного слэша:

```
// литерал регулярного выражения
var re = /\\/gm;

// конструктор
var re = new RegExp("\\\\", "gm");
```

Как видите, литералы регулярных выражений обеспечивают более краткую форму записи и не вынуждают вас мыслить терминами конструкторов классов. Поэтому предпочтительнее использовать литералы.

Кроме того, при использовании конструктора RegExp() придется экранировать кавычки и дважды экранировать символы обратного слэша, как показано в предыдущем примере, где потребовалось указать четыре обратных слэша, чтобы обеспечить совпадение с одним символом. Это удлиняет регулярные выражения и делает их более сложными для чтения и внесения изменений. Регулярные выражения сами по себе являются достаточно сложными конструкциями, и не следует пренебрегать любыми возможностями, позволяющими упростить их, поэтому старайтесь пользоваться литералами.

Синтаксис литералов регулярных выражений

В литералах регулярных выражений шаблон, совпадение с которым требуется отыскать, заключается в символы слэша. Вслед за вторым

слэшем можно добавить модификатор шаблона в виде символа без кавычек:

- `g` – глобальный поиск
- `m` – поиск в многострочном тексте
- `i` – поиск без учета регистра символов

Модификаторы шаблона могут указываться в любых комбинациях и в любом порядке:

```
var re = /pattern/gmi;
```

Использование литералов регулярных выражений позволяет писать более краткий программный код при вызове таких методов, как `String.prototype.replace()`, способных принимать объекты регулярных выражений в качестве параметров.

```
var no_letters = "abc123XYZ".replace(/[a-z]/gi, "");
console.log(no_letters); // 123
```

Единственная причина, оправдывающая использование конструктора `RegExp()`, – когда шаблон не известен заранее и создается как строка во время выполнения.

Еще одно отличие литералов регулярных выражений от конструктора заключается в том, что объекты из литералов создаются всего один раз на этапе синтаксического анализа. Если одно и то же регулярное выражение создается в цикле, будет возвращаться один и тот же объект со всеми значениями свойств (таких как `lastIndex`), установленными в первый раз. Взгляните на следующий пример, иллюстрирующий, что один и тот же объект возвращается дважды.

```
function getRE() {
    var re = /[a-z]/;
    re.foo = "bar";
    return re;
}

var reg = getRE(),
    re2 = getRE();

console.log(reg === re2); // true
reg.foo = "baz";
console.log(re2.foo); // "baz"
```

Такое поведение литералов было изменено в **ES5, и теперь литералы** каждый раз создают новые объекты. Соответствующие изменения уже внесены в реализацию многих броузеров, поэтому вы не должны полагаться на это поведение.

И последнее замечание: при вызове конструктора RegExp() без оператора new (то есть как функции, а не как конструктора) он ведет себя точно так же, как и при вызове с оператором new.

Объекты-обертки значений простых типов

В языке JavaScript имеется пять простых типов данных: число, строка, логическое значение, null и undefined. За исключением null и undefined, для остальных трех типов существуют так называемые *объекты-обертки*. Объекты-обертки могут быть созданы с помощью встроенных конструкторов Number(), String() и Boolean().

Чтобы понять различия между простым числом и объектом числа, взгляните на следующий пример:

```
// простое число
var n = 100;
console.log(typeof n);    // "number"

// объект Number
var nobj = new Number(100);
console.log(typeof nobj); // "object"
```

Объекты-обертки обладают рядом интересных свойств и методов, например, объекты-числа обладают такими методами, как toFixed() и toExponential(). Объекты-строки обладают методами substring(), charAt() и toLowerCase() (помимо других) и свойством length. Эти методы очень удобны в работе, что может служить веским основанием к использованию объекта вместо элементарного значения. Однако методы могут применяться и к элементарным значениям – при вызове метода элементарное значение временно преобразуется в объект и ведет себя как объект.

```
// элементарная строка, используемая как объект
var s = "hello";
console.log(s.toUpperCase()); // "HELLO"

// даже само значение может действовать как объект
"monkey".slice(3, 6);         // "key"

// то же относится и к числам
(22 / 7).toPrecision(3);      // "3.14"
```

Поскольку элементарные значения могут действовать как объекты, нет никаких причин использовать более длинную форму записи с использованием конструктора. Например, нет никакой необходимости писать new String("hi");, когда можно просто использовать "hi":

```
// избегайте такого использования:
var s = new String("my string");
```

```
var n = new Number(101);
var b = new Boolean(true);

// лучше и проще:
var s = "my string";
var n = 101;
var b = true;
```

Объекты-обертки можно использовать, когда необходимо добавить к значению дополнительные свойства и обеспечить сохранение информации о состоянии. Поскольку элементарные значения не являются объектами, к ним невозможно добавить дополнительные свойства.

```
// элементарная строка
var greet = "Hello there";

// значение будет временно преобразовано в объект,
// чтобы предоставить возможность вызвать метод split()
greet.split(' ')[0]; // "Hello"

// попытка добавить свойство к элементарному значению не вызовет ошибку
greet.smile = true;

// но она не даст положительных результатов
typeof greet.smile;  // "undefined"
```

В предыдущем фрагменте переменная `greet` лишь временно преобразуется в объект, чтобы обеспечить возможность доступа к методу/свойству. При этом, если бы переменная `greet` была объявлена как объект с помощью вызова конструктора `new String()`, дополнительное свойство `smile` было бы добавлено к объекту. Расширение строк, чисел и логических значений дополнительными свойствами редко используется на практике. Однако если это действительно необходимо, то вам, вероятно, следует воспользоваться конструкторами объектов-оберток.

При вызове конструкторов объектов-оберток без оператора `new` они преобразуют свои аргументы в элементарные значения:

```
typeof Number(1);            // "number"
typeof Number("1");          // "number"
typeof Number(new Number()); // "number"
typeof String(1);            // "string"
typeof Boolean(1);           // "boolean"
```

Объекты Error

В JavaScript имеется множество встроенных конструкторов объектов ошибок, таких как `Error()`, `SyntaxError()`, `TypeError()` и других, которые используются в инструкции `throw`. Объекты ошибок, создаваемые этими конструкторами, имеют следующие свойства:

`name`

Содержит значение свойства `name` функции-конструктора, создающей объект. Это может быть строка **«Error», соответствующая универсальному конструктору**, или более узкоспециализированное значение, такое как «RangeError».

`message`

Строка, которая передается конструктору при создании объекта.

Объекты ошибок обладают и другими свойствами, определяющими, например, номер строки и имя файла, где была обнаружена ошибка, но эти дополнительные свойства по-разному реализованы в разных броузерах и потому ненадежны.

С другой стороны, инструкция `throw` способна принимать любые объекты, не обязательно созданные с помощью какого-либо из конструкторов семейства `Error`. Благодаря этому у вас есть возможность возбуждать ошибки, передавая собственные объекты. Такие объекты ошибок могут иметь свойства `name`, `message` и любые другие для сохранения информации, необходимой обработчику в инструкции `catch`. Вы можете проявить изобретательность и использовать свои объекты ошибок для восстановления приложения в нормальное состояние.

```
try {
    // произошло что-то неприятное, возбудить ошибку
    throw {
        name: "MyErrorType",         // нестандартный тип ошибки
        message: "oops",
        extra: "This was rather embarrassing",
        remedy: genericErrorHandler // какой обработчик
                                    // должен обрабатывать ошибку
    };
} catch (e) {
    // известить пользователя
    alert(e.message);               // "oops"

    // обработать ошибку
    e.remedy();                     // вызовет genericErrorHandler()
}
```

Если конструкторы объектов ошибок вызываются как обычные функции (без оператора `new`), они ведут себя точно так же, как конструкторы (при вызове с оператором `new`), и возвращают те же объекты ошибок.

В заключение

В этой главе вы познакомились с различными шаблонами применения литералов, которые являются простыми альтернативами использования функций-конструкторов. В этой главе обсуждались:

- Литералы объектов – элегантный способ создания объектов в виде списков пар ключ-значение, разделенных запятыми, заключенных в фигурные скобки.
- Функции-конструкторы – встроенные конструкторы (вместо которых практически всегда проще использовать более краткую форму записи в виде литералов) и собственные конструкторы.
- Способы, гарантирующие, что конструкторы всегда будут вести себя как конструкторы независимо от использования оператора `new`.
- Литералы массивов – списки значений, разделенных запятыми, заключенные в квадратные скобки.
- JSON – формат представления данных, состоящий из литералов объекта и массива.
- Литералы регулярных выражений.
- Другие встроенные конструкторы, которых следует избегать: `String()`, `Number()`, `Boolean()` и различные конструкторы `Error()`.

Вообще говоря, необходимость в использовании встроенных конструкторов, за исключением конструктора `Date()`, редко возникает на практике. В следующей таблице перечислены эти конструкторы вместе с соответствующими им литералами.

Встроенные конструкторы (желательно избегать)	**Литералы и элементарные значения (предпочтительнее)**
`var o = new Object();`	`var o = {};`
`var a = new Array();`	`var a = [];`

`var re = new RegExp(`
`  "[a-z]",`
`  "g"`
`);`
	`var re = /[a-z]/g;`

`var s = new String();`	`var s = "";`
`var n = new Number();`	`var n = 0;`
`var b = new Boolean();`	`var b = false;`

`throw new Error("uh-oh");`	`throw {`
	`  name: "Error",`
	`  message: "uh-oh"`
	`};`
	... или
	`throw Error("uh-oh");`

4

Функции

Работа с функциями – один из базовых навыков программиста на JavaScript, потому что они очень широко используются в этом языке программирования. Функции в JavaScript решают большой объем задач, для которых в других языках программирования могут быть предусмотрены иные синтаксические конструкции.

В этой главе вы познакомитесь с различными способами определения функций в JavaScript, узнаете о функциях-выражениях и функциях-объявлениях, а также увидите, как действует локальная область видимости и механизм подъема переменных. Затем вашему вниманию будет представлено множество шаблонов, которые помогут разработать свой API (то есть создать более удачные интерфейсы к своим функциям), реализовать инициализацию объектов (с минимальным количеством глобальных переменных) и увеличить производительность (точнее, отказаться от выполнения лишней работы).

А теперь углубимся в изучение функций, начав с рассмотрения и разъяснения важнейших основ.

Основы

Функции в языке JavaScript обладают двумя основными свойствами, которые делают их особенными. Во-первых, функции являются *обычными* объектами и, во-вторых, они образуют собственные области видимости.

Функции – это объекты, которые:

- Могут создаваться динамически в процессе выполнения программы.
- Могут присваиваться переменным, ссылки на них могут копироваться в другие переменные, могут быть расширены дополнительными свойствами и, за исключением некоторых особых случаев, могут быть удалены.

- Могут передаваться как аргументы другим функциям и могут возвращаться другими функциями.
- Могут иметь собственные свойства и методы.

То есть возможно, что функция A, будучи объектом, будет обладать свойствами и методами, одним из которых является функция B. Функция B в свою очередь может принимать функцию C как аргумент и возвращать другую функцию, D. На первый взгляд все это может показаться слишком сложным. Но вы сможете по достоинству оценить их мощь, гибкость и выразительность как только освоитесь с различными способами применения функций. Вообще говоря, функции в языке JavaScript представляют собой обычные объекты, которые обладают дополнительным свойством – они могут вызываться, то есть выполняться.

В том, что функции являются объектами, легко убедиться, если посмотреть на конструктор new Function() в действии:

```
// антишаблон
// исключительно в демонстрационных целях
var add = new Function('a, b', 'return a + b');
add(1, 2); // вернет 3
```

Не вызывает никаких сомнений, что add() в этом фрагменте является объектом – в конце концов, он был создан конструктором. При этом применение конструктора Function() считается далеко не лучшим решением (аналогично использованию функции eval()), потому что программный код передается в виде строки и интерпретируется. Кроме того, такой программный код сложнее писать (и читать) из-за необходимости экранировать кавычки и предпринимать дополнительные усилия для оформления внутри функции отступов, позволяющих повысить удобочитаемость.

Вторая важная особенность функций – они образуют собственные области видимости. В языке JavaScript локальная область видимости не задается фигурными скобками. Другими словами, блок программного кода не определяет отдельную область видимости. Только функции обладают этим свойством. Любая переменная, объявленная с помощью инструкции var внутри функции, становится локальной переменной, невидимой за пределами функции. Когда говорится, что фигурные скобки не создают локальную область видимости, это означает, что переменная, определенная с помощью инструкции var внутри условной инструкции if или циклов for и while, не будет интерпретироваться как локальная переменная по отношению к этим инструкциями if, for или while. Переменная может быть локальной только по отношению к окружающей ее функции, и, если такой функции не существует, переменная станет глобальной. Как уже говорилось в главе 2, необходимо стремиться минимизировать количество глобальных переменных, поэтому функции совершенно необходимы, чтобы сохранить видимость переменных под контролем.

Устранение неоднозначностей в терминологии

Давайте прервемся ненадолго, чтобы условиться о терминах, описывающих программный код, который используется для определения функций, потому что при обсуждении шаблонов точное и согласованное применение понятий важно не менее чем сам код.

Взгляните на следующий фрагмент:

```
// именованная функция-выражение
var add = function add(a, b) {
    return a + b;
};
```

В этом фрагменте показана функция, выраженная как *именованная функция-выражение*.

Если опустить имя (второй идентификатор `add` в примере) в функции-выражении, вы получите *неименованную* функцию-выражение, или просто *функцию-выражение*, которую часто еще называют *анонимной функцией*. Например:

```
// функция-выражение, она же анонимная функция
var add = function (a, b) {
    return a + b;
};
```

То есть «функция-выражение» – это более широкий термин, а термин «именованная функция-выражение» описывает частный случай, когда для функции-выражения определяется необязательное имя.

Если вы опускаете второй идентификатор `add` и останавливаетесь на неименованной функции-выражении, это никак не влияет на ее определение и способы обращения к ней. Единственное отличие состоит в том, что свойство `name` объекта функции будет содержать пустую строку. Свойство `name` – это расширение языка (наличие этого свойства не предусматривается стандартом ECMA), реализованное во многих окружениях. Если оставить второй идентификатор `add`, то свойство `add.name` будет содержать строку «add». Свойство `name` удобно использовать при работе с отладчиком, таким как Firebug, или для организации рекурсивных вызовов функции самой себя. В остальных случаях его можно просто опустить.

Наконец, существуют *функции-объявления*. Они очень похожи на функции в других языках программирования:

```
function foo() {
    // тело функции
}
```

С точки зрения синтаксиса именованные функции-выражения и функции-объявления очень похожи, особенно если результат функции-выра-

жения не присваивается переменной (как мы увидим в шаблоне применения функций обратного вызова, далее в этой главе). Иногда даже нет иного способа заметить различия между функцией-объявлением и именованной функцией-выражением, кроме как взглянуть на контекст, в котором определяется функция, как будет показано в следующем разделе.

Синтаксические различия между этими двумя типами функций заключаются в завершающей точке с запятой. В функциях-объявлениях заключительную точку с запятой можно опустить, но она является обязательной в функциях-выражениях, и вы всегда должны использовать ее, несмотря на то, что механизм автоматической вставки точки с запятой может сделать это вместо вас.

Также достаточно часто вам может встретиться термин *функция-литерал*. Под этим названием может подразумеваться либо функция-выражение, либо именованная функция-выражение. Вследствие такой неоднозначности лучше воздерживаться от использования этого термина.

Объявления и выражения: имена и подъем

Так какие же функции использовать – функции-объявления или функции-выражения? В случаях, когда синтаксически невозможно использовать функции-объявления, эта дилемма разрешается сама собой. Например, когда требуется передать объект функции в виде параметра или определить метод в литерале объекта:

```
// это функция-выражение,
// которая передается как аргумент функции callMe
callMe(function () {
    // Это неименованная функция-выражение,
    // также известная как анонимная функция
});

// это именованная функция-выражение
callMe(function me() {
    // Это именованная функция-выражение,
    // с именем "me"
});

// еще одна функция-выражение
var myobject = {
    say: function () {
        // Это функция-выражение
    }
};
```

Функции-объявления могут появляться только в «программном коде», то есть внутри других функций или в глобальной области видимости. Определения этих функций не могут присваиваться переменным или свойствам или передаваться другим функциям в виде параметров. Ниже приводится пример допустимых способов использования функций-объявлений, где все функции, `foo()`, `bar()` и `local()`, определяются с использованием шаблона функций-объявлений:

```
// глобальная область видимости
function foo() {}

function local() {
    // локальная область видимости
    function bar() {}
    return bar;
}
```

Свойство name функций

При выборе способа определения функций необходимо учитывать еще одно обстоятельство – доступность свойства `name`. Напомню, что наличие этого свойства не определяется стандартом, но оно доступно во многих окружениях. В функциях-объявлениях и в именованных функциях-выражениях свойство `name` определено. Наличие этого свойства в анонимных функциях-выражениях зависит от реализации – оно может отсутствовать (IE) или присутствовать, но содержать пустую строку (Firefox, WebKit):

```
function foo() {}            // функция-объявление
var bar = function () {};    // функция-выражение
var baz = function baz() {}; // именованная функция-выражение

foo.name; // "foo"
bar.name; // ""
baz.name; // "baz"
```

Свойство `name` обеспечивает дополнительные удобства при отладке программного кода в Firebug или в других отладчиках. Когда отладчику потребуется вывести сообщение об ошибке, возникшей при выполнении функции, он сможет проверить наличие свойства `name` и использовать его значение в качестве дополнительной подсказки. Свойство `name` также используется для организации рекурсивных вызовов функции. Если вас эти возможности не интересуют, можно использовать неименованные функции-выражения, которые выглядят проще и компактнее.

Против функций-объявлений и в пользу функций-выражений говорит еще и то обстоятельство, что функции-выражения лишний раз подчеркивают, что функции в JavaScript – это объекты, а не какие-то специализированные конструкции языка.

Технически вполне возможно определить именованную функцию-выражение и присвоить ее переменной с другим именем, например:

```
var foo = function bar() {};
```

Однако поведение такого определения некорректно реализовано в некоторых броузерах (IE), поэтому использовать такой прием не рекомендуется.

Подъем функций

Из предыдущего обсуждения можно было бы заключить, что по своему поведению функции-объявления во многом схожи с *именованными* функциями-выражениями. Однако это не совсем так, и различия заключаются в поведении, которое называется подъемом.

Определение термина *подъем* отсутствует в стандарте ECMA-Script, но он часто используется на практике и достаточно точно описывает поведение.

Как вы уже знаете, объявления всех переменных, независимо от того, в каком месте внутри функции они находятся, на этапе интерпретации как бы «поднимаются» в начало функции. То же относится и к функциям, потому что фактически они являются объектами, которые присваиваются переменным. Но «фишка» в том, что для функций-объявлений вместе с именем функции «поднимается» и ее определение, а не только имя. Взгляните на следующий фрагмент:

```
// антишаблон
// исключительно в демонстрационных целях

// глобальные функции
function foo() {
    alert('global foo');
}
function bar() {
    alert('global bar');
}

function hoistMe() {

    console.log(typeof foo); // "function"
    console.log(typeof bar); // "undefined"

    foo(); // "local foo"
    bar(); // TypeError: bar is not a function
```

```
    // функция-объявление:
    // имя 'foo' и его определение "поднимаются" вместе
    function foo() {
        alert('local foo');
    }

    // функция-выражение:
    // "поднимается" только имя 'bar',
    // без реализации
    var bar = function () {
        alert('local bar');
    };
}
hoistMe();
```

Этот пример демонстрирует, что, как и в случае с обычными переменными, простое присутствие определений `foo` и `bar` в теле функции `hoistMe()` вызывает их подъем в начало функции и делает недоступными глобальные функции `foo` и `bar`. Разница состоит в том, что вместе с локальным именем `foo` вверх поднимается и *определение* этой функции. А определение функции `bar()` не поднимается – поднимается только объявление имени. Именно поэтому, пока поток выполнения не достигнет определения функции `bar()`, она будет оставаться неопределенной и не сможет использоваться как функция (мешая при этом «увидеть» глобальную функцию `bar()` в цепочке областей видимости).

Теперь, когда мы в необходимом объеме познакомились с основами и терминологией, можно перейти к изучению некоторых шаблонов использования функций в языке JavaScript, и начнем мы с функций обратного вызова. Важно не забывать о двух особенностях функций в JavaScript:

- Они являются объектами
- Они образуют локальную область видимости

Функции обратного вызова

Функции – это объекты, то есть они могут передаваться как аргументы другим функциям. Если, например, функция `introduceBugs()` передается как параметр функции `writeCode()`, это оначает, что в какой-то момент функция `writeCode()` выполнит (или вызовет) функцию `introduceBugs()`. В этом случае функция `introduceBugs()` называется *функцией обратного вызова*:

```
function writeCode(callback) {
    // выполнение некоторых операций...
    callback();
    // ...
}
```

```
function introduceBugs() {
    // ... вносит ошибку
}

writeCode(introduceBugs);
```

Обратите внимание, что при передаче `introduceBugs()` в виде аргумента функции `writeCode()` она передается без круглых скобок. Наличие круглых скобок вызывает выполнение функции, тогда как в данном случае нам необходимо всего лишь передать ссылку на эту функцию и дать функции `writeCode()` возможность выполнить ее (то есть произвести обратный вызов) в нужный момент времени.

Пример использования функции обратного вызова

Для начала рассмотрим пример без использования функции обратного вызова, а затем перепишем его заново. Допустим, что у нас имеется некоторая функция общего назначения, выполняющая ряд сложных операций и возвращающая большой набор данных. Назовем эту универсальную функцию, например `findNodes()`. Она будет выполнять обход дерева DOM страницы и возвращать массив элементов страницы, представляющих для нас интерес:

```
var findNodes = function () {
    var i = 100000, // большой, тяжелый цикл
        nodes = [], // хранилище для результатов
        found;      // следующий найденный узел
    while (i) {
        i -= 1;
        // здесь находится сложная логика выбора узлов...
        nodes.push(found);
    }
    return nodes;
};
```

Было бы желательно сохранить эту функцию максимально универсальной, чтобы она просто возвращала массив узлов DOM, не выполняя никаких операций с фактическими элементами дерева. Логику изменения узлов можно вынести в отдельную функцию, например с именем `hide()`, которая, как следует из ее имени, делает элементы страницы невидимыми:

```
var hide = function (nodes) {
    var i = 0, max = nodes.length;
    for (; i < max; i += 1) {
        nodes[i].style.display = "none";
    }
};
```

```
// вызов функций
hide(findNodes());
```

Однако такая реализация не отличается эффективностью, потому что функция `hide()` снова выполняет обход узлов в массиве, полученном от функции `findNodes()`. Эффективность можно было бы повысить, если бы удалось отказаться от этого цикла и узлы скрывались бы по мере их обнаружения функцией `findNodes()`. Но если включить логику скрытия в функцию `findNodes()`, она перестанет быть универсальной – из-за образовавшейся *тесной связи* между логикой поиска и логикой модификации узлов. Воспользуемся шаблоном применения функции обратного вызова – оформим операцию скрытия узла дерева в виде отдельной функции обратного вызова и делегируем ей полномочия выполнения операции:

```
// измененный вариант функции findNodes(),
// принимающий функцию обратного вызова
var findNodes = function (callback) {
    var i = 100000,
        nodes = [],
        found;

    // проверить, является ли объект callback функцией
    if (typeof callback !== "function") {
        callback = false;
    }

    while (i) {
        i -= 1;
        // здесь находится сложная логика выбора узлов...

        // теперь вызвать функцию callback:
        if (callback) {
            callback(found);
        }

        nodes.push(found);
    }
    return nodes;
};
```

Реализация получилась достаточно простой. Единственное, что пришлось добавить в функцию `findNodes()`, – проверку наличия необязательного параметра `callback` и его вызов. Параметр `callback` в измененной версии функции `findNodes()` является необязательным, поэтому данная функция может использоваться так же, как и раньше, и не окажет влияния на работоспособность уже имеющегося программного кода, опирающегося на прежний вариант реализации.

Реализация функции `hide()` теперь выглядит еще проще благодаря тому, что в ней не нужно выполнять цикл по элементам массива:

```
// функция обратного вызова
var hide = function (node) {
    node.style.display = "none";
};

// отыскать узлы и скрыть их
findNodes(hide);
```

Функция обратного вызова может быть уже существующей функцией, как это показано в предыдущем примере, или анонимной функцией, которая создается при вызове основной функции. Например, ниже показано, как можно реализовать отображение скрытых узлов с помощью все той же универсальной функции `findNodes()`:

```
// передача анонимной функции обратного вызова
findNodes(function (node) {
    node.style.display = "block";
});
```

Функции обратного вызова и их области видимости

В предыдущем примере обратный вызов производился, как показано ниже:

```
callback(parameters);
```

В большинстве случаев этого вполне достаточно, но в некоторых ситуациях в качестве функции обратного вызова используется не анонимная или глобальная функция, а метод объекта. Если такой метод обратного вызова использует ссылку `this` для обращения к своему объекту, это может стать причиной неожиданного поведения метода.

Представьте, что в качестве функции обратного вызова используется метод `paint()` объекта с именем `myapp`:

```
var myapp = {};
myapp.color = "green";
myapp.paint = function (node) {
    node.style.color = this.color;
};
```

И этот метод передается следующей функции `findNodes()`:

```
var findNodes = function (callback) {
    // ...
    if (typeof callback === "function") {
        callback(found);
    }
    // ...
};
```

Если теперь выполнить вызов `findNodes(myapp.paint)`, он будет действовать не так, как ожидается, потому что свойство `this.color` не будет определено внутри метода. Ссылка `this` будет указывать на глобальный объект, потому что `findNodes()` – это глобальная функция. Если бы `findNodes()` была методом объекта с именем `dom` (например: `dom.findNodes()`), то внутри метода обратного вызова ссылка `this` указывала бы на объект `dom`, а не на объект `myapp`.

Чтобы решить эту проблему, необходимо вместе с методом обратного вызова передать объект, которому принадлежит этот метод:

```
findNodes(myapp.paint, myapp);
```

Кроме того, нам потребовалось бы изменить функцию `findNodes()` и связать в ней объект с методом:

```
var findNodes = function (callback, callback_obj) {
    //...
    if (typeof callback === "function") {
        callback.call(callback_obj, found);
    }
    // ...
};
```

Подробнее о связывании и использовании функций `call()` и `apply()` мы поговорим в следующих главах.

Другая возможность связать объект и метод, используемый в качестве функции обратного вызова, заключается в том, чтобы передать метод в виде строки, благодаря чему отпадает необходимость дважды упоминать имя объекта в вызове функции. Другими словами, вызов:

```
findNodes(myapp.paint, myapp);
```

можно преобразовать в:

```
findNodes("paint", myapp);
```

А функцию `findNodes()` изменить, как показано ниже:

```
var findNodes = function (callback, callback_obj) {

    if (typeof callback === "string") {
        callback = callback_obj[callback];
    }

    //...
    if (typeof callback === "function") {
        callback.call(callback_obj, found);
    }
    // ...
};
```

Обработчики асинхронных событий

Прием на основе функций обратного вызова имеет множество применений; например, при подключении обработчиков событий к элементам страницы вы фактически передаете указатель на функцию обратного вызова, которая должна вызываться при возникновении события. Ниже приводится простой пример использования `console.log()` в качестве функции обратного вызова для обработки события `click` в документе:

```
document.addEventListener("click", console.log, false);
```

В клиентских сценариях выполнением большинства операций управляют события. Когда завершается загрузка страницы, возникает событие `load`. Затем пользователь начинает взаимодействовать со страницей, чем вызывает появление различных событий, таких как `click`, `keypress`, `mouseover`, `mousemove` и так далее. Язык JavaScript особенно хорошо подходит для программирования сценариев, управляемых событиями, потому что прием использования функций обратного вызова позволяет писать программы, которые выполняются *асинхронно*, то есть не по порядку.

В Голливуде широко известна фраза: «Не звоните нам, мы сами вам позвоним», которая произносится, когда на одну и ту же роль пробуется множество претендентов. Группа подбора актеров просто не смогла бы работать, если бы постоянно отвечала на звонки претендентов. В асинхронном программировании на JavaScript наблюдается похожая ситуация. Только вместо номера телефона оставляется функция обратного вызова, которая будет вызвана в нужное время. Может так получиться, что некоторые функции обратного вызова никогда не будут вызваны, потому что некоторые события могут так никогда и не произойти. Например, пользователь может никогда не щелкнуть на кнопке «Купить!», и ваша функция проверки номера кредитной карты никогда не будет вызвана.

Предельное время ожидания

Другой пример применения функций обратного вызова – при использовании объектом `window` методов управления таймером в броузерах: `setTimeout()` и `setInterval()`. Эти методы также могут принимать и вызывать функции обратного вызова:

```
var thePlotThickens = function () {
    console.log('500ms later...');
};
setTimeout(thePlotThickens, 500);
```

Обратите внимание еще раз, что функция `thePlotThickens` передается как переменная, без круглых скобок, потому что мы не собираемся вызывать ее немедленно, а просто хотим передать ссылку на нее методу

setTimeout() для использования в будущем. Передача строки "thePlotThickens()" вместо ссылки на функцию является распространенным антишаблоном, напоминающим использование функции eval().

Функции обратного вызова в библиотеках

Функции обратного вызова – это простой и мощный прием, который может пригодиться при проектировании библиотек. Программный код, входящий в состав библиотеки, должен быть максимально универсальным, что позволит использовать его в самых разных ситуациях, и функции обратного вызова способны помочь вам в достижении этой универсальности. Вам не нужно стремиться предугадывать и предусматривать реализацию всех возможных способов применения, потому что это приведет к раздуванию библиотеки, и большинство пользователей библиотеки никогда не будут использовать все имеющиеся в ней возможности. Вместо этого вам следует сконцентрироваться на реализации базовых функциональных возможностей и предоставить «зацепки» в форме функций обратного вызова, которые позволят легко конструировать библиотечные методы, расширять и настраивать их поведение.

Возвращение функций

Функции – это объекты, поэтому их можно использовать в качестве возвращаемых значений. То есть функция не обязательно должна возвращать какое-то конкретное значение или массив данных. Функция может возвращать другую, более специализированную функцию или создавать другую функцию по мере необходимости в зависимости от входных параметров.

Ниже приводится простой пример. Функция выполняет некоторые операции, возможно, инициализирует некоторые значения и возвращает результат. Возвращаемым результатом может быть другая функция, которую также можно вызвать:

```
var setup = function () {
    alert(1);
    return function () {
        alert(2);
    };
};

// использование функции setup
var my = setup(); // выведет диалог с текстом "1"
my();             // выведет диалог с текстом "2"
```

Поскольку функция setup() обертывает возвращаемую функцию, образуется замыкание, которое можно использовать для хранения некото-

рых частных данных, доступных возвращаемой функции и недоступных за ее пределами. Примером таких частных данных может служить счетчик, наращиваемый при каждом обращении к нему:

```
var setup = function () {
    var count = 0;
    return function () {
        return (count += 1);
    };
};

// пример использования
var next = setup();
next(); // вернет 1
next(); // 2
next(); // 3
```

Самоопределяемые функции

Функции могут определяться динамически и присваиваться переменным. Если создать новую функцию и присвоить ее переменной, которая уже хранит другую функцию, старая функция будет затерта новой. Это напоминает переустановку указателя на новую функцию. Все это можно реализовать в теле старой функции. В этом случае функция переопределит саму себя новой реализацией. На первый взгляд такой прием выглядит сложнее, чем это есть на самом деле, поэтому давайте рассмотрим простой пример:

```
var scareMe = function () {
    alert("Boo!");
    scareMe = function () {
        alert("Double boo!");
    };
};

// вызов самоопределяемой функции
scareMe(); // Boo!
scareMe(); // Double boo!
```

Этот прием удобно использовать, когда требуется выполнить некоторые операции по инициализации, причем эти операции должны быть выполнены всего один раз. Поскольку нет никаких причин многократно выполнять некоторые операции, когда этого можно избежать, часть функции может оказаться невостребованной. В подобных ситуациях можно применять самоопределяемые функции, изменяющие собственную реализацию.

Совершенно очевидно, что использование этого шаблона способно положительно сказаться на производительности приложения, если переопределенная функция выполняет меньше работы.

Этот шаблон имеет еще одно название: «отложенное определение функции», потому что окончательное определение функции откладывается до первого обращения к ней, и, как правило, новая функция выполняет меньше работы.

Недостаток этого шаблона состоит в том, что все свойства, добавленные к прежней функции, будут потеряны в момент переопределения. Кроме того, если прежняя функция использовалась под другим именем, например, была присвоена другой переменной или была задействована как метод объекта, то в таких случаях переопределение функции никогда не произойдет и будет выполняться тело прежней функции.

Рассмотрим пример, где функция `scareMe()` используется как обычный объект:

1. Добавляется новое свойство.
2. Объект функции присваивается новой переменной.
3. Функция используется как метод.

Взгляните на следующий фрагмент:

```
// 1. добавление нового свойства
scareMe.property = “properly”;

// 2. присваивание другой переменной
var prank = scareMe;

// 3. использование в качестве метода
var spooky = {
    boo: scareMe
};

// вызов под новым именем
prank();                          // “Boo!”
prank();                          // “Boo!”
console.log(prank.property);      // “properly”

// вызов как метода
spooky.boo();                     // “Boo!”
spooky.boo();                     // “Boo!”
console.log(spooky.boo.property); // “properly”

// использование самоопределяемой функции
scareMe();                        // Double boo!
scareMe();                        // Double boo!
console.log(scareMe.property);    // undefined
```

Как видите, при присваивании функции новой переменной переопределение не происходит, как вы, возможно, ожидали. При каждом вызове функции под именем `prank()` она выводит один и тот же текст «Boo!».

Хотя переопределение глобальной функции `scareMe()` происходит, но переменная `prank` продолжает ссылаться на прежний объект функции, обладающий свойством `property`. То же самое происходит, когда функция используется в качестве метода `boo()` объекта `spooky`. В обоих случаях при вызове выполняется переопределение глобальной ссылки `scareMe()`, поэтому когда дело доходит до вызова функции под этим именем, она оказывается уже измененной и при первом же вызове выводит текст «Double boo». Кроме того, теряется возможность получить доступ к прежнему свойству `scareMe.property`.

Немедленно вызываемые функции

Немедленно вызываемая функция – это синтаксическая конструкция, позволяющая вызвать функцию немедленно, в точке ее определения. Например:

```
(function () {
    alert('watch out!');
}());
```

Этот прием может применяться только к функциям-выражениям (как к именованным, так и к анонимным) и служит для немедленного вызова функции сразу после ее создания. Определение термина *немедленно вызываемая функция* отсутствует в стандарте ECMAScript, но он достаточно краток и позволяет описывать и обсуждать шаблон.

Шаблон состоит из следующих частей:

- Определение функции-выражения. (Этот прием не действует с функциями-объявлениями.)
- Добавление круглых скобок в конце, которые заставляют интерпретатор выполнить функцию немедленно.
- Добавление круглых скобок, окружающих всю эту конструкцию (они необходимы, только если функция не присваивается какой-нибудь переменной).

Часто также используется следующий синтаксис (обратите внимание на местоположение закрывающей круглой скобки), но инструмент JSLint отдает предпочтение первому варианту:

```
(function () {
    alert('watch out!');
})();
```

Этот шаблон удобен тем, что он предоставляет программному коду *изолированную* область видимости. Представьте себе такую распространенную ситуацию: сценарий должен выполнить некоторые операции по инициализации после загрузки страницы, например подключить обработчики событий, создать объекты и так далее. Все эти действия должны выполняться только один раз, поэтому нет смысла создавать

именованную функцию. В ходе выполнения этих операций необходимо создать несколько временных переменных, которые будут не нужны по окончании инициализации, и было бы нежелательно, чтобы все эти переменные остались существовать в глобальном пространстве имен. Именно в подобных ситуациях можно воспользоваться немедленно вызываемой функцией, чтобы обернуть программный код локальной областью видимости и не допустить засорения глобального пространства имен временными переменными:

```
(function () {

    var days = ['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat'],
        today = new Date(),
        msg = 'Today is ' + days[today.getDay()] + ', ' + today.getDate();

    alert(msg);

}()); // "Today is Fri, 13"
```

Если бы этот фрагмент кода не был обернут немедленно вызываемой функцией, переменные `days`, `today` и `msg` превратились бы в глобальные переменные и остались бы существовать после выполнения этого фрагмента.

Параметры немедленно вызываемых функций

Немедленно вызываемым функциям можно передавать дополнительные аргументы, как показано в следующем примере:

```
// выведет:
// I met Joe Black on Fri Aug 13 2010 23:26:59 GMT-0800 (PST)

(function (who, when) {

    console.log("I met " + who + " on " + when);

}("Joe Black", new Date()));
```

Обычно немедленно вызываемым функциям в качестве аргумента передается глобальный объект, чтобы к нему можно было обращаться без использования свойства `window`; это позволяет использовать программный код в средах, отличных от броузеров:

```
(function (global) {

    // глобальный объект доступен через аргумент `global`

}(this));
```

Обратите внимание, что вообще-то не рекомендуется передавать немедленно вызываемым функциям слишком большое количество парамет-

ров, потому что впоследствии это может привести к тому, что придется просматривать всю функцию от начала до конца и обратно, чтобы понять, как эти параметры используются.

Значения, возвращаемые немедленно вызываемыми функциями

Как и любая другая функция, немедленно вызываемая функция способна возвращать некоторые значения, которые можно присваивать переменным:

```
var result = (function () {
    return 2 + 2;
}());
```

Того же эффекта можно добиться, опустив круглые скобки, окружающие функцию, потому что они не требуются, когда значение, возвращаемое немедленно вызываемой функцией, присваивается переменной. Опустив первую пару круглых скобок в предыдущем примере, мы получим следующее:

```
var result = function () {
    return 2 + 2;
}();
```

Такой синтаксис выглядит проще, но иногда он может вводить в заблуждение. Не заметив заключительную пару скобок `()` в конце функции, легко можно решить, что переменная `result` – это ссылка на функцию. Фактически же переменная `result` получает значение, возвращаемое функцией, в данном случае число `4`.

Еще один способ, позволяющий добиться того же эффекта:

```
var result = (function () {
    return 2 + 2;
})();
```

В предыдущих примерах немедленно вызываемая функция возвращает простое целочисленное значение. Однако немедленно вызываемые функции способны возвращать значения любых типов, включая другие функции. В этом случае вы можете использовать область видимости немедленно вызываемой функции для хранения некоторых частных данных, используемых внутри возвращаемой функции.

В следующем примере немедленно вызываемая функция возвращает другую функцию, которая присваивается переменной `getResult` и просто возвращает значение `res`, которое было предварительно вычислено немедленно вызываемой функцией и сохранено в замыкании:

```
var getResult = (function () {
    var res = 2 + 2;
    return function () {
```

```
        return res;
    };
}());
```

Немедленно вызываемые функции могут также использоваться при определении свойств объектов. Представьте, что необходимо определить свойство объекта, значение которого едва ли будет изменяться на протяжении всего времени существования объекта, но, чтобы получить начальное значение, необходимо произвести некоторые вычисления. Эти вычисления можно производить с помощью немедленно вызываемой функции, как показано в примере ниже:

```
var o = {
    message: (function () {
        var who = "me",
            what = "call";
        return what + " " + who;
    }()),
    getMsg: function () {
        return this.message;
    }
};

// пример использования usage
o.getMsg(); // "call me"
o.message;  // "call me"
```

В этом примере `o.message` – это свойство, хранящее строку, а не функцию, но, чтобы определить начальное значение свойства, потребовалась функция, которая вызывается сразу после загрузки сценария.

Преимущества и особенности использования

Немедленно вызываемые функции широко используются на практике. Они помогают выполнять необходимые операции, не оставляя за собой глобальных переменных. Все переменные, определяемые внутри таких функций, будут локальными, и вам не придется беспокоиться о засорении глобального пространства имен временными переменными.

Иногда можно услышать такие названия немедленно вызываемых функций, как «самовызывающиеся» или «самовыполняющиеся» функции, потому что функция вызывает саму себя сразу после определения.

Данный шаблон также часто используется в букмарклетах (броузерных закладках, bookmarklets), потому что букмарклеты могут выполняться в контексте любой страницы и для них важно соблюдать чистоту глобального пространства имен (и тем самым обеспечить ненавязчивость программного кода букмарклета).

Этот шаблон также дает возможность выделить отдельные особенности в самодостаточные модули. Представьте, что имеется некоторая статичная страница, которая прекрасно обходится без JavaScript. Вдохновившись принципом постепенного расширения, вы решили добавить в нее некоторый программный код, расширяющий возможности страницы. Этот программный код (вы можете называть его «модулем» или «особенностью») можно заключить в немедленно вызываемую функцию и убедиться, что страница по-прежнему прекрасно работает без него. Затем вы можете добавлять дополнительные расширения, удалять их, тестировать, давать пользователям возможность запрещать их и так далее.

Такой функциональный элемент (назовем его `module1`) можно определить с помощью следующего шаблона:

```
// имя module1 определено в файле module1.js
(function () {

    // реализация модуля module1...

}());
```

Следуя этому же шаблону, вы можете создавать другие модули. Затем, когда придет время перенести программный код на действующий сайт, вы сможете решить, какие из реализованных особенностей готовы к работе, и включить соответствующие файлы с помощью вашего сценария сборки.

Немедленная инициализация объектов

Другой способ, позволяющий защитить глобальное пространство имен от засорения, напоминает шаблон немедленно вызываемых функций, описанный выше, и заключается в следовании шаблону *немедленной инициализации объектов*. Этот шаблон опирается на использование объектов, обладающих методом `init()`, который вызывается сразу после создания объекта. Функция `init()` реализует все необходимые операции по инициализации.

Ниже приводится пример использования шаблона немедленной инициализации объектов:

```
({
    // здесь можно определить параметры настройки
    // или конфигурационные константы
    maxwidth: 600,
    maxheight: 400,

    // также можно определять вспомогательные методы
    gimmeMax: function () {
```

```
        return this.maxwidth + "x" + this.maxheight;
    },

    // инициализация
    init: function () {
        console.log(this.gimmeMax());
        // операции по инициализации...
    }
}).init();
```

С точки зрения синтаксиса этот шаблон напоминает создание обычного объекта с использованием литерала. Литерал точно так же заключается в круглые скобки (оператор группировки), благодаря которым интерпретатор JavaScript воспринимает то, что в фигурных скобках, как литерал объекта, а не как блок программного кода. (Это не инструкция `if` и не цикл `for`.) После закрывающей круглой скобки производится немедленный вызов метода `init()`.

В первую пару круглых скобок можно заключать не только определение объекта, но и вызов метода `init()`. Другими словами, обе следующие конструкции являются допустимыми:

```
({...}).init();
({...}.init());
```

Этот шаблон обладает теми же преимуществами, что и шаблон немедленно вызываемых функций: он позволяет защитить глобальное пространство имен при выполнении операций по инициализации. С точки зрения синтаксиса такой подход может показаться более замысловатым, чем просто заключение программного кода в анонимную функцию, но, если для инициализации потребуется реализовать достаточно сложные операции (что часто случается на практике), он обеспечит дополнительное структурирование процедуры инициализации. Например, частные вспомогательные функции легко заметны, так как они являются свойствами временного объекта, тогда как при использовании шаблона немедленно вызываемых функций они, вероятнее всего, будут обычными функциями, разбросанными по сценарию.

Недостаток этого шаблона заключается в том, что многие компрессоры JavaScript могут не так эффективно сжимать этот шаблон, как программный код, обернутый простой функцией. Частным свойствам и методам не будут присвоены более короткие имена, потому что с точки зрения компрессора это выглядит небезопасным. К моменту написания этих строк Google Closure Compiler был единственным компрессором, который в «расширенном» режиме мог присваивать более короткие имена свойствам объектов с немедленной инициализацией, превращая предыдущий фрагмент в следующую строку:

```
({d:600,c:400,a:function(){return this.d+"x"+this.c},b:function(){console.
log(this.a())}}).b();
```

Этот шаблон в основном используется для однократного выполнения операций и не обеспечивает возможность доступа к объекту после завершения выполнения метода `init()`. Если вам потребуется сохранить ссылку на объект, это можно легко сделать, вернув объект из метода `init()`.

Выделение ветвей, выполняющихся на этапе инициализации

Выделение ветвей, выполняющихся на этапе инициализации (или во время загрузки), – это один из шаблонов оптимизации. Если известно, что некоторые условия не будут изменяться во время выполнения программы, есть смысл реализовать проверку этих условий так, чтобы проверка выполнялась только один раз. Типичным примером может служить определение типа броузера (или поддерживаемых им возможностей).

Например, после того как вы определили, что `XMLHttpRequest` поддерживается как встроенный объект, можно быть твердо уверенными, что тип броузера, в котором выполняется сценарий, не изменится в течение работы программы и вашему программному коду не придется иметь дело с объектами ActiveX. Поскольку среда не может измениться, нет смысла снова и снова проверять тип броузера (и приходить к одним и тем же выводам) всякий раз, когда вам потребуется создать очередной объект XHR.

Определение вычисляемых стилей элементов DOM или подключение обработчиков событий – другие кандидаты на применение шаблона выделения ветвей, выполняющихся на этапе инициализации. Большинству разработчиков за время разработки клиентских сценариев хотя бы раз приходилось создавать вспомогательный объект с методами подключения и удаления обработчиков событий, подобный показанному в следующем примере:

```
// ДО
var utils = {
    addListener: function (el, type, fn) {
        if (typeof window.addEventListener === 'function') {
            el.addEventListener(type, fn, false);
        } else if (typeof document.attachEvent === 'function') { // IE
            el.attachEvent('on' + type, fn);
        } else { // устаревшие броузеры
            el['on' + type] = fn;
        }
    },
    removeListener: function (el, type, fn) {
        // почти то же самое...
```

```
    }
};
```

Минус такой реализации заключается в том, что она несколько неэффективна. Всякий раз, когда вызывается `utils.addListener()` или `utils.removeListener()`, снова и снова выполняются одни и те же проверки.

Используя шаблон выделения ветвей, выполняющихся на этапе инициализации, можно обеспечить однократное определение особенностей, поддерживаемых броузером на этапе загрузки сценария. В этот момент можно определить, какая функция будет работать на протяжении всего времени жизни страницы. Следующий пример демонстрирует, как можно решить эту задачу:

```
// ПОСЛЕ

// интерфейс
var utils = {
    addListener: null,
    removeListener: null
};

// реализация
if (typeof window.addEventListener === 'function') {
    utils.addListener = function (el, type, fn) {
        el.addEventListener(type, fn, false);
    };
    utils.removeListener = function (el, type, fn) {
        el.removeEventListener(type, fn, false);
    };
} else if (typeof document.attachEvent === 'function') { // IE
    utils.addListener = function (el, type, fn) {
        el.attachEvent('on' + type, fn);
    };
    utils.removeListener = function (el, type, fn) {
        el.detachEvent('on' + type, fn);
    };
} else { // устаревшие броузеры
    utils.addListener = function (el, type, fn) {
        el['on' + type] = fn;
    };
    utils.removeListener = function (el, type, fn) {
        el['on' + type] = null;
    };
}
```

Сейчас самое время сказать несколько слов против определения типа броузера. При использовании этого шаблона не следует делать скоропалительных выводов о том, какие особенности поддерживаются броузером. Например, если вы определили, что броузер не поддерживает

метод window.addEventListener, не следует тут же делать вывод, что сценарий выполняется в броузере IE, который не поддерживает встроенный объект XMLHttpRequest, несмотря на то, что на определенном этапе развития броузера это так и было. Бывают ситуации, когда можно с достаточной степенью уверенности говорить о существовании поддержки одной особенности исходя из наличия поддержки другой особенности. Например, по наличию метода .addEventListener можно судить о наличии метода .removeEventListener, но в большинстве случаев поддержка одних особенностей не зависит от других. Лучше всего определять наличие особенностей независимо друг от друга и применять шаблон выделения ветвей, выполняющихся на этапе инициализации, чтобы такие проверки выполнялись только один раз.

Свойства функций – шаблон мемоизации

Функции – это объекты, поэтому они могут иметь свойства. Фактически функции уже при создании имеют некоторые предопределенные свойства и методы. Например, каждая функция, независимо от того, как она создавалась, автоматически получает свойство length, содержащее число аргументов, ожидаемых функцией:

```
function func(a, b, c) {}
console.log(func.length); // 3
```

В любой момент времени вы можете добавлять к функциям свои свойства. Эти нестандартные свойства могут использоваться, например, для кэширования результатов (возвращаемого значения) функции, чтобы при следующем обращении к функции ей не приходилось выполнять сложные и продолжительные вычисления. Прием кэширования результатов функции известен еще под названием *мемоизация* (*memorization*).

В следующем примере функция myFunc создает свойство cache, доступное как myFunc.cache. Свойство cache – это объект (хеш), в котором параметр param, переданный функции, используется в качестве ключа, а результат вычислений – в качестве значения. Результат может быть любой сложной структурой данных, какая только может вам потребоваться:

```
var myFunc = function (param) {
    if (!myFunc.cache[param]) {
        var result = {};
        // ... продолжительные операции ...
        myFunc.cache[param] = result;
    }
    return myFunc.cache[param];
};

// создание хранилища результатов
myFunc.cache = {};
```

В предыдущем фрагменте предполагается, что функция принимает единственный аргумент `param` элементарного типа (например, строка). Если ваша функция принимает большее число параметров, имеющих более сложные типы, в качестве универсального решения можно было бы порекомендовать сериализовать их. Например, аргументы в виде объектов можно было бы преобразовать в строку JSON и использовать эту строку как ключ в объекте `cache`:

```
var myFunc = function () {

    var cachekey = JSON.stringify(Array.prototype.slice.call(arguments)),
        result;

    if (!myFunc.cache[cachekey]) {
        result = {};
        // ... продолжительные операции ...
        myFunc.cache[cachekey] = result;
    }
    return myFunc.cache[cachekey];
};

// создание хранилища результатов
myFunc.cache = {};
```

Имейте в виду, что при сериализации «идентичность» объектов не проверяется. Если у вас имеется два различных объекта, которые по случайности обладают одними и теми же свойствами, обоим этим объектам в кэше будет соответствовать одна и та же запись.

Другой способ реализовать предыдущую функцию основан на использовании свойства `arguments.callee`, ссылающегося на функцию вместо имени функции. В настоящее время это еще возможно, но имейте в виду, что в строгом режиме, определяемом стандартом ECMAScript 5, свойство `arguments.callee` не поддерживается:

```
var myFunc = function (param) {

    var f = arguments.callee,
        result;

    if (!f.cache[param]) {
        result = {};
        // ... продолжительные операции ...
        f.cache[param] = result;
    }
    return f.cache[param];
};

// создание хранилища результатов
myFunc.cache = {};
```

Объекты с параметрами

Шаблон использования объектов с параметрами позволяет обеспечить более простой прикладной интерфейс и особенно полезен для тех, кто занимается разработкой библиотек или другого программного кода, который включается в состав других программ.

Ни для кого не секрет, что требования к программному обеспечению изменяются в процессе его развития. Часто случается так, что разработка начинается с учетом одних требований, но в процессе разработки добавляются все новые и новые функциональные возможности.

Представьте, что вы пишете функцию `addPerson()`, которая принимает имя и фамилию и добавляет их в некоторый список:

```
function addPerson(first, last) {...}
```

Позднее появляется необходимость вместе с именем и фамилией сохранять еще и дату рождения, а также по возможности пол и адрес. Исходя из этого вы изменяете функцию и добавляете в нее новые параметры (помещая необязательные параметры в конец списка):

```
function addPerson(first, last, dob, gender, address) {...}
```

В настоящий момент сигнатура функции получилась уже достаточно длинной. И тут вы узнаете, что совершенно необходимо добавить еще один обязательный параметр, в котором будет передаваться имя пользователя. Теперь программист вынужден будет указывать даже необязательные параметры и внимательно следить, чтобы не перепутать порядок их следования:

```
addPerson("Bruce", "Wayne", new Date(), null, null, "batman");
```

Передавать большое количество параметров очень неудобно. Лучше было бы заменить все параметры одним параметром и сделать его объектом. Давайте создадим такой параметр и назовем его `conf`:

```
addPerson(conf);
```

Теперь вызов функции можно будет оформить, как показано ниже:

```
var conf = {
    username: "batman",
    first: "Bruce",
    last: "Wayne"
};
addPerson(conf);
```

Объекты с параметрами обладают следующими достоинствами:

- Не требуется запоминать количество и порядок следования параметров
- Можно не указывать необязательные параметры

- Программный код, в котором используются объекты с параметрами, легко читается и прост в сопровождении
- Упрощается возможность добавления и удаления параметров

А из недостатков можно указать следующие:

- Необходимо помнить имена параметров
- Имена свойств не поддаются сжатию с помощью компрессоров

Этот шаблон с успехом может использоваться в функциях, которые, например, создают элементы DOM или изменяют стили CSS элементов, потому что элементы и стили могут иметь огромное количество свойств и атрибутов.

Каррирование

В оставшейся части главы будут обсуждаться темы, связанные с *каррированием* (*currying*) и с *частично* применимыми функциями. Но, прежде чем погрузиться в изучение этих тем, давайте сначала точно определим, что подразумевается под словами *применение функций*.

Применение функций

В некоторых функциональных языках программирования принято говорить, что функция *применяется*, а не *вызывается*. В языке JavaScript мы можем пользоваться той же терминологией – мы можем применять функции с помощью метода `Function.prototype.apply()`, потому что функции в языке JavaScript фактически являются объектами, которые имеют методы.

Ниже приводится пример применения функции:

```
// определение функции
var sayHi = function (who) {
    return "Hello" + (who ? ", " + who : "") + "!";
};

// вызов функции
sayHi();                    // "Hello!"
sayHi('world');             // "Hello, world!"

// применение функции
sayHi.apply(null, ["hello"]); // "Hello, hello!"
```

Как следует из этого примера, *вызов* функции и ее *применение* дают один и тот же результат. Метод `apply()` принимает два параметра: в первом передается объект, который будет передан функции через ссылку `this`, а во втором – массив аргументов, который затем будет преобразован в объект `arguments`, похожий на массив, доступный внутри функции. Если в первом параметре передать значение `null`, то ссылка `this` бу-

дет указывать на глобальный объект, что и происходит, когда функция вызывается не как метод определенного объекта.

Когда функция вызывается как метод объекта, методу `apply()` передается ссылка на этот объект, а не значение `null` (как в предыдущем примере). В следующем примере в первом аргументе методу `apply()` передается ссылка на объект:

```
var alien = {
    sayHi: function (who) {
        return "Hello" + (who ? ", " + who : "") + "!";
    }
};

alien.sayHi('world');                  // "Hello, world!"
alien.sayHi.apply(alien, ["humans"]); // "Hello, humans!"
```

В этом примере внутри функции `sayHi()` ссылка `this` будет указывать на объект `alien`. В предыдущем примере она будет указывать на глобальный объект.

Эти два примера демонстрируют, что вызов функции, как оказывается, – это не более чем синтаксический сахар[1], эквивалентный применению функций.

Обратите внимание, что помимо метода `apply()` объект `Function.prototype` обладает еще и методом `call()`, но и он является всего лишь синтаксическим сахаром, реализованным поверх метода `apply()`. Есть случаи, когда полезно пользоваться этим приемом: если функция принимает единственный параметр, можно избежать необходимости создавать массив с одним элементом:

```
// второй способ более эффективен, в нем не требуется создавать массив
alien.sayHi.apply(alien, ["humans"]); // "Hello, humans!"
alien.sayHi.call(alien, "humans");    // "Hello, humans!"
```

Частичное применение

Теперь, когда мы знаем, что вызов функции фактически является применением множества аргументов к функции, можно попробовать ответить на вопрос: как обеспечить возможность передачи функции не всех аргументов, а только их части? Это похоже на то, как бы вы действовали в обычной жизни, выполняя арифметическое действие вручную.

Допустим, что имеется функция `add()`, вычисляющая сумму двух чисел: `x` и `y`. Следующий фрагмент демонстрирует, как бы вы находили сумму исходя из того, что `x` имеет значение 5, а `y` имеет значение 4:

---

[1] Синтаксический сахар (syntactic sugar) – термин, обозначающий дополнения синтаксиса, которые не добавляют новых возможностей, а делают использование языка более удобным для человека. – *Прим. перев.*

```
// исключительно в демонстрационных целях
// это недопустимый программный код в JavaScript

// у нас имеется такая функция
function add(x, y) {
    return x + y;
}

// и нам известны значения аргументов
add(5, 4);

// шаг 1 – подстановка первого аргумента
function add(5, y) {
    return 5 + y;
}

// шаг 2 – подстановка второго аргумента
function add(5, 4) {
    return 5 + 4;
}
```

В этом фрагменте шаги 1 и 2 представлены недопустимым программным кодом в JavaScript, но они наглядно показывают, как бы вы решали эту задачу вручную. Вы взяли бы значение первого аргумента и заменили бы неизвестное значение `x` известным значением 5. Затем то же самое проделали бы с остальными аргументами.

Шаг 1 в этом примере можно было бы назвать частичным применением: мы применили только первый аргумент. Производя частичное применение, вы не получите результат (решение), а вместо этого получите другую функцию.

Следующий фрагмент демонстрирует использование воображаемого метода `partialApply()`:

```
var add = function (x, y) {
    return x + y;
};

// полное применение
add.apply(null, [5, 4]); // 9

// частичное применение
var newadd = add.partialApply(null, [5]);

// применение аргумента к новой функции
newadd.apply(null, [4]); // 9
```

Как видите, в результате частичного применения получается другая функция, которую можно вызывать с оставшимися аргументами. Фактически она является эквивалентом воображаемой функции `add(5)(4)`, так как `add(5)` возвращает функцию, которая может быть вызвана с ар-

гументом (4). И снова привычную форму записи add(5, 4) можно воспринимать как синтаксический сахар, используемый вместо формы записи add(5)(4).

Теперь вернемся обратно на землю: функции в языке JavaScript не имеют метода partialApply() и по умолчанию не ведут себя подобным образом. Но вы можете реализовать такое поведение, потому что JavaScript – достаточно динамичный язык программирования, чтобы позволить это.

Процесс, в результате которого появляются функции, обладающие возможностью частичного применения, называется *каррированием* (*currying*).

Каррирование

Термин *каррирование* не имеет никакого отношения к индийским пряностям. Он происходит от имени математика Хаскелла Карри (Haskell Curry). (Язык программирования Haskell также назван в его честь.) Каррирование – это процесс преобразования, в данном случае – процесс преобразования функции. Процесс каррирования можно было бы также назвать *шейнфинкелизацией* (*schönfinkelisation*), в честь еще одного математика, Моисея Исаевича Шейнфинкеля (Moses Schönfinkel), первым предложившего это преобразование.

Так как же выполняется шейнфинкелизация (или каррирование) функций? В других – функциональных – языках программирования такая возможность может быть встроена непосредственно в сам язык, и все функции по умолчанию уже могут быть каррированы. А в JavaScript мы можем изменить реализацию функции add() так, чтобы она предусматривала возможность частичного применения.

Рассмотрим следующий пример:

```
// каррированная функция add()
// принимает неполный список аргументов
function add(x, y) {
    var oldx = x, oldy = y;
    if (typeof oldy === "undefined") { // частичное применение
        return function (newy) {
            return oldx + newy;
        };
    }
    // полное применение
    return x + y;
}

// проверка
typeof add(5); // "function"
add(3)(4);     // 7
```

```
// создать и сохранить новую функцию
var add2000 = add(2000);
add2000(10);   // 2010
```

В этом фрагменте первый вызов `add()` создает замыкание с внутренней функцией, которая возвращается в виде результата. Оригинальные значения `x` и `y` сохраняются в замыкании в частных переменных `oldx` и `oldy`. Первая из них, `oldx`, используется внутренней функцией. В ситуации полного применения, когда функции `add()` передаются оба аргумента `x` и `y`, она просто складывает их. Такая реализация функции `add()` несколько избыточна, и сделано это исключительно в демонстрационных целях. Более компактная версия показана в следующем фрагменте – в ней отсутствуют переменные `oldx` и `oldy`, потому что оригинальный аргумент `x` неявно сохраняется в замыкании, а вместо переменной `newy` используется `y`:

```
// каррированная функция add()
// принимает неполный список аргументов
function add(x, y) {
    if (typeof y === "undefined") { // partial
        return function (y) {
            return x + y;
        };
    }
    // полное применение
    return x + y;
}
```

В этих примерах реализация обработки ситуации частичного применения находится в самой функции `add()`. Но существует ли более универсальный способ? Другими словами, возможно ли преобразовать произвольную функцию в новую, принимающую неполный список параметров? В следующем фрагменте демонстрируется функция, назовем ее `schonfinkelize()`, которая как раз это и делает. Мы использовали имя `schonfinkelize()` отчасти потому, что оно сложнее в произношении, а отчасти потому, что оно звучит как глагол (имя «curry» в данном случае несколько неоднозначно), а нам необходим глагол, чтобы подчеркнуть, что функция выполняет преобразование.

Ниже приводится реализация функции, выполняющей каррирование:

```
function schonfinkelize(fn) {
    var slice = Array.prototype.slice,
        stored_args = slice.call(arguments, 1);
    return function () {
        var new_args = slice.call(arguments),
            args = stored_args.concat(new_args);
        return fn.apply(null, args);
    };
}
```

Функция schonfinkelize() получилась чуть более сложной, чем могла бы быть, но только из-за того, что объект arguments в JavaScript не является настоящим массивом. Метод slice(), заимствованный из объекта Array.prototype, помогает нам превратить объект arguments в массив и получить более удобный способ работы с ним. Когда функция schonfinkelize() вызывается в первый раз, она сохраняет ссылку на метод slice() в частной переменной (с именем slice), а также сохраняет переданные ей аргументы (в переменной stored_args), отбрасывая первый аргумент, потому что в первом аргументе передается каррируемая функция. Затем schonfinkelize() возвращает новую функцию. Когда новая функция вызывается, она имеет доступ (благодаря замыканию) к аргументам, сохраненным ранее в stored_args, и к ссылке на метод slice. Новая функция просто объединяет ранее примененные аргументы (stored_args) с новыми (new_args) и затем применяет их к оригинальной функции fn (ссылка на которую также сохраняется в замыкании).

Теперь, вооружившись новым универсальным способом каррирования функций, выполним несколько экспериментов:

```
// обычная функция
function add(x, y) {
    return x + y;
}

// каррировать существующую функцию и получить новую
var newadd = schonfinkelize(add, 5);
newadd(4);                                  // 9

// другой вариант – вызвать новую функцию сразу же
schonfinkelize(add, 6)(7);                  // 13
```

Возможности функции преобразования schonfinkelize() не ограничиваются единственным параметром или одношаговым преобразованием. Ниже приводятся еще несколько примеров ее использования:

```
// обычная функция
function add(a, b, c, d, e) {
    return a + b + c + d + e;
}

// может обрабатывать любое количество аргументов
schonfinkelize(add, 1, 2, 3)(5, 5);         // 16

// может выполнять каррирование в два этапа
var addOne = schonfinkelize(add, 1);
addOne(10, 10, 10, 10);                     // 41
var addSix = schonfinkelize(addOne, 2, 3);
addSix(5, 5);                               // 16
```

Когда использовать каррирование

Если обнаружится, что вы неоднократно вызываете одну и ту же функцию, передавая ей практически одни и те же параметры, эта функция наверняка является отличным кандидатом на каррирование. Вы можете создать новую функцию, используя прием применения части параметров к оригинальной функции. Новая функция будет хранить повторяющиеся параметры (благодаря чему вам не придется передавать их каждый раз) и использовать их для заполнения полного списка аргументов, ожидаемых оригинальной функцией.

В заключение

Знание особенностей функций и правил их использования в языке JavaScript имеет критически важное значение. В этой главе обсуждались терминология и основы применения функций. Вы узнали о двух важных особенностях функций в JavaScript, а именно:

1. Функции – это *обычные объекты*, они могут передаваться как обычные значения и расширяться новыми свойствами и методами.
2. Функции образуют *локальную область видимости*, которую невозможно создать с помощью простой пары фигурных скобок. Кроме того, не забывайте, что объявления локальных переменных неявно поднимаются в начало локальной области видимости.

В JavaScript существуют следующие синтаксические разновидности функций:

1. *Именованные функции-выражения.*
2. *Функции-выражения* (точно такие же функции, что и выше, но не имеющие имени), также известные как *анонимные функции.*
3. *Функции-объявления*, похожие на функции в других языках программирования.

После знакомства с основами и синтаксисом функций мы рассмотрели множество полезных шаблонов, сгруппированных в следующие категории:

1. *Шаблоны API*, которые помогают создавать более простые и удобные интерфейсы для функций. Эта категория включает:

   Шаблон применения функций обратного вызова

   Предусматривает передачу функций в виде аргументов.

   Объекты с параметрами

   Помогает удерживать количество аргументов функций под контролем.

Возвращение функций

Когда возвращаемым значением одной функции является другая функция.

Каррирование

Когда на основе существующей создается новая функция с частично подготовленным списком аргументов.

2. *Шаблоны инициализации*, помогающие выполнять операции инициализации и настройки (весьма типичные для веб-страниц и приложений) ясным структурированным способом без засорения глобального пространства имен временными переменными. В их числе мы рассмотрели:

   Немедленно вызываемые функции

   Автоматически выполняются сразу же после определения.

   Немедленную инициализацию объектов

   Структурирование операций инициализации в виде анонимного объекта, в котором имеется метод, вызываемый немедленно после объявления объекта.

   Выделение ветвей, выполняющихся на этапе инициализации

   Помогает выделить программный код, выполняемый только один раз на этапе инициализации, в противоположность программному коду, который выполняется многократно в течение времени жизни приложения.

3. *Шаблоны оптимизации производительности*, помогающие повысить скорость выполнения. В их числе:

   Мемоизация

   Использование свойств функций для хранения вычисленных ранее возвращаемых значений, чтобы избежать необходимости вычислять их повторно.

   Самоопределяемые функции

   Функции, которые сами определяют для себя новое тело, чтобы, начиная со второго вызова, выполнять меньше операций.

5

Шаблоны создания объектов

Создание объектов в JavaScript выполняется достаточно просто – для этого достаточно воспользоваться литералом объекта или вызвать функцию-конструктор. В этой главе мы познакомимся с несколькими дополнительными приемами, используемыми при создании объектов.

Язык программирования JavaScript отличается простотой и очевидностью, и в нем отсутствуют многие специализированные синтаксические конструкции, имеющиеся в других языках, такие как пространства имен, модули, пакеты, частные свойства и статические члены. В этой главе вы познакомитесь с общими шаблонами, позволяющими реализовать или просто представлять иначе все эти особенности.

Сначала в этой главе мы познакомимся с приемами создания пространств имен и модулей, с возможностью объявления зависимостей и выделения изолированного пространства имен. Все это поможет вам организовать и структурировать программный код в приложениях и ослабить влияние механизма подразумеваемых глобальных переменных. В числе других тем мы обсудим частные и привилегированные члены, статические и частные статические члены, объекты-константы, составление цепочек из вызовов методов и еще один способ определения конструкторов.

Пространство имен

Пространства имен помогают уменьшить количество глобальных переменных, необходимых нашим программам, и одновременно избежать конфликтов имен и чрезмерного употребления префиксов.

В синтаксисе языка JavaScript отсутствуют конструкции определения пространств имен, но эту особенность легко реализовать другими способами. Вместо того чтобы засорять глобальное пространство имен большим количеством функций, объектов и других переменных, мож-

но создать один (в идеале только один) глобальный объект, который будет служить пространством имен для приложения или библиотеки. После этого вы сможете добавлять всю необходимую функциональность в этот объект.

Взгляните на следующий пример:

```
// ДО: 5 глобальных переменных
// Внимание: антишаблон

// конструкторы
function Parent() {}
function Child() {}

// переменная
var some_var = 1;

// пара объектов
var module1 = {};
module1.data = {a: 1, b: 2};
var module2 = {};
```

Такой программный код легко переписать иным способом, создав единственный глобальный объект, назовем его MYAPP, и превратив все функции и переменные в свойства этого глобального объекта:

```
// ПОСЛЕ: 1 глобальная переменная

// глобальный объект
var MYAPP = {};

// конструкторы
MYAPP.Parent = function () {};
MYAPP.Child = function () {};

// переменная
MYAPP.some_var = 1;

// объект-контейнер
MYAPP.modules = {};

// вложенные объекты
MYAPP.modules.module1 = {};
MYAPP.modules.module1.data = {a: 1, b: 2};
MYAPP.modules.module2 = {};
```

В качестве имени глобального объекта пространства имен вы можете выбрать, например, имя вашего приложения или библиотеки, доменное имя или название своей компании. Часто разработчики следуют соглашению об именовании, согласно которому имена глобальных переменных конструируются только из ЗАГЛАВНЫХ СИМВОЛОВ, благодаря чему они сразу будут бросаться в глаза тем, кто будет читать программ-

ный код. (Но имейте в виду, что то же соглашение часто используется при выборе имен для констант.)

Этот шаблон обеспечивает возможность создания пространств имен для вашего программного кода и позволяет избежать конфликтов имен как внутри ваших приложений, так и между вашим программным кодом и сторонними библиотеками, например с библиотеками JavaScript или с комплектами виджетов. Этот шаблон отлично подходит для большинства задач, но он имеет некоторые недостатки:

- Увеличивается объем ввода с клавиатуры, так как приходится снабжать префиксом каждую переменную и каждую функцию, что в результате увеличивает размер загружаемого файла.
- Наличие единственного экземпляра глобального объекта подразумевает, что любая часть программного кода может внести в него изменения, при этом остальная часть программного кода будет пользоваться измененным объектом.
- Удлинение имен вложенных переменных подразумевает увеличение времени, необходимого на разрешение имен.

Эти недостатки устраняет шаблон изолированного пространства имен, который обсуждается ниже в этой главе.

Универсальная функция для создания пространства имен

С ростом сложности программы некоторые фрагменты программного кода приходится выносить в отдельные файлы и подключать их при определенных условиях. Вследствие этого становится безосновательным предполагать, что ваш файл первым определит некоторое пространство имен или свойство в нем. Вполне возможно, что некоторые свойства, которые предполагается добавить, уже существуют, и вы можете затереть их по неосторожности. Поэтому, прежде чем добавлять свойство или создавать пространство имен, желательно убедиться, что оно еще не создано, как показано в следующем примере:

```
// небезопасный способ
var MYAPP = {};

// так лучше
if (typeof MYAPP === "undefined") {
    var MYAPP = {};
}

// или немного короче
var MYAPP = MYAPP || {};
```

Легко понять, что такие проверки быстро могут превратиться в огромный объем повторяющегося программного кода. Например, если потребуется создать свойство `MYAPP.modules.module2`, необходимо будет вы-

полнить три проверки, по одной для каждого создаваемого объекта или свойства. Поэтому было бы очень удобно иметь функцию, которая взяла бы на себя выполнение всех операций, необходимых для создания пространства имен. Назовем эту функцию `namespace()` и положим, что она должна использоваться, как показано ниже:

```
// применение функции пространства имен
MYAPP.namespace('MYAPP.modules.module2');

// этот вызов эквивалентен следующей конструкции:
// var MYAPP = {
//     modules: {
//         module2: {}
//     }
// };
```

Далее приводится пример реализации этой функции, в котором использован принцип неразрушения, то есть если пространство имен с заданным именем уже существует, оно не будет создано заново:

```
var MYAPP = MYAPP || {};

MYAPP.namespace = function (ns_string) {
    var parts = ns_string.split('.'),
        parent = MYAPP,
        i;

    // отбросить начальный префикс - имя глобального объекта
    if (parts[0] === "MYAPP") {
        parts = parts.slice(1);
    }

    for (i = 0; i < parts.length; i += 1) {
        // создать свойство, если оно отсутствует
        if (typeof parent[parts[i]] === "undefined") {
            parent[parts[i]] = {};
        }
        parent = parent[parts[i]];
    }
    return parent;
};
```

Такая реализация делает допустимыми все следующие варианты использования функции:

```
// присваивать возвращаемое значение локальной переменной
var module2 = MYAPP.namespace('MYAPP.modules.module2');
module2 === MYAPP.modules.module2; // true

// опускать начальный префикс `MYAPP`
MYAPP.namespace('modules.module51');
```

```
// создавать глубоко вложенные пространства имен
MYAPP.namespace('once.upon.a.time.there.was.this.long.nested.property');
```

На рис. 5.1 показано, как выглядят пространства имен, созданные в предыдущем примере, в расширении Firebug.

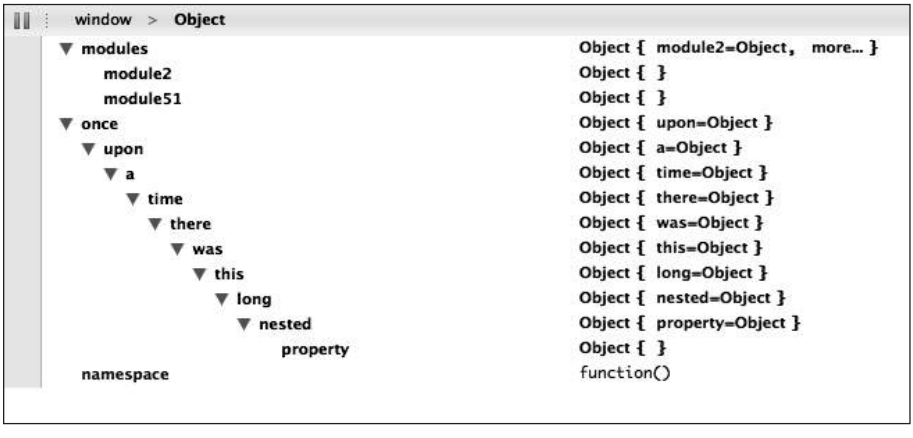

Рис. 5.1. *Пространство MYAPP в Firebug*

Объявление зависимостей

Библиотеки JavaScript часто имеют модульную архитектуру и определяют собственные пространства имен, что позволяет подключать к приложению только необходимые модули. Например, в библиотеке YUI2 существует глобальная переменная `YAHOO`, которая играет роль пространства имен, и ряд модулей, являющихся свойствами этой глобальной переменной, такие как `YAHOO.util.Dom` (модуль DOM) и `YAHOO.util.Event` (модуль Events).

Было бы неплохо объявлять модули, необходимые программному коду, в начале функции или модуля. Для такого объявления достаточно лишь локальных переменных, ссылающихся на требуемые модули:

```
var myFunction = function () {
    // зависимости
    var event = YAHOO.util.Event,
        dom = YAHOO.util.Dom;

    // остальная часть функции
    // использует переменные event и dom...
};
```

Это удивительно простой шаблон, обладающий многочисленными преимуществами:

- Явное объявление зависимостей сообщает пользователям вашего программного кода, какие файлы необходимо подключить к странице.
- Опережающее объявление в начале функции упрощает поиск и разрешение зависимостей.
- Операции с локальными переменными (такими как `dom`) всегда выполняются быстрее, чем с глобальными (такими как `YAHOO`), и даже быстрее, чем с вложенными свойствами глобальной переменной (такими как `YAHOO.util.Dom`), поэтому их использование позволяет увеличить производительность. Применение шаблона объявления зависимостей позволяет выполнять разрешение глобальных имен в функции только один раз. После этого везде в функции будет использоваться локальная переменная, обеспечивая более высокую скорость выполнения.
- Улучшенные компрессоры JavaScript, такие как YUICompressor и Google Closure Compiler, способны переименовывать локальные переменные (в результате чего переменной `event` наверняка будет присвоено односимвольное имя, например `A`), уменьшая объем программного кода, но они никогда не переименовывают глобальные переменные, потому что это небезопасно.

Следующий фрагмент иллюстрирует влияние шаблона объявления зависимостей на сжатие программного кода. Хотя функция `test2()`, следующая шаблону, выглядит немного сложнее, потому что содержит больше строк кода из-за добавления дополнительной переменной, тем не менее после компрессии она оказывается меньше, а это означает, что размер загружаемого файла также уменьшится:

```
function test1() {
    alert(MYAPP.modules.m1);
    alert(MYAPP.modules.m2);
    alert(MYAPP.modules.m51);
}

/*
Тело сжатой функции test1:
alert(MYAPP.modules.m1);alert(MYAPP.modules.m2);alert(MYAPP.modules.m51)
*/

function test2() {
    var modules = MYAPP.modules;
    alert(modules.m1);
    alert(modules.m2);
    alert(modules.m51);
}

/*
Тело сжатой функции test2:
var a=MYAPP.modules;alert(a.m1);alert(a.m2);alert(a.m51)
*/
```

Частные свойства и методы

В языке JavaScript нет специальных средств объявления частных (private), защищенных (protected) или общедоступных (public) свойств и методов, как в языке Java или в других языках. Все члены объектов в этом языке являются общедоступными:

```
var myobj = {
    myprop: 1,
    getProp: function () {
        return this.myprop;
    }
};

console.log(myobj.myprop);    // `myprop` - общедоступный член
console.log(myobj.getProp()); // getProp() - также общедоступный член
```

То же справедливо и при использовании функций-конструкторов для создания объектов – все члены являются общедоступными:

```
function Gadget() {
    this.name = 'iPod';
    this.stretch = function () {
        return 'iPad';
    };
}
var toy = new Gadget();
console.log(toy.name);      // `name` - общедоступный член
console.log(toy.stretch()); // stretch() - общедоступный член
```

Частные члены

Несмотря на отсутствие в языке специальных средств определения частных членов, их все-таки можно создать, используя для этого замыкания. Функция-конструктор может образовывать замыкание, и любые переменные, ставшие частью этого замыкания, не будут доступны за пределами объекта. Однако такие частные члены останутся доступными для общедоступных методов – методов, определяемых внутри конструктора и являющихся частью интерфейса возвращаемого объекта. Давайте рассмотрим пример создания частного члена, недоступного за пределами объекта:

```
function Gadget() {
    // частный член
    var name = 'iPod';
    // общедоступная функция
    this.getName = function () {
        return name;
    };
}
var toy = new Gadget();
```

```
// имя `name` не определено, частный член
console.log(toy.name);      // undefined

// общедоступный метод может обратиться к частному члену `name`
console.log(toy.getName()); // "iPod"
```

Как видите, в JavaScript можно легко создавать частные члены. Все, что для этого необходимо, – обернуть данные, которые вы хотели бы оставить частными, в функцию, сделав их локальными по отношению к этой функции и, следовательно, недоступными за пределами функции.

Привилегированные методы

Под понятием *привилегированный метод* не подразумевается какая-либо особая синтаксическая конструкция – так называются общедоступные методы, обладающие доступом к частным членам (то есть имеющие более высокие привилегии).

В предыдущем примере `getName()` – привилегированный метод, потому что он имеет «особый» доступ к частному свойству `name`.

Нежелательный доступ к частным членам

Существует несколько краевых случаев, связанных с нежелательным получением доступа к частным членам:

- В некоторых ранних версиях Firefox функции `eval()` можно было передать второй параметр, определяющий контекстный объект, что позволяло вторгаться в частную область видимости функции. Точно так же свойство `__parent__` в Mozilla Rhino позволяет получить доступ к частной области видимости. Эти проблемы отсутствуют в широко используемых современных броузерах.
- Если привилегированный метод возвращает частную переменную непосредственно и эта переменная является объектом или массивом, внешний программный код сможет изменить такую частную переменную, потому что объекты и массивы передаются по ссылке.

Рассмотрим вторую проблему немного подробнее. Следующая реализация конструктора `Gadget` выглядит достаточно невинной:

```
function Gadget() {
    // частный член
    var specs = {
        screen_width: 320,
        screen_height: 480,
        color: "white"
    };

    // общедоступная функция
    this.getSpecs = function () {
```

```
        return specs;
    };
}
```

Проблема здесь заключается в том, что метод `getSpecs()` возвращает ссылку на объект `specs`. Это дает пользователю конструктора `Gadget` возможность изменять, казалось бы, скрытую и частную переменную `specs`:

```
var toy = new Gadget(),
    specs = toy.getSpecs();

specs.color = "black";
specs.price = "free";

console.dir(toy.getSpecs());
```

Результат работы этого фрагмента в консоли Firebug показан на рис. 5.2.

***Рис. 5.2.** Частный объект был изменен*

Чтобы избежать этого неожиданного эффекта, не следует возвращать ссылки на объекты и массивы, которые должны оставаться частными. Одно из возможных решений состоит в том, чтобы в методе `getSpecs()` создавать и возвращать новый объект, содержащий только те данные, которые могут представлять интерес для получателя этого объекта. Этот прием известен как принцип минимально необходимых полномочий (Principle of Least Authority, POLA), который гласит, что вы никогда не должны отдавать больше, чем это необходимо. В данном случае, если получателю объекта `Gadget` требуется определить, уместится ли этот объект в области определенного размера, ему достаточно будет передать только размеры. Поэтому вместо того чтобы отдавать все, что имеется, можно создать метод `getDimensions()`, который будет возвращать новый объект, содержащий только ширину и высоту. При таком подходе надобность в методе `getSpecs()` вообще может отпасть.

Другой подход, когда действительно требуется вернуть все данные, заключается в том, чтобы создать копию объекта `specs` с помощью универсальной функции копирования объектов. В следующей главе вашему вниманию будут представлены две такие функции: одна с именем `extend()`, создающая поверхностную копию указанного объекта (она копирует только параметры верхнего уровня), и другая с именем `extendDeep()`, создающая полную копию, выполняя рекурсивное копирование всех свойств и вложенных в них свойств.

Частные члены и литералы объектов

До сих пор все примеры реализации частных свойств, которые мы видели, были основаны на использовании конструкторов. А как быть в случаях, когда объекты определяются в виде литералов? Возможно ли в таких ситуациях создавать частные члены?

Как вы уже видели, чтобы обеспечить сокрытие данных, их необходимо обернуть функцией. Так, в случае литералов объектов замыкание можно создать с помощью дополнительной анонимной функции, вызываемой немедленно. Например:

```
var myobj; // это будет объект
(function () {
    // частные члены
    var name = "my, oh my";

    // реализация общедоступных членов
    // обратите внимание на отсутствие инструкции `var`
    myobj = {
        // привилегированный метод
        getName: function () {
            return name;
        }
    };
}());
myobj.getName(); // "my, oh my"
```

Та же идея положена в основу следующего примера, имеющего несколько иную реализацию:

```
var myobj = (function () {
    // частные члены
    var name = "my, oh my";

    // реализация общедоступных членов
    return {
        getName: function () {
            return name;
        }
    };
}());

myobj.getName(); // "my, oh my"
```

Этот шаблон является также основой шаблона, известного под названием «модуль», исследованием которого мы займемся чуть ниже.

Частные члены и прототипы

Один из недостатков создания частных членов с применением конструкторов заключается в том, что они создаются всякий раз, когда вызывается конструктор для создания нового объекта.

Фактически эта проблема относится ко всем членам, добавляемым внутри конструкторов. Чтобы сэкономить свои усилия и память, общие для всех экземпляров свойства и методы можно добавить в свойство `prototype` конструктора. При таком подходе общие члены будут совместно использоваться всеми экземплярами, созданными с помощью одного и того же конструктора. Аналогичным образом можно определять частные свойства, совместно используемые всеми экземплярами. Для этого необходимо применить комбинацию из двух шаблонов: частные свойства внутри конструкторов и частные свойства в литералах объектов. Так как свойство `prototype` является обычным объектом, его можно определить в виде литерала.

Как это сделать, показано в следующем примере:

```
function Gadget() {
    // частный член
    var name = 'iPod';

    // общедоступная функция
    this.getName = function () {
        return name;
    };
}

Gadget.prototype = (function () {
    // частный член
    var browser = "Mobile Webkit";

    // общедоступные члены прототипа
    return {
        getBrowser: function () {
            return browser;
        }
    };
}());

var toy = new Gadget();
console.log(toy.getName());    // "собственный" привилегированный метод
console.log(toy.getBrowser()); // привилегированный метод прототипа
```

Объявление частных функций общедоступными методами

Шаблон открытия касается частных методов, к которым вы открываете доступ как к общедоступным методам. Этот шаблон может пригодиться, когда все функциональные возможности объекта являются критически важными для его нормального функционирования, и для вас было бы желательно защитить их настолько, насколько это возможно. Но в то же время необходимо предоставить возможность доступа к некоторым из этих методов извне, так как это могло бы быть полезным

в некоторых ситуациях. Когда вы открываете доступ к методам, вы делаете их уязвимыми – общедоступные методы могут легко изменяться пользователями, порой непреднамеренно. В стандарте ECMAScript 5 имеется возможность зафиксировать объект, которая не поддерживалась в предыдущих версиях языка. Представляем вашему вниманию шаблон открытия (этот термин, введенный Кристианом Хейлманном (Christian Heilmann), первоначально употреблялся применительно к шаблону «открытия модуля»).

Рассмотрим пример, основанный на одном из шаблонов сокрытия данных – определении частных членов в литералах объектов:

```
var myarray;

(function () {

    var astr = "[object Array]",
        toString = Object.prototype.toString;

    function isArray(a) {
        return toString.call(a) === astr;
    }

    function indexOf(haystack, needle) {
        var i = 0,
            max = haystack.length;
        for (; i < max; i += 1) {
            if (haystack[i] === needle) {
                return i;
            }
        }
        return -1;
    }

    myarray = {
        isArray: isArray,
        indexOf: indexOf,
        inArray: indexOf
    };

}());
```

Здесь есть две частные переменные и две частные функции, `isArray()` и `indexOf()`. В конце немедленно вызываемой функции объект `myarray` заполняется функциональными возможностями, доступ к которым требуется открыть. В данном случае одно и то же частное свойство `indexOf()` открывается под двумя разными именами: `indexOf` в стиле ECMAScript 5 и `inArray` в духе языка PHP. Проверим наш объект `myarray`:

```
myarray.isArray([1,2]);                // true
myarray.isArray({0: 1});               // false
myarray.indexOf(["a", "b", "z"], "z"); // 2
myarray.inArray(["a", "b", "z"], "z"); // 2
```

Если теперь произойдет, например, что-то неожиданное с общедоступным методом `indexOf()`, частный метод `indexOf()` останется нетронутым, благодаря чему метод `inArray()` сохранит свою работоспособность:

```
myarray.indexOf = null;
myarray.inArray(["a", "b", "z"], "z"); // 2
```

Шаблон «модуль»

Шаблон «модуль» получил широкое распространение благодаря предоставляемой им возможности структурировать и организовать программный код по мере увеличения его объема. В отличие от других языков, в JavaScript отсутствует специальная синтаксическая конструкция для создания пакетов, но шаблон «модуль» обеспечивает все необходимое для создания независимых фрагментов программного кода, которые можно рассматривать как «черные ящики» и добавлять, заменять или удалять их из своей программы в соответствии с потребностями (возможно, изменяющимися с течением времени) вашей программы.

Шаблон «модуль» является комбинацией нескольких шаблонов, описанных выше в этой книге, а именно:

- Пространств имен
- Немедленно вызываемых функций
- Частных и привилегированных членов
- Объявления зависимостей

Первый шаг заключается в создании пространства имен. Воспользуемся функцией `namespace()`, реализованной выше в этой главе, и создадим вспомогательный модуль, содержащий методы для работы с массивами:

```
MYAPP.namespace('MYAPP.utilities.array');
```

Следующий шаг – определение модуля. На этом этапе используется немедленно вызываемая функция, образующая частную область видимости, если это необходимо. Немедленно вызываемая функция возвращает объект – собственно модуль с общедоступным интерфейсом, который может использоваться программами, в которые добавляется модуль:

```
MYAPP.utilities.array = (function () {
    return {
        // будет реализовано позже...
    };
}());
```

Затем необходимо добавить общедоступные методы:

```
MYAPP.utilities.array = (function () {
    return {
        inArray: function (needle, haystack) {
            // ...
        },
        isArray: function (a) {
            // ...
        }
    };
}());
```

В частной области видимости, образуемой немедленно вызываемой функцией, можно объявить частные свойства и методы, если это необходимо. В самом начале немедленно вызываемой функции следует также поместить все объявления зависимостей, необходимых для модуля. Вслед за объявлениями переменных можно поместить программный код инициализации модуля, который будет вызываться всего один раз. Окончательный результат – объект, возвращаемый немедленно вызываемой функцией, содержащий общедоступные члены модуля:

```
MYAPP.namespace('MYAPP.utilities.array');

MYAPP.utilities.array = (function () {

        // зависимости
    var uobj = MYAPP.utilities.object,
        ulang = MYAPP.utilities.lang,

        // частные свойства
        array_string = "[object Array]",
        ops = Object.prototype.toString;

        // частные методы
        // ...

        // конец инструкции var

    // реализация необязательной процедуры инициализации
    // ...

    // общедоступные члены
    return {

        inArray: function (needle, haystack) {
            for (var i = 0, max = haystack.length; i < max; i += 1) {
                if (haystack[i] === needle) {
                    return true;
                }
            }
        },
```

```
        isArray: function (a) {
            return ops.call(a) === array_string;
        }
        // ... другие методы и свойства
    };
}());
```

Шаблон «модуль» широко используется на практике и является рекомендуемым способом организации программного кода, особенно по мере увеличения его объема.

Шаблон открытия модуля

Мы уже рассматривали шаблон открытия в этой главе, когда обсуждали шаблоны сокрытия данных. Шаблон «модуль» может быть модифицирован аналогичным способом, когда все методы модуля являются частными и в конце определения модуля принимается решение, к каким из них следует открыть доступ.

Так, предыдущий пример можно преобразовать, как показано ниже:

```
MYAPP.utilities.array = (function () {

        // частные свойства
    var array_string = "[object Array]",
        ops = Object.prototype.toString,

        // частные методы
        inArray = function (haystack, needle) {
            for (var i = 0, max = haystack.length; i < max; i += 1) {
                if (haystack[i] === needle) {
                    return i;
                }
            }
            return -1;
        },

        isArray = function (a) {
            return ops.call(a) === array_string;
        };

        // конец инструкции var

    // открытие доступа к некоторым методам
    return {
        isArray: isArray,
        indexOf: inArray
    };
}());
```

Модули, создающие конструкторы

В предыдущем примере создается объект MYAPP.utilities.array, но иногда гораздо удобнее создавать объекты с помощью конструкторов. Для этой цели также можно использовать шаблон «модуль». Единственное отличие состоит в том, что немедленно вызываемая функция, оборачивающая модуль, возвращает функцию, а не объект.

Взгляните на следующий пример использования шаблона «модуль», который создает функцию-конструктор MYAPP.utilities.Array:

```
MYAPP.namespace('MYAPP.utilities.Array');

MYAPP.utilities.Array = (function () {

        // зависимости
    var uobj = MYAPP.utilities.object,
        ulang = MYAPP.utilities.lang,

        // частные свойства и методы...
        Constr;

        // конец инструкции var

    // реализация необязательной процедуры инициализации
    // ...

    // общедоступные методы -- конструктор
    Constr = function (o) {
        this.elements = this.toArray(o);
    };

    // общедоступные методы -- прототип
    Constr.prototype = {
        constructor: MYAPP.utilities.Array,
        version: "2.0",
        toArray: function (obj) {
            for (var i = 0, a = [], len = obj.length; i < len; i += 1) {
                a[i] = obj[i];
            }
            return a;
        }
    };

    // вернуть конструктор,
    // создающий новое пространство имен
    return Constr;

}());
```

Ниже показано, как можно использовать новый конструктор:

```
var arr = new MYAPP.utilities.Array(obj);
```

Импортирование глобальных переменных в модули

В одной из распространенных разновидностей шаблона предусматривается возможность передачи аргументов в немедленно вызываемую функцию, обертывающую модуль. Функции можно передавать любые значения, но на практике ей обычно передаются ссылки на глобальные переменные и иногда даже сам глобальный объект. Импортирование глобальных переменных позволяет увеличить скорость разрешения глобальных имен внутри немедленно вызываемой функции благодаря тому, что импортируемые переменные становятся локальными для этой функции:

```
MYAPP.utilities.module = (function (app, global) {

    // ссылки на объект global
    // и на глобальное пространство имен app
    // теперь будут локальными переменными

}(MYAPP, this));
```

Шаблон изолированного пространства имен

Шаблон изолированного пространства имен призван устранить недостатки, присущие обычному шаблону пространства имен, а именно:

- В приложении может быть только одна глобальная переменная, определяющая пространство имен. В шаблоне пространства имен не предусматривается возможность создания двух версий одного и того же пространства имен приложения или библиотеки в пределах одной и той же страницы, потому что обе версии используют одно и то же глобальное имя, например `MYAPP`.
- Требуются длинные имена, например `MYAPP.utilities.array`, которые сложнее вводить с клавиатуры, а во время выполнения на их разрешение затрачивается больше времени.

Как следует из его названия, шаблон изолированного пространства имен предоставляет каждому модулю окружение, изолированное от других модулей и их пространств имен.

Этот шаблон широко используется, например, в библиотеке YUI версии 3, но имейте в виду, что следующее ниже обсуждение – это лишь пример реализации шаблона, который не может рассматриваться как описание реализации изолированных пространств имен в библиотеке YUI3.

Глобальный конструктор

В шаблоне пространства имен имеется единственный глобальный объект; в шаблоне изолированного пространства имен единственным глобальным объектом является конструктор: назовем его `Sandbox()`. Этот конструктор используется для создания объектов пространств имен и принимает функцию обратного вызова, которая будет играть роль изолированного окружения для вашего программного кода.

Ниже показано, как используется изолированное пространство имен:

```
new Sandbox(function (box) {
    // здесь находится ваш программный код...
});
```

Объект `box` – это аналог объекта `MYAPP` в шаблоне пространства имен; он должен включать все библиотеки, необходимые для работы вашего программного кода.

Добавим в этот шаблон еще пару улучшений:

- С помощью некоторой доли волшебства (шаблон принудительного использования оператора `new` из главы 3) можно обеспечить необходимое поведение конструктора, даже когда он вызывается без оператора `new`.
- Конструктор `Sandbox()` может принимать дополнительные аргументы с настройками, определяющие имена модулей, необходимых для работы этого экземпляра объекта. Мы ставим своей целью добиться модульности кода, поэтому вся функциональность, обеспечиваемая конструктором `Sandbox()`, будет распределена по модулям.

Теперь рассмотрим несколько примеров программного кода, использующего конструктор с этими дополнительными улучшениями.

Вы можете опустить оператор `new` и создать объект, использующий некоторые фиктивные модули «ajax» и «event»:

```
Sandbox(['ajax', 'event'], function (box) {
    // console.log(box);
});
```

Следующий пример похож на предыдущий, но на этот раз имена модулей передаются в виде отдельных аргументов:

```
Sandbox('ajax', 'dom', function (box) {
    // console.log(box);
});
```

Теперь давайте определим, что шаблонный символ «*» в качестве аргумента будет означать «все доступные модули». Для удобства предположим также, что если при вызове конструктора имена модулей не передаются, то будет подразумеваться наличие аргумента со значением `'*'`. С учетом этого дополнения два способа подключения всех доступных модулей будут выглядеть, как показано ниже:

```
Sandbox('*', function (box) {
    // console.log(box);
});

Sandbox(function (box) {
    // console.log(box);
});
```

И еще один пример использования шаблона, иллюстрирующий возможность создания нескольких экземпляров объекта изолированного пространства имен; вы можете даже вкладывать их друг в друга, не вызывая конфликтов между ними:

```
Sandbox('dom', 'event', function (box) {

    // использует модули dom и event

    Sandbox('ajax', function (box) {
        // другой объект "box" изолированного пространства имен
        // этот объект "box" отличается от объекта
        // "box", находящегося за пределами этой функции

        //...

        // конец пространства имен, использующего модуль Ajax
    });

    // здесь модуль Ajax недоступен

});
```

Как видно из этих примеров, используя шаблон изолированного пространства имен, можно защитить глобальное пространство имен, обернув свой программный код функцией обратного вызова.

В случае необходимости можно также использовать в своих интересах тот факт, что функции являются объектами, и сохранять какие-то данные в «статических» свойствах конструктора `Sandbox()`.

И наконец, можно создавать различные экземпляры пространств имен в зависимости от типов подключаемых модулей, которые будут действовать независимо друг от друга.

А теперь посмотрим, как реализовать конструктор `Sandbox()` и модули, необходимые для поддержки его функциональности.

Добавление модулей

Прежде чем перейти к фактической реализации конструктора, рассмотрим, как можно реализовать добавление модулей.

Функция-конструктор `Sandbox()` кроме всего прочего является еще и объектом, поэтому мы можем добавить к ней статическое свойство с име-

нем modules. Это свойство будет еще одним объектом, содержащим пары ключ-значение, где ключами будут служить имена модулей, а значениями – функции, реализующие отдельные модули:

```
Sandbox.modules = {};

Sandbox.modules.dom = function (box) {
    box.getElement = function () {};
    box.getStyle = function () {};
    box.foo = "bar";
};

Sandbox.modules.event = function (box) {
    // при необходимости к прототипу Sandbox можно обратиться так:
    // box.constructor.prototype.m = "mmm";
    box.attachEvent = function () {};
    box.detachEvent = function () {};
};

Sandbox.modules.ajax = function (box) {
    box.makeRequest = function () {};
    box.getResponse = function () {};
};
```

В этом примере мы добавили модули dom, event и ajax, которые обычно используются в библиотеках или в сложных веб-приложениях.

Функции, реализующие модули, принимают текущий экземпляр box в качестве параметра и могут добавлять дополнительные свойства и методы к этому экземпляру.

Реализация конструктора

Теперь перейдем к реализации конструктора Sandbox() (естественно, вы можете выбрать для конструктора другое имя, более подходящее для вашей библиотеки или приложения):

```
function Sandbox() {
        // преобразовать аргументы в массив
    var args = Array.prototype.slice.call(arguments),
        // последний аргумент - функция обратного вызова
        callback = args.pop(),
        // имена модулей могут передаваться в форме массива
        // или в виде отдельных параметров
        modules = (args[0] && typeof args[0] === "string") ? args : args[0],
        i;

    // проверить, была ли функция вызвана
    // как конструктор
    if (!(this instanceof Sandbox)) {
        return new Sandbox(modules, callback);
    }
```

```
        // добавить свойства к объекту `this`, если это необходимо:
        this.a = 1;
        this.b = 2;

        // добавить модули в базовый объект `this`
        // отсутствие аргументов с именами модулей или аргумент со значением "*"
        // предполагает необходимость включения "всех модулей"
        if (!modules || modules === '*') {
            modules = [];
            for (i in Sandbox.modules) {
                if (Sandbox.modules.hasOwnProperty(i)) {
                    modules.push(i);
                }
            }
        }

        // инициализировать необходимые модули
        for (i = 0; i < modules.length; i += 1) {
            Sandbox.modules[modules[i]](this);
        }

        // вызвать функцию обратного вызова
        callback(this);
    }

    // добавить свойства к прототипу, если это необходимо
    Sandbox.prototype = {
        name: "My Application",
        version: "1.0",
        getName: function () {
            return this.name;
        }
    };
```

Ключевыми пунктами реализации конструктора являются:

- Проверка того, указывает ли ссылка `this` на экземпляр `Sandbox`, и если нет (когда функция `Sandbox()` вызывается без оператора `new`), то вызвать функцию как конструктор.
- Внутри конструктора допускается добавлять новые свойства к объекту `this`. Точно так же допускается добавлять свойства к прототипу конструктора.
- Информация о необходимых модулях может передаваться в виде массива имен, в виде отдельных аргументов или в виде аргумента со значением `'*'` (это значение используется по умолчанию в случае отсутствия аргумента), означающего, что необходимо включить все имеющиеся модули. Обратите внимание, что в данном примере реализации мы не беспокоимся о загрузке необходимых модулей, находящихся в отдельных файлах, но такую возможность необходимо

предусмотреть. Она поддерживается, например, библиотекой YUI3. Вы можете загрузить только основной модуль (известный также как «запускающий»), а все остальные модули будут загружаться из файлов, имена которых следуют соглашению, что имена файлов соответствуют именам модулей.

- Определившись с перечнем необходимых модулей, мы инициализируем их, то есть вызываем функции, реализующие модули.
- Последний аргумент конструктора – функция обратного вызова. Эта функция вызывается сразу после создания нового экземпляра объекта. Она играет роль изолированного пространства имен и получает объект `box`, заполненный всей необходимой функциональностью.

Статические члены

Статическими называются свойства и методы, которые не изменяются от экземпляра к экземпляру. В языках, опирающихся на использование классов, статические члены создаются с помощью специального синтаксиса и используются, как если бы они были членами самого класса. Например, статический метод `max()` некоторого класса `MathUtils` мог бы вызываться как `MathUtils.max(3, 5)`. Это пример общедоступного статического члена, который может использоваться без участия экземпляров класса. Однако точно так же могут существовать частные статические члены, недоступные для программного кода, использующего класс, но доступные для совместного использования всем экземплярам класса. Давайте посмотрим, как можно реализовать частные и общедоступные статические члены в JavaScript.

Общедоступные статические члены

В JavaScript отсутствует специальный синтаксис объявления статических членов. Тем не менее можно использовать тот же синтаксис, что и в языках на основе классов, добавляя свойства к функции-конструктору. Это возможно благодаря тому, что конструкторы, как и все остальные функции, являются объектами и могут иметь свойства. Шаблон мемоизации, обсуждавшийся в предыдущей главе, использует ту же идею – добавление свойств к функциям.

В следующем примере определяется конструктор `Gadget` со статическим методом `isShiny()` и с обычным методом экземпляров `setPrice()`. Метод `isShiny()` является статическим, потому что для его вызова не требуется создавать экземпляр объекта мини-приложения (gadget), как не обязательно иметь какое-то мини-приложение, чтобы сказать, что все мини-приложения имеют привлекательный вид (shiny). Метод `setPrice()`, напротив, требует наличия конкретного экземпляра объекта, так как различные украшательства могут иметь разную цену:

```
// конструктор
var Gadget = function () {};

// статический метод
Gadget.isShiny = function () {
    return "you bet";
};

// обычный метод, добавляемый в прототип
Gadget.prototype.setPrice = function (price) {
    this.price = price;
};
```

Теперь попробуем вызвать эти методы. Статический метод `isShiny()` вызывается непосредственно относительно конструктора, тогда как для вызова обычного метода необходим конкретный экземпляр:

```
// вызов статического метода
Gadget.isShiny(); // "you bet"

// создать экземпляр и вызвать обычный метод
var iphone = new Gadget();
iphone.setPrice(500);
```

Попытка вызвать метод экземпляра как статический метод окажется безуспешной, как безуспешной окажется и попытка вызвать статический метод с использованием экземпляра `iphone`:

```
typeof Gadget.setPrice; // "undefined"
typeof iphone.isShiny;  // "undefined"
```

Однако иногда бывает удобно иметь возможность вызывать статические методы и относительно экземпляров. Это легко можно реализовать, добавив в прототип новый метод, который будет играть роль ссылки, указывающей на фактический статический метод:

```
Gadget.prototype.isShiny = Gadget.isShiny;
iphone.isShiny(); // "you bet"
```

В подобных случаях нужно быть особенно осторожными при работе со ссылкой `this` внутри статических методов. При вызове метода в виде `Gadget.isShiny()` ссылка `this` внутри `isShiny()` будет указывать на функцию-конструктор `Gadget`. При вызове метода в виде `iphone.isShiny()` ссылка `this` внутри будет указывать на объект `iphone`.

В последнем примере демонстрируется, как можно реализовать метод, который может вызываться статически и не статически и который будет вести себя по-разному в зависимости от способа вызова. В данном случае способ вызова определяется с помощью оператора `instanceof`:

```
// конструктор
var Gadget = function (price) {
```

```
    this.price = price;
};

// статический метод
Gadget.isShiny = function () {

    // следующая инструкция выполняется всегда
    var msg = "you bet";

    if (this instanceof Gadget) {
        // эта ветка вызывается, когда метод вызывается не статически
        msg += ", it costs $" + this.price + '!';
    }

    return msg;
};

// обычный метод, добавляемый в прототип
Gadget.prototype.isShiny = function () {
    return Gadget.isShiny.call(this);
};
```

Проверим вызов статического метода:

```
Gadget.isShiny(); // "you bet"
```

Проверим вызов нестатического метода:

```
var a = new Gadget('499.99');
a.isShiny();      // "you bet, it costs $499.99!"
```

Частные статические члены

Мы обсудили общедоступные статические методы, а теперь давайте посмотрим, как можно реализовать частные статические члены. Под частными статическими членами подразумеваются члены, которые:

- Совместно используются всеми объектами, созданными с использованием одной и той же функции-конструктора
- Недоступны за пределами конструктора

Рассмотрим пример реализации частного статического свойства `counter` в конструкторе `Gadget`. В этой главе мы уже обсуждали частные свойства, и в этой части мы можем использовать тот же самый шаблон – создать функцию, образующую замыкание и обертывающую частные члены, оформить эту функцию как немедленно вызываемую и возвращающую новую функцию. Возвращаемую функцию необходимо присвоить переменной `Gadget`, в результате чего получить новый конструктор:

```
var Gadget = (function () {

    // статическая переменная/свойство
    var counter = 0;
```

```
    // вернуть новую реализацию конструктора
    return function () {
        console.log(counter += 1);
    };

}()); // выполняется немедленно
```

Новый конструктор `Gadget` просто увеличивает значение частного свойства `counter` и выводит его в консоль. Создав несколько экземпляров, можно заметить, что счетчик `counter` совместно используется всеми экземплярами:

```
var g1 = new Gadget(); // выведет 1
var g2 = new Gadget(); // выведет 2
var g3 = new Gadget(); // выведет 3
```

Так как при создании каждого экземпляра происходит увеличение значения свойства `counter`, оно может использоваться для уникальной идентификации объектов, создаваемых с помощью конструктора `Gadget`. Иногда бывает полезно иметь возможность определять значение уникального идентификатора, так почему бы не реализовать доступ к счетчику в виде привилегированного метода? Ниже приводится пример, построенный на основе предыдущего, куда добавлен привилегированный метод `getLastId()`, обеспечивающий доступ к частному свойству:

```
// конструктор
var Gadget = (function () {

    // статическая переменная/свойство
    var counter = 0,
        NewGadget;

    // реализация нового конструктора
    NewGadget = function () {
        counter += 1;
    };

    // привилегированный метод
    NewGadget.prototype.getLastId = function () {
        return counter;
    };

    // переопределить конструктор
    return NewGadget;

}()); // вызывается немедленно
```

Проверим новую реализацию:

```
var iphone = new Gadget();
iphone.getLastId();          // 1
var ipod = new Gadget();
```

```
ipod.getLastId();            // 2
var ipad = new Gadget();
ipad.getLastId();            // 3
```

Статические свойства (и частные, и общедоступные) могут быть весьма удобны. Они могут быть методами или содержать данные, которые не имеют отношения к конкретному экземпляру и которые не требуется создавать для каждого экземпляра в отдельности. В главе 7, при обсуждении шаблона единственного объекта (singleton), вы увидите пример реализации, в которой статические свойства используются для реализации конструкторов, обеспечивающих возможность создания только одного объекта.

Объекты-константы

В языке JavaScript отсутствует возможность определять константы, тогда как многие современные окружения предлагают для создания констант инструкцию `const`.

В качестве обходного маневра обычно предлагается использовать соглашение об именовании и вводить имена переменных, значения которых не должны изменяться, заглавными символами. Это соглашение используется для именования встроенных объектов JavaScript:

```
Math.PI;           // 3.141592653589793
Math.SQRT2;        // 1.4142135623730951
Number.MAX_VALUE; // 1.7976931348623157e+308
```

Выбирая имена для своих констант, вы тоже можете следовать этому соглашению и добавлять их как статические свойства функций-конструкторов:

```
// конструктор
var Widget = function () {
    // реализация...
};

// константы
Widget.MAX_HEIGHT = 320;
Widget.MAX_WIDTH = 480;
```

Это же соглашение об именовании может применяться при создании объектов из литералов; константы могут быть обычными свойствами, имена которых содержат только заглавные символы.

Если вам необходимо по-настоящему неизменяемое значение, можно создать частное свойство и реализовать метод, возвращающий его значение. Во многих случаях такой подход к решению проблемы будет, вероятно, излишним, если можно просто следовать соглашениям об именовании, но тем не менее такая возможность существует.

В примере ниже приводится реализация универсального объекта `constant` со следующими методами:

`set(`*name*`, `*value*`)`

Для определения новой константы.

`isDefined(`*name*`)`

Для проверки существования константы с именем *name*.

`get(`*name*`)`

Для получения значения константы.

В этой реализации в качестве констант допускается использовать только элементарные значения. Кроме того, объект проверяет имена констант при объявлении, чтобы они не совпадали с именами встроенных свойств, таких как `toString` или `hasOwnProperty`, используя для этого метод `hasOwnProperty()`, и дополнительно добавляет случайные префиксы к именам всех констант:

```
var constant = (function () {
    var constants = {},
        ownProp = Object.prototype.hasOwnProperty,
        allowed = {
            string: 1,
            number: 1,
            boolean: 1
        },
        prefix = (Math.random() + "_").slice(2);
    return {
        set: function (name, value) {
            if (this.isDefined(name)) {
                return false;
            }
            if (!ownProp.call(allowed, typeof value)) {
                return false;
            }
            constants[prefix + name] = value;
            return true;
        },

        isDefined: function (name) {
            return ownProp.call(constants, prefix + name);
        },

        get: function (name) {
            if (this.isDefined(name)) {
                return constants[prefix + name];
            }
            return null;
        }
```

```
    };
}());
```

Опробуем эту реализацию:

```
// проверить, определена ли константа
constant.isDefined("maxwidth"); // false

// определить
constant.set("maxwidth", 480);  // true

// проверить еще раз
constant.isDefined("maxwidth"); // true

// попытаться переопределить
constant.set("maxwidth", 320);  // false

// значение не изменилось?
constant.get("maxwidth");       // 480
```

Шаблон цепочек

Шаблон цепочек предоставляет возможность вызова методов объектов друг за другом без необходимости присваивать предыдущее возвращаемое значение промежуточной переменной и размещать инструкции вызова в нескольких строках:

```
myobj.method1("hello").method2().method3("world").method4();
```

При создании методов, не имеющих какого-либо осмысленного возвращаемого значения, можно возвращать из них ссылку `this` – экземпляр объекта, метод которого был вызван. Это позволит пользователям вашего объекта вызывать следующий метод, объединив его в цепочку с предыдущим:

```
var obj = {
    value: 1,
    increment: function () {
        this.value += 1;
        return this;
    },
    add: function (v) {
        this.value += v;
        return this;
    },
    shout: function () {
        alert(this.value);
    }
};
```

```
// цепочка из вызовов методов
obj.increment().add(3).shout(); // 5

// то же самое, но методы вызываются по одному
obj.increment();
obj.add(3);
obj.shout();                    // 5
```

Достоинства и недостатки шаблона цепочек

Одно из достоинств шаблона цепочек состоит в том, что его применение помогает сэкономить на вводе с клавиатуры и писать более компактный программный код, который читается почти как обычное предложение.

Другое достоинство в том, что этот шаблон заставляет задумываться о разбиении программного кода на маленькие и более узкоспециализированные функции, в противоположность большим функциям, в которые пытаются поместить слишком многое. В конечном итоге это повышает удобство в сопровождении.

Недостаток заключается в том, что такой программный код сложнее в отладке. Вы можете знать, что ошибка происходит в определенной строке, но в этой строке может происходить много всего. Если один из методов, входящих в цепочку, терпит неудачу, не порождая сообщение об ошибке, у вас нет никакой подсказки о том, где произошла ошибка. Роберт Мартин (Robert Martin), автор книги «Clean Code»[1], считает это недостаток столь значимым, что вообще назвал этот шаблон «крушением поезда».

Но, как бы то ни было, полезно знать этот шаблон, и когда вы пишете метод, который не имеет возвращаемого значения, всегда можно возвращать из него ссылку `this`. Этот шаблон широко используется, например, в библиотеке jQuery. А если вы заглянете в DOM API, то увидите, что он также позволяет составлять цепочки, такие как:

```
document.getElementsByTagName('head')[0].appendChild(newnode);
```

Метод method()

Язык JavaScript может вызывать замешательство у программистов, привыкших мыслить в терминах классов. Именно поэтому некоторые разработчики стараются сделать JavaScript более похожим на языки, где используются классы. Одна из таких попыток, принадлежащая Дугласу Крокфорду (Douglas Crockford), предлагает создание метода `method()`. Оглядываясь назад, он признает, что идея сделать JavaScript

[1] Мартин Роберт «Чистый код: создание, анализ и рефакторинг. Библиотека программиста». – Пер. с англ. – Питер, 2010.

более похожим на языки с классами была ошибочной, но это достаточно любопытный шаблон, а кроме того, вы можете столкнуться с ним в некоторых приложениях.

Применение функций-конструкторов напоминает использование классов в языке Java. Эти функции также позволяют добавлять свойства экземпляров внутри конструктора. Однако добавлять методы таким способом достаточно неэффективно, так как в конечном итоге они создаются отдельно для каждого экземпляра, что приводит к увеличению потребления памяти. Именно поэтому методы, общие для всех экземпляров, должны добавляться к свойству `prototype` конструктора. Для многих разработчиков свойство `prototype` может казаться чем-то инородным, поэтому вы можете спрятать его за методом с более привычным именем.

Дополнительная функциональность, добавляемая в язык ради удобства, часто называется *синтаксическим сахаром*. В данном случае метод `method()` также можно было бы назвать «методом-сахаром».

Способ определения «класса» с использованием метода `method()` выглядит, как показано ниже:

```
var Person = function (name) {
    this.name = name;
}.
    method('getName', function () {
        return this.name;
    }).
    method('setName', function (name) {
        this.name = name;
        return this;
    });
```

Обратите внимание, что в цепочку с конструктором объединен вызов метода `method()`, с которым в свою очередь объединен в цепочку еще один вызов метода `method()` и так далее. Такое следование шаблону цепочек, описанному выше, помогает определять целые «классы» в одной инструкции.

Метод `method()` принимает два параметра:

- Имя нового метода
- Реализацию метода

Затем новый метод добавляется к «классу» `Person`. Реализация – это просто еще одна функция, а внутри этой функции, как и следовало ожидать, ссылка `this` будет указывать на объект, созданный конструктором `Person`.

Ниже приводится пример использования `Person()` для создания нового объекта:

```
var a = new Person('Adam');
a.getName();                // 'Adam'
a.setName('Eve').getName(); // 'Eve'
```

Также обратите внимание, что использование шаблона цепочек стало возможным благодаря тому, что метод `setName()` возвращает ссылку `this`.

И наконец, ниже приводится реализация метода `method()`:

```
if (typeof Function.prototype.method !== "function") {
    Function.prototype.method = function (name, implementation) {
        this.prototype[name] = implementation;
        return this;
    };
}
```

Метод `method()` сначала проверяет, не был ли реализован метод с указанным именем ранее. Если нет, работа продолжается, и в прототип конструктора добавляется функция, переданная в аргументе `implementation`. В данном случае ссылка `this` ссылается на функцию-конструктор, прототип которой расширяется новым методом.

В заключение

В этой главе вы познакомились с различными шаблонами создания объектов, которые выходят за рамки стандартных способов использования литералов объектов и функций-конструкторов.

Вы познакомились с шаблоном *организации пространств имен*, позволяющим предотвратить засорение глобального пространства имен и помогающим организовать и структурировать программный код. Вы также познакомились с простым, но удивительно полезным шаблоном *объявления зависимостей*. Затем мы подробно обсудили шаблоны сокрытия данных, включая *частные* члены, *привилегированные* методы, некоторые особые случаи сокрытия данных, возможность создания частных членов с использованием литералов объектов и *открытие* доступа к частным методам. Все эти шаблоны являются строительными блоками популярного и мощного *шаблона «модуль»*.

Затем вы познакомились с *шаблоном организации изолированного пространства имен*, являющимся альтернативой шаблону пространств имен, который позволяет создавать независимые окружения для вашего программного кода и модулей.

В завершение дискуссии мы подробно рассмотрели *объекты-константы*, *статические методы* (общедоступные и частные), шаблон составления *цепочек* из методов и любопытный метод `method()`.

6

Шаблоны повторного использования программного кода

Повторное использование программного кода – важная и интересная тема, хотя бы потому, что вполне естественно стремиться писать как можно меньше и как можно больше использовать имеющийся программный код, уже написанный вами или кем-то другим. Особенно если это проверенный, удобный в сопровождении, легко расширяемый и хорошо документированный программный код.

Первое, с чем ассоциируется повторное использование программного кода, – это наследование, и значительная часть главы будет посвящена этой теме. Вы увидите несколько способов реализации «классического» и неклассического наследования. Но не забывайте о главной цели – нам требуется обеспечить повторное использование программного кода. Наследование – это один из путей (*средств*) достижения этой цели. И он не единственный. Вы увидите, как можно компоновать объекты из других объектов, как использовать объекты-смеси и как можно заимствовать и повторно использовать только необходимые функциональные возможности, не наследуя что-либо еще.

Приступая к решению задачи повторного использования программного кода, помните совет по созданию объектов, данный «бандой четырех» в своей книге[1]: «Метод составления объектов из других объектов должен пользоваться преимуществом перед наследованием».

---

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования». – Пер. с англ. – Питер, 2001.

Классические и современные шаблоны наследования

Вам наверняка часто приходилось слышать термин «классическое наследование» в дискуссиях, где обсуждалась тема наследования в JavaScript, поэтому давайте сначала проясним, что означает слово *классическое*. Этот термин не означает нечто старинное, устоявшееся или общепринятый способ выполнения каких-либо действий. Этот термин происходит от слова «класс».

Во многих языках программирования существует понятие классов, играющих роль кальки, используемой при создании объектов. В таких языках каждый объект является *экземпляром* определенного класса, и (как в языке Java, например) нет никакой возможности создать объект, если нет соответствующего класса. В JavaScript благодаря отсутствию классов понятие экземпляра класса не имеет никакого смысла. Объекты в JavaScript – это просто списки пар ключ-значение, которые можно создавать и изменять на лету.

Но в JavaScript есть функции-конструкторы, а синтаксис с использованием оператора `new` очень напоминает синтаксис использования классов.

В языке Java вы могли бы написать такую инструкцию:

```
Person adam = new Person();
```

И в JavaScript можно написать нечто похожее:

```
var adam = new Person();
```

Кроме того обстоятельства, что Java является языком со строгим контролем типов и, чтобы объявить переменную `adam`, необходимо указать ее тип `Person`, в остальном инструкции выглядят практически одинаковыми. Вызов конструктора в JavaScript выглядит так, как если бы `Person` была именем класса, но не забывайте, что `Person` – это всего лишь функция. Сходство синтаксиса заставляет многих разработчиков мыслить на языке JavaScript в терминах классов и разрабатывать идеи и шаблоны программирования, которые предполагают наличие классов. Такие реализации мы и будем называть «классическими». Кроме того, «современными» мы будем называть все остальные шаблоны, которые не вынуждают нас думать о классах.

Когда дело доходит до принятия шаблона наследования для реализации вашего проекта, у вас на выбор остается совсем немного вариантов. Вы можете отказаться от применения современных шаблонов, только если другие разработчики вашего совместного проекта чувствуют себя неуверенно, когда проект не выражен в терминах использования классов.

В этой главе сначала рассматриваются классические шаблоны, а затем современные.

Ожидаемый результат при использовании классического наследования

Цель реализации классического наследования в том, чтобы объекты, создаваемые одной функцией-конструктором `Child()`, приобретали свойства, присущие другому конструктору `Parent()`.

Хотя мы и обсуждаем классические шаблоны, давайте тем не менее не будем употреблять слово «класс». Термин «функция-конструктор», или просто «конструктор», длиннее, но он более точный и однозначный. Старайтесь вообще не употреблять слово «класс» в своем коллективе, потому что в контексте языка JavaScript для разных людей это слово может иметь разные значения.

Ниже приводится пример реализации конструкторов `Parent()` и `Child()`:

```
// родительский конструктор
function Parent(name) {
    this.name = name || 'Adam';
}

// добавление дополнительной функциональности в прототип
Parent.prototype.say = function () {
    return this.name;
};

// пустой дочерний конструктор
function Child(name) {}

// здесь происходит магия наследования
inherit(Child, Parent);
```

Здесь у нас имеются родительский и дочерний конструкторы, метод `say()`, добавленный в прототип родительского конструктора, и вызов функции `inherit()`, которая устанавливает зависимость наследования. Функция `inherit()` не является частью языка, поэтому вам придется написать ее самостоятельно. Давайте рассмотрим несколько примеров ее реализации универсальным способом.

Классический шаблон №1: шаблон по умолчанию

Наиболее часто используемый базовый способ заключается в том, чтобы создать объект с помощью конструктора `Parent()` и присвоить этот объект свойству `prototype` конструктора `Child()`. Ниже приводится первый вариант реализации функции `inherit()`:

```
function inherit(C, P) {
    C.prototype = new P();
}
```

Важно помнить, что свойство `prototype` должно ссылаться на объект, а не на функцию, поэтому здесь в него записывается ссылка на экземпляр (объект), созданный с помощью родительского конструктора, а не на сам конструктор. Проще говоря, обратите внимание на оператор `new` – он совершенно необходим для правильной работы этого шаблона.

Когда позднее в своем приложении вы будете создавать объекты с помощью выражения `new Child()`, они будут наследовать функциональность экземпляра `Parent()` через прототип, как показано в следующем примере:

```
var kid = new Child();
kid.say(); // "Adam"
```

Обход цепочки прототипов

При использовании этого шаблона экземпляры наследуют не только собственные свойства (свойства экземпляра, добавляемые к ссылке `this` в конструкторе, такие как `name`), но также свойства и методы прототипа (такие как `say()`).

Давайте посмотрим, как действует цепочка прототипов в этом шаблоне наследования. В нашем рассуждении будем представлять себе объекты как блоки памяти, которые могут содержать данные и ссылки на другие блоки. Создавая новый объект с помощью выражения `new Parent()`, вы создаете один такой блок (на рис. 6.1 он изображен под номером 2). Он хранит значение свойства `name`. Если попытаться вызвать метод `say()` (например, так: `(new Parent).say()`), интерпретатор обнаружит, что в блоке 2 этот метод отсутствует. Но благодаря скрытой ссылке `__proto__`, указывающей на свойство `prototype` функции конструктора `Parent()`, вы получаете доступ к блоку 1 (`Parent.prototype`), который имеет метод `say()`. Все это происходит автоматически, и вам не нужно беспокоиться

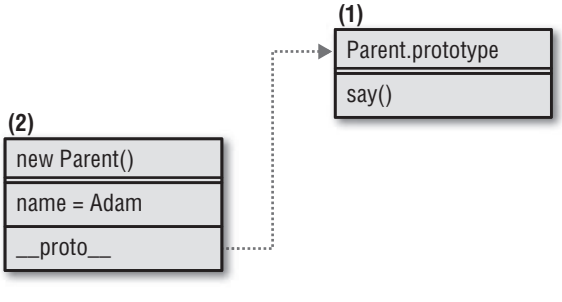

Рис. 6.1. *Цепочка прототипов к конструктору Parent()*

об этом, но совершенно необходимо знать, как действует этот механизм и где находятся данные, к которым вы обращаетесь и которые, возможно, изменяете. Обратите внимание, что ссылка `__proto__`, используемая здесь для описания работы цепочки прототипов, не является частью языка, но предоставляется некоторыми окружениями (например, Firefox).

Теперь посмотрим, что происходит, когда новый объект создается инструкцией `var kid = new Child()` после вызова функции `inherit()`. Диаграмма отношений объектов изображена на рис. 6.2.

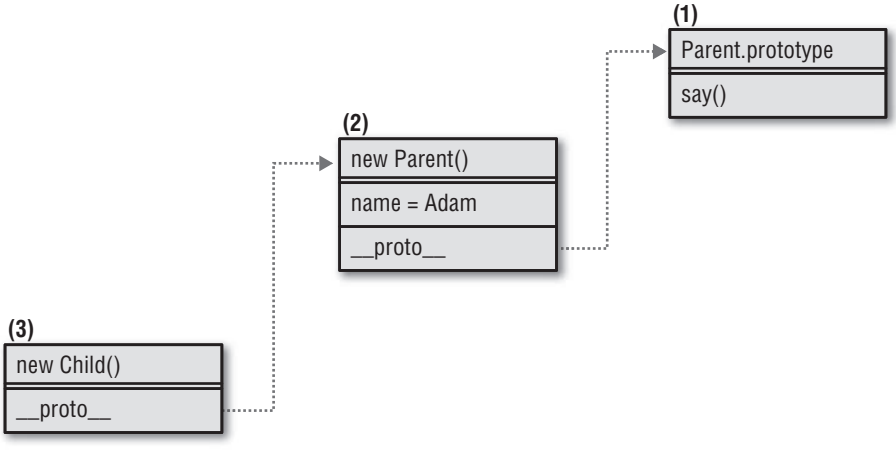

Рис. 6.2. *Цепочка прототипов после установления наследования*

Конструктор `Child()` в нашем примере пустой и не добавляет никаких свойств к `Child.prototype`. Поэтому выражение `new Child()` будет создавать практически пустые объекты, не имеющие никаких свойств, кроме скрытой ссылки `__proto__`. В данном случае ссылка `__proto__` будет указывать на объект, созданный выражением `new Parent()` внутри функции `inherit()`.

Что же произойдет теперь, если попытаться вызвать метод `kid.say()`? Объект под номером 3 не имеет такого метода, поэтому, следуя по цепочке прототипов, процедура поиска перейдет к объекту под номером 2. Этот объект также не имеет искомого метода, поэтому процедура поиска перейдет к объекту 1, в котором такой метод имеется. Внутри метода `say()` используется ссылка на свойство `this.name`, которое требуется отыскать. Поиск запускается снова. В данном случае ссылка `this` указывает на объект под номером 3, в котором отсутствует свойство `name`. Затем будет проверен объект под номером 2, в котором присутствует свойство `name` со значением «Adam».

Наконец, сделаем еще один шаг. Допустим, что имеется такой программный код:

```
var kid = new Child();
kid.name = "Patrick";
kid.say(); // "Patrick"
```

Как будет выглядеть цепочка в этом случае, показано на рис. 6.3.

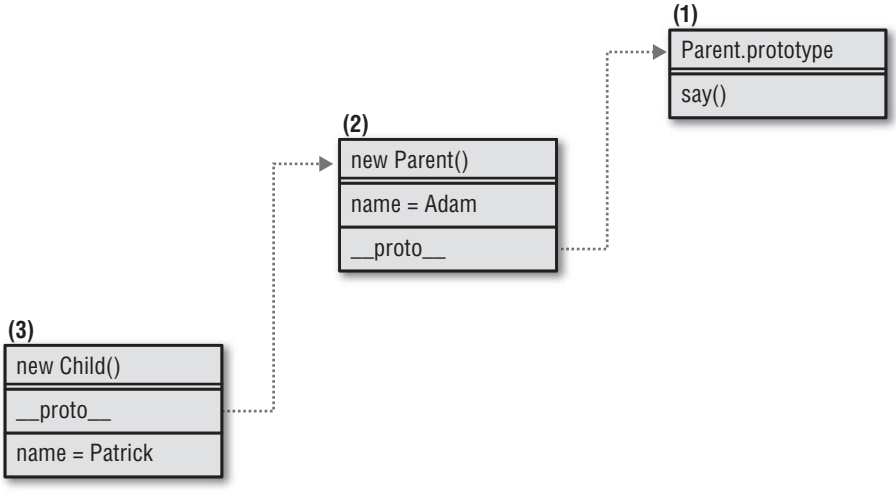

Рис. 6.3. *Цепочка прототипов после установления наследования и добавления свойства в дочерний объект*

Операция присваивания нового значения свойству `kid.name` не изменяет значение свойства `name` в объекте 2, а создает новое свойство в объекте `kid` под номером 3. Теперь при вызове метода `kid.say()` поиск метода `say`, как и прежде, будет выполнен в объекте 3, затем в объекте 2 и, наконец, он будет обнаружен в объекте 1. Но поиск свойства `this.name` (то есть свойства `kid.name`) на этот раз завершится быстрее, потому что оно сразу же будет найдено в объекте 3.

Если удалить новое свойство с помощью инструкции `delete kid.name`, то при последующих обращениях к свойству `name` будет обнаруживаться свойство `name` в объекте 2.

Недостатки шаблона №1

Одним из недостатков этого шаблона является то обстоятельство, что дочерние объекты наследуют не только свойства прототипа родительского объекта, но и собственные свойства, добавленные к нему. В большинстве случаев наследовать собственные свойства нежелательно, по-

тому что чаще всего они характерны для конкретного экземпляра и не могут повторно использоваться дочерними объектами.

Главное правило при использовании конструкторов – повторно используемые члены должны добавляться в прототип.

Еще один недостаток заключается в том, что использование универсальной функции `inherit()` не позволяет передать параметры, полученные при вызове дочерним конструктором, от дочернего конструктора родительскому. Взгляните на следующий пример:

```
var s = new Child('Seth');
s.say(); // "Adam"
```

Результат не совсем тот, какого вы могли бы ожидать. Можно было бы организовать передачу параметров родительскому конструктору внутри дочернего конструктора, но тогда пришлось бы устанавливать отношение наследования при создании каждого нового дочернего объекта, что очень неэффективно, так как при этом будут создаваться все новые и новые родительские объекты.

Классический шаблон №2: заимствование конструктора

Следующий шаблон решает проблему передачи аргументов дочернего конструктора родительскому конструктору, выполняя связывание дочернего объекта со ссылкой `this` и передавая любые аргументы:

```
function Child(a, c, b, d) {
    Parent.apply(this, arguments);
}
```

При таком подходе наследуются только свойства, добавляемые внутри родительского конструктора. Но свойства, добавленные в прототип, не наследуются.

При использовании шаблона заимствования конструктора дочерние объекты получают копии унаследованных членов, в отличие от классического шаблона №1, где они получают только ссылки. Отличия этого шаблона демонстрируются в следующем примере:

```
// родительский конструктор
function Article() {
    this.tags = ['js', 'css'];
}
var article = new Article();

// объект сообщения в блоге наследует свойства объекта article
// через классический шаблон №1
function BlogPost() {}
```

```
BlogPost.prototype = article;
var blog = new BlogPost();
// обратите внимание, что выше нет необходимости
// использовать выражение `new Article()`,
// потому что уже имеется доступный экземпляр

// статическая страница наследует свойства объекта article
// через шаблон заимствования конструктора
function StaticPage() {
    Article.call(this);
}
var page = new StaticPage();

alert(article.hasOwnProperty('tags')); // true
alert(blog.hasOwnProperty('tags'));    // false
alert(page.hasOwnProperty('tags'));    // true
```

В этом фрагменте родительский конструктор `Article()` наследуется двумя разными способами. Шаблон по умолчанию позволяет объекту `blog` получить доступ к свойству `tags` через прототип, поэтому он не имеет собственного свойства, и функция `hasOwnProperty()` возвращает `false`. Объект `page` получает собственное свойство `tags`, потому что при использовании шаблона заимствования конструктора новый объект получает копию родительского члена `tags` (а не ссылку на него).

Обратите внимание на различия, возникающие при попытке изменить значение унаследованного свойства `tags`:

```
blog.tags.push('html');
page.tags.push('php');
alert(article.tags.join(', ')); // "js, css, html"
```

В этом примере предпринимается попытка изменить значение свойства `tags` дочернего объекта `blog`, но при такой организации данных одновременно изменяется свойство родительского объекта, потому что оба свойства, `blog.tags` и `article.tags`, ссылаются на один и тот же массив. Попытка изменить значение свойства `page.tags` не приводит к изменению свойства в родительском объекте `article`, потому что свойство `page.tags` является отдельной копией, созданной в результате наследования.

Цепочка прототипов

Давайте посмотрим, как выглядит цепочка прототипов при использовании этого шаблона и знакомых уже конструкторов `Parent()` и `Child()`. Конструктор `Child()` необходимо немного изменить в соответствии с новым шаблоном:

```
// родительский конструктор
function Parent(name) {
    this.name = name || 'Adam';
}
```

```
// добавление дополнительной функциональности в прототип
Parent.prototype.say = function () {
    return this.name;
};

// дочерний конструктор
function Child(name) {
    Parent.apply(this, arguments);
}

var kid = new Child("Patrick");
kid.name;       // "Patrick"
typeof kid.say; // "undefined"
```

На рис. 6.4 можно заметить отсутствие связи между объектами `new Child` и `Parent`. Это обусловлено тем, что свойство `Child.prototype` вообще не используется здесь, и оно ссылается на пустой объект. При использовании этого шаблона объект `kid` получит собственное свойство `name`, но он не унаследует метод `say()`, и попытка вызвать его приведет к ошибке. Акт наследования в данном случае – это однократная операция, которая скопировала собственные свойства родительского объекта в дочерний объект и только; никаких ссылок `__proto__` при этом не сохранилось.

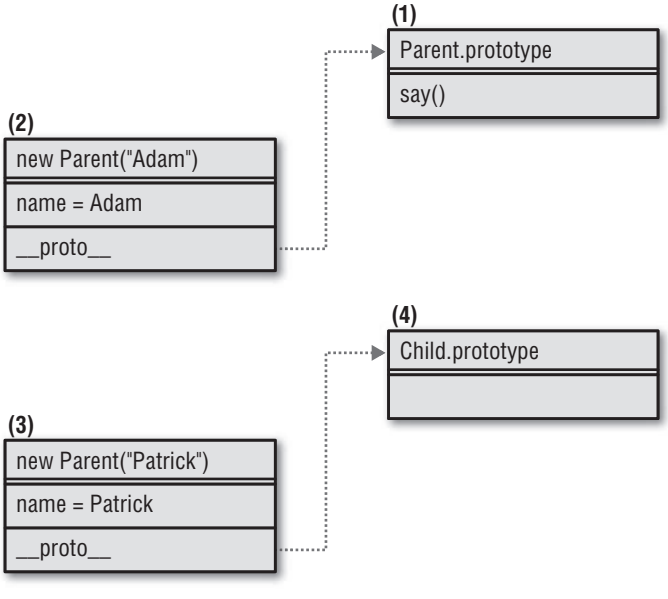

Рис. 6.4. *Цепочка прототипов разрывается при использовании шаблона заимствования конструктора*

Множественное наследование при заимствовании конструкторов

Шаблон заимствования конструктора позволяет реализовать множественное наследование просто за счет заимствования нескольких конструкторов:

```
function Cat() {
    this.legs = 4;
    this.say = function () {
        return "meaowww";
    }
}

function Bird() {
    this.wings = 2;
    this.fly = true;
}

function CatWings() {
    Cat.apply(this);
    Bird.apply(this);
}

var jane = new CatWings();
console.dir(jane);
```

Результат выполнения этого фрагмента изображен на рис. 6.5. При наличии в родительских объектах одноименных свойств дочернему объекту достанется последнее из них.

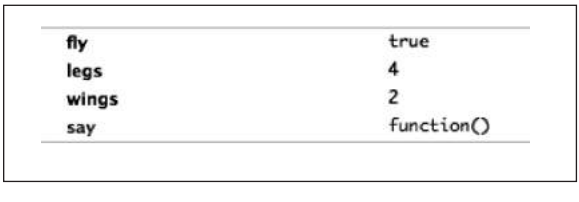

Рис. 6.5. *Объект CatWings в Firebug*

Достоинства и недостатки шаблона заимствования конструктора

Недостаток этого шаблона очевиден – при его использовании не наследуются свойства прототипа, тогда как прототип, о чем уже говорилось выше, является местом, куда следует добавлять совместно используемые методы и свойства, которые не должны создаваться отдельно для каждого экземпляра.

Преимущество же заключается в том, что дочерние объекты получают настоящие копии собственных свойств родительских объектов, благодаря чему исключается риск случайного изменения значения родительского свойства.

Так как же сделать, чтобы дочерние объекты наследовали также и свойства прототипа, как в предыдущем случае, и объект `kid` получил доступ к методу `say()`? На этот вопрос отвечает следующий шаблон.

Классический шаблон №3: заимствование и установка прототипа

При объединении двух предыдущих шаблонов необходимо сначала заимствовать конструктор, а затем сохранить в свойстве `prototype` дочернего объекта ссылку на новый экземпляр конструктора:

```
function Child(a, c, b, d) {
    Parent.apply(this, arguments);
}
Child.prototype = new Parent();
```

Преимущество такого решения в том, что дочерний объект получает копии собственных членов родительского объекта и ссылку на функциональные возможности родительского объекта (реализованные в виде членов прототипа). Дочерний объект также может передавать любые аргументы родительскому конструктору. Этот шаблон, пожалуй, наиболее близок к поведению механизма наследования в языке Java – дочерний объект наследует все, что имеет родительский объект, и в то же время он может безбоязненно изменять собственные свойства, не рискуя изменить значения свойств родительского объекта.

Недостаток заключается в необходимости дважды вызывать родительский конструктор, что снижает эффективность. В результате этого собственные свойства (такие как `name` в данном случае) наследуются дважды.

Давайте рассмотрим пример, выполняющий тестирование этого шаблона:

```
// родительский конструктор
function Parent(name) {
    this.name = name || 'Adam';
}

// добавление дополнительной функциональности в прототип
Parent.prototype.say = function () {
    return this.name;
};

// дочерний конструктор
function Child(name) {
```

```
    Parent.apply(this, arguments);
}
Child.prototype = new Parent();

var kid = new Child("Patrick");
kid.name;                       // "Patrick"
kid.say();                      // "Patrick"
delete kid.name;
kid.say();                      // "Adam"
```

В отличие от предыдущего шаблона, теперь дочерний объект наследует метод `say()`. Можно также заметить, что свойство `name` было унаследовано дважды, и после удаления собственной копии дочерний объект получает доступ к другому свойству, унаследованному по цепочке прототипов.

На рис. 6.6 изображены взаимоотношения между объектами в данном примере. Они напоминают взаимоотношения на рис. 6.3, но были получены иным способом.

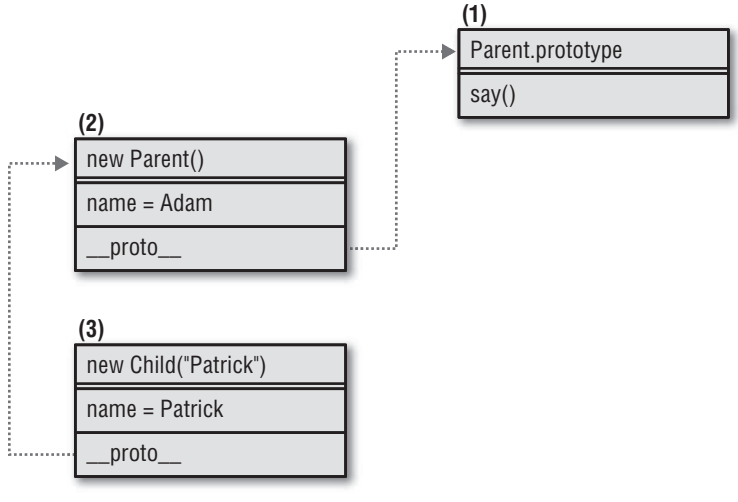

Рис. 6.6. *В дополнение к наследованию собственных членов сохраняется цепочка прототипов*

Классический шаблон №4: совместное использование прототипа

В отличие от предыдущего шаблона классического наследования, в котором требуется выполнить два вызова родительского конструктора, следующий шаблон вообще не предусматривает вызов родительского конструктора.

При таком способе наследования доступ к повторно используемым членам обеспечивается прототипом, а не ссылкой `this`. То есть все, что должно наследоваться дочерними объектами, должно находиться в родительском прототипе. В этом случае достаточно просто присвоить родительский прототип дочернему прототипу:

```
function inherit(C, P) {
    C.prototype = P.prototype;
}
```

Благодаря этому образуется короткая цепочка прототипов, обеспечивающая высокую скорость поиска, так как все объекты фактически будут совместно использовать один и тот же прототип. Но эта особенность является и недостатком, потому что в случае, если один из дочерних объектов, находящихся где-то в цепочке наследования, изменит прототип, это скажется на всех его родительских объектах, расположенных выше в цепочке наследования.

Как видно на рис. 6.7, и дочерний, и родительский объекты используют один и тот же прототип и обладают доступом к методу `say()`. Однако дочерний объект не имеет свойства `name`.

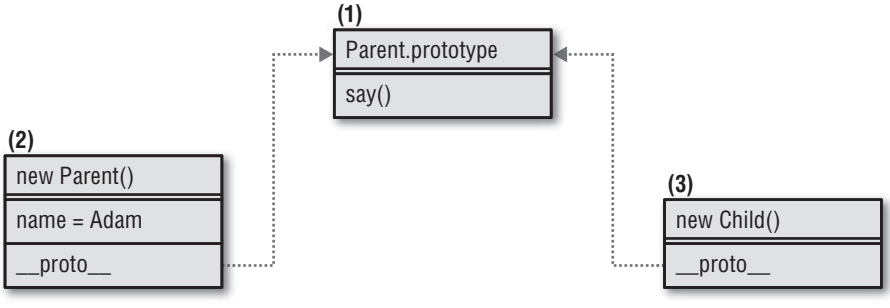

Рис. 6.7. *Взаимоотношения между объектами при совместном использовании одного и того же прототипа*

Классический шаблон №5: временный конструктор

Следующий шаблон решает проблему, возникающую при совместном использовании прототипа, разрывая прямую связь между родительским и дочерним прототипами, но сохраняя при этом преимущества, которые дает наличие цепочки прототипов.

Ниже приводится реализация этого шаблона, где мы вводим пустую функцию `F()`, играющую роль передаточного звена между предком и потомком. Свойство `prototype` функции `F()` ссылается на родительский прототип. А прототипом потомка является экземпляр пустой функции:

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
}
```

Поведение этого шаблона несколько отличается от поведения шаблона по умолчанию (классический шаблон №1), потому что здесь потомок наследует только свойства прототипа (как видно на рис. 6.8).

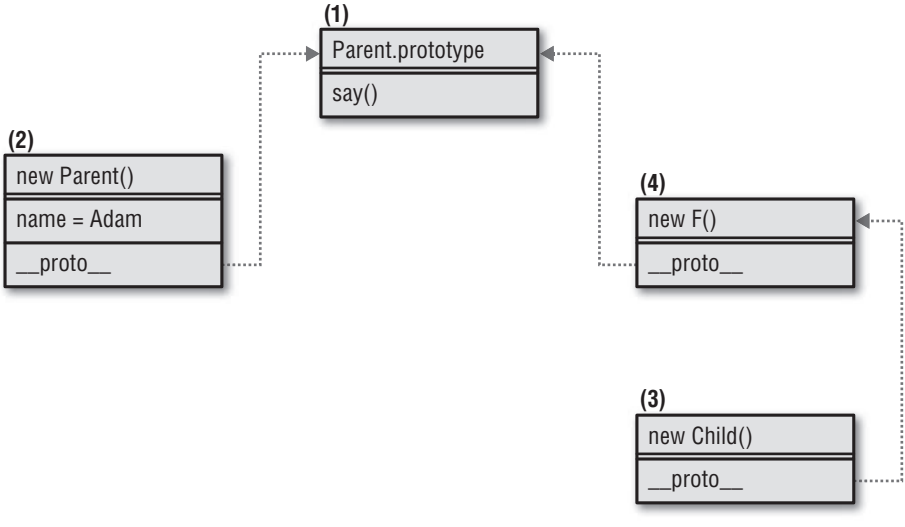

Рис. 6.8. *Классическое наследование при использовании временного (промежуточного) конструктора F()*

Обычно это наиболее предпочтительное поведение, потому что прототип является местом сосредоточения повторно используемой функциональности. В этом шаблоне все члены, добавляемые родительским конструктором, не наследуются потомками.

Создадим новый дочерний объект и исследуем особенности его поведения:

```
var kid = new Child();
```

Если вы попытаетесь обратиться к свойству `kid.name`, то получите значение `undefined`. В данном примере свойство `name` является собственным свойством родительского объекта, а так как при такой организации наследования конструктор `new Parent()` не вызывается, это свойство не создается. При попытке вызвать метод `kid.say()` он не будет найден в объекте с номером 3, и поиск будет продолжен в цепочке прототипов. В объекте с номером 4 этот метод также отсутствует, но он имеется в объекте с номером 1, и это тот блок в памяти, который будет совместно исполь-

зоваться всеми конструкторами, наследующими `Parent()`, и всеми объектами, созданными этими дочерними конструкторами.

Сохранение суперкласса

Предыдущий шаблон можно несколько расширить, добавив в него ссылку на оригинального предка. Это напоминает доступ к суперклассу в других языках и может оказаться полезным в некоторых ситуациях.

Назовем это свойство `uber`, потому что слово «super» является зарезервированным словом, а слово «superclass» может натолкнуть непосвященного разработчика на мысль, что в JavaScript есть классы. Ниже приводится улучшенная реализация этого классического шаблона:

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
    C.uber = P.prototype;
}
```

Установка указателя на конструктор

И последнее, что можно добавить в эту практически идеальную функцию классического наследования, – это установка указателя на функцию-конструктор, который может понадобиться в будущем.

Если не переустановить указатель на конструктор, все дочерние объекты будут полагать, что их конструктором является `Parent()`, который не особо полезен. То есть, используя предыдущую реализацию функции `inherit()`, можно наблюдать следующее поведение объектов:

```
// предок, потомок, наследование
function Parent() {}
function Child() {}
inherit(Child, Parent);

// проверка
var kid = new Child();
kid.constructor.name;       // "Parent"
kid.constructor === Parent; // true
```

Свойство `constructor` редко используется на практике, но оно может пригодиться для проведения интроспекции объектов во время выполнения. В это свойство можно записать ссылку на желаемый конструктор, не оказывая влияния на функциональные возможности, потому что это свойство используется исключительно для информирования.

Заключительная версия функции реализации шаблона классического наследования будет выглядеть так:

```
function inherit(C, P) {
    var F = function () {};
    F.prototype = P.prototype;
    C.prototype = new F();
    C.uber = P.prototype;
    C.prototype.constructor = C;
}
```

Функция, похожая на эту, присутствует в библиотеке YUI (и, возможно, в других библиотеках) и обеспечивает возможность классического наследования в языке без классов, если вы решите, что этот подход лучше всего подходит для вашего проекта.

Этот шаблон также называется *шаблоном с промежуточной функцией*, или *шаблоном с промежуточным конструктором*, потому что временный конструктор используется в нем как промежуточное звено для получения доступа к родительскому прототипу.

Одно из очевидных усовершенствований этого шаблона заключается в том, чтобы избавиться от необходимости создавать временный (промежуточный) конструктор всякий раз, когда потребуется организовать наследование. Вполне достаточно создать его однажды и просто изменять его свойство `prototype`. Для этой цели можно использовать немедленно вызываемую функцию и сохранять промежуточную функцию в замыкании:

```
var inherit = (function () {
    var F = function () {};
    return function (C, P) {
        F.prototype = P.prototype;
        C.prototype = new F();
        C.uber = P.prototype;
        C.prototype.constructor = C;
    }
}());
```

Функция klass()

Многие библиотеки JavaScript имитируют классы, вводя дополнительные синтаксические конструкции (синтаксический сахар). Все реализации отличаются, но имеют некоторые общие черты:

- Вводится соглашение об имени метода, который будет считаться конструктором класса, например `initialize`, `_init` или нечто похожее, и будет вызываться автоматически.
- Классы наследуют другие классы.

- Обеспечивается доступ к родительскому классу (суперклассу) из дочернего класса.

Давайте здесь изменим правила и позволим себе в этой (и только в этой) части главы использовать слово «класс», потому что темой этого раздела является имитация классов.

Не углубляясь в подробности, рассмотрим пример реализации имитации классов в языке JavaScript. Во-первых, посмотрим, как эта имитация будет выглядеть с точки зрения клиента:

```
var Man = klass(null, {
    __construct: function (what) {
        console.log("Man's constructor");
        this.name = what;
    },
    getName: function () {
        return this.name;
    }
});
```

Вспомогательная синтаксическая конструкция имеет вид функции с именем `klass()`. В некоторых реализациях она имеет форму конструктора `Klass()` или расширенного свойства `Object.prototype`, но в данном примере мы оставим ее в виде простой функции.

Функция принимает два параметра: родительский класс, который должен быть унаследован, и реализацию нового класса в виде литерала объекта. Поддавшись влиянию языка PHP, давайте установим соглашение, по которому конструктором класса будет метод с именем `__construct`. В предыдущем фрагменте создается новый класс с именем `Man`, который не наследует никакой другой класс (то есть наследует класс `Object`). Класс `Man` имеет собственное свойство `name`, создаваемое внутри конструктора `__construct`, и метод `getName()`. Класс – это функция-конструктор, поэтому следующий программный код вполне допустим (и выглядит как создание экземпляра класса):

```
var first = new Man('Adam'); // выведет "Man's constructor"
first.getName();             // "Adam"
```

Теперь расширим этот класс и создадим класс `SuperMan`:

```
var SuperMan = klass(Man, {
    __construct: function (what) {
        console.log("SuperMan's constructor");
    },
    getName: function () {
        var name = SuperMan.uber.getName.call(this);
        return "I am " + name;
    }
});
```

Здесь первым параметром функции `klass()` передается имя наследуемого класса `Man`. Обратите также внимание, что в функции `getName()` сначала вызывается функция `getName()`, унаследованная от родительского класса, с использованием статического свойства `uber` (суперкласс) класса `SuperMan`. Давайте протестируем этот класс:

```
var clark = new SuperMan('Clark Kent');
clark.getName(); // "I am Clark Kent"
```

Первая инструкция сначала выведет в консоль строку «Man's constructor», а затем строку «Superman's constructor». В некоторых языках программирования родительский конструктор вызывается автоматически при каждом вызове дочернего конструктора, так почему бы не имитировать и эту особенность?

Применение оператора `instanceof` возвращает вполне ожидаемый результат:

```
clark instanceof Man;      // true
clark instanceof SuperMan; // true
```

А теперь посмотрим, как могла бы быть реализована функция `klass()`:

```
var klass = function (Parent, props) {

    var Child, F, i;

    // 1.
    // новый конструктор
    Child = function () {
        if (Child.uber && Child.uber.hasOwnProperty("__construct")) {
            Child.uber.__construct.apply(this, arguments);
        }
        if (Child.prototype.hasOwnProperty("__construct")) {
            Child.prototype.__construct.apply(this, arguments);
        }
    };

    // 2.
    // наследование
    Parent = Parent || Object;
    F = function () {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.uber = Parent.prototype;
    Child.prototype.constructor = Child;

    // 3.
    // добавить реализацию методов
    for (i in props) {
        if (props.hasOwnProperty(i)) {
            Child.prototype[i] = props[i];
```

```
            }
        }

        // вернуть сформированный "класс"
        return Child;
    };
```

Реализация функции `klass()` может быть разбита на три части, представляющие интерес:

1. Создается функция-конструктор `Child()`. Эта функция возвращается в конце и будет играть роль класса. В ней вызывается метод `__construct`, если он существует. Но перед этим с помощью свойства `uber` вызывается родительский метод `__construct` (опять же, если он существует). В некоторых случаях свойство `uber` может быть не определено, например, когда наследуется объект `Object`, как это было при определении класса `Man`.
2. Вторая часть устанавливает взаимоотношения наследования. Используется последний шаблон классического наследования, обсуждавшийся в предыдущем разделе. Здесь появилось только одно новшество: в качестве родительского класса используется `Object`, если в аргументе `Parent` не был передан другой класс.
3. В последнем разделе выполняется обход реализаций всех методов (таких как `__construct` и `getName` в примере), составляющих фактическое определение класса, и производится их добавление в прототип функции `Child`.

В каких случаях лучше использовать этот шаблон? Скажем так, желательно было бы вообще его не использовать, потому что он вводит сбивающее с толку понятие класса, которое технически отсутствует в языке. Он добавляет новые правила и новый синтаксис, которые необходимо знать и помнить. Однако если вы или ваши коллеги легко разбираетесь в классах и в то же время испытываете неуверенность при работе с прототипами, вам имеет смысл изучить этот шаблон. Он позволяет полностью забыть о прототипах и, что самое приятное, использовать синтаксис и соглашения, напоминающие какой-то из ваших любимых языков.

Наследование через прототип

Наше обсуждение «современных» бесклассовых шаблонов мы начнем с шаблона *наследования через прототип*. Этот шаблон никак не связан с понятием классов – здесь одни объекты наследуют другие объекты. Представить такой тип наследования можно так: имеется некоторый объект, который можно было бы использовать повторно, и вам требуется создать второй объект, использующий функциональные возможности первого объекта.

Ниже показано, как можно было бы реализовать эту схему наследования:

```
// объект, который наследуется
var parent = {
    name: "Papa"
};

// новый объект
var child = object(parent);

// проверка
alert(child.name); // "Papa"
```

В этом фрагменте имеется объект с именем `parent`, созданный с помощью литерала объекта, и создается другой объект с именем `child`, который обладает теми же свойствами и методами, что и родительский объект `parent`. Объект `child` был создан с помощью функции `object()`. Эта функция отсутствует в языке JavaScript (не путайте ее с функцией-конструктором `Object()`), поэтому нам необходимо определить ее.

Так же как и в последнем классическом шаблоне, можно было бы определить пустую временную функцию-конструктор `F()`, присвоить родительский объект свойству `prototype` функции `F()` и в конце вернуть новый экземпляр временного конструктора:

```
function object(o) {
    function F() {}
    F.prototype = o;
    return new F();
}
```

Цепочка прототипов, образующаяся при использовании шаблона наследования через прототип, изображена на рис. 6.9. Здесь дочерний объект `child` всегда создается как пустой объект, не имеющий собственных свойств, но обладающий доступом ко всем функциональным возможностям родительского объекта `parent` благодаря ссылке `__proto__`.

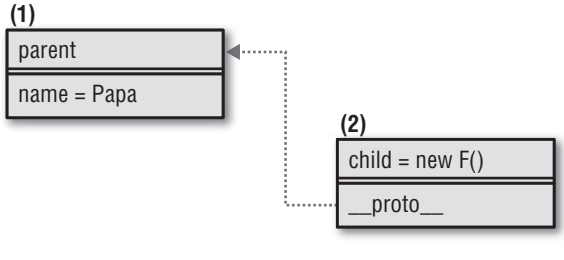

Рис. 6.9. *Шаблон наследования через прототип*

Обсуждение

В шаблоне наследования через прототип родительский объект необязательно должен создаваться с применением литерала (хотя этот способ является наиболее типичным). С тем же успехом создание родительского объекта может производиться с помощью функции-конструктора. Однако учтите, что в последнем случае унаследованы будут не только свойства прототипа конструктора, но и «собственные» свойства объекта:

```
// родительский конструктор
function Person() {
    // “собственное” свойство
    this.name = “Adam”;
}

// свойство, добавляемое в прототип
Person.prototype.getName = function () {
    return this.name;
};

// создать новый объект типа Person
var papa = new Person();

// наследник
var kid = object(papa);

// убедиться, что было унаследовано не только
// свойство прототипа, но и собственное свойство
kid.getName(); // “Adam”
```

Другая разновидность этого шаблона предполагает возможность наследования только объекта прототипа существующего конструктора. Не забывайте, что объекты наследуют объекты независимо от того, каким способом создаются родительские объекты. Ниже приводится немного измененная реализация предыдущего примера:

```
// родительский конструктор
function Person() {
    // “собственное” свойство
    this.name = “Adam”;
}
// свойство, добавляемое в прототип
Person.prototype.getName = function () {
    return this.name;
};

// наследник
var kid = object(Person.prototype);

typeof kid.getName; // “function”, потому что присутствует в прототипе
typeof kid.name;    // “undefined”, потому что наследуется только прототип
```

Дополнения в стандарте ECMAScript 5

В стандарте ECMAScript 5 шаблон наследования через прототип стал официальной частью языка. Этот шаблон реализован в виде метода `Object.create()`. Другими словами, вам не потребуется создавать собственную функцию, похожую на `object()`; она уже будет встроена в язык:

```
var child = Object.create(parent);
```

Метод `Object.create()` принимает дополнительный параметр – объект. Свойства этого объекта будут добавлены во вновь созданный дочерний объект как собственные свойства. Это позволяет создавать дочерние объекты и определять отношения наследования единственным вызовом метода. Например:

```
var child = Object.create(parent, {
    age: { value: 2 }        // описание в соответствии со стандартом ECMA5
});
child.hasOwnProperty("age"); // true
```

Реализацию шаблона наследования через прототип можно также найти в некоторых библиотеках JavaScript. Например, в библиотеке YUI3 имеется метод `Y.Object()`:

```
YUI().use('*', function (Y) {
    var child = Y.Object(parent);
});
```

Наследование копированием свойств

Теперь рассмотрим другой способ организации наследования – наследование копированием свойств. В этом шаблоне один объект получает доступ к функциональности другого объекта за счет простого копирования свойств. Ниже приводится пример реализации функции `extend()`, осуществляющей этот шаблон:

```
function extend(parent, child) {
    var i;
    child = child || {};
    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            child[i] = parent[i];
        }
    }
    return child;
}
```

В этом примере выполняется обход и копирование членов родительского объекта. В данной реализации параметр `child` является необязательным – если опустить этот параметр, будет создан новый объект, в про-

тивном случае функция дополнит существующий объект свойствами родительского объекта:

```
var dad = {name: "Adam"};
var kid = extend(dad);
kid.name; // "Adam"
```

Данная реализация выполняет так называемое «поверхностное копирование» свойств объекта. Чтобы выполнить полное копирование, необходимо проверить, является ли копируемое свойство объектом или массивом, и если это так, следует рекурсивно обойти все вложенные свойства и также скопировать их. В случае поверхностного копирования (так как в JavaScript объекты передаются по ссылке) при изменении в дочернем объекте свойства, которое в свою очередь является объектом, изменение будет внесено и в родительский объект. Поверхностное копирование является наиболее предпочтительным для методов (функции также являются объектами и передаются по ссылке), но может приводить к неприятным сюрпризам при наличии свойств, являющихся объектами или массивами. Взгляните на следующий пример:

```
var dad = {
    counts: [1, 2, 3],
    reads: {paper: true}
};
var kid = extend(dad);
kid.counts.push(4);
dad.counts.toString();   // "1,2,3,4"
dad.reads === kid.reads; // true
```

Теперь изменим функцию `extend()` так, чтобы она выполняла полное копирование. Для этого необходимо проверить, является ли свойство объектом, и если это так, рекурсивно скопировать его свойства. Кроме того, необходимо проверить, является ли свойство истинным объектом или массивом. Для этого воспользуемся решением, предложенным в главе 3. Итак, версия `extend()`, выполняющая полное копирование, могла бы выглядеть, как показано ниже:

```
function extendDeep(parent, child) {
    var i,
        toStr = Object.prototype.toString,
        astr = "[object Array]";

    child = child || {};

    for (i in parent) {
        if (parent.hasOwnProperty(i)) {
            if (typeof parent[i] === "object") {
                child[i] = (toStr.call(parent[i]) === astr) ? [] : {};
                extendDeep(parent[i], child[i]);
```

```
                } else {
                    child[i] = parent[i];
                }
            }
        }
        return child;
    }
```

Проверим новую реализацию, выполняющую полное копирование объектов, и убедимся, что изменения в дочернем объекте не оказывают влияния на родительский объект:

```
var dad = {
    counts: [1, 2, 3],
    reads: {paper: true}
};
var kid = extendDeep(dad);

kid.counts.push(4);
kid.counts.toString();   // "1,2,3,4"
dad.counts.toString();   // "1,2,3"

dad.reads === kid.reads; // false
kid.reads.paper = false;

kid.reads.web = true;
dad.reads.paper;         // true
```

Этот шаблон копирования свойств прост и получил широкое распространение. Например, в Firebug (расширение для броузера Firefox, написанное на языке JavaScript) имеется метод `extend()`, выполняющий поверхностное копирование. В библиотеке jQuery метод `extend()` выполняет полное копирование. Библиотека YUI3 предлагает метод `Y.clone()`, который не только выполняет полное копирование свойств, но еще копирует функции, связывая их с дочерним объектом. (Подробнее о связывании рассказывается ниже в этой главе.)

Особенно следует отметить, что прототипы в этом шаблоне вообще никак не используются; шаблон предусматривает работу только с объектами и их собственными свойствами.

Смешивание

Давайте рассмотрим шаблон смешивания, взяв за основу идею наследования копированием свойств и развив ее еще дальше. Вместо копирования одного объекта можно предусмотреть возможность копирования произвольного количества объектов, смешав их свойства в новом объекте.

Реализация выглядит просто; достаточно обойти все аргументы и скопировать каждое свойство каждого объекта, переданного функции:

```
function mix() {
    var arg, prop, child = {};
    for (arg = 0; arg < arguments.length; arg += 1) {
        for (prop in arguments[arg]) {
            if (arguments[arg].hasOwnProperty(prop)) {
                child[prop] = arguments[arg][prop];
            }
        }
    }
    return child;
}
```

Теперь, когда у вас имеется универсальная функция смешивания, вы сможете передать ей произвольное количество объектов и получить в результате новый объект, обладающий свойствами всех исходных объектов. Например:

```
var cake = mix(
    {eggs: 2, large: true},
    {butter: 1, salted: true},
    {flour: "3 cups"},
    {sugar: "sure!"}
);
```

На рис. 6.10 показан результат вызова функции `console.dir(cake)`, которая вывела в консоль Firebug свойства нового смешанного объекта `cake`.

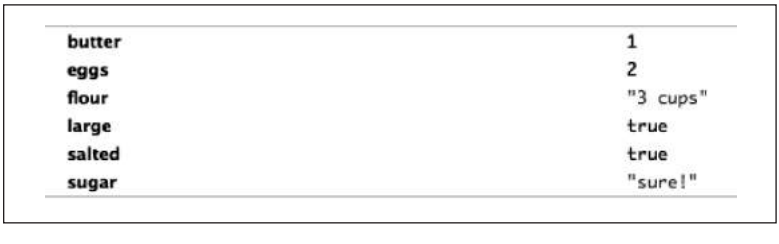

Рис. 6.10. *Результат исследования объекта cake в Firebug*

Если вы уже использовали прием смешивания объектов в других языках, где он является частью языка, вы могли бы ожидать, что изменения в одном или более родительских объектах отразятся на объекте-потомке, однако в данной реализации это не так. Здесь мы просто копируем собственные свойства родительских объектов и разрываем связь с ними.

Заимствование методов

Иногда интерес представляют только один-два метода в существующем объекте. У вас может появиться желание повторно использовать их, но при этом вам ни к чему устанавливать с этим объектом отношения родитель-потомок. То есть вам может требоваться использовать часть методов, не наследуя при этом все остальные методы, которые вам не нужны. Этого можно добиться с помощью шаблона заимствования методов, который опирается на использование методов `call()` и `apply()`. Вы уже видели этот шаблон в этой книге и даже в этой главе – например, в реализации функции `extendDeep()`.

Как вы уже знаете, функции в языке JavaScript являются объектами, и они обладают некоторыми интересными методами, такими как `call()` и `apply()`. Единственное отличие между этими двумя методами состоит в том, что первый из них принимает массив с параметрами, который передается вызываемому методу, а второй принимает параметры в виде простого списка аргументов. Эти методы можно использовать для заимствования функциональности из существующих объектов:

```
// пример call()
notmyobj.doStuff.call(myobj, param1, p2, p3);
// пример apply()
notmyobj.doStuff.apply(myobj, [param1, p2, p3]);
```

Здесь имеется метод `myobj` и некоторый другой объект `notmyobj`, обладающий интересующим вас методом `doStuff()`. Вместо того чтобы продираться через тернии наследования и наследовать в объекте `myobj` множество методов объекта `notmyobj`, которые никогда не потребуются, можно просто временно заимствовать метод `doStuff()`.

Вы передаете свой объект со всеми необходимыми параметрами, и заимствуемый метод будет временно связан с вашим объектом, как если бы это был его собственный метод. Фактически ваш объект маскируется под другой объект, чтобы воспользоваться понравившимся вам методом. Этот прием похож на наследование, но без необходимости платить за это «высокую цену» (здесь под «высокой ценой» подразумевается наследование дополнительных свойств и методов, которые вам не нужны).

Пример: заимствование методов массива

Типичный пример использования этого шаблона – заимствование методов массива.

Массивы обладают множеством полезных методов, которые отсутствуют в объектах, напоминающих массивы, таких как `arguments`. Однако объект `arguments` может заимствовать методы массива, например `slice()`, как показано в следующем примере:

```
function f() {
    var args = [].slice.call(arguments, 1, 3);
    return args;
}

// пример
f(1, 2, 3, 4, 5, 6); // вернет [2,3]
```

В этом примере создается пустой массив, для того чтобы получить возможность использовать его метод. Можно пойти чуть более длинным путем и получить тот же результат, заимствовав метод непосредственно из прототипа конструктора `Array` с помощью инструкции `Array.prototype.slice.call(...)`. Эта инструкция выглядит длиннее, но она экономит время, необходимое на создание пустого массива.

Заимствование и связывание

Объект, на который указывает ссылка `this` внутри метода, заимствованного с помощью `call()`/`apply()` или в результате простого присваивания, определяется выражением вызова. Но иногда бывает нужно предварительно «зафиксировать» значение ссылки `this`, то есть связать ее с определенным объектом.

Рассмотрим следующий пример. Здесь имеется объект с именем `one`, обладающий методом `say()`:

```
var one = {
    name: "object",
    say: function (greet) {
        return greet + ", " + this.name;
    }
};

// проверка
one.say('hi'); // "hi, object"
```

И другой объект `two`, не имеющий метода `say()`, но он может заимствовать этот метод из объекта `one`:

```
var two = {
    name: "another object"
};

one.say.apply(two, ['hello']); // "hello, another object"
```

В последнем случае ссылка `this` внутри метода `say()` указывает на объект `two`, поэтому обращение к свойству `this.name` возвращает строку «another object». А что произойдет, если сохранить ссылку на функцию в глобальной переменной или передать ее как функцию обратного вызо-

ва? В клиентских сценариях возникает множество событий и часто используются функции обратного вызова, поэтому такая необходимость действительно может возникнуть:

```
// в случае присваивания функции переменной
// ссылка `this` будет указывать на глобальный объект
var say = one.say;
say('hoho');                // "hoho, undefined"

// передача в виде функции обратного вызова
var yetanother = {
    name: "Yet another object",
    method: function (callback) {
        return callback('Hola');
    }
};
yetanother.method(one.say); // "Hola, undefined"
```

В обоих случаях ссылка `this` внутри метода `say()` указывает на глобальный объект, и программный код действует не так, как ожидалось. Чтобы исправить ошибку (то есть связать объект с методом), можно использовать простую функцию:

```
function bind(o, m) {
    return function () {
        return m.apply(o, [].slice.call(arguments));
    };
}
```

Данная функция `bind()` принимает объект `o` и метод `m`, связывает их вместе и возвращает другую функцию. Возвращаемая функция имеет доступ к ссылкам `o` и `m` через замыкание. То есть даже после выхода из функции `bind()` внутренняя функция будет иметь доступ к ссылкам `o` и `m`, которые всегда будут указывать на оригинальный объект и метод. Создадим новую функцию с помощью `bind()`:

```
var twosay = bind(two, one.say);
twosay('yo'); // "yo, another object"
```

Как видите, даже в случае, когда `twosay()` создается как глобальная функция, ссылка `this` будет указывать не на глобальный объект, а на объект `two`, который был передан функции `bind()`. Неважно, как будет вызвана функция `twosay()`, ссылка `this` внутри нее всегда будет указывать на объект `two`.

Однако за роскошь связывания приходится платить ценой создания дополнительного замыкания.

Function.prototype.bind()

Стандартом ECMAScript 5 предусмотрен дополнительный метод `bind()` объекта `Function.prototype`, такой же простой в использовании, как методы `apply()` и `call()`, выполняющий связывание, как показано ниже:

```
var newFunc = obj.someFunc.bind(myobj, 1, 2, 3);
```

В данном случае вызов метода связывает функцию `someFunc()` с объектом `myobj` и определяет значения трех первых аргументов, ожидаемых функцией `someFunc()`. Это выражение также является примером частичного применения функции, о котором рассказывалось в главе 4.

Давайте посмотрим, как можно реализовать метод `Function.prototype.bind()` для использования в программах, выполняющихся в окружениях, не поддерживающих стандарт ES5:

```
if (typeof Function.prototype.bind === "undefined") {
    Function.prototype.bind = function (thisArg) {
        var fn = this,
            slice = Array.prototype.slice,
            args = slice.call(arguments, 1);

        return function () {
            return fn.apply(thisArg, args.concat(slice.call(arguments)));
        };
    };
}
```

Эта реализация наверняка покажется вам знакомой; в ней используется прием частичного применения и слияния списков аргументов – тех, что передаются методу `bind()` (кроме первого), и тех, что передаются при вызове новой функции, возвращаемой методом `bind()`. Ниже приводится пример ее использования:

```
var twosay2 = one.say.bind(two);
twosay2('Bonjour'); // "Bonjour, another object"
```

В этом примере методу `bind()` не передаются никакие параметры кроме объекта, с которым требуется связать метод `say()`. В следующем примере выполняется частичное применение за счет передачи дополнительного аргумента:

```
var twosay3 = one.say.bind(two, 'Enchanté');
twosay3(); // "Enchanté, another object"
```

В заключение

Существует масса способов реализации наследования в JavaScript. Очень полезно изучать и понимать различные шаблоны, потому что это поможет вам улучшить понимание языка. В этой главе вы познакоми-

лись с несколькими классическими и с несколькими современными шаблонами реализации наследования.

Однако наследование – это, пожалуй, не та проблема, с которой вы будете часто сталкиваться в процессе разработки. Отчасти потому, что эта проблема фактически уже решена тем или иным способом в библиотеках, которые вы будете использовать, а отчасти потому, что в JavaScript редко приходится выстраивать длинные и сложные иерархии наследования. В языках программирования со строгим контролем типов наследование может быть единственным способом повторного использования программного кода. В JavaScript зачастую имеются более простые и элегантные способы, включая заимствование методов, их связывание, копирование свойств и смешивание свойств из нескольких объектов.

Помните, что целью является повторное использование программного кода, а наследование – это лишь один из способов ее достижения.

7

Шаблоны проектирования

Шаблоны проектирования, представленные «бандой четырех» в уже упоминавшейся книге, предлагают решение наиболее типичных задач, связанных с архитектурой объектно-ориентированного программного обеспечения. Они достаточно давно используются на практике и доказали свою полезность во многих ситуациях. Именно поэтому и вам будет полезно познакомиться с ними и обсудить их.

Хотя эти шаблоны проектирования могут применяться в любом языке программирования, тем не менее многие годы они изучаются с позиций языков со строгим контролем типов и со статическими классами, таких как C++ и Java.

JavaScript, будучи динамическим нетипизированным языком, опирающимся на использование прототипов, иногда позволяет удивительно легко и даже тривиально реализовать некоторые их этих шаблонов.

В качестве первого примера рассмотрим шаблон единственного объекта (singleton), демонстрирующий, насколько сильно может отличаться реализация в JavaScript от реализации в языках на основе статических классов.

Единственный объект

Суть шаблона единственного объекта состоит в том, чтобы обеспечить возможность создать только один экземпляр определенного класса. Это означает, что при попытке создать второй экземпляр того же класса вызывающая программа должна получить объект, созданный при первой попытке.

И как же это относится к JavaScript? В JavaScript нет классов – только объекты. Когда создается новый объект, фактически в программе не существует объектов, похожих на него, и новый объект уже является

единственным. При создании простого объекта с помощью литерала также получается единственный объект:

```
var obj = {
    myprop: 'my value'
};
```

В JavaScript объекты никогда не равны между собой, если только они не являются одним и тем же объектом, поэтому даже если создать второй такой же объект с тем же самым набором членов, он не будет тем же объектом, что и первый:

```
var obj2 = {
    myprop: 'my value'
};
obj === obj2; // false
obj == obj2;  // false
```

Поэтому можно смело сказать, что всякий раз, когда объект создается с использованием литерала объекта, фактически создается единственный объект, и для этого не требуется использовать какие-либо специальные синтаксические конструкции.

Обратите внимание: иногда под шаблоном единственного объекта (singleton) в контексте JavaScript подразумевается шаблон «модуль», обсуждавшийся в главе 5.

Использование оператора new

В JavaScript отсутствуют классы, поэтому с технической точки зрения буквальное определение единственного объекта не имеет смысла. Но в JavaScript есть оператор `new`, который используется для создания объектов с помощью функций-конструкторов, и иногда может потребоваться реализовать шаблон единственного объекта с помощью этого синтаксиса. Идея заключается в том, чтобы при создании нескольких объектов с помощью оператора `new` и одного и того же конструктора вы получали бы просто новые ссылки на один и тот же объект.

Внимание: следующее обсуждение представляет не практический, а скорее теоретический интерес – как пример реализации обходных решений проблем, связанных с особенностями некоторых языков программирования (со строгим контролем типов) с классами, в которых функции не являются обычными объектами.

Ожидаемое поведение иллюстрирует следующий фрагмент (здесь предполагается, что вы не являетесь приверженцем теории о множественных Вселенных и принимаете идею существования единственной Вселенной (Universe)):

```
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

В этом примере новый объект `uni` создается только при первом вызове конструктора. При втором вызове (а также третьем, четвертом и так далее) возвращается тот же самый объект `uni`. Именно поэтому выполняется условие `uni === uni2`, так как в действительности это всего лишь две ссылки, указывающие на один и тот же объект. Но как добиться этого в языке JavaScript?

Для этого необходимо, чтобы конструктор `Universe` запоминал ссылку `this` на объект и затем возвращал ее при последующих вызовах. Добиться этого можно несколькими способами:

- Для хранения экземпляра можно использовать глобальную переменную. Однако использовать этот способ не рекомендуется, так как он нарушает общепринятый постулат, что глобальные переменные – это зло. Кроме того, глобальную переменную можно изменить по неосторожности. Поэтому далее мы не будем рассматривать этот способ.
- Ссылку на экземпляр можно сохранить в статическом свойстве конструктора. Функции в JavaScript являются объектами, поэтому они могут иметь собственные свойства. Вы можете создать свойство, например `Universe.instance`, и сохранить ссылку на объект в нем. Это простое и понятное решение, но оно имеет один недостаток – свойство `instance` является общедоступным и может быть изменено внешним программным кодом, в результате чего есть риск потерять свой экземпляр.
- Можно сохранить экземпляр в замыкании. Это предотвратит возможность изменения экземпляра из-за пределов конструктора за счет создания дополнительного замыкания.

Давайте рассмотрим примеры реализации второго и третьего способов.

Экземпляр в статическом свойстве

Ниже приводится пример сохранения единственного экземпляра в статическом свойстве конструктора `Universe`:

```
function Universe() {

    // имеется ли экземпляр, созданный ранее?
    if (typeof Universe.instance === "object") {
        return Universe.instance;
    }

    // создать новый экземпляр
    this.start_time = 0;
    this.bang = "Big";
```

```
    // сохранить его
    Universe.instance = this;

    // неявный возврат экземпляра:
    // return this;
}

// проверка
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

Как видите, это достаточно простое решение, обладающее единственным недостатком – свойство `instance` является общедоступным. Маловероятно, что какой-то другой программный код изменит это свойство по ошибке (гораздо менее вероятно, чем в случае, если бы экземпляр сохранялся в глобальной переменной), тем не менее такая возможность существует.

Экземпляр в замыкании

Другой способ реализовать шаблон единственного экземпляра заключается в использовании замыкания для ограничения доступа к экземпляру. Это можно реализовать с помощью шаблона частных статических членов, обсуждавшегося в главе 5. Секрет состоит в том, чтобы переопределить конструктор:

```
function Universe() {

    // сохраненный экземпляр
    var instance = this;

    // создать новый экземпляр
    this.start_time = 0;
    this.bang = "Big";

    // переопределить конструктор
    Universe = function () {
        return instance;
    };
}

// проверка
var uni = new Universe();
var uni2 = new Universe();
uni === uni2; // true
```

При первом обращении вызывается оригинальный конструктор, возвращающий ссылку `this` как обычно. При втором, третьем и так далее обращении вызывается уже переопределенный конструктор. Новый кон-

структор обладает доступом к частной переменной `instance` благодаря замыканию и просто возвращает ее.

Фактически данная реализация является также примером использования шаблона самоопределяемых функций, представленного в главе 4. Недостаток этого шаблона, как уже обсуждалось ранее, состоит в том, что при переопределении функции (в данном случае конструктора `Universe()`) она теряет все свойства, которые могли быть добавлены между моментом ее определения и моментом переопределения. В данном конкретном случае все, что будет добавлено в прототип функции `Universe()`, окажется недоступно экземпляру, созданному оригинальной реализацией.

Ниже показано, как эта проблема может проявлять себя на практике:

```
// добавить свойство в прототип
Universe.prototype.nothing = true;

var uni = new Universe();

// добавить еще одно свойство в прототип
// уже после создания первого объекта
Universe.prototype.everything = true;

var uni2 = new Universe();
```

Проверка:

```
// объект имеет доступ только
// к оригинальному прототипу
uni.nothing;                    // true
uni2.nothing;                   // true
uni.everything;                 // undefined
uni2.everything;                // undefined

// это выражение дает ожидаемый результат:
uni.constructor.name;           // “Universe”

// а это нет:
uni.constructor === Universe;  // false
```

Причина, по которой ссылка `uni.constructor` больше не указывает на конструктор `Universe()`, состоит в том, что она по-прежнему указывает на оригинальный конструктор, а не на переопределенный.

Если необходимо сохранить свойства прототипа и ссылку на конструктор, эту проблему можно решить внесением нескольких улучшений:

```
function Universe() {

    // сохраненный экземпляр
    var instance;
```

```
    // переопределить конструктор
    Universe = function Universe() {
        return instance;
    };

    // перенести свойства прототипа
    Universe.prototype = this;

    // создать экземпляр
    instance = new Universe();

    // переустановить указатель на конструктор
    instance.constructor = Universe;

    // добавить остальную функциональность
    instance.start_time = 0;
    instance.bang = "Big";

    return instance;
}
```

Теперь все проверки должны давать ожидаемые результаты:

```
// добавить свойство в прототип и создать экземпляр
Universe.prototype.nothing = true;     // true
var uni = new Universe();
Universe.prototype.everything = true;  // true
var uni2 = new Universe();

// тот же самый экземпляр
uni === uni2;                          // true

// все свойства прототипа доступны
// независимо от того, когда они были добавлены
uni.nothing && uni.everything && uni2.nothing && uni2.everything; // true
// обычные свойства объекта также доступны
uni.bang;                              // "Big"
// ссылка на конструктор содержит правильный указатель
uni.constructor === Universe;          // true
```

Другой вариант решения проблемы заключается в обертывании конструктора и ссылки на экземпляр немедленно вызываемой функцией. При первом обращении конструктор создаст объект и сохранит ссылку на него в частной переменной `instance`. При повторных обращениях конструктор будет просто возвращать значение частной переменной. Все проверки, проведенные в предыдущем фрагменте, будут возвращать ожидаемые результаты и для новой реализации:

```
var Universe;

(function () {
```

```
    var instance;

    Universe = function Universe() {

        if (instance) {
            return instance;
        }

        instance = this;

        // добавить остальную функциональность
        this.start_time = 0;
        this.bang = "Big";
    };
}());
```

Фабрика

Назначение фабрики в том, чтобы создавать объекты. Этот шаблон обычно реализуется в виде классов или в виде статических методов классов и преследует следующие цели:

- Выполнение повторяющихся операций, необходимых при создании похожих объектов
- Предложить пользователям фабрики способ создания объектов без необходимости знать их тип (класс) на этапе компиляции

Второй пункт наиболее важен в языках со статическими классами, где может оказаться совсем непросто создать экземпляр класса, неизвестного заранее (на этапе компиляции). В JavaScript эта часть шаблона реализуется очень просто.

Объекты, создаваемые фабричным методом (или классом), обычно наследуют один и тот же родительский объект; они являются подклассами, реализующими специализированные функциональные возможности. Иногда общим предком является тот же класс, который содержит фабричный метод.

Рассмотрим пример реализации этого шаблона, в котором имеются:

- Общий родительский конструктор `CarMaker`.
- Статический метод объекта `CarMaker` с именем `factory()`, который создает объекты-автомобили.
- Специализированные конструкторы `CarMaker.Compact`, `CarMaker.SUV` и `CarMaker.Convertible`, наследующие конструктор `CarMaker`. Все эти конструкторы определяются как статические свойства предка, благодаря чему предотвращается засорение глобального пространства имен, а кроме того, при такой организации мы точно знаем, где искать их, когда они нам потребуются.

Сначала посмотрим, как будет использоваться окончательная реализация:

```
var corolla  = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');
corolla.drive();  // "Vroom, I have 4 doors"
solstice.drive(); // "Vroom, I have 2 doors"
cherokee.drive(); // "Vroom, I have 24 doors"
```

Эта часть:

```
var corolla = CarMaker.factory('Compact');
```

является, вероятно, самой узнаваемой в шаблоне фабрики. Имеется метод, принимающий тип объекта в виде строки и создающий объект указанного типа. Нет никаких конструкторов, вызываемых с оператором `new`, никаких литералов объектов – только функция, создающая объект, опираясь на тип, определяемый строкой.

Ниже приводится пример реализации шаблона фабрики, который обеспечивает работоспособность программного кода из предыдущего фрагмента:

```
// родительский конструктор
function CarMaker() {}

// метод предка
CarMaker.prototype.drive = function () {
    return "Vroom, I have " + this.doors + " doors";
};

// статический фабричный метод
CarMaker.factory = function (type) {
    var constr = type,
        newcar;

    // сообщить об ошибке, если конструктор
    // для запрошенного типа отсутствует
    if (typeof CarMaker[constr] !== "function") {
        throw {
            name: "Error",
            message: constr + " doesn't exist"
        };
    }

    // в этой точке известно, что требуемый конструктор существует
    // поэтому определим отношения наследования с предком,
    // но только один раз
    if (typeof CarMaker[constr].prototype.drive !== "function") {
        CarMaker[constr].prototype = new CarMaker();
    }
```

```
        // создать новый экземпляр
        newcar = new CarMaker[constr]();
        // дополнительно можно вызвать какие-либо методы
        // и затем вернуть объект...
        return newcar;
    };

    // специализированные конструкторы
    CarMaker.Compact = function () {
        this.doors = 4;
    };
    CarMaker.Convertible = function () {
        this.doors = 2;
    };
    CarMaker.SUV = function () {
        this.doors = 24;
    };
```

В реализации шаблона фабрики нет ничего особенно сложного. Все, что необходимо сделать, – это отыскать функцию-конструктор, которая создаст объект требуемого типа. В данном случае для отображения типов объектов на конструкторы, создающие их, используется простое соглашение об именовании. Часть, где определяются отношения наследования, – это пример реализации повторяющихся операций, общих для всех конструкторов, которые было бы разумно вынести в фабричные методы.

Встроенная фабрика объектов

В качестве примера «природной фабрики» рассмотрим встроенный глобальный конструктор `Object()`. Он также ведет себя, как фабрика, потому что создает различные объекты исходя из входных данных. Если передать конструктору простое число, он создаст объект, задействовав конструктор `Number()`. То же справедливо в отношении строк и логических значений. При любых других значениях, включая отсутствие входных значений, будут создаваться обычные объекты.

Ниже приводятся несколько примеров создания объектов и проверки их поведения. Обратите внимание, что конструктор `Object()` может вызываться как с оператором `new`, так и без него:

```
var o = new Object(),
    n = new Object(1),
    s = Object('1'),
    b = Object(true);

// проверка
o.constructor === Object;  // true
n.constructor === Number;  // true
```

```
s.constructor === String;  // true
b.constructor === Boolean; // true
```

Тот факт, что конструктор `Object()` также является фабрикой, не имеет особого практического значения. Просто важно было отметить, что шаблон фабрики используется повсюду.

Итератор

Шаблон итератора применяется, когда имеется объект, содержащий совокупность данных. Эти данные могут храниться в виде сложной структуры, а вам необходимо обеспечить удобный доступ к каждому элементу этой структуры. Пользователи вашего объекта не обязаны знать, как организованы ваши данные, – им необходим доступ к отдельным элементам.

Объект, реализующий шаблон итератора, должен предоставить метод `next()`. При последующем обращении метод `next()` должен вернуть следующий элемент, и только вам решать, что означает понятие «следующий» для вашей конкретной структуры данных.

Предположим, что имеется объект `agg`, и вам необходимо обеспечить доступ к каждому элементу простым вызовом метода `next()` в цикле, как показано ниже:

```
var element;
while (element = agg.next()) {
    // выполнить некоторые операции над элементом ...
    console.log(element);
}
```

Объект с данными, реализующий шаблон итератора, обычно предоставляет еще один удобный метод `hasNext()`, чтобы пользователи объекта имели возможность определить, не был ли достигнут конец данных. Другой способ обеспечения последовательного доступа ко всем элементам, на этот раз с использованием метода `hasNext()`, можно представить, как показано ниже:

```
while (agg.hasNext()) {
    // выполнить некоторые операции над элементом...
    console.log(agg.next());
}
```

Теперь, когда мы получили представление о способах использования, можно перейти к реализации объекта с данными.

При реализации шаблона итератора имеет смысл обеспечить сокрытие самих данных и указателя (индекса) на следующий доступный элемент. Чтобы продемонстрировать типичную реализацию, предположим, что данные организованы в виде обычного массива, а «специальная» логи-

ка извлечения следующего элемента будет возвращать каждый второй элемент массива:

```
var agg = (function () {

    var index = 0,
        data = [1, 2, 3, 4, 5],
        length = data.length;

    return {

        next: function () {
            var element;
            if (!this.hasNext()) {
                return null;
            }
            element = data[index];
            index = index + 2;
            return element;
        },

        hasNext: function () {
            return index < length;
        }
    };
}());
```

Чтобы упростить доступ к текущему элементу и обеспечить возможность выполнять обход элементов несколько раз, объект может предоставлять следующие вспомогательные методы:

`rewind()`

Переустанавливает указатель в начало.

`current()`

Возвращает текущий элемент, чего нельзя сделать с помощью метода `next()`, который передвигает указатель дальше.

Реализация этих методов не представляет никакой сложности:

```
var agg = (function () {

    // [пропустить...]

    return {

        // [пропустить...]

        rewind: function () {
            index = 0;
        },
```

```
        current: function () {
            return data[index];
        }

    };
}());
```

Теперь протестируем итератор:

```
// этот цикл выведет значения 1, 3 и 5
while (agg.hasNext()) {
    console.log(agg.next());
}

// возврат
agg.rewind();
console.log(agg.current()); // 1
```

Этот фрагмент выведет в консоль значения: 1, 3, 5 (в цикле) и 1 (после вызова метода `rewind()`).

Декоратор

При использовании шаблона декораторов дополнительную функциональность можно добавлять к объекту динамически во время выполнения. В языках со статическими классами реализовать такую возможность было бы гораздо сложнее. В JavaScript объекты допускают возможность изменения, поэтому процедура добавления функциональности в объект не представляет из себя ничего сложного.

Важной особенностью шаблона декораторов является возможность его использования для определения желаемого поведения объектов. Имея простой объект, обладающий некоторой базовой функциональностью, вы можете выбирать из множества доступных декораторов те, которыми желательно было бы расширить этот простой объект, и в каком порядке, если порядок имеет значение.

Пример использования

Рассмотрим пример использования шаблона. Допустим, что вы разрабатываете веб-приложение для интернет-магазина. Каждая новая покупка – это новый объект `sale`. Объект `sale` хранит цену товара, которую можно получить вызовом метода `sale.getPrice()`. В зависимости от некоторых обстоятельств этот объект можно было бы декорировать дополнительными функциональными возможностями. Представьте себе ситуацию, когда покупатель находится в Канаде, в провинции Квебек. В этом случае он должен заплатить федеральный налог и местный налог провинции Квебек. Следуя терминологии шаблона декораторов, вы бы сказали, что «декорируете» объект декоратором федерального на-

лога и декоратором местного налога провинции Квебек. Дополнительно объект можно было бы декорировать механизмом форматирования цены. Реализация этого примера могла бы выглядеть, как показано ниже:

```
var sale = new Sale(100);       // цена 100 долларов
sale = sale.decorate('fedtax'); // добавить федеральный налог
sale = sale.decorate('quebec'); // добавить местный налог
sale = sale.decorate('money');  // форматировать как денежную сумму
sale.getPrice();                // "$112.88"
```

Другой покупатель может находиться в провинции, в которой не взимается местный налог, но при этом вам могло бы потребоваться отформатировать цену для представления в канадских долларах:

```
var sale = new Sale(100);       // цена 100 долларов
sale = sale.decorate('fedtax'); // добавить федеральный налог
sale = sale.decorate('cdn');    // форматировать как сумму в CDN
sale.getPrice();                // "CDN$ 105.00"
```

Как видите, декораторы обеспечивают весьма гибкий способ добавления функциональности и настройки поведения объектов во время выполнения. А теперь посмотрим, как реализовать этот шаблон.

Реализация

Один из способов реализации шаблона декораторов заключается в создании для каждого декоратора объекта, содержащего методы, которые должны быть переопределены. Каждый декоратор фактически наследует объект, расширенный предыдущим декоратором. Каждый декорированный метод вызывает одноименный метод объекта `uber` (унаследованного объекта) и, получив от него некоторое значение, выполняет необходимые дополнительные операции.

То есть в первом примере использования при обращении к методу `sale.getPrice()` вызывается метод декоратора `money` (рис. 7.1). Но поскольку каждый декорированный метод сначала вызывает метод родительского объекта, то метод `getPrice()` декоратора `money` сначала вызовет метод `getPrice()` объекта `quebec`, а он в свою очередь вызовет метод `getPrice()` объекта `fedtax` и так далее. Эта цепочка продолжается вплоть до оригинального, недекорированного метода `getPrice()`, реализованного в конструкторе `Sale()`.

Реализация начинается с конструктора и метода прототипа:

```
function Sale(price) {
    this.price = price || 100;
}
Sale.prototype.getPrice = function () {
    return this.price;
};
```

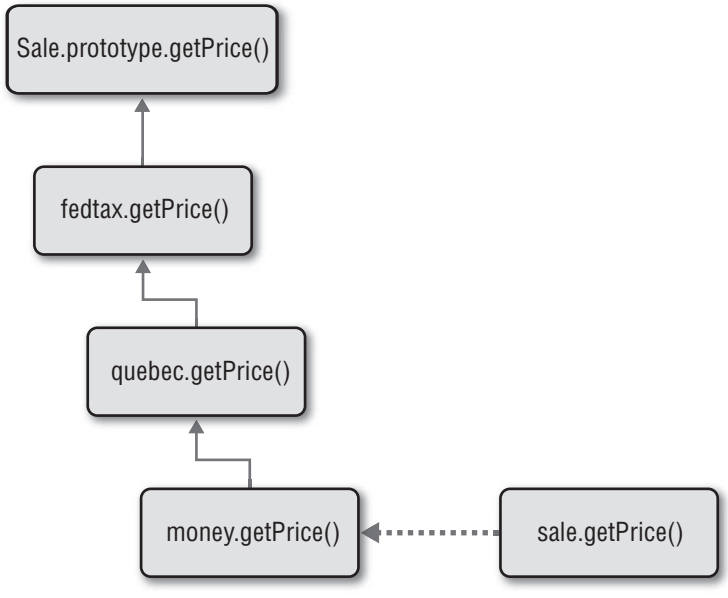

Рис. 7.1. *Реализация шаблона декораторов*

Все объекты-декораторы реализуются как свойства конструктора:

```
Sale.decorators = {};
```

Давайте рассмотрим пример реализации одного из декораторов. Это объект, реализующий адаптированный метод `getPrice()`. Обратите внимание, что этот метод сначала получает значение, обратившись к родительскому методу, а затем изменяет это значение:

```
Sale.decorators.fedtax = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 5 / 100;
        return price;
    }
};
```

Аналогичным способом реализуются остальные необходимые декораторы. Они могут быть реализованы в виде расширений базового конструктора `Sale()`. Они могут находиться даже в отдельных файлах и создаваться сторонними разработчиками:

```
Sale.decorators.quebec = {
    getPrice: function () {
        var price = this.uber.getPrice();
        price += price * 7.5 / 100;
```

```
        return price;
    }
};

Sale.decorators.money = {
    getPrice: function () {
        return "$" + this.uber.getPrice().toFixed(2);
    }
};

Sale.decorators.cdn = {
    getPrice: function () {
        return "CDN$ " + this.uber.getPrice().toFixed(2);
    }
};
```

Наконец, давайте посмотрим, как можно реализовать «волшебный» метод `decorate()`, который соединяет все части воедино. Напомню, что он вызывается, как показано ниже:

```
sale = sale.decorate('fedtax');
```

Строка `'fedtax'` соответствует имени объекта, реализованного в виде свойства `Sale.decorators.fedtax`. Новый объект `newobj`, являющийся результатом декорирования, будет наследовать объект, к которому применяется декоратор (это будет или оригинальный объект, или объект, получившийся в результате применения предыдущего декоратора), то есть объект `this`. Для организации наследования воспользуемся шаблоном временного конструктора, представленным в предыдущей главе. Кроме того, необходимо установить значение свойства `uber` объекта `newobj`, чтобы потомок имел доступ к родительским свойствам и методам. Затем скопируем все необходимые свойства из декоратора во вновь созданный декорированный объект `newobj`. И в конце вернем объект `newobj` вызывающей программе. В нашем примере использования он станет новым дополненным объектом `sale`:

```
Sale.prototype.decorate = function (decorator) {
    var F = function () {},
        overrides = this.constructor.decorators[decorator],
        i, newobj;
    F.prototype = this;
    newobj = new F();
    newobj.uber = F.prototype;
    for (i in overrides) {
        if (overrides.hasOwnProperty(i)) {
            newobj[i] = overrides[i];
        }
    }
    return newobj;
};
```

Реализация с использованием списка

Теперь рассмотрим немного иную реализацию, в которой используются преимущества динамической природы JavaScript и вообще не используется наследование. Кроме того, вместо вызова каждого предыдущего метода в цепочке декораторов мы просто будем передавать результат вызова предыдущего метода следующему.

Такая реализация позволяет также выполнять операцию, *обратную декорированию*, то есть убирать декораторы, или просто удалять элементы из списка декораторов.

Это поможет немного упростить порядок использования декораторов, так как в новой реализации отпадает необходимость присваивания значения, возвращаемого методом `decorate()`. В данной реализации метод `decorate()` вообще не будет выполнять никаких операций над объектом – он будет просто добавлять элементы в список:

```
var sale = new Sale(100); // цена 100 долларов
sale.decorate('fedtax');  // добавить федеральный налог
sale.decorate('quebec');  // добавить местный налог
sale.decorate('money');   // форматировать как денежную сумму
sale.getPrice();          // "$112.88"
```

Список декораторов будет храниться как собственное свойство конструктора `Sale()`:

```
function Sale(price) {
    this.price = (price > 0) || 100;
    this.decorators_list = [];
}
```

Сами декораторы по-прежнему реализуются как свойства `Sale.decorators`. Обратите внимание, что методы `getPrice()` теперь стали проще, так как отпала необходимость вызывать родительский метод `getPrice()`, чтобы получить промежуточный результат; этот результат будет передаваться им в виде параметра:

```
Sale.decorators = {};

Sale.decorators.fedtax = {
    getPrice: function (price) {
        return price + price * 5 / 100;
    }
};

Sale.decorators.quebec = {
    getPrice: function (price) {
        return price + price * 7.5 / 100;
    }
};
```

```
Sale.decorators.money = {
    getPrice: function (price) {
        return "$" + price.toFixed(2);
    }
};
```

Самое интересное происходит в родительских методах `decorate()` и `getPrice()`. В предыдущей реализации метод `decorate()` был достаточно сложным, а метод `getPrice()` – простым. В данной реализации все наоборот: метод `decorate()` просто добавляет новый элемент в список, а метод `getPrice()` выполняет всю основную работу. Под этой работой подразумевается обход списка декораторов и вызов каждого из методов `getPrice()`, которым передается результат предыдущего вызова:

```
Sale.prototype.decorate = function (decorator) {
    this.decorators_list.push(decorator);
};

Sale.prototype.getPrice = function () {
    var price = this.price,
        i,
        max = this.decorators_list.length,
        name;
    for (i = 0; i < max; i += 1) {
        name = this.decorators_list[i];
        price = Sale.decorators[name].getPrice(price);
    }
    return price;
};
```

Эта вторая реализация шаблона декораторов получилась проще, и в ней не используется механизм наследования. Методы декораторов также получились немного проще. Вся работа выполняется методом, который «согласен» на декорирование. В этом примере реализации метод `getPrice()` является единственным методом, допускающим возможность декорирования. Если вам потребуется добавить возможность декорирования и к другим методам, то фрагмент программного кода, выполняющий обход списка декораторов, необходимо будет повторить в каждом таком методе. Однако эту операцию легко оформить в виде вспомогательного метода, который принимает метод и делает его «декорируемым». В такой реализации свойство `decorators_list` можно было бы превратить в объект со свойствами, имена которых будут совпадать с именами методов, а значениями будут массивы объектов-декораторов.

Стратегия

Шаблон стратегии позволяет выбирать тот или иной алгоритм во время выполнения. Пользователи вашего программного кода могут работать с одним и тем же интерфейсом и выбирать из множества доступных ал-

горитмов тот, который лучше подходит для решения определенной задачи в зависимости от сложившихся условий.

Примером использования шаблона стратегии может служить решение проблемы проверки полей формы. Вы можете определить единственный объект, выполняющий проверку, с методом `validate()`. Этот метод будет вызываться независимо от конкретного типа формы и всегда возвращать один и тот же результат – список данных, не прошедших проверку, и сообщения об ошибках.

Но в каждом конкретном случае, в зависимости от формы и проверяемых данных, пользователи вашего объекта, реализующего проверку, смогут выбирать различные виды проверок. Ваш объект будет выбирать лучшую стратегию решения задачи и делегировать выполнение проверок конкретных данных соответствующим алгоритмам.

Пример проверки данных

Допустим, что имеется следующий набор данных, возможно полученный из формы на странице, и вам необходимо проверить их допустимость:

```
var data = {
    first_name: "Super",
    last_name: "Man",
    age: "unknown",
    username: "o_O"
};
```

Чтобы объект, выполняющий проверку, смог выбрать наилучшую стратегию для данного конкретного примера, вам необходимо сначала настроить его и задать набор правил определения допустимости данных.

Предположим, что значение `last_name` не является обязательным, а в поле `first_name` допускается указывать любые комбинации символов, но необходимо, чтобы поле `age` содержало число, а поле `username` содержало бы только буквы и цифры – никакие специальные символы в этом поле недопустимы. Конфигурация объекта могла бы выглядеть так:

```
validator.config = {
    first_name: 'isNonEmpty',
    age: 'isNumber',
    username: 'isAlphaNum'
};
```

Теперь, когда у нас имеется настроенный объект, выполняющий проверку, можно вызвать его метод `validate()` и вывести в консоль все сообщения об ошибках:

```
validator.validate(data);
if (validator.hasErrors()) {
    console.log(validator.messages.join("\n"));
}
```

Этот фрагмент мог бы вывести следующие сообщения:

```
Invalid value for *age*, the value can only be a valid number, e.g. 1, 3.14
or 2010
Invalid value for *username*, the value can only contain characters and
numbers, no special symbols
```

*(Недопустимое значение в поле ***age***, значением может быть только число, например 1, 3.14 или 2010.*
*Недопустимое значение в поле *username*, значение может содержать только буквы и цифры, специальные символы недопустимы)*

Теперь посмотрим, как реализован объект, выполняющий проверку. Доступные алгоритмы проверки данных являются объектами с предопределенным интерфейсом – они реализуют метод `validate()` и содержат однострочное свойство с текстом для вывода в сообщениях об ошибках:

```
// проверяет наличие значения
validator.types.isNonEmpty = {
    validate: function (value) {
        return value !== "";
    },
    instructions: "the value cannot be empty"
};

// проверяет, является ли значение числом
validator.types.isNumber = {
    validate: function (value) {
        return !isNaN(value);
    },
    instructions: "the value can only be a valid number, e.g. 1, 3.14 or 2010"
};

// проверяет, содержит ли значение только буквы и цифры
validator.types.isAlphaNum = {
    validate: function (value) {
        return !/[^a-z0-9]/i.test(value);
    },
    instructions: "the value can only contain characters and numbers,
                   no special symbols"
};
```

И наконец, сам объект `validator`, выполняющий проверку:

```
var validator = {

    // все доступные проверки
    types: {},

    // сообщения об ошибках
    // в текущем сеансе проверки
    messages: [],
```

```
    // текущие параметры проверки
    // имя: тип проверки
    config: {},

    // интерфейсный метод
    // аргумент `data` – это пары ключ => значение
    validate: function (data) {

        var i, msg, type, checker, result_ok;

        // удалить все сообщения
        this.messages = [];

        for (i in data) {

            if (data.hasOwnProperty(i)) {

                type = this.config[i];
                checker = this.types[type];

                if (!type) {
                    continue; // проверка не требуется
                }
                if (!checker) { // ай-яй-яй
                    throw {
                        name: "ValidationError",
                        message: "No handler to validate type " + type
                    };
                }

                result_ok = checker.validate(data[i]);
                if (!result_ok) {
                    msg = "Invalid value for *" + i + "*, " +
                          checker.instructions;
                    this.messages.push(msg);
                }
            }
        }
        return this.hasErrors();
    },

    // вспомогательный метод
    hasErrors: function () {
        return this.messages.length !== 0;
    }
};
```

Как видите, объект `validator` **является достаточно универсальным и мог бы использоваться в таком виде во многих ситуациях, когда требуется проверить данные. Его можно улучшать, добавляя в него больше вари-**

антов проверок. Когда вы начнете использовать его в разных ситуациях, вы очень скоро соберете отличную коллекцию разнообразных проверок. При этом для создания нового типа проверки достаточно будет определить конфигурацию объекта `validator` и вызвать его метод `validate()`.

Фасад

Фасад (façade) – простой шаблон; он только предоставляет альтернативный интерфейс для объекта. При проектировании считается хорошей практикой делать методы короткими и не выполнять в них слишком большое количество операций. Следуя такой практике, вы будете получать большее количество методов, чем в случае реализации *супер* методов с большим количеством параметров. Бывает, что два или более методов часто вызываются вместе. В подобных ситуациях есть смысл создать новый метод, обертывающий повторяющуюся комбинацию вызовов других методов.

Например, для обработки событий в броузерах часто используются следующие методы:

`stopPropagation()`

: Используется в обработчиках событий, чтобы запретить дальнейшее всплытие события вверх по дереву DOM.

`preventDefault()`

: Запрещает выполнение действий, предусмотренных по умолчанию (например, переход по ссылке или отправку формы).

Это два совершенно разных метода, выполняющих разные задачи, и они должны существовать отдельно, но в то же время они часто вызываются друг за другом. Поэтому, чтобы не повторять вызовы этих двух методов снова и снова, можно создать фасадный метод, который будет вызывать их:

```
var myevent = {
    // ...
    stop: function (e) {
        e.preventDefault();
        e.stopPropagation();
    }
    // ...
};
```

Шаблон фасада также может использоваться для сокрытия различий между броузерами за фасадными методами. В продолжение предыдущего примера можно было бы добавить программный код, учитывающий отличия интерфейса событий в IE:

```
var myevent = {
    // ...
    stop: function (e) {
        // прочие броузеры
        if (typeof e.preventDefault === "function") {
            e.preventDefault();
        }
        if (typeof e.stopPropagation === "function") {
            e.stopPropagation();
        }
        // IE
        if (typeof e.returnValue === "boolean") {
            e.returnValue = false;
        }
        if (typeof e.cancelBubble === "boolean") {
            e.cancelBubble = true;
        }
    }
    // ...
};
```

Шаблон фасада также удобно использовать при перепроектировании интерфейсов и реорганизации программного кода. Чтобы изменить реализацию объекта, может потребоваться некоторое время (предполагается, что речь идет о сложном объекте). В это же время может разрабатываться новый программный код, использующий этот объект. Вы можете сначала продумать новый интерфейс объекта и создать фасад для старого объекта, имитирующий новый интерфейс. При таком подходе, когда вы полностью замените старый объект новым, вам не придется вносить существенные изменения в программный код, использующий этот объект, так как он уже будет использовать обновленный интерфейс.

Прокси-объект

В шаблоне прокси-объекта (промежуточного объекта) один объект играет роль интерфейса другого объекта. Этот шаблон отличается от шаблона фасада, где все, что от вас требуется, – это создать удобные методы, объединяющие в себе вызовы других методов. Промежуточный объект располагается между пользовательским программным кодом и объектом и регулирует доступ к этому объекту.

Этот шаблон может показаться излишеством, тем не менее его использование может дать прирост производительности. Промежуточный объект служит защитником основного объекта (иногда его называют «действительным объектом») и стремится максимально уменьшить количество обращений к действительному объекту.

Примером использования этого шаблона может служить реализация того, что мы называем *отложенной инициализацией*. Представьте, что

инициализация действительного объекта – достаточно дорогостоящая операция, и может случиться так, что пользовательский программный код потребует выполнить инициализацию объекта, но никогда не будет использовать его. В подобных ситуациях нам может помочь прокси-объект, являющийся интерфейсом к действительному объекту. Прокси-объект может принять запрос на инициализацию, но не передавать его дальше, пока не станет очевидным, что действительный объект на самом деле необходим.

На рис. 7.2 изображена ситуация, когда пользовательский программный код запрашивает инициализацию, а прокси-объект отвечает, что все в порядке, но на самом деле запрос не передается действительному объекту, пока не станет очевидным, что пользовательский программный код собирается воспользоваться объектом. Только в этом случае прокси-объект передаст оба запроса – запрос на инициализацию и запрос на выполнение операции.

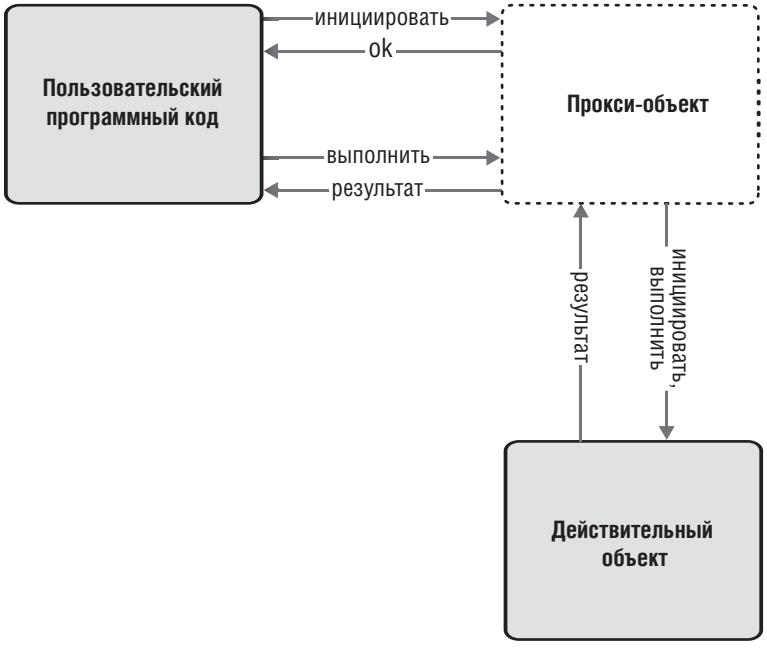

Рис. 7.2. *Взаимодействие пользовательского программного кода с действительным объектом через прокси-объект*

Пример

Шаблон прокси-объекта особенно полезен, когда действительный объект выполняет дорогостоящие операции. В веб-приложениях одной из самых дорогостоящих операций является операция выполнения запро-

са по сети, поэтому их желательно объединять в общие HTTP-запросы насколько это возможно. Давайте рассмотрим пример, который как раз это и делает и демонстрирует применение шаблона прокси-объекта.

Вывод дополнительной информации о видеофильме

В этом примере мы создадим небольшое приложение, которое проигрывает выбранные видеофильмы (рис. 7.3). Вы можете опробовать этот пример и познакомиться с программным кодом по адресу *http://www.jspatterns.com/book/7/proxy.html*.

Рис. 7.3. *Вывод дополнительной информации о видеофильме*

На странице выводится список названий видеофильмов. Когда пользователь щелкает на названии фильма, под названием открывается дополнительная область, в которой выводится расширенная информация о видеофильме и предоставляется возможность просмотреть фильм. Подробное описание и адрес URL видеофильма изначально отсутствуют на странице – их необходимо получить, выполнив запрос к веб-службе. Веб-служба способна принимать в одном запросе сразу несколько идентификаторов видеофильмов; благодаря этому мы можем повысить скорость работы приложения, уменьшив число HTTP-запросов и получая информацию сразу о нескольких видеофильмах.

Наше приложение дает возможность распахнуть сразу несколько (или даже все) описаний видеофильмов, что дает нам отличную возможность объединить сразу несколько обращений к веб-службе в единый запрос.

Без прокси-объекта

Главными «действующими лицами» в приложении являются два объекта:

`videos`

Отвечает за развертывание/свертывание области с дополнительной информацией (метод `videos.getInfo()`) и проигрывание видеофильма (метод `videos.getPlayer()`).

`http`

Отвечает за взаимодействие с сервером посредством метода `http.makeRequest()`.

В отсутствие прокси-объекта метод `videos.getInfo()` будет вызывать метод `http.makeRequest()` для получения информации о каждом видеофильме в отдельности. После добавления прокси-объекта он станет новым действующим лицом с именем `proxy`, будет располагаться между объектами `videos` и `http` и передавать запросы методу `makeRequest()`, объединяя их по мере возможности.

Сначала мы рассмотрим реализацию приложения без прокси-объекта, а затем добавим прокси-объект, чтобы повысить отзывчивость приложения.

HTML

Разметка HTML – это просто список ссылок:

```
<p><span id="toggle-all">Toggle Checked</span></p>
<ol id="vids">
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2158073">
        Gravedigger</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--4472739">Save Me</a></li>
```

```
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--45286339">Crush</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2144530">
        Don't Drink The Water</a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--217241800">
        Funny the Way It Is </a></li>
    <li><input type="checkbox" checked><a
        href="http://new.music.yahoo.com/videos/--2144532">
        What Would You Say</a></li>
</ol>
```

Обработчики событий

Теперь рассмотрим обработчики событий. Сначала определим вспомогательную функцию с коротким именем `$` (для удобства):

```
var $ = function (id) {
    return document.getElementById(id);
};
```

Используя прием делегирования событий (подробнее об этом шаблоне рассказывается в главе 8), сосредоточим обработку всех щелчков мышью на элементах упорядоченного списка с атрибутом `id="vids"` в единственной функции:

```
$('vids').onclick = function (e) {
    var src, id;

    e = e || window.event;
    src = e.target || e.srcElement;

    if (src.nodeName !== "A") {
        return;
    }

    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    e.returnValue = false;

    id = src.href.split('--')[1];

    if (src.className === "play") {
        src.parentNode.innerHTML = videos.getPlayer(id);
        return;
    }

    src.parentNode.id = "v" + id;
    videos.getInfo(id);
};
```

В едином обработчике событий нам необходимо по-разному обрабатывать щелчки, вызывающие развертывание/свертывание области с дополнительной информацией (вызовом метода `getInfo()`) и запускающие воспроизведение фильма (когда целевой элемент разметки содержит класс CSS с именем «play»). В последнем случае известно, что область с дополнительной информацией уже развернута и можно вызвать метод `getPlayer()`. Идентификаторы видеофильмов извлекаются из адресов URL в атрибутах `href`.

Другой обработчик щелчков мышью вызывается в случае щелчка на элементе с атрибутом `id="toggle-all"` и выполняет развертывание/свертывание всех областей с дополнительной информацией. Фактически он точно так же вызывает метод `getInfo()`, но уже в цикле:

```
$('toggle-all').onclick = function (e) {

    var hrefs,
        i,
        max,
        id;

    hrefs = $('vids').getElementsByTagName('a');
    for (i = 0, max = hrefs.length; i < max; i += 1) {
        // пропустить ссылки "play"
        if (hrefs[i].className === "play") {
            continue;
        }
        // пропустить не выбранные элементы списка
        if (!hrefs[i].parentNode.firstChild.checked) {
            continue;
        }

        id = hrefs[i].href.split('--')[1];
        hrefs[i].parentNode.id = "v" + id;
        videos.getInfo(id);
    }
};
```

Объект videos

Объект `videos` обладает тремя методами:

`getPlayer()`

Возвращает разметку HTML, необходимую для проигрывания видеоролика Flash (не имеет отношения к нашей дискуссии).

`updateList()`

Функция обратного вызова. Ей передаются все данные, полученные от веб-службы, и она воспроизводит разметку HTML для развернутой области с дополнительной информацией. В этом методе тоже не происходит ничего интересного для нас.

`getInfo()`

Этот метод отвечает за отображение областей с дополнительной информацией и вызывает метод объекта `http`, передавая ему метод `updateList()` в качестве функции обратного вызова.

Ниже приводится фрагмент реализации объекта:

```
var videos = {

    getPlayer: function (id) {...},
    updateList: function (data) {...},

    getInfo: function (id) {

        var info = $('info' + id);

        if (!info) {
            http.makeRequest([id], "videos.updateList");
            return;
        }

        if (info.style.display === "none") {
            info.style.display = '';
        } else {
            info.style.display = 'none';
        }

    }
};
```

Объект http

Объект `http` имеет всего один метод, который отправляет запрос в формате JSONP веб-службе YQL компании Yahoo!:

```
var http = {
    makeRequest: function (ids, callback) {
        var url = 'http://query.yahooapis.com/v1/public/yql?q=',
            sql = 'select * from music.video.id where ids IN ("%ID%")',
            format = "format=json",
            handler = "callback=" + callback,
            script = document.createElement('script');

        sql = sql.replace('%ID%', ids.join('","'));
        sql = encodeURIComponent(sql);

        url += sql + '&' + format + '&' + handler;
        script.src = url;

        document.body.appendChild(script);
    }
};
```

YQL (Yahoo! Query Language – язык запросов Yahoo!) – это *мета* веб-служба, которая предоставляет возможность использовать SQL-подобный синтаксис для обращения к другим веб-службам без необходимости изучать прикладной интерфейс каждой из имеющихся веб-служб.

Когда пользователь запрашивает информацию по всем шести видеофильмам одновременно, веб-службе отправляется шесть самостоятельных запросов YQL, которые выглядят, как показано ниже:

```
select * from music.video.id where ids IN ("2158073")
```

Добавляем прокси-объект

Реализация, описанная выше, прекрасно справляется с возложенной на нее задачей, но можно сделать ее еще лучше. После появления на сцене объекта `proxy` он принимает на себя всю ответственность за взаимодействия между объектами `http` и `videos`. Он старается объединить запросы под управлением простой логики – с помощью буфера ожидания, в котором запрос хранится не более 50 миллисекунд. Объект `videos` обращается не к веб-службе непосредственно, а вызывает метод прокси-объекта. После этого прокси-объект ожидает в течение 50 миллисекунд и передает запрос дальше. Если в течение этого времени от объекта `videos` поступит новый запрос, он будет объединен с предыдущим. Задержка в 50 миллисекунд практически незаметна для пользователя, но она помогает объединить запросы и ускорить получение информации после щелчка на строке «Toggle Checked», когда необходимо развернуть сразу несколько областей с дополнительной информацией. Кроме того, это способствует существенному снижению нагрузки на сервер, так как в этом случае серверу приходится обрабатывать меньшее количество запросов.

Объединенный запрос YQL на получение информации о двух фильмах имеет следующий вид:

```
select * from music.video.id where ids IN ("2158073", "123456")
```

Новая версия программного кода содержит лишь одно изменение – метод `videos.getInfo()` теперь вызывает `proxy.makeRequest()`, а не `http.makeRequest()`, как показано ниже:

```
proxy.makeRequest(id, videos.updateList, videos);
```

Прокси-объект подготавливает очередь для хранения идентификаторов видеофильмов, принятых в течение последних 50 миллисекунд, и по истечении установленного интервала времени очищает очередь, вызывая метод объекта `http` и передавая ему собственную функцию обратного вызова, потому что функция `videos.updateList()` способна обрабатывать только одну запись с данными.

Реализация прокси-объекта приводится ниже:

```
var proxy = {
    ids: [],
    delay: 50,
    timeout: null,
    callback: null,
    context: null,
    makeRequest: function (id, callback, context) {

        // добавить в очередь
        this.ids.push(id);

        this.callback = callback;
        this.context = context;

        // установить предельное время ожидания
        if (!this.timeout) {
            this.timeout = setTimeout(function () {
                proxy.flush();
            }, this.delay);
        }
    },
    flush: function () {

        http.makeRequest(this.ids, "proxy.handler");

        // сбросить таймер и очистить очередь
        this.timeout = null;
        this.ids = [];

    },
    handler: function (data) {
        var i, max;

        // единственный видеофильм
        if (parseInt(data.query.count, 10) === 1) {
            proxy.callback.call(proxy.context, data.query.results.Video);
            return;
        }

        // несколько видеофильмов
        for (i = 0, max = data.query.results.Video.length; i < max; i += 1)
{
            proxy.callback.call(proxy.context, data.query.results.Video[i]);
        }
    }
};
```

Введение прокси-объекта обеспечило возможность объединения нескольких запросов к веб-службе в один запрос за счет незначительного изменения первоначального программного кода.

На рис. 7.4 и 7.5 иллюстрируются ситуации, когда серверу отправляются три запроса по отдельности (без прокси-объекта) и когда запросы объединены в один запрос за счет прокси-объекта.

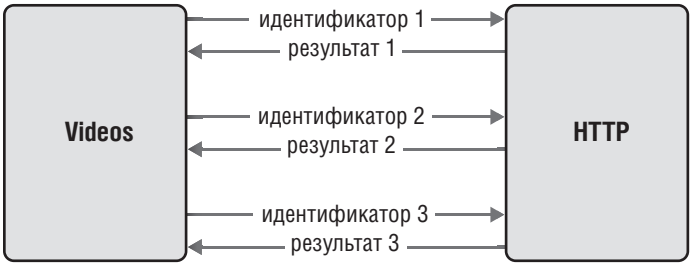

Рис. 7.4. *Три запроса к серверу*

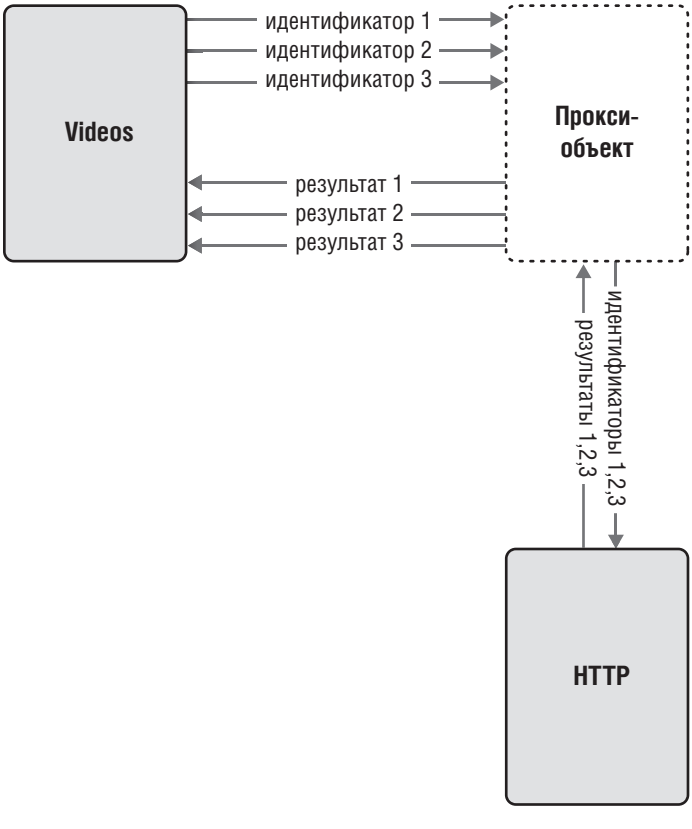

Рис. 7.5. *Использование прокси-объекта позволяет объединять запросы и уменьшать количество запросов к серверу*

Прокси-объект как кэш

В нашем примере клиентский объект (`videos`) достаточно интеллектуален, чтобы не запрашивать повторно информацию об одном и том же видеофильме. Но так бывает не всегда. Прокси-объект может взять на себя защиту реального объекта `http`, предусмотрев сохранение результатов предыдущих запросов в новом свойстве `cache` (рис. 7.6). В этом случае, если объект `videos` запросит информацию для одного и того же видеофильма второй раз, прокси-объект сможет извлечь ее из кэша и сэкономить время, необходимое на выполнение запроса по сети.

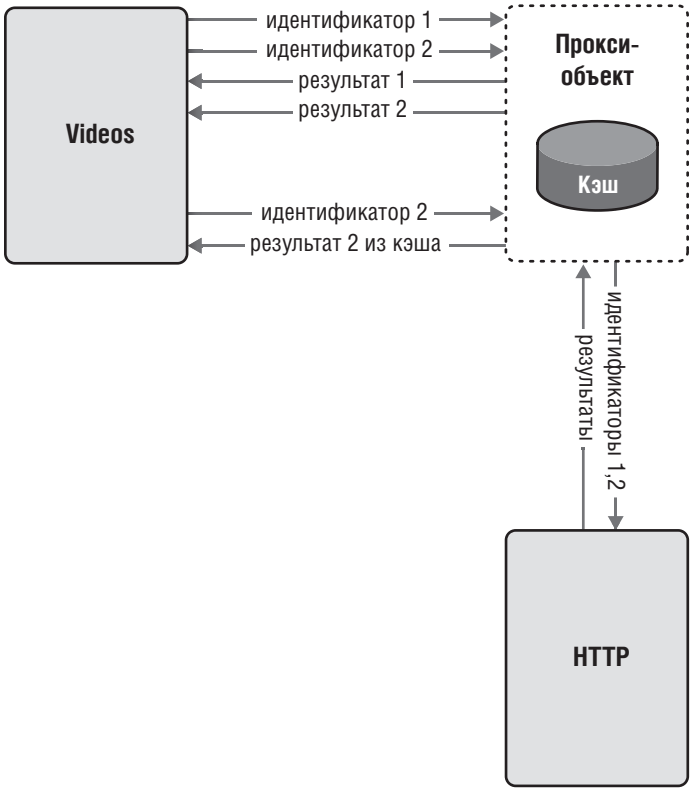

Рис. 7.6. *Кэш прокси-объекта*

Посредник

Приложения – большие и маленькие – состоят из отдельных объектов. Все эти объекты должны взаимодействовать друг с другом способом, который не усложнял бы сопровождение программного кода и позволял бы изменять одну часть приложения, не оказывая отрицательного влияния на другую. По мере развития приложения в него добавляется

все больше и больше объектов. Затем в процессе реорганизации объекты могут удаляться и перегруппировываться. Когда объекты знают друг о друге слишком много и взаимодействуют напрямую (вызывают методы друг друга и изменяют значения свойств), это приводит к образованию нежелательной *тесной связи* между ними. Когда объекты слишком тесно связаны, становится сложнее изменять их так, чтобы не оказать влияния на другие объекты. Тогда внесение даже простейших изменений превращается в нетривиальную задачу, и становится практически невозможным оценить время, необходимое на внесение изменений.

Применение шаблона посредника (mediator) упрощает подобные ситуации, способствуя *ослаблению связей* и помогая повысить удобство сопровождения (рис. 7.7). Использование этого шаблона исключает прямые взаимодействия между независимыми объектами (*коллегами*) за счет введения объекта-*посредника*. Когда какой-либо из объектов-коллег изменяет свое состояние, он извещает об этом посредника, а посредник сообщает об изменениях всем остальным коллегам, которые должны знать об этом.

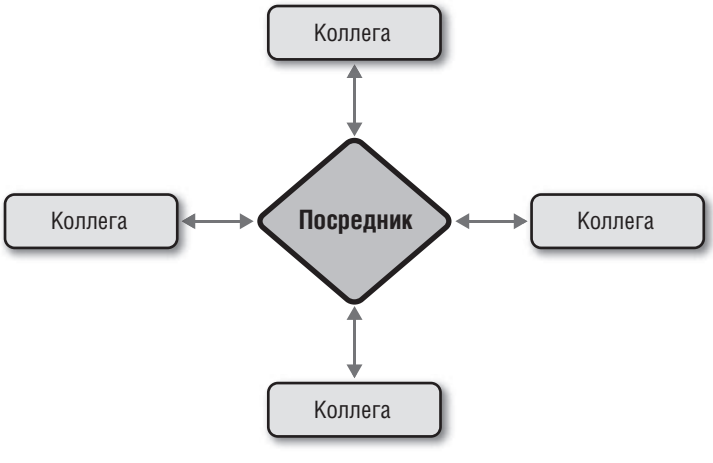

Рис. 7.7. *Действующие лица в шаблоне посредника*

Пример использования шаблона посредника

Рассмотрим пример использования шаблона посредника. В качестве приложения мы создадим игру, где двум игрокам дается полминуты, в течение которой они соревнуются – кто успеет больше раз нажать клавишу на клавиатуре. Первый игрок должен нажимать клавишу 1, а второй игрок – клавишу 0 (так им не придется драться за клавиатуру). Табло на экране будет отображать текущий счет.

В игре участвуют следующие объекты:

- Игрок 1
- Игрок 2
- Табло
- Посредник

Посредник знает о существовании всех остальных объектов. Он получает информацию от устройства ввода (клавиатура), обслуживает события нажатия клавиш, определяет, какой игрок нажал клавишу, и извещает его (рис. 7.8). Игрок делает ход (что означает увеличение его счета на единицу) и извещает об этом посредника. Посредник передает обновленный счет на табло, которое отображает его.

Кроме посредника ни один объект ничего не знает о существовании других объектов. Это существенно упрощает внесение изменений в игру, например, ничего не стоит добавить еще одного игрока или еще одно табло, отображающее время, оставшееся до конца игры.

Действующую версию игры и ее исходные тексты вы найдете по адресу *http://jspatterns.com/book/7/mediator.html*.

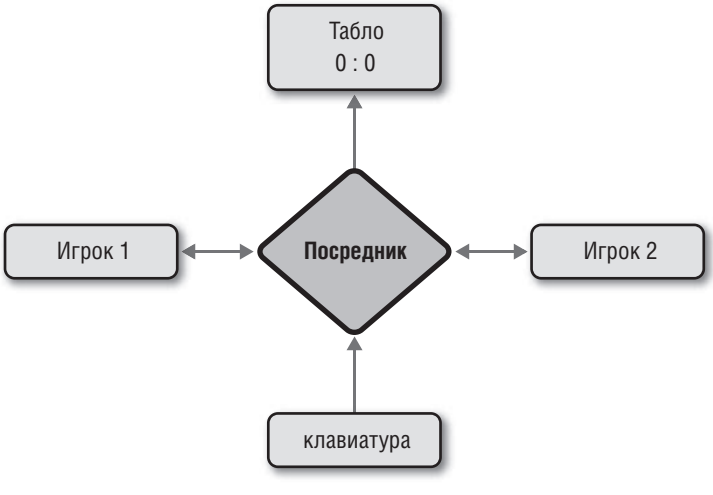

***Рис. 7.8.** Участники игры*

Объекты, представляющие игроков, создаются с помощью конструктора `Player()` и обладают собственными свойствами `points` и `name`. Метод `play()` прототипа увеличивает количество очков на единицу и затем извещает об этом посредника:

```
function Player(name) {
    this.points = 0;
```

```
    this.name = name;
}
Player.prototype.play = function () {
    this.points += 1;
    mediator.played();
};
```

Объект, представляющий табло, имеет метод `update()`, который вызывается объектом-посредником после получения каждого извещения от игроков. Табло ничего не знает об игроках и не сохраняет счет – этот объект просто отображает счет, полученный от посредника:

```
var scoreboard = {

    // элемент HTML, который должен обновляться
    element: document.getElementById('results'),

    // обновляет счет на экране
    update: function (score) {

        var i, msg = '';
        for (i in score) {
            if (score.hasOwnProperty(i)) {
                msg += '<p><strong>' + i + '<\/strong>: ';
                msg += score[i];
                msg += '<\/p>';
            }
        }
        this.element.innerHTML = msg;
    }
};
```

А теперь рассмотрим реализацию объекта-посредника `mediator`. Он инициализирует игру, создавая объекты, представляющие игроков, в методе `setup()`, и сохраняет ссылки на них в своем свойстве `players`. Метод `played()` вызывается объектами, представляющими игроков, после того как они сделают ход. Этот метод обновляет хеш `score` и передает его объекту табло `scoreboard` для отображения. Последний метод `keypress()` обрабатывает события от клавиатуры, определяет, какой игрок нажал клавишу, и извещает соответствующий объект:

```
var mediator = {

    // все игроки
    players: {},

    // инициализация
    setup: function () {
        var players = this.players;
        players.home = new Player('Home');
```

```
            players.guest = new Player('Guest');
        },

        // обновляет счет, если кто-то из игроков сделал ход
        played: function () {
            var players = this.players,
                score = {
                    Home:  players.home.points,
                    Guest: players.guest.points
                };

            scoreboard.update(score);
        },

        // обработчик действий пользователя
        keypress: function (e) {
            e = e || window.event; // IE
            if (e.which === 49) {  // key "1"
                mediator.players.home.play();
                return;
            }
            if (e.which === 48) {  // key "0"
                mediator.players.guest.play();
                return;
            }
        }
    };
```

И последнее – запуск и остановка игры:

```
    // Старт!
    mediator.setup();
    window.onkeypress = mediator.keypress;

    // Игра завершится через 30 секунд
    setTimeout(function () {
        window.onkeypress = null;
        alert('Game over!');
    }, 30000);
```

Наблюдатель

Шаблон *наблюдатель* (observer) широко применяется при разработке клиентских сценариев на JavaScript. Все события, возникающие в броузерах (`mouseover`, `keypress` и другие), являются примерами реализации этого шаблона. Этот шаблон иногда называют *механизмом собственных событий*, то есть событий, которые возбуждаются программно, в отличие от тех, что возбуждаются броузером. Еще одно название этого шаблона – *подписчик/издатель*.

Основная идея шаблона состоит в ослаблении связи между объектами. Вместо вызова одним объектом метода другого объекта некоторый объект подписывается на получение извещений об определенных событиях от другого объекта. Подписчик также называется наблюдателем, а объект, за которым ведется наблюдение, называется издателем, или объектом наблюдения. Издатель оповещает (вызывает) всех подписчиков о наступлении какого-то важного события, и часто сообщение передается им в форме объекта события.

Пример 1: подписка на журнал

Рассмотрим на конкретном примере, как реализовать этот шаблон. Допустим, что имеется некоторое издательство `paper`, которое выпускает ежедневную газету и ежемесячный журнал. Издательство будет уведомлять подписчика `joe` об этих событиях.

Объекту `paper` необходимо свойство `subscribers`, являющееся массивом, в котором будет храниться список подписчиков. Факт подписки оформляется простым занесением подписчика в этот массив. Когда происходит какое-либо событие, объект `paper` просматривает список подписчиков и отправляет им уведомления. Под уведомлением понимается вызов метода объекта-подписчика. Это означает, что при оформлении подписки подписчик передает ссылку на один из своих методов методу `subscribe()` объекта `paper`.

Кроме того, объект `paper` предоставляет метод `unsubscribe()`, удаляющий подписчика из списка. Последний важный метод объекта `paper` – метод `publish()`, который будет вызывать методы подписчиков. Подводя итоги, объекту-издателю потребуются следующие члены:

`subscribers`

Массив.

`subscribe()`

Добавляет подписчика в массив.

`unsubscribe()`

Удаляет подписчика из массива.

`publish()`

Просматривает список подписчиков и вызывает методы, указанные при оформлении подписки.

Всем трем методам необходимо передавать параметр `type`, потому что издатель может возбуждать различные события (выпускать журнал и газету), а подписчики могут предпочесть подписаться только на одно из них.

Так как эти члены являются универсальными для любого объекта-издателя, имеет смысл реализовать их как часть отдельного объекта.

После этого мы сможем добавлять их (вспомните шаблон смешивания) в любые объекты и превращать их в объекты-издатели.

Ниже приводится пример реализации универсальных методов объектов-издателей, где определяются все необходимые члены, перечисленные выше, а также дополнительный вспомогательный метод `visitSubscribers()`:

```
var publisher = {
    subscribers: {
        any: []                // тип события: подписчик
    },
    subscribe: function (fn, type) {
        type = type || 'any';
        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push(fn);
    },
    unsubscribe: function (fn, type) {
        this.visitSubscribers('unsubscribe', fn, type);
    },
    publish: function (publication, type) {
        this.visitSubscribers('publish', publication, type);
    },
    visitSubscribers: function (action, arg, type) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
            i,
            max = subscribers.length;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i](arg);
            } else {
                if (subscribers[i] === arg) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

Далее приводится функция, которая принимает объект и превращает его в объект-издатель, добавляя в него универсальные методы издателя:

```
function makePublisher(o) {
    var i;
    for (i in publisher) {
        if (publisher.hasOwnProperty(i) && typeof publisher[i]==="function")
{
```

```
            o[i] = publisher[i];
        }
    }
    o.subscribers = {any: []};
}
```

Теперь реализуем объект paper. Все, что он может делать, – это издавать ежедневную газету и ежемесячный журнал:

```
var paper = {
    daily: function () {
        this.publish("big news today");
    },
    monthly: function () {
        this.publish("interesting analysis", "monthly");
    }
};
```

Создадим объект paper:

```
makePublisher(paper);
```

Теперь, когда у нас готов объект-издатель, можно перейти к реализации объекта-подписчика joe, который обладает двумя методами:

```
var joe = {
    drinkCoffee: function (paper) {
        console.log('Just read ' + paper);
    },
    sundayPreNap: function (monthly) {
        console.log('About to fall asleep reading this ' + monthly);
    }
};
```

Далее paper подписывает joe (то есть joe подписывается в издательстве paper):

```
paper.subscribe(joe.drinkCoffee);
paper.subscribe(joe.sundayPreNap, 'monthly');
```

Как видите, joe указывает один метод, который должен вызываться по любому («any») событию, и другой метод, который должен вызываться по событию издания ежемесячного журнала («monthly»). Теперь попробуем возбудить несколько событий:

```
paper.daily();
paper.daily();
paper.daily();
paper.monthly();
```

Все эти публикации приведут к вызову соответствующих методов объекта joe, в результате чего в консоли появятся следующие строки:

```
Just read big news today
Just read big news today
Just read big news today
About to fall asleep reading this interesting analysis
```

Самое интересное здесь заключается в том, что объекту `paper` заранее ничего неизвестно об объекте `joe`, а объекту `joe` – об объекте `paper`. Здесь нет также никакого объекта-посредника, который знает о существовании всех объектов. Объекты, участвующие во взаимодействиях, слабо связаны между собой, и ничего не меняя в них, мы можем добавить в объект `paper` любое количество подписчиков; кроме того, объект `joe` в любой момент может аннулировать подписку.

Давайте разовьем этот пример еще дальше и превратим объект `joe` в объект-издатель. (В конце концов, в блогах и микроблогах любой может быть издателем.) Благодаря превращению объекта `joe` в объект-издатель он сможет посылать информацию об изменении своего состояния в микроблог:

```
makePublisher(joe);
joe.tweet = function (msg) {
    this.publish(msg);
};
```

Теперь представьте себе, что в отделе по связям с общественностью, принадлежащем издательству, было решено постоянно знакомиться с тем, что пишут их читатели в микроблогах, и подписаться на события объекта `joe`, указав свой метод `readTweets()`:

```
paper.readTweets = function (tweet) {
    alert('Call big meeting! Someone ' + tweet);
};
joe.subscribe(paper.readTweets);
```

Теперь, как только `joe` что-то напишет в своем микроблоге, издательство `paper` тут же получит уведомление:

```
joe.tweet("hated the paper today");
```

То есть издательство получит такое уведомление: «Call big meeting! Someone hated the paper today».

Получить полные исходные тексты этого примера, а также поэкспериментировать с ним вы сможете, обратившись по адресу *http://jspatterns.com/book/7/observer.html*.

Пример 2: игра на нажатие клавиш

Рассмотрим еще один пример. Мы реализуем игру на нажатие клавиш, которую использовали в качестве примера при обсуждении шаблона посредника, но на этот раз мы применим шаблон наблюдателя. Чтобы

несколько усовершенствовать ее, будем считать, что число игроков неограничено. В этом примере по-прежнему имеется конструктор `Player()`, с помощью которого создаются объекты, представляющие игроков, и объект `scoreboard`, представляющий табло. Только объект `mediator` теперь превратился в объект `game`.

В реализации на основе шаблона посредника объект `mediator` заранее знает о существовании всех объектов-участников и вызывает их методы. Объект `game` в реализации на основе шаблона наблюдателя ничего этого не делает; вместо этого он позволяет объектам подписываться на интересующие их события. Например, объект `scoreboard` подпишется в объекте `game` на событие «scorechange».

Для начала вернемся к объекту, реализующему универсальные методы издателя, и немного модифицируем его интерфейс, чтобы сделать его немного ближе к миру броузеров:

- Методы `publish()`, `subscribe()` и `unsubscribe()` мы переименуем в `fire()`, `on()` и `remove()`.
- Тип события `type` будет использоваться повсеместно, поэтому мы сделаем его первым аргументом этих трех функций.
- Помимо функций подписчики смогут определять дополнительный контекст, чтобы обеспечить возможность использовать методы подписчиков в качестве функций обратного вызова, передавая им ссылку `this`, указывающую на сам объект.

Новая версия объекта `publisher` теперь будет выглядеть так:

```
var publisher = {
    subscribers: {
        any: []
    },
    on: function (type, fn, context) {
        type = type || 'any';
        fn = typeof fn === "function" ? fn : context[fn];

        if (typeof this.subscribers[type] === "undefined") {
            this.subscribers[type] = [];
        }
        this.subscribers[type].push({fn: fn, context: context || this});
    },
    remove: function (type, fn, context) {
        this.visitSubscribers('unsubscribe', type, fn, context);
    },
    fire: function (type, publication) {
        this.visitSubscribers('publish', type, publication);
    },
    visitSubscribers: function (action, type, arg, context) {
        var pubtype = type || 'any',
            subscribers = this.subscribers[pubtype],
```

```
            i,
            max = subscribers ? subscribers.length : 0;

        for (i = 0; i < max; i += 1) {
            if (action === 'publish') {
                subscribers[i].fn.call(subscribers[i].context, arg);
            } else {
                if (subscribers[i].fn === arg &&
                    subscribers[i].context === context) {
                    subscribers.splice(i, 1);
                }
            }
        }
    }
};
```

Новая версия конструктора `Player()` теперь будет выглядеть так:

```
function Player(name, key) {
    this.points = 0;
    this.name = name;
    this.key = key;
    this.fire('newplayer', this);
}

Player.prototype.play = function () {
    this.points += 1;
    this.fire('play', this);
};
```

Новым здесь является параметр конструктора `key`, определяющий клавишу на клавиатуре, которую должен нажать игрок, чтобы получить одно очко. (Прежде клавиши определялись заранее.) Кроме того, при каждом создании нового объекта, представляющего игрока, возбуждается событие «newplayer». Аналогично, когда кто-то из игроков делает ход, возбуждается событие «play».

Объект `scoreboard` остается без изменений – он просто обновляет счет на экране.

Новый объект `game` способен следить за всеми игроками, поэтому он объявляет счет и возбуждает событие «scorechange». Он также будет подписываться на все события «keypress», возбуждаемые броузером, и имеет полную информацию о том, какая клавиша какому игроку соответствует:

```
var game = {

    keys: {},

    addPlayer: function (player) {
```

```
            var key = player.key.toString().charCodeAt(0);
            this.keys[key] = player;
        },

        handleKeypress: function (e) {
            e = e || window.event; // IE
            if (game.keys[e.which]) {
                game.keys[e.which].play();
            }
        },

        handlePlay: function (player) {
            var i,
                players = this.keys,
                score = {};

            for (i in players) {
                if (players.hasOwnProperty(i)) {
                    score[players[i].name] = players[i].points;
                }
            }
            this.fire('scorechange', score);
        }
    };
```

Функция `makePublisher()`, превращающая любой объект в издателя, осталась без изменений. Объект `game` превращается в издателя (чтобы иметь возможность возбуждать событие «scorechange»). Точно так же в издателя превращается `Player.protoype`, чтобы каждый объект, представляющий игрока, мог возбуждать события «play» и «newplayer», на получение которых могут подписаться все, кому они будут интересны:

```
makePublisher(Player.prototype);
makePublisher(game);
```

Объект `game` подписывается на события «play» и «newplayer» (а также на событие «keypress» от броузера), а объект `scoreboard` подписывается на событие «scorechange»:

```
Player.prototype.on("newplayer", "addPlayer",  game);
Player.prototype.on("play",      "handlePlay", game);
game.on("scorechange", scoreboard.update, scoreboard);
window.onkeypress = game.handleKeypress;
```

Как видно из этого фрагмента, метод `on()` позволяет подписчикам указывать функцию обратного вызова не только в виде ссылки на функцию (`scoreboard.update`), но и в виде строки (`"addPlayer"`). При использовании строковой формы необходимо дополнительно определять контекст вызова (например, `game`).

Последний этап в подготовке к игре – динамическое создание произвольного количества объектов, представляющих игроков (с клавишами, которые они должны нажимать):

```
var playername, key;
while (1) {
    playername = prompt("Add player (name)");
    if (!playername) {
        break;
    }
    while (1) {
        key = prompt("Key for " + playername + "?");
        if (key) {
            break;
        }
    }
    new Player(playername, key);
}
```

Вот и все, что нужно для этого варианта игры. Полные исходные тексты игры вы найдете на странице *http://jspatterns.com/book/7/observer-game.html* и там же сможете поиграть в нее.

Обратите внимание, что в реализации шаблона посредника объект-посредник должен знать о существовании всех остальных объектов, чтобы вызывать нужные методы в нужное время и координировать всю игру. В данном примере объект `game` менее интеллектуальный; он полагается на то, что другие объекты ожидают получения извещений о событиях и предпринимают нужные действия: например, объект табло `scoreboard` прослушивает событие «scorechange». В результате связь между объектами ослабляется еще больше (чем меньше объект знает, тем лучше), но при этом усложняется слежение за тем, кто и какие события ожидает. В этом примере игры подписка всех объектов на получение событий выполняется в одном месте, но с ростом размеров приложения вызовы метода `on()` могут оказаться в самых разных местах (например, в программном коде инициализации каждого отдельного объекта). Это усложнит отладку, так как в этом случае не будет единственного места, куда можно будет заглянуть и понять, что происходит. Применяя шаблон наблюдателя, вы уходите от последовательного выполнения программного кода, при котором вы просматриваете сценарий от начала и до конца программы.

В заключение

В этой главе вы познакомились с несколькими популярными шаблонами проектирования и увидели, как можно реализовать их на языке JavaScript. Ниже перечислены шаблоны проектирования, которые обсуждались здесь:

Единственный объект

Обеспечивает возможность создания только одного экземпляра «класса». Мы рассмотрели несколько подходов к реализации шаблона на тот случай, если вам потребуется имитировать классы с помощью функций-конструкторов и обеспечить возможность использования Java-подобного синтаксиса. При этом технически все объекты в JavaScript являются единственными. А кроме того, иногда разработчики под термином «singleton» подразумевают объекты, созданные с применением шаблона «модуль».

Фабрика

Метод, создающий объекты, типы которых определяются в виде строки во время выполнения.

Итератор

Определение прикладного интерфейса, позволяющего выполнять обход сложных структур данных.

Декоратор

Расширение возможностей объектов во время выполнения за счет добавления в них функциональных возможностей, определяемых объектами-декораторами.

Стратегия

При сохранении прежнего интерфейса обеспечивает выбор наилучшей стратегии для решения определенных задач.

Фасад

Предоставление более удобного прикладного интерфейса за счет создания новых методов, обертывающих часто используемые (или неудачно спроектированные) методы.

Прокси-объект

Обертывание объекта с целью управления доступом к нему и предотвращения выполнения лишних дорогостоящих операций за счет их объединения или выполнения только тогда, когда они действительно необходимы.

Посредник

Способствует ослаблению связей между объектами за счет отказа от прямых взаимодействий между ними и введения объекта-посредника, берущего на себя роль управления взаимодействиями.

Наблюдатель

Способствует ослаблению связей между объектами за счет создания «объектов наблюдения», которые посылают извещения всем объектам, наблюдающим за ними, когда происходит какое-то интересное событие (этот шаблон также называется подписчик/издатель или «механизм собственных событий»).

8

Шаблоны для работы с деревом DOM и броузерами

В предыдущих главах книги больше внимания уделялось базовым возможностям языка JavaScript (ECMAScript), чем шаблонам использования JavaScript в броузерах. В этой главе, напротив, будут исследоваться шаблоны, предназначенные для использования в броузерах, потому что броузеры являются наиболее типичной средой выполнения программ на языке JavaScript. Когда люди говорят, что им не нравится JavaScript, чаще всего они подразумевают разработку сценариев для броузеров. И это вполне понятно, если учесть массу несовместимостей основных объектов и реализаций DOM в броузерах. Совершенно очевидно, что будут полезны любые приемы программирования, способные сделать разработку клиентских сценариев менее трудоемкой.

В этой главе шаблоны разделены на несколько категорий, включая манипулирование деревом DOM, обработка событий, удаленные взаимодействия, стратегии загрузки сценариев JavaScript и приемы развертывания приложений JavaScript на действующих веб-сайтах.

Но сначала подведем вкратце теоретическую базу под тему разработки клиентских сценариев.

Разделение на составные части

Разработка веб-приложения состоит из решения следующих трех задач:

Информационное наполнение

Документ HTML.

Представление

Стили CSS, определяющие внешний вид документа.

Поведение

Сценарии JavaScript, обеспечивающие взаимодействие с пользователем и реализующие динамические изменения в документе.

Стремление решать эти задачи максимально независимо друг от друга повышает доступность приложения для работы с самыми разнообразными пользовательскими агентами – броузерами с графическим интерфейсом, текстовыми броузерами, вспомогательными устройствами для лиц с ограниченными возможностями, мобильными устройствами и так далее. Кроме того, *разделение на составные части* непосредственно связано с воплощением идеи *последовательного улучшения* – вы начинаете с базовой реализации (включающей только разметку HTML), которая сможет восприниматься простейшими пользовательскими агентами, и добавляете в нее все более широкие возможности, которые поддерживаются более совершенными пользовательскими агентами. Если броузер поддерживает CSS, следовательно, пользователь увидит документ в более привлекательном оформлении. Если броузер поддерживает JavaScript, то документ станет больше похожим на полноценное приложение, обладающее еще более широкими возможностями.

На практике разделение на составные части влечет за собой:

- Тестирование страницы с отключенными стилями CSS, чтобы увидеть, пригодна ли страница к использованию в таком виде и насколько удобочитаемым выглядит ее содержимое.
- Тестирование страницы с отключенной поддержкой JavaScript, чтобы убедиться, что страница позволяет решать задачи, стоящие перед пользователем, все ссылки действуют (отсутствуют ссылки с атрибутом `href="#"`) и все формы позволяют вводить и отправлять данные.
- Отказ от использования встроенных обработчиков событий (таких как `onclick`) или встроенных атрибутов `style`, так как они не принадлежат к уровню информационного наполнения.
- Использование семантически значимых элементов HTML, таких как заголовки и списки.

Уровень JavaScript (поведение) должен оформляться в *ненавязчивом* стиле, то есть он не должен быть единственным способом решения задач, стоящих перед пользователем. При отображении в броузерах, не поддерживающих JavaScript, страница должна оставаться работоспособной, а поддержка JavaScript не должна быть обязательным условием. Сценарии JavaScript должны лишь расширять возможности страницы.

Для решения проблем, связанных с различиями между броузерами, часто используется прием *определения поддерживаемых возможностей*. Согласно ему вы не должны полагаться на определение типа пользовательского агента, а должны проверять наличие в текущем окружении свойств и методов, которые собираетесь использовать. Вообще говоря, прием определения типа броузера расценивается как антишаблон. Бы-

вают ситуации, когда без этого не обойтись, однако этот прием следует рассматривать как крайнюю меру и использовать только в случаях, когда для принятия решения невозможно применить прием определения поддерживаемых возможностей (или это приводит к существенному снижению производительности):

```
// антишаблон
if (navigator.userAgent.indexOf('MSIE') !== -1) {
    document.attachEvent('onclick', console.log);
}

// предпочтительнее
if (document.attachEvent) {
   document.attachEvent('onclick', console.log);
}

// или более конкретно
if (typeof document.attachEvent !== "undefined") {
    document.attachEvent('onclick', console.log);
}
```

Кроме того, разделение на составные части помогает упростить разработку, сопровождение и обновление существующих веб-приложений, потому что вы знаете, где искать проблему, если что-то пошло не так. Если возникает ошибка JavaScript, вам нет необходимости просматривать файлы HTML или CSS, чтобы исправить ее.

Работа с деревом DOM

Работа с деревом DOM страницы – одна из наиболее типичных задач клиентских сценариев на JavaScript. Это также основной источник проблем (и дурной славы JavaScript), потому что методы DOM в разных броузерах реализованы по-разному. Именно поэтому использование хорошей библиотеки JavaScript, скрывающей различия между броузерами, может существенно ускорить разработку.

Рассмотрим некоторые шаблоны, рекомендуемые для доступа и модификации дерева DOM, основное внимание в которых уделяется высокой производительности.

Доступ к дереву DOM

Операции доступа к дереву DOM являются достаточно дорогостоящими; это самое узкое место сценариев JavaScript с точки зрения производительности. Обусловлено это тем, что дерево DOM обычно реализовано отдельно от механизма JavaScript. С точки зрения броузера в этом есть определенный смысл, так как приложению на JavaScript может вообще не потребоваться обращаться к дереву DOM. А кроме того, для работы с деревом DOM страницы могут использоваться другие языки программирования, отличные от JavaScript (например, VBScript в IE).

Таким образом, необходимо стремиться свести к минимуму операции обращения к дереву DOM. Это означает, что:

- Следует избегать обращений к элементам DOM внутри циклов
- Желательно присваивать ссылки на элементы DOM локальным переменным и работать с этими переменными
- Следует использовать интерфейс селекторов, где это возможно
- Следует сохранять значение свойства `length` в локальной переменной при выполнении итераций через коллекции HTML (об этом уже упоминалось в главе 2)

Взгляните на следующий пример, где второй цикл, несмотря на больший размер, оказывается в десятки, а то и в сотни раз быстрее первого в зависимости от броузера:

```
// антишаблон
for (var i = 0; i < 100; i += 1) {
    document.getElementById("result").innerHTML += i + ", ";
}

// предпочтительнее - изменяет значение локальной переменной
var i, content = "";
for (i = 0; i < 100; i += 1) {
    content += i + ",";
}
document.getElementById("result").innerHTML += content;
```

В следующем фрагменте второй пример (использующий локальную переменную) является более предпочтительным, хотя он содержит на одну строку программного кода больше и использует дополнительную переменную:

```
// антишаблон
var padding = document.getElementById("result").style.padding,
    margin  = document.getElementById("result").style.margin;

// предпочтительнее
var style   = document.getElementById("result").style,
    padding = style.padding,
    margin  = style.margin;
```

Под использованием интерфейса селекторов подразумевается применение методов:

```
document.querySelector("ul .selected");
document.querySelectorAll("#widget .class");
```

Эти методы принимают строки селекторов CSS и возвращают списки узлов DOM, соответствующих указанным критериям отбора. Методы селекторов доступны в современных броузерах (и в IE начиная с версии 8) и всегда действуют быстрее, чем любая другая реализация отбо-

ра узлов, основанная на применении других методов DOM. Последние версии популярных библиотек JavaScript используют преимущества интерфейса селекторов, поэтому обязательно убедитесь, что вы пользуетесь самой свежей версией своей любимой библиотеки.

Кроме того, добавляйте атрибут `id=""` в элементы, к которым приходится обращаться чаще всего, потому что это обеспечит возможность простого и быстрого их поиска с помощью метода `document.getElementById(myid)`.

Манипулирование деревом DOM

Помимо обращений к элементам DOM, вам часто будет требоваться изменять их, удалять или добавлять новые элементы. Внесение изменений в дерево DOM может заставлять броузер перерисовывать страницу, а также *перераспределять* элементы (пересчитывать их координаты и размеры), что может оказаться весьма дорогостоящей операцией.

Здесь снова действует основной принцип – стараться свести к минимуму количество операций, модифицирующих дерево DOM, что означает накапливать изменения, выполняя их за пределами «живого» дерева DOM документа.

Если вам необходимо создать относительно крупную ветвь дерева, вы должны стараться не добавлять ее в дерево, пока она не будет полностью готова. Для этой цели можно создать *фрагмент документа*, содержащий необходимые вам узлы.

Ниже показано, как *не следует* добавлять узлы:

```
// антишаблон
// узлы добавляются в дерево по мере их создания

var p, t;

p = document.createElement('p');
t = document.createTextNode('first paragraph');
p.appendChild(t);
document.body.appendChild(p);

p = document.createElement('p');
t = document.createTextNode('second paragraph');
p.appendChild(t);
document.body.appendChild(p);
```

Гораздо предпочтительнее создать фрагмент документа, изменить его в «автономном режиме» и добавить в дерево DOM, когда он будет полностью готов. При его добавлении в дерево DOM будет добавлено содержимое фрагмента, а не сам фрагмент. И это действительно очень удобно. Итак, фрагмент документа является отличным способом обернуть множество узлов, даже когда они не находятся внутри общего родительского элемента (например, когда абзацы не находятся внутри элемента `div`).

Ниже приводится пример использования фрагмента документа:

```
var p, t, frag;

frag = document.createDocumentFragment();

p = document.createElement('p');
t = document.createTextNode('first paragraph');
p.appendChild(t);
frag.appendChild(p);

p = document.createElement('p');
t = document.createTextNode('second paragraph');
p.appendChild(t);
frag.appendChild(p);

document.body.appendChild(frag);
```

В этом примере документ изменяется только один раз, вызывая единственную операцию перерисовывания страницы, тогда как предыдущий пример антишаблона вызывает перерисовывание страницы при добавлении каждого абзаца.

Фрагмент документа удобно использовать при *добавлении* новых узлов в дерево. Однако при *изменении* существующих узлов дерева также следует стремиться свести количество изменений к минимуму. Для этого можно создать копию требуемой ветви дерева, выполнить необходимые изменения в копии и, когда она будет готова, заменить ею оригинал:

```
var oldnode = document.getElementById('result'),
    clone = oldnode.cloneNode(true);

// выполнить операции над копией...

// когда копия будет готова:
oldnode.parentNode.replaceChild(clone, oldnode);
```

События

Другой областью, полной несовместимостей между разными броузерами и способной приводить в отчаяние, является работа с событиями, такими как `click`, `mouseover` и так далее. И снова библиотеки JavaScript помогут избежать выполнения двойной работы для поддержания IE (до версии 9) и реализаций, соответствующих требованиям консорциума W3C.

А теперь рассмотрим основные этапы; это может оказаться полезным для тех, у кого нет возможности использовать существующие библиотеки в простых страницах и в решениях на скорую руку, а также для тех, кто создает собственную библиотеку.

Обработка событий

Обработка событий начинается с подключения обработчиков к элементам. Представьте, что у вас имеется кнопка, щелчок на которой должен приводить к увеличению счетчика на единицу. Можно было бы добавить встроенный обработчик события в виде атрибута `onclick`, который будет одинаково действовать во всех броузерах, но это нарушает принцип разделения на составные части и принцип последовательного улучшения. Поэтому вы должны подключать обработчики на JavaScript за пределами разметки.

Допустим, имеется следующая разметка:

```
<button id="clickme">Click me: 0</button>
```

Можно присвоить функцию свойству `onclick` узла, но только один раз:

```
// не самое оптимальное решение
var b = document.getElementById('clickme'),
    count = 0;
b.onclick = function () {
    count += 1;
    b.innerHTML = "Click me: " + count;
};
```

Если потребуется иметь несколько функций, вызываемых по событию щелчка мышью, вы не сможете в рамках этого шаблона подключить их, при этом обеспечивая слабую связь между ними. Технически можно выполнить проверку наличия обработчика события `onclick`, добавить его вызов в свою функцию и заменить значение свойства `onclick` ссылкой на свою функцию. Однако существует гораздо более простое решение, основанное на использовании метода `addEventListener()`. Данный метод отсутствует в IE вплоть до версии 8, поэтому в этих броузерах необходимо использовать метод `attachEvent()`.

При обсуждении шаблона выделения ветвей, выполняющихся на этапе инициализации (глава 4), мы рассматривали пример реализации утилиты, выполняющей подключение обработчиков событий в броузерах любых типов. Поэтому не будем вдаваться в подробности, а просто подключим обработчик к нашей кнопке:

```
var b = document.getElementById('clickme');
if (document.addEventListener) {        // W3C
    b.addEventListener('click', myHandler, false);
} else if (document.attachEvent) {      // IE
    b.attachEvent('onclick', myHandler);
} else {                                // крайний случай
    b.onclick = myHandler;
}
```

Теперь, когда пользователь щелкнет мышью на кнопке, будет вызвана функция `myHandler()`. А сейчас определим функцию, которая будет выводить увеличенное значение счетчика в виде надписи на кнопке «Click me: 0». Чтобы сделать задачу более интересной, предположим, что имеется несколько кнопок и для всех используется общий обработчик `myHandler()`. Сохранять ссылки на все кнопки и счетчики для каждой из них будет неэффективно, учитывая, что необходимую информацию можно получить из объекта события, который создается при каждом щелчке.

Давайте сначала приведем решение, а потом прокомментируем его:

```
function myHandler(e) {

    var src, parts;

    // получить объект события и элемент-источник
    e = e || window.event;
    src = e.target || e.srcElement;

    // фактическое решение: обновить надпись
    parts = src.innerHTML.split(": ");
    parts[1] = parseInt(parts[1], 10) + 1;
    src.innerHTML = parts[0] + ": " + parts[1];

    // предотвратить дальнейшее всплытие события
    if (typeof e.stopPropagation === "function") {
        e.stopPropagation();
    }
    if (typeof e.cancelBubble !== "undefined") {
        e.cancelBubble = true;
    }

    // предотвратить выполнение действия по умолчанию
    if (typeof e.preventDefault === "function") {
        e.preventDefault();
    }
    if (typeof e.returnValue !== "undefined") {
        e.returnValue = false;
    }

}
```

Действующий пример доступен по адресу *http://jspatterns.com/book/8/click.html*.

Функция обработки события состоит из четырех частей:

- В первой части мы получаем доступ к объекту события, содержащему информацию о событии и ссылку на элемент, вызвавший это событие. Объект события передается, когда обработчик подключается

как функция обратного вызова, но не в случае присваивания функции свойству `onclick` – в этой ситуации объект события доступен как свойство `window.event` глобального объекта.

- Вторая часть является фактическим решением поставленной задачи – изменяет текст надписи.
- Далее предотвращается дальнейшее распространение события. В данном конкретном примере этот шаг можно было бы опустить, но в общем случае, если этого не сделать, событие продолжит всплытие вверх по дереву документа и может достигнуть даже объекта `window`. И снова предотвратить всплытие можно двумя разными способами: стандартным – в броузерах, следующих стандартам W3C (вызвав метод `stopPropagation()`), и специальным – в IE (установив свойство `cancelBubble` в значение `true`).
- В последней части предотвращается выполнение действия по умолчанию, если это необходимо. Некоторые события (такие как щелчок на ссылке или на кнопке отправки формы) вызывают выполнение действия, предусмотренного по умолчанию, но вы можете предотвратить их, вызвав метод `preventDefault()` (или для IE, установив свойство `returnValue` в значение `false`).

Как видите, приходится выполнять немало двойной работы, поэтому есть смысл написать свою утилиту с фасадными методами для использования в обработчиках событий, как обсуждалось в главе 7.

Делегирование событий

Шаблон делегирования событий позволяет использовать преимущества, которые дает всплытие событий, и уменьшить количество обработчиков событий, подключенных к отдельным узлам. Например, если внутри элемента `div` имеется 10 кнопок, можно подключить единственный обработчик событий к элементу `div` вместо обработчика для каждой кнопки.

Рассмотрим пример с тремя кнопками внутри элемента `div` (рис. 8.1). Действующий пример реализации шаблона делегирования событий доступен по адресу *http://jspatterns.com/book/8/click-delegate.html*.

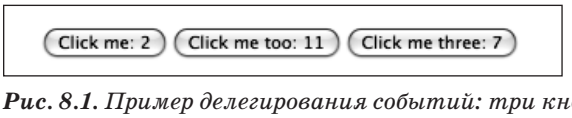

Рис. 8.1. *Пример делегирования событий: три кнопки, увеличивающие значения в своих надписях по щелчку*

Итак, у нас имеется следующая разметка:

```
<div id="click-wrap">
  <button>Click me: 0</button>
```

```
    <button>Click me too: 0</button>
    <button>Click me three: 0</button>
</div>
```

Вместо того чтобы подключать обработчики к каждой из кнопок, мы подключим один общий обработчик к обертывающему их элементу `div`. В качестве обработчика можно использовать функцию `myHandler()` из предыдущего примера с одним небольшим изменением: необходимо отфильтровать щелчки, которые нас не интересуют. В данном случае интерес для нас представляют только щелчки на кнопках, поэтому все остальные щелчки, выполненные в пределах элемента `div`, мы будем игнорировать.

Для этого в функции `myHandler()` можно было бы проверить, совпадает ли значение свойства `nodeName` источника события со строкой «button»:

```
// ...
// получить объект события и элемент-источник
e = e || window.event;
src = e.target || e.srcElement;

if (src.nodeName.toLowerCase() !== "button") {
    return;
}
// ...
```

Недостаток шаблона делегирования событий заключается в том, что для его реализации требуется написать дополнительный программный код, отфильтровывающий события, происходящие в контейнере и не представляющие интереса для нас. Но его преимущества – более высокая производительность и ясность программного кода – перевешивают недостатки, поэтому этот шаблон настоятельно рекомендуется к использованию.

Современные библиотеки JavaScript упрощают прием делегирования событий, предоставляя удобные прикладные интерфейсы. Например, в библиотеке YUI3 имеется метод `Y.delegate()`, который позволяет определить селектор CSS, соответствующий обертывающему элементу, и другой селектор, соответствующий узлам, представляющим интерес. Удобство заключается в том, что ваша функция-обработчик события никогда не будет вызываться для обработки событий, возникших за пределами интересующих вас узлов. В данном случае подключить обработчик событий можно было бы следующей простой инструкцией:

```
Y.delegate('click', myHandler, "#click-wrap", "button");
```

А так как библиотека YUI берет на себя сокрытие различий между броузерами и автоматически определяет источник события, реализацию обработчика можно было бы значительно упростить:

```
function myHandler(e) {

    var src = e.currentTarget,
        parts;

    parts = src.get('innerHTML').split(": ");
    parts[1] = parseInt(parts[1], 10) + 1;
    src.set('innerHTML', parts[0] + ": " + parts[1]);

    e.halt();
}
```

Действующий пример доступен по адресу *http://jspatterns.com/book/8/click-y-delegate.html*.

Сценарии, работающие продолжительное время

Вы наверняка обращали внимание, что иногда броузер предупреждает, что сценарий выполняется слишком долго, и предлагает остановить его. Никакому разработчику не хотелось бы, чтобы подобное происходило с его приложением независимо от того, насколько сложную задачу оно решает.

Кроме того, когда сценарий интенсивно работает, интерфейс броузера перестает откликаться на действия пользователя. Это производит отрицательное впечатление на пользователя, и таких ситуаций следует избегать.

В JavaScript нет возможности реализовать многопоточное выполнение программ, но его можно имитировать с помощью метода `setTimeout()` или с помощью механизма фоновых вычислений (web workers), реализованного в современных броузерах.

setTimeout()

Идея использования метода `setTimeout()` состоит в том, чтобы разбить большой объем работы на маленькие фрагменты и выполнять фрагменты с интервалом 1 мс. Использование столь коротких интервалов времени может замедлить решение всей задачи в целом, но пользовательский интерфейс при этом сохранит отзывчивость и пользователь будет чувствовать себя более комфортно.

Назначенный тайм-аут в 1 миллисекунду (или даже 0 миллисекунд) в реальности окажется больше в зависимости от броузера и операционной системы. Предельное время ожидания 0 миллисекунд означает не «немедленно», а «как можно скорее». В Internet Explorer, например, самый короткий интервал, который способен отмерять таймер, равен 15 миллисекундам.

Фоновые вычисления (web workers)

Последние версии броузеров предлагают другое решение для сценариев, выполняющихся длительное время: механизм фоновых вычислений (web workers). Этот механизм обеспечивает поддержку фонового потока выполнения в броузере. Вы можете поместить сценарий, выполняющий тяжелые вычисления, в отдельный файл, например *my_web_worker.js*, и затем вызывать его из главной программы (страницы), как показано ниже:

```
var ww = new Worker('my_web_worker.js');
ww.onmessage = function (event) {
    document.body.innerHTML +=
        "<p>message from the background thread: " + event.data + "</p>";
};
```

Ниже приводится исходный текст такого фонового задания, в котором простая арифметическая операция выполняется 1e8 раз (1 с 8 нулями):

```
var end = 1e8, tmp = 1;

postMessage('hello there');

while (end) {
    end -= 1;
    tmp += end;
    if (end === 5e7) { // число 5e7 - половина числа 1e8
        postMessage('halfway there, `tmp` is now ' + tmp);
    }
}
postMessage('all done');
```

Для взаимодействия с вызывающей программой фоновое задание использует метод `postMessage()`, а вызывающая программа подписывается на событие `onmessage`, вместе с которым получает изменения. Функция обратного вызова `onmessage` принимает в виде аргумента объект события со свойством `data`, в котором фоновое задание может передать любые данные. Аналогичным способом вызывающая программа может передавать данные фоновому заданию, вызывая (в данном примере) `ww.postMessage()`, а фоновое задание может подписаться на получение этих данных с помощью функции обратного вызова `onmessage`.

Предыдущий пример выведет в окне броузера следующие сообщения:

```
message from the background thread: hello there
message from the background thread: halfway there,
                                   `tmp` is now 3749999975000001
message from the background thread: all done
```

Удаленные взаимодействия

Современные веб-приложения часто используют механизмы удаленных взаимодействий для обмена данными с сервером без полной перезагрузки текущей страницы. Это позволяет создавать веб-приложения, еще больше напоминающие обычные приложения. Давайте рассмотрим некоторые из способов взаимодействий с сервером из JavaScript.

XMLHttpRequest

`XMLHttpRequest` – это специальный объект (функция-конструктор), доступный в большинстве современных броузеров и позволяющий выполнять HTTP-запросы из JavaScript. Чтобы выполнить запрос, требуется три этапа:

1. Создать объект `XMLHttpRequest` (будем называть его XHR для краткости).
2. Указать функцию обратного вызова, которая будет вызываться при изменении состояния объекта запроса.
3. Отправить запрос.

Первый этап реализуется просто:

```
var xhr = new XMLHttpRequest();
```

При этом в IE версий ниже 7 функциональность XHR была реализована в виде объекта ActiveX, поэтому для создания объекта запроса необходимо было выполнить специальные действия.

Второй этап – подключение функции обратного вызова для обработки события `readystatechange`:

```
xhr.onreadystatechange = handleResponse;
```

Последний этап – отправка запроса, который выполняется с помощью двух методов `open()` и `send()`. Метод `open()` определяет метод HTTP-запроса (например: GET, POST) и адрес URL. Метод `send()` передает объекту любые данные, если запрос выполняется методом POST, или пустую строку, если запрос выполняется методом GET. Последний параметр метода `open()` определяет, будет ли запрос выполняться асинхронно. Под асинхронностью подразумевается, что броузер не будет приостанавливать работу в ожидании ответа. Такой режим выполнения запроса не создает неприятных ощущений у пользователя, поэтому если нет каких-либо веских причин для обратного, в параметре, управляющем асинхронностью, всегда следует передавать значение `true`:

```
xhr.open("GET", "page.html", true);
xhr.send();
```

Ниже приводится полный действующий пример, извлекающий содержимое новой страницы и обновляющий содержимое текущей страницы (демонстрационная версия доступна по адресу *http://jspatterns.com/book/8/xhr.html*):

```
var i, xhr, activeXids = [
    'MSXML2.XMLHTTP.3.0',
    'MSXML2.XMLHTTP',
    'Microsoft.XMLHTTP'
];

if (typeof XMLHttpRequest === "function") {    // встроенный объект XHR
    xhr = new XMLHttpRequest();
} else {                                       // для IE до версии 7
    for (i = 0; i < activeXids.length; i += 1) {
        try {
            xhr = new ActiveXObject(activeXids[i]);
            break;
        } catch (e) {}
    }
}

xhr.onreadystatechange = function () {
    if (xhr.readyState !== 4) {
        return false;
    }
    if (xhr.status !== 200) {
        alert("Error, status code: " + xhr.status);
        return false;
    }
    document.body.innerHTML += "<pre>" + xhr.responseText + "<\/pre>";
};

xhr.open("GET", "page.html", true);
xhr.send("");
```

Несколько замечаний к этому примеру:

- В IE версии 6 и ниже процедура создания нового объекта XHR немного более сложная. Мы обходим в цикле список идентификаторов ActiveX, начиная от самой последней версии и заканчивая более ранними, и пытаемся создать объект, обернув операцию создания в блок `try-catch`.
- Функция обратного вызова проверяет значение свойства `readyState` объекта `xhr`. Это свойство может иметь пять возможных значений от 0 до 4, где 4 означает «завершено». Если объект не находится в состоянии «завершено», мы переходим к ожиданию следующего события `readystatechange`.

- По достижении состояния «завершено» функция обратного вызова проверяет значение свойства `status` объекта `xhr`. Это свойство содержит код состояния HTTP, например 200 (OK) или 404 (не найдено). Нас интересует только ответ с кодом состояния 200 – все остальные коды интерпретируются как ошибка (это сделано с целью упростить реализацию, в противном случае пришлось бы проверять все остальные допустимые коды состояния).
- Как видно из листинга, эта реализация проверяет поддерживаемый способ создания объекта XHR всякий раз, когда выполняется запрос. В предыдущих главах мы видели несколько шаблонов (таких как шаблон выделения ветвей, выполняющихся на этапе инициализации), с помощью которых вы сможете переписать этот пример так, чтобы проверка выполнялась только один раз.

Формат JSONP

Формат JSONP (JSON with padding – JSON с дополнением) – это еще один способ организации удаленных взаимодействий. В отличие от XHR, он не подпадает под действие политики «того же домена» в броузерах, поэтому его следует использовать с особой осторожностью из-за возможных проблем, связанных с безопасностью данных, загружаемых со сторонних сайтов.

Ответом на запрос XHR может быть документ любого типа:

- XML (исторически)
- Фрагмент HTML (что весьма обычно)
- Данные в формате JSON (простой и удобный формат)
- Простой текстовый файл или документ в любом другом формате

При использовании формата JSONP данные, чаще всего в формате JSON, обертываются в вызов функции, где имя функции передается в запросе.

Обычно адрес URL запроса JSONP имеет примерно такой вид:

http://example.org/getdata.php?callback=myHandler

где *getdata.php* может быть любой страницей или сценарием. Параметр `callback` определяет функцию на JavaScript, которая должна использоваться для обработки ответа.

Затем адрес URL загружается в динамический элемент `<script>`, например:

```
var script = document.createElement("script");
script.src = url;
document.body.appendChild(script);
```

Сервер возвращает некоторые данные в формате JSON, которые передаются в виде параметра функции обратного вызова. Фактически вы подключаете к странице новый сценарий, содержащий вызов функции. Например:

```
myHandler({"hello": "world"});
```

Пример использования JSONP: игра «крестики–нолики»

Давайте попробуем задействовать JSONP в реализации игры «крестики-нолики», где в качестве игроков выступают клиент (броузер) и сервер. Оба будут генерировать случайные числа в диапазоне от 1 до 9, а для получения значения со стороны сервера будет использоваться JSONP (рис. 8.2).

Действующая реализация игры доступна по адресу *http://jspatterns.com/book/8/ttt.html*.

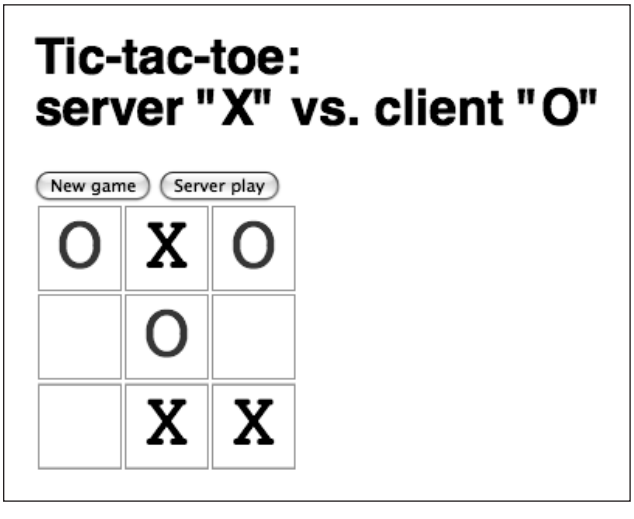

Рис. 8.2. *Игра «крестики-нолики», в которой используется JSONP*

На странице имеются две кнопки: кнопка запуска новой игры и кнопка получения информации о ходе сервера (клиент будет выполнять свой ход автоматически с некоторой задержкой):

```
<button id="new">New game</button>
<button id="server">Server play</button>
```

Игровое поле представляет собой таблицу с девятью ячейками и с соответствующими значениями атрибутов `id`. Например:

```
<td id="cell-1">&amp;nbsp;</td>
<td id="cell-2">&amp;nbsp;</td>
```

```
<td id="cell-3"> </td>
...
```

Сама игра реализована в виде глобального объекта `ttt`:

```
var ttt = {
    // ячейки, занятые к настоящему моменту
    played: [],

    // реализация функции с более коротким именем, для удобства
    get: function (id) {
        return document.getElementById(id);
    },

    // предусмотреть обработку щелчков мышью
    setup: function () {
        this.get('new').onclick = this.newGame;
        this.get('server').onclick = this.remoteRequest;
    },

    // очистка игрового поля
    newGame: function () {
        var tds = document.getElementsByTagName("td"),
            max = tds.length,
            i;
        for (i = 0; i < max; i += 1) {
            tds[i].innerHTML = " ";
        }
        ttt.played = [];
    },

    // выполняет запрос
    remoteRequest: function () {
        var script = document.createElement("script");
        script.src = "server.php?callback=ttt.serverPlay&played=" +
                     ttt.played.join(',');
        document.body.appendChild(script);
    },

    // функция обратного вызова, получает ход сервера
    serverPlay: function (data) {
        if (data.error) {
            alert(data.error);
            return;
        }
        data = parseInt(data, 10);
        this.played.push(data);

        this.get('cell-' + data).innerHTML = '<span class="server">X<\/span>';

        setTimeout(function () {
```

```
                ttt.clientPlay();
            }, 300); // имитирует задумчивость клиента
        },

        // ход клиента
        clientPlay: function () {
            var data = 5;

            if (this.played.length === 9) {
                alert("Game over");
                return;
            }

            // продолжать генерировать случайные числа 1-9,
            // пока не будет обнаружена незанятая ячейка
            while (this.get('cell-' + data).innerHTML !== " ") {
                data = Math.ceil(Math.random() * 9);
            }
            this.get('cell-' + data).innerHTML = 'O';
            this.played.push(data);

        }

    };
```

Объект `ttt` хранит список занятых ячеек в свойстве `ttt.played` и отправляет его серверу, благодаря чему сервер может выбрать новое число, соответствующее номеру незанятой ячейки. В случае ошибки сервер вернет ответ:

```
ttt.serverPlay({"error": "Error description here"});
```

Как следует из примера, функция обратного вызова в JSONP должна быть общедоступной функцией, необязательно глобальной, но может быть методом глобального объекта. При отсутствии ошибок сервер вернет ответ с вызовом метода:

```
ttt.serverPlay(3);
```

Число 3 здесь означает ячейку с номером 3 – случайный выбор, сделанный сервером. В данном примере данные настолько просты, что не требуется даже использовать формат JSON; единственное значение – это все, что требуется.

Обмен данными с использованием фреймов и изображений

Альтернативный способ запуска удаленного сценария заключается в использовании фреймов. С помощью JavaScript можно создать плавающий фрейм (`iframe`) и изменить адрес URL в его атрибуте `src`. Новый

адрес URL может содержать данные и вызовы функций, изменяющие вызывающую страницу – родительскую страницу за пределами плавающего фрейма (`iframe`).

Простейшая форма удаленных взаимодействий – необходимо только отправить данные на сервер и не требуется получить ответ. В подобных случаях можно создать новый элемент изображения и указать в его атрибуте `src` адрес URL сценария на сервере:

```
new Image().src = "http://example.org/some/page.php";
```

Этот шаблон называется *маяком*; он может пригодиться, если требуется отправлять данные, которые отслеживает сервер, например, для сбора статистической информации о посетителях. При использовании шаблона маяка не предполагается получать ответ, однако на практике (внимание: это антишаблон) сервер обычно возвращает изображение GIF с размерами 1×1. Предпочтительнее было бы возвращать HTTP-ответ «204 No Content». В этом случае клиенту будет отправляться только заголовок ответа без дополнительного содержимого.

Развертывание сценариев JavaScript

Когда дело доходит до включения сценариев JavaScript в работу, следует принять во внимание несколько проблем, связанных с производительностью. Давайте обсудим в общих чертах наиболее важные из них. Более детальное обсуждение этих проблем вы найдете в книгах «High Performance Web Sites» и «Even Faster Web Sites», выпущенных издательством O'Reilly.

Объединение сценариев

Первое правило, которому необходимо следовать при создании быстро загружающихся страниц, заключается в максимальном уменьшении количества внешних элементов, потому что для выполнения каждого отдельного запроса HTTP требуется некоторое время. В случае с JavaScript это означает, что скорость загрузки страницы можно существенно повысить за счет объединения внешних файлов сценариев.

Допустим, страница использует библиотеку jQuery. Это один файл *.js*. Кроме того, вы задействовали в странице несколько расширений jQuery, каждое из которых находится в отдельном файле. Таким образом, еще до того как вы напишете хотя бы одну строку программного кода, вам может потребоваться загрузить 4–5 файлов. Есть смысл объединить все эти файлы в один, особенно если учесть, что некоторые из них имеют достаточно маленький размер (2–3 килобайта), а затраты времени на выполнение нескольких запросов HTTP могут превысить время самой загрузки. Чтобы объединить сценарии, достаточно создать новый файл и вставить в него содержимое всех отдельных файлов.

Разумеется, объединение файлов должно выполняться непосредственно перед передачей программного кода в эксплуатацию, а не во время разработки, потому что это может существенно осложнить отладку.

Однако объединение файлов имеет свои недостатки:

- Это дополнительная операция перед запуском сценария, однако ее легко можно автоматизировать и выполнять из командной строки, например с помощью команды `cat` в Linux/UNIX:

```
$ cat jquery.js jquery.quickselect.js jquery.limit.js > all.js
```

- Теряются некоторые преимущества кэширования – при внесении изменений в один из файлов придется выполнить операцию объединения заново. Именно поэтому для крупных проектов желательно заранее планировать выпуск новых версий или поддерживать два пакета: один – с файлами, которые, вероятно, будут подвергаться изменениям, и один «базовый» пакет, который вообще не предполагается изменять.
- Необходимо продумать вопросы именования и нумерации версий пакетов, например использовать в имени файла сведения о дате его выпуска *all_20100426.js* или контрольную сумму содержимого файла.

Эти недостатки можно отнести скорее к неудобствам, но получаемые в итоге преимущества стоят усилий.

Сжатие и компрессия

В главе 2 мы говорили о сжатии программного кода. Очень важно сделать операцию сжатия частью процедуры подготовки программного кода к передаче в эксплуатацию.

С точки зрения пользователя нет никакого смысла загружать все комментарии, содержащиеся в вашем программном коде, наличие или отсутствие которых никак не сказывается на работоспособности приложения.

Выгода от сжатия может отличаться в зависимости от того, как часто вы используете комментарии и пробельные символы, а также в зависимости от используемого инструмента, выполняющего сжатие. Но в среднем в результате сжатия получается экономия порядка 50% от первоначального размера файла.

Необходимо также уделять внимание компрессии файлов сценариев. Достаточно один раз настроить сервер на использование компрессии gzip и постоянно пользоваться приростом скорости загрузки. Даже если вы пользуетесь услугами хостинга у поставщика, который не позволяет вам сильно вмешиваться в настройки сервера, тем не менее большинство поставщиков услуг позволяют использовать файлы *.htaccess* с настройками сервера Apache. Благодаря этому вы можете добавить в файл

.htaccess, находящийся в вашем корневом каталоге с веб-документами, следующее:

```
AddOutputFilterByType DEFLATE text/html text/css text/plain text/xml
application/JavaScript application/json
```

Компрессия в среднем дает уменьшение размеров файлов до 70%. Объединив компрессию со сжатием, можно смело ожидать, что пользователи будут загружать файлы, размер которых составляет всего 15% от первоначального размера исходных файлов, не подвергавшихся сжатию и компрессии.

Заголовок Expires

Вопреки широко распространенному мнению, файлы остаются в кэше броузера совсем недолго. Вы можете повлиять на это и увеличить время нахождения ваших файлов в кэше с помощью заголовка `Expires`.

И снова это вопрос однократной настройки сервера, которую можно выполнить в файле *.htaccess*:

```
ExpiresActive On
ExpiresByType application/x-JavaScript "access plus 10 years"
```

Недостаток такого подхода состоит в том, что если потребуется внести изменения в этот файл, вам также придется переименовать его, но вы наверняка и так уже предусмотрели это, определив соглашения об именовании своих пакетов.

Использование CDN

Аббревиатура CDN происходит от Content Delivery Network (сеть доставки содержимого). Это оплачиваемая (иногда достаточно дорогостоящая) услуга хостинга, которая позволит вам распространить копии ваших файлов по различным центрам обработки и хранения данных, разбросанным по всему миру, которые смогут быстрее обслуживать ваших пользователей, при этом в вашем коде будет использоваться тот же самый адрес URL.

Даже если у вас нет средств на оплату услуг CDN, вы можете воспользоваться некоторыми бесплатными предложениями:

- Компания Google размещает у себя множество популярных, свободно распространяемых библиотек, подключив которые, вы бесплатно сможете пользоваться преимуществами сети CDN, принадлежащей этой компании.
- Корпорация Microsoft размещает у себя библиотеку jQuery и свои собственные библиотеки поддержки технологии Ajax.
- Компания Yahoo! размещает библиотеку YUI в своей собственной сети CDN.

Стратегии загрузки

На первый взгляд задача подключения сценариев к странице не представляет сложности – вы добавляете элемент <script> и либо вставляете программный код JavaScript в него, либо связываете его с внешним файлом с помощью атрибута src:

```
// вариант 1
<script>
    console.log("hello world");
</script>
// вариант 2
<script src="external.js"></script>
```

Однако существует ряд дополнительных шаблонов и принципов, которые необходимо учитывать, если вы ставите перед собой цель создать высокопроизводительное веб-приложение.

Следует отметить, что разработчики имеют обыкновение использовать некоторые атрибуты элемента <script>:

language="JavaScript"

Во многих случаях используется форма написания с заглавными буквами «JavaScript» и иногда указывается номер версии. Атрибут language использовать нежелательно, потому что по умолчанию и так предполагается, что в качестве языка программирования используется JavaScript. Номер версии не оказывает никакого влияния, и в настоящее время его присутствие считается ошибкой.

type="text/JavaScript"

Этот атрибут считается обязательным, согласно стандартам HTML4 и XHTML1, но его также не следует использовать, потому что по умолчанию броузеры в любом случае принимают программный код JavaScript. В HTML5 этот атрибут считается необязательным. Кроме удовлетворения требований различных программ проверки разметки, нет никаких других причин использовать этот атрибут.

defer

(Или еще лучше – атрибут async, определяемый спецификацией HTML5) обеспечивает способ, хотя и не получивший широкого распространения, указать, что загрузка внешнего файла сценария не должна блокировать работу остальной части страницы. Подробнее о блокировании рассказывается ниже.

Местоположение элемента <script>

Элементы <script> блокируют загрузку других элементов страницы. Броузеры способны загружать сразу несколько элементов, но встречая элемент <script>, ссылающийся на внешний файл, они приостанавли-

вают загрузку страницы, пока внешний файл сценария не будет загружен и выполнен. Это замедляет загрузку страницы, особенно если в процессе загрузки встречается несколько таких элементов.

Чтобы минимизировать влияние эффекта блокировки, можно поместить элемент <script> в конец страницы, непосредственно перед закрывающим тегом </body>. Благодаря этому загрузка сценария не будет блокировать загрузку других ресурсов. К этому моменту все остальные элементы страницы уже будут загружены и представлены на обозрение пользователю.

Наихудший антишаблон заключается в перечислении отдельных файлов в заголовке документа:

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <!-- АНТИШаблоН -->
    <script src="jquery.js"></script>
    <script src="jquery.quickselect.js"></script>
    <script src="jquery.lightbox.js"></script>
    <script src="myapp.js"></script>
</head>
<body>
    ...
</body>
</html>
```

Предпочтительнее было бы объединить все файлы в один:

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <script src="all_20100426.js"></script>
</head>
<body>
    ...
</body>
</html>
```

А еще лучше – поместить объединенный сценарий в самый конец страницы:

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<body>
    ...
```

```
    <script src="all_20100426.js"></script>
</body>
</html>
```

Фрагментирование средствами HTTP

Протокол HTTP поддерживает возможность фрагментирования документов. Это позволяет отправлять страницу по частям. То есть если у вас имеется сложная веб-страница, вы можете не ждать, пока будут выполнены все операции на стороне сервера, прежде чем начнется передача более или менее статической части страницы.

Простейший прием состоит в том, чтобы обеспечить отправку содержимого раздела `<head>` в виде первого фрагмента и отправку остальной части страницы, как только она будет готова. Другими словами, страницу можно оформить, как показано ниже:

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<!-- end of chunk #1 -->
<body>
    ...
    <script src="all_20100426.js"></script>
</body>
</html>
<!-- end of chunk #2 -->
```

При таком усовершенствовании следовало бы перенести сценарии JavaScript обратно в раздел `<head>`, сделав их частью первого фрагмента. Благодаря этому броузер выполнит загрузку сценариев вместе с заголовком документа, пока остальная страница подготавливается сервером:

```
<!doctype html>
<html>
<head>
    <title>My App</title>
    <script src="all_20100426.js"></script>
</head>
<!-- end of chunk #1 -->
<body>
    ...
</body>
</html>
<!-- end of chunk #2 -->
```

Но еще лучше было бы создать третий фрагмент в самом конце страницы, содержащий только сценарии. При этом можно было бы поместить

в первый фрагмент часть тела страницы, содержащую статическую информацию:

```
<!doctype html>
<html>
<head>
    <title>My App</title>
</head>
<body>
    <div id="header">
        <img src="logo.png" />
        ...
    </div>
    <!-- end of chunk #1 -->

    ... Остальное тело страницы ...

    <!-- end of chunk #2 -->
    <script src="all_20100426.js"></script>
</body>
</html>
<!-- end of chunk #3 -->
```

Этот прием соответствует идее последовательного улучшения и ненавязчивого JavaScript. Сразу за концом второго фрагмента HTML у вас будет полностью загруженная и готовая к использованию страница в том виде, как если бы броузер не поддерживал JavaScript. После того как будет загружен третий фрагмент со сценариями JavaScript, они расширят возможности страницы, добавив в нее все бантики и рюшечки.

Динамические элементы <script> для неблокирующей загрузки сценариев

Как уже упоминалось выше, сценарии JavaScript блокируют загрузку файлов, следующих за ними. Однако имеется несколько приемов, предотвращающих это:

- Загрузка сценариев с помощью запросов XHR и последующим их выполнением с помощью `eval()`. Этот прием подвержен ограничениям политики единого домена-источника, а кроме того, вовлекает в работу функцию `eval()`, применение которой само по себе считается антишаблоном.
- Использование атрибутов `defer` и `async`, но они действуют не во всех броузерах.
- Использование динамических элементов `<script>`.

Последний прием является наиболее привлекательным. Он напоминает описанный выше шаблон использования JSONP – вы создаете новый

элемент <script>, определяете значение его атрибута src и добавляете этот элемент в страницу.

Следующий пример загрузит файл JavaScript асинхронно, не блокируя загрузку остальной части страницы:

```
var script = document.createElement("script");
script.src = "all_20100426.js";
document.documentElement.firstChild.appendChild(script);
```

Недостаток такого шаблона заключается в том, что его нельзя использовать для загрузки других сценариев, зависящих от *main.js*. Файл *main.js* в этом примере загружается асинхронно, поэтому нет никаких гарантий, что он будет загружен к тому моменту, когда дело дойдет до выполнения других сценариев, следующих за ним и использующих объекты, определяемые в нем.

Чтобы преодолеть этот недостаток, можно все встроенные сценарии не выполнять немедленно, сразу после загрузки, а собирать в массив в виде функций. А после загрузки основного сценария можно будет выполнить все функции, собранные в буферный массив. Это решение предусматривает три этапа.

Сначала как можно выше на странице необходимо создать массив для сохранения в нем встроенных сценариев:

```
var mynamespace = {
    inline_scripts: []
};
```

Затем необходимо обернуть все встроенные сценарии функциями и добавить их в массив inline_scripts. Другими словами:

```
// было:
// <script>console.log("I am inline");</script>

// стало:
<script>
mynamespace.inline_scripts.push(function () {
    console.log("I am inline");
});
</script>
```

И наконец, предусмотреть в главном сценарии цикл обхода буфера встроенных сценариев и их выполнения:

```
var i, scripts = mynamespace.inline_scripts, max = scripts.length;
for (i = 0; i < max; max += 1) {
    scripts[i]();
}
```

Добавление элемента <script>

Часто сценарии добавляются в раздел <head> документа, однако их можно добавлять в любые элементы, включая <body> (как показано в примере использования JSONP).

Для добавления сценария в раздел <head> в предыдущем примере было использовано свойство documentElement, потому что оно представляет элемент <html>, первым вложенным элементом в котором является элемент <head>:

```
document.documentElement.firstChild.appendChild(script);
```

Также часто используется следующий способ:

```
document.getElementsByTagName("head")[0].appendChild(script);
```

Эти приемы отлично подходят в случаях, когда вся разметка находится под полным вашим контролем, но представьте, что вы создаете виджет или рекламный элемент и понятия не имеете о том, на каких страницах он будет размещаться. С технической точки зрения страница может не иметь ни раздела <head>, ни раздела <body> – хотя свойство document.body будет доступно всегда, даже в отсутствие тега <body>:

```
document.body.appendChild(script);
```

Но, как бы то ни было, на странице, где выполняется сценарий, всегда будет присутствовать как минимум один тег – тег <script>. Если на странице не будет тега <script> (со встроенным сценарием или со ссылкой на внешний файл), то ваш сценарий просто не будет выполнен. Вы можете воспользоваться этим фактом и с помощью метода insertBefore() добавить свой элемент <script> перед первым доступным элементом <script> на странице:

```
var first_script = document.getElementsByTagName('script')[0];
first_script.parentNode.insertBefore(script, first_script);
```

Здесь first_script – это элемент <script>, гарантированно присутствующий на странице, а script – новый элемент <script>, созданный вами.

Отложенная загрузка

Под отложенной загрузкой понимается загрузка внешнего файла после того, как будет возбуждено событие load. При использовании этого приема часто бывает выгодно разбить большой объем кода на две части:

- В первую часть помещается программный код, выполняющий инициализацию страницы и подключающий обработчики событий к элементам пользовательского интерфейса.
- Во вторую часть помещается программный код, который потребуется только после того, как пользователь выполнит некоторые ма-

нипуляции, или при других условиях, которые могут возникнуть спустя некоторое время.

Цель приема состоит в том, чтобы обеспечить последовательную загрузку страницы и предоставить пользователю возможность включиться в работу как можно скорее. После этого, пока пользователь занят изучением страницы, можно загрузить оставшуюся ее часть в фоновом режиме.

Загрузка второй группы сценариев JavaScript реализуется простым динамическим добавлением элемента `<script>` в заголовок или в тело страницы:

```
    ... Тело страницы ...

    <!-- end of chunk #2 -->
    <script src="all_20100426.js"></script>
    <script>
    window.onload = function () {
        var script = document.createElement("script");
        script.src = "all_lazy_20100426.js";
        document.documentElement.firstChild.appendChild(script);
    };
    </script>
</body>
</html>
<!-- end of chunk #3 -->
```

Для большинства приложений часть программного кода, загружаемая второй, будет иметь больший объем, чем первая базовая часть, потому что наиболее интересные «операции» (такие как перетаскивание элементов мышью, применение объекта XHR и воспроизведение анимационных эффектов) выполняются только после того, как пользователь инициирует их.

Загрузка по требованию

Предыдущий шаблон предусматривает безусловную загрузку дополнительных сценариев JavaScript, предполагая, что этот программный код потребуется наверняка. Но возможно ли реализовать загрузку только тех сценариев, которые действительно необходимы?

Представьте, что на странице имеется панель с различными вкладками. Щелчок на вкладке приводит к выполнению запроса XHR для получения содержимого вкладки и воспроизведению анимационного эффекта, плавно изменяющего цвет. А что если это единственное место на странице, где необходимы библиотеки поддержки XHR и анимационных эффектов, и что если пользователь может вообще никогда не щелкнуть ни на одной вкладке?

Воспользуйтесь шаблоном загрузки по требованию. Для этого можно реализовать функцию или метод `require()`, принимающую имя файла сценария, который требуется загрузить, и функцию обратного вызова, которая будет вызвана по окончании загрузки сценария.

Такую функцию `require()` можно было бы использовать, как показано ниже:

```
require("extra.js", function () {
    functionDefinedInExtraJS();
});
```

Давайте посмотрим, как можно было бы реализовать такую функцию. Загрузить дополнительный сценарий просто – достаточно воспользоваться шаблоном динамического элемента `<script>`. Выяснение момента, когда будет закончена загрузка этого сценария, реализуется немного сложнее из-за различий, существующих между броузерами:

```
function require(file, callback) {

    var script = document.getElementsByTagName('script')[0],
        newjs = document.createElement('script');

    // IE
    newjs.onreadystatechange = function () {
        if (newjs.readyState === 'loaded' || newjs.readyState === 'complete'){
            newjs.onreadystatechange = null;
            callback();
        }
    };

    // другие броузеры
    newjs.onload = function () {
        callback();
    };

    newjs.src = file;
    script.parentNode.insertBefore(newjs, script);
}
```

Несколько комментариев к реализации:

- В IE необходимо подписаться на получение события `readystatechange` и ждать, пока свойство `readyState` не примет значение «загружено» или «завершено». Во всех остальных броузерах этого не требуется.
- В броузерах Firefox, Safari и Opera необходимо подписаться на получение события `load`, присвоив обработчик события свойству `onload`.
- Этот прием не действует в Safari 2. Если необходимо обеспечить поддержку этого броузера, вам придется с помощью таймера органи-

зовать периодическую проверку наличия какой-либо переменной, определяемой в дополнительном файле. Как только такая переменная появится, это будет означать, что сценарий был загружен и выполнен.

Для проверки этой реализации можно создать на стороне сервера сценарий с искусственной задержкой (имитирующей задержку в сети) с именем *ondemand.js.php*, например:

```
<?php
header('Content-Type: application/JavaScript');
sleep(1);
?>
function extraFunction(logthis) {
    console.log('loaded and executed');
    console.log(logthis);
}
```

Теперь проверим работу функции `require()`:

```
require('ondemand.js.php', function () {
    extraFunction('loaded from the parent page');
    document.body.appendChild(document.createTextNode('done!'));
});
```

Этот фрагмент выведет в консоль две строки и добавит в страницу строку «done!». Действующий пример вы найдете по адресу *http://jspatterns.com/book/7/ondemand.html*.

Предварительная загрузка сценариев JavaScript

В шаблонах с отложенной загрузкой и загрузкой по требованию сценарии, необходимые странице, загружаются после загрузки самой страницы. Кроме того, мы можем загрузить сценарии, которые не требуются текущей странице, но с высокой степенью вероятности потребуются следующей. В этом случае, когда пользователь переходит ко второй странице, сценарии оказываются уже загруженными и создается ощущение, что страница загружается очень быстро.

Предварительная загрузка может быть реализована за счет применения шаблона с динамическим элементом `<script>`. Но это означает, что сценарий будет загружен и выполнен. Синтаксический анализ предварительно загруженного сценария не только требует некоторого времени, но его выполнение может привести к появлению ошибок JavaScript, потому что предварительно загруженный сценарий предполагает, что выполняется в контексте второй страницы и, например, ожидает отыскать определенные узлы DOM.

Однако существует прием, позволяющий загружать сценарии без последующего их анализа и выполнения интерпретатором; этот прием также может использоваться для загрузки стилей CSS и изображений.

В IE можно выполнить запрос, применив уже знакомый вам шаблон маяка:

```
new Image().src = "preloadme.js";
```

В других броузерах вместо элемента <script> можно использовать тег <object> и определить в его атрибуте data адрес URL сценария:

```
var obj = document.createElement('object');
obj.data = "preloadme.js";
document.body.appendChild(obj);
```

Чтобы предотвратить отображение этого объекта на странице, необходимо также записать значение 0 в его атрибуты width и height.

Можно создать универсальную функцию или метод preload() и использовать ее в реализации шаблона выделения ветвей, выполняющихся на этапе инициализации (глава 4), обслуживающей различия между броузерами:

```
var preload;
if (/*@cc_on!@*/false) {     // Определение IE с помощью условного комментария
    preload = function (file) {
        new Image().src = file;
    };
} else {
    preload = function (file) {
        var obj = document.createElement('object'),
            body = document.body;

        obj.width = 0;
        obj.height = 0;
        obj.data = file;
        body.appendChild(obj);
    };
}
```

Пример использования новой функции:

```
preload('my_web_worker.js');
```

Недостаток этого шаблона заключается в использовании приема определения типа броузера, но без этого не обойтись, потому что в данном случае прием определения поддерживаемых возможностей не позволяет точно выяснить поведение броузера. В этом шаблоне, например, теоретически можно было бы проверить, возвращает ли выражение typeof Image значение "function", и пользоваться этой проверкой вместо приема определения типа броузера. Однако этот прием оказывается бесполезным, потому что все броузеры поддерживают вызов new Image(), но в некоторых из них для изображений предусмотрен отдельный кэш, а это означает, что компоненты, предварительно загруженные как изобра-

жения, не смогут извлекаться второй страницей из кэша как сценарии, и их потребуется загружать повторно.

Существует интересный способ определения типа броузера – по наличию поддержки условных комментариев. Он намного безопаснее, чем анализ значения свойства `navigator.userAgent`, потому что это значение легко может быть изменено пользователем.

Инструкция:

```
var isIE = /*@cc_on!@*/false;
```

установит переменную `isIE` в значение `false` во всех броузерах (потому что они игнорируют комментарии), но только не в IE из-за оператора отрицания `!` в условном комментарии. Вот как эту инструкцию «видит» IE:

```
var isIE = !false; // true
```

Шаблон предварительной загрузки может применяться не только к сценариям, но и к любым другим компонентам. Он будет полезен, например, на страницах, где пользователю предлагается ввести имя и пароль. Пока пользователь вводит свое имя, вы можете использовать это время для предварительной загрузки данных (разумеется, не содержащих никакой секретной информации), потому что более чем вероятно, что пользователь перейдет к следующей странице.

В заключение

В предыдущих главах книги основное внимание уделялось шаблонам использования базовых возможностей языка JavaScript, не зависящих от среды выполнения, тогда как в этой главе, напротив, исследовались шаблоны, предназначенные только для использования в броузерах.

Мы рассмотрели:

- Идеи разделения на составные части (HTML: содержимое, CSS: представление, JavaScript: поведение), ненавязчивого JavaScript, а также прием определения поддерживаемых возможностей, рекомендуемый в противовес приему определения типа броузера. (Однако в конце главы мы увидели пример, когда эта рекомендация оказывается неприменима на практике.)
- Операции с деревом DOM – шаблоны, ускоряющие доступ и манипулирование элементами дерева DOM главным образом за счет объединения операций, потому что любое обращение к DOM является дорогостоящей операцией.
- События, приемы обработки событий, не зависящие от типов броузеров, и использование приема делегирования событий для уменьше-

ния количества обработчиков событий и повышения производительности.

- Два шаблона организации выполнения продолжительных вычислений – использование функции `setTimeout()` для разбиения операций на мелкие фрагменты и использование механизма фоновых вычислений (web workers), реализованного в современных броузерах.
- Различные шаблоны удаленных взаимодействий и обмена данными между сервером и клиентом – XHR, JSONP и шаблон маяка, основанный на использовании плавающих фреймов и элементов изображений.
- Приемы, используемые при развертывании сценариев JavaScript на действующем сервере: объединение файлов сценариев, сжатие и компрессия (что позволяет уменьшить объем загружаемых файлов на 85%), а также пользование услугами CDN и применение заголовков `Expires`, позволяющих увеличить преимущества механизма кэширования.
- Шаблоны включения сценариев в страницу, обеспечивающие повышение производительности, в том числе: выбор местоположения элементов `<script>`, использование преимуществ механизма фрагментирования, предусмотренного протоколом HTTP. Кроме того, мы познакомились с различными шаблонами, снижающими время начальной загрузки больших сценариев, такими как отложенная загрузка, предварительная загрузка и загрузка по требованию.

Алфавитный указатель

К

Л

М

Н

P

C

У

Ф

Ц

Ч

Ш

Э