БАКАЛАВРИАТ

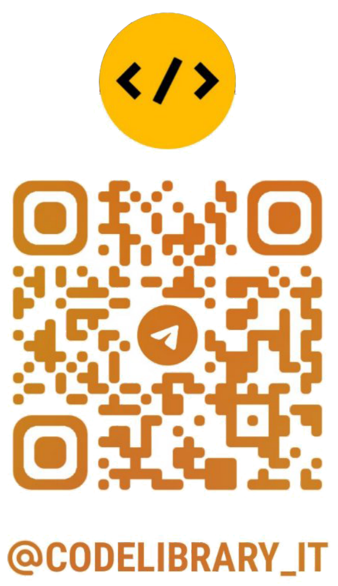
И. Б. Государев

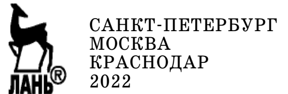
ВВЕДЕНИЕ В ВЕБ-РАЗРАБОТКУ НА ЯЗЫКЕ JavaScript

И. Б. ГОСУДАРЕВ

ВВЕДЕНИЕ В ВЕБ-РАЗРАБОТКУ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие





САНКТ-ПЕТЕРБУРГ
МОСКВА
КРАСНОДАР
2022

УДК 004.43
ББК 32.973.26-018.1я73
Г 72

Государев И. Б.

Г 72 Введение в веб-разработку на языке JavaScript: Учебное пособие. — СПб.: Издательство «Лань», 2022. — 144 с.: ил. — (Учебники для вузов. Специальная литература).

ISBN 978-5-8114-3539-5

В учебно-методическом пособии рассматриваются фундаментальные основы и прикладные аспекты использования языка JavaScript для клиентской и серверной разработки веб-ресурсов. Проанализированы основные тенденции развития наиболее распространённого языка клиентского веб-программирования в свете внедрения новых стандартов ECMAScript. Изучающим язык предложены задания развивающего и проблемного типа, нацеленные на формирование профессиональных компетенций в области веб-разработки.

Пособие инновационно по своей структуре: все примеры и задания доступны в его интерактивной части (сайт kodaktor.ru), которая является зарегистрированным в Роспатенте средством электронного обучения и содержит не только образцы кода, но и скринкасты по ряду рассматриваемых в текстовой части вопросов.

УДК 004.43
ББК 32.973.26-018.1я73

Обложка
Е. А. ВЛАСОВА

© Издательство «Лань», 2022
© И. Б. Государев, 2022
© Издательство «Лань»,
художественное оформление, 2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5
Релевантность пособия 5
Терминология 6
Замечание о коде и его стиле 7
Замечание об инструментах 7
Рекомендации студенту 8
Рекомендации преподавателю 9
Веб-портфолио 10
ГЛАВА 1. СИНТАКСИС И ОСНОВНЫЕ ПОНЯТИЯ JAVASCRIPT 11
Подготовка к лекции 11
Обзор языка JavaScript 13
Консоль 14
Операторы, выражения и инструкции 16
ГЛАВА 2. ЗАВИСИМОСТИ 24
Менеджмент зависимостей 24
Запуск зависимостей 28
ГЛАВА 3. ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ 32
Преобразование типов 33
Тернарный оператор и ветвление 36
ГЛАВА 4. ФУНКЦИИ 40
Линтинг 40
IIFE и стрелки 48
ГЛАВА 5. ФУНКЦИЯ КАК ТИП ДАННЫХ 52
Промисификация 58
Замыкание 61
Модели реализации коллбэка 61
Веб-воркеры 62
ГЛАВА 6. СТРОКИ И МАССИВЫ 65
Задачи с массивами 69
Деструктуризация массива 70
Оператор rest/spread 72
ГЛАВА 7. ЛИТЕРАЛЬНЫЕ ОБЪЕКТЫ 77
Деструктуризация объектов 81
Методы 84
Энумерабельность и итерабельность 86
ГЛАВА 8. ПРОТОТИПЫ И КОНСТРУКТОРЫ 94
Собственные ключи 97
Цепочка прототипов 98
Классы ES2015 101
super 104
ГЛАВА 9. МОДУЛЬНОСТЬ И ТРАНСПИЛЯЦИЯ 106
Транспиляция 108
Сборка проекта 112

ЗАКЛЮЧЕНИЕ 119
ПРИЛОЖЕНИЕ 1 120
ПРИЛОЖЕНИЕ 2 124
ПРИЛОЖЕНИЕ 3 125
ПРИЛОЖЕНИЕ 4 135
ЛИТЕРАТУРА 140

ВВЕДЕНИЕ

«Веб-разработка» — это словосочетание, указывающее на область деятельности, которую в английском языке называют web development. Это не то же самое, что веб-дизайн. Речь не идёт о художественном подходе к веб-ресурсам, замысле, видении, эргономичности, юзабилити, эстетике. Речь идёт о создании решений поставленных задач («веб-решений») с помощью языковых и инструментальных средств, в первую очередь языков программирования и разметки.

В силу исторических причин язык программирования JavaScript приобрёл особый статус языка, объединяющего клиентскую и серверную веб-разработку (см. последнюю главу). Этот факт и побудил посвятить ему данное учебное пособие.

Релевантность пособия

Веб-разработка, находясь на переднем крае IT, является областью, в которой новые сущности и связи между ними возникают буквально ежедневно. Они возникают в самом вебе (в блогах разработчиков в первую очередь). Некоторые из них исчезают (https://github.com/GossJS/11lines). Любая попытка зафиксировать эти сущности и связи в традиционном виде печатного текста интересна в основном исторически. Данное пособие нужно в первую очередь для того, чтобы сориентировать читателя в некоторых важных направлениях, предложить примеры терминов, заданий, решений, ресурсов. Оно тоже фиксирует некоторое состояние дел в области веб-разработки, но не претендует на актуальность, так как это невозможно в принципе. Его электронная составляющая (сайт Кодактор, см. ниже «Замечание об инструментах») вместе с репозиторием GossJS служат средством обеспечения актуальности сообщаемой информации через включённые в текст ссылки.

Если вы видите скриншот окна программы (например, консоли браузера), то следует понимать, что в момент чтения текста окно может выглядеть иначе. Поэтому необходимо сверять одно с другим, выяснять актуальное состояние вопроса, обращаясь к соответствующим ресурсам. Назовём это фоновыми информационными запросами (ФИЗ). Прекрасным средством удовлетворения этих запросов является сайт stackoverflow, который к тому же развивает коммуникационную культуру веб-разработчика.

Если создатель веба Тим Бернерс-Ли связывал слово «паутина» с соответствующей сетевой структурой информации, то сегодня оно явно приобрело другие коннотации: изменчивости, некоторой эфемерности, сочетания хрупкости, гибкости и прочности.

Пособие адресовано широкому кругу читателей, но в первую очередь — студентам магистратуры, изучающим веб-разработку, веб-технологии, веб-платформу электронного обучения. В частности, это магистратуры направлений «09.04.02 — Информационные системы и технологии» и «44.04.01 — Педагогическое образование». Главы пособия соответствуют разделам модулей,

изучавшихся магистрантами программ «Веб-технологии» (ИТМО, 2017–2018) и «Корпоративное электронного обучение» (РГПУ им. А. И. Герцена, 2017–2018). В частности, на базе ИТМО автором разработана и реализована программа дисциплины «Исследование экосистем веб-языков и веб-технологий», изучаемая во втором семестре первого курса магистратуры. В разделе «Веб-портфолио» приведён пример веб-портфолио студентов по этой дисциплине.

Это пособие, повторимся, не дублирует справочники и руководства. Русское издание фундаментального руководства Дэвида Флэнагана [1] занимает более тысячи страниц, и такие объёмы в учебных изданиях невообразимы. Автор старался построить связный рассказ о важных, развивающихся чертах языка, акцентируя внимание на интересных, неоднозначных моментах, противоречиях и пробелах, подстёгивающих исследователя. Этот нарратив построен в том числе на вопросах, упражнениях и отражает исследовательский путь автора. Изучение языка программирования — мощный способ развития сознания, особенно если подходить к делу критически, отчасти философски, привлекая исследовательские приёмы анализа, сравнения, обобщения, аналогии, моделирования и другие.

Терминология

Веб-разработка, находясь на переднем крае IT, является областью, в которой новые термины появляются буквально ежедневно. Академический стиль текста с присущим ему стремлением опираться на известные, устоявшиеся слова и словосочетания, «одобренные» «авторитетными источниками», в этой области невозможен. Это явление особенно выпукло выглядит в местном словоупотреблении, потому что новые термины появляются в английском языке как постоянно обновляющийся результат живой совместной работы тысяч специалистов, не для каждого из которых английский язык является родным. Для некоторых сущностей в русском языке нет не только отдельного слова или словосочетания, но даже и относительно краткой фразы, которая бы понятным образом описывала обозначаемый словом смысл. В таких случаях наиболее вероятный путь — использовать слово в оригинальном написании или в некоторой транслитерации/транскрипции. Вот несколько примеров, актуальных на момент написания данного текста:

1. Понятие «гуглить», которое вряд ли нуждается в комментариях.

2. Понятие «веб», которое стали писать русскими буквами и не пытаются переводить как «паутина». Аналогично: «браузер», а не «обозреватель» и «блог», а не «дневник в паутине».

3. Понятие, обозначаемое словосочетанием landing page. Типичный вариант ссылки на это понятие — «лендинг» (хотя ещё в 2011 г. переводчиками одной из книг на эту тему было предложено словосочетание «целевая страница»).

4. Понятие, обозначаемое словом callback. Хотя существует «официальный» перевод («функция обратного вызова»), в ежедневном словоупотреблении мы часто встречаем «коллбэк».

Употребление таких слов, как коллбэк, «снижает стиль», делает его в известной мере неформальным, но это практически неизбежно, хотя затрудняет работу по редактированию авторского текста. Из-за постоянно меняющихся ресурсов теряет смысл деятельность по созданию ссылок на эти ресурсы — нет никакой гарантии, что сохранится адрес и/или содержание ресурса. Читателю придётся взять на себя работу по гуглению и поддержанию своих представлений об этих словах в актуальном состоянии.

Замечание о коде и его стиле

Для более комфортной работы с кодом важны удобные средства редактирования, которые включают *подсветку синтаксиса* (https://kodaktor.ru/highlight) и *выявление проблемных мест* (типа «переменная объявлена, но нигде не используется»).

Эти задачи решаются адекватной настройкой редактора или IDE, при этом если говорить конкретно о выявлении ошибок, существует элемент сложившейся JavaScript-экосистемы, известный как линтер eslint, который может быть настроен на совместную работу с выбранным редактором. Пакет eslint желательно установить сразу и оперативно освоить работу с ним (https://kodaktor.ru/g/eslint). Вносимые линтером правки как бы насильственно держат разработчика в рамках некоторого *стиля*, который в совокупности регламентируется набором настроек (например, от airbnb). Подсказки линтера содержат термины, значение которых необходимо знать веб-разработчику (например, sparse arrays).

На сегодняшний день использование систем контроля версий является фактическим стандартом, но здесь хотелось бы выделить конкретную возможность таких инструментов, как Git сравнивать между собой различные версии одного и того же файла или разные файлы и визуализировать разницу в очень наглядной форме. Для экспорта получившейся визуализации удобно использовать пакет diff2html-cli — образец его применения показан в начале главы 2. Различные примеры цветовых схем подсветки приведены в примерах на сайте kodaktor.ru и в электронной версии пособия.

Замечание об инструментах

Пособие по своей природе связано с сайтом kodaktor.ru (кодактор), являясь его печатным образом. При этом задействуются следующие дополнительные инструменты:

- https://github.com/GossJS — репозитории с примерами кода;
- https://www.youtube.com/user/gossoudarev/videos?view=0&flow=grid — скринкасты и видеоролики с устными объяснениями материала;
- https://ide.c9.io/gossoudarev/js-study1 — проект на платформе Cloud9 для совместной работы с кодом;
- https://kodaktor.ru/g/onlineeditors — различные онлайн-редакторы кода.

Рекомендации студенту

Уважаемый студент, пожалуйста, используйте информацию этого пособия как один из способов ориентироваться в веб-разработке. Применяйте критическое мышление для сравнения изложенного здесь с другими источниками. Всякий раз, встречая странное или незнакомое слово, выясняйте его значение (ФИЗ) в Google, других книгах и научных статьях. Сравнивайте решение задач на JavaScript и на других языках (PHP, Go, Dart, ...), сравнивайте устройство этих языков по тем параметрам, которые допускают сопоставление.

Важно исследовать язык (в данном случае JavaScript). К этому относятся, в том числе, следующие действия:

- работа со статистикой по языку (использование в различных проектах и т. п.);
- измерение времени и пространства, которые тратятся на выполнение/вычисление;
- выяснение идей/концептов, стоящих за характеристиками языка;
- поиск ошибок, противоречий;
- сравнение реального поведения с документацией;
- формулирование предложений по совершенствованию;
- попытки написания собственных расширений/библиотек/фреймворков/компиляторов.

Некоторые методы исследования специфичны для используемой среды. Например, запустив node, в консоли мы можем ввести начало имени команды (Refl) и нажать клавишу табуляции, чтобы функция автодополнения предложила нам доступные варианты дальнейшей записи (Reflect). Так можно получить как бы быструю встроенную справку.

Пререквизитами к освоению материала являются знания и умения в области языков разметки (HTML5, CSS, SVG, XML) и алгоритмизации. Вам также нужно иметь представление о командной строке Linux или родственных систем, о системе контроля версий Git и системе контейнеризации Docker. Весьма желательно установить эти продукты на свой компьютер. Для комфортной работы с node на компьютере, на котором основной операционной системой является Windows, лучше установить виртуальную машину Linux (например, Ubuntu для VirtualBox). Ещё одним способом является использование Vagrant (https://github.com/GossJS/vagrant).

Подготовьте своё рабочее место к изучению и практикованию JavaScript:

- установите платформу node.js по инструкциям с официального сайта (https://nodejs.org/en/download/ и https://kodaktor.ru/g/node);
- установите и настройте удобный редактор кода (рекомендуется Atom.io);
- установите как минимум два браузера Google Chrome и Mozilla FireFox;
- подготовьте свой репозиторий, в котором можно хранить код, а также ссылки на другие варианты размещения работающих примеров (результатов выполнения заданий) и который можно рассматривать как веб-портфолио (см. ниже).

Примеры кода, работающего в браузере, а также упражнения для самостоятельной работы даются в основном в виде ссылок на сайт **kodaktor.ru**. Частично по таким адресам изложены и некоторые теоретические материалы. Адреса этих материалов достаточно короткие, и их можно набрать на клавиатуре (поэтому не используются сокращатели).

Пример: https://kodaktor.ru/js01

Чтобы эта ссылка была воспринята максимально однозначно, здесь она продублирована в виде QR-кода:

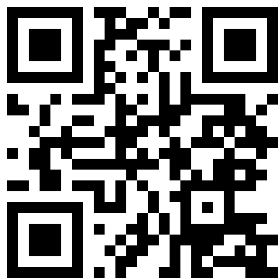


Рис. 1. QR-код для ссылки https://kodaktor.ru/js01

Доступный по такому адресу документ будем называть словом «борд» («board» — «доска»), а часть адреса после kodaktor.ru (в примере выше это ***js01***) будем называть «ключом борда».

Некоторые адреса кодактора, которые выглядят как борды, являются программами, микросервисами для решения некоторых задач. В этом случае эти адреса называются запросами (в них есть параметры, от которых зависит результат): например, запрос https://kodaktor.ru/sleep/?n=6 позволяет получить паузу длительностью в 6 секунд и две временные отметки с этой разницей.

Обо всех опечатках, замечаниях, предложениях большая просьба сообщать по адресу электронной почты webmaster@kodaktor.ru.

Рекомендации преподавателю

В пособии автор не говорит от первого лица и не выступает в роли преподавателя, который пользуется «скрытыми» приёмами. Вместо этого применяется тактика раскрытия приёма, что в случае образовательного процесса тождественно описанию методики самообучения (учения, learning). Это нацелено на достижение следующего эффекта: читатель должен видеть себя как преподавателя и как обучающегося одновременно и спрашивать себя, понятно ли, доступна ли скорость подачи информации и т. д. Иными словами, речь идёт о развитии рефлексии.

Когда это полезно, в тексте возникают методические примечания. Преподаватель может рассматривать их как советы по изложению соответствующего материала, в основном в очной форме. При дистанционной работе их нужно адаптировать к соответствующему режиму взаимодействия.

У преподавания таких технологий, как веб и JavaScript есть как минимум одна достаточно дискомфортная отличительная черта, обусловленная динами-

кой изменений: материалы к занятиям приходится проверять не позднее чем за несколько часов до их проведения. В некоторых случаях разумно записывать скринкасты и логи происходящего, в противном случае всегда есть шанс непосредственно на лекции столкнуться с неожиданным поведением, которому трудно найти объяснение или исправить. Конечно, при известном навыке экспромта и сотрудничестве аудитории это не обязательно должно быть проблемой, но всё же есть смысл предусмотреть разные сценарии хода событий.

При проведении лекционных занятий и в преимущественно дистанционной форме целесообразно использовать инструменты вещания кода и совместной работы (среда c9, scrimba.com, kodaktor.ru и др.), возможно, в сочетании с вещанием видео и экрана (в какой-либо вебинарной среде или, например, на канале youtube.com).

При проведении практических занятий полезно заранее распределить темы для подготовки к выступлениям/сообщениям/докладам: https://kodaktor.ru/g/keynote — в этом случае практическое занятие может быть организовано как мастер-класс обучающегося.

В основе каждой из глав находится содержательная тема (например, «Зависимости»), по которой формулируются направления самостоятельной работы. В них может быть сделан упор на исследовательскую составляющую (например, выполнить сравнительный анализ функциональности менеджеров npm и yarn) или на практический аспект работы (например, разработать собственный шаблон проекта на JavaScript и сценарий развёртывания). Далее на основе задания самостоятельной работы формулируются задания лабораторной работы (если такие часы предусмотрены в учебном плане). Они носят более конкретный характер и сопровождаются теоретическим пояснением и набором инструкций.

Скринкасты могут заменять наборы инструкций и теоретические пояснения. В качестве скринкастов могут фигурировать записи живого вещания во время лекционного занятия.

Веб-портфолио

Автор данного пособия ввёл в рассмотрение русскоязычный термин «веб-портфолио», который обозначает веб-ресурс, используемый для хранения и мониторинга прогресса обучающегося (https://kodaktor.ru/g/webportfolio). На момент написания этого текста наиболее удобным и не зависящим от обстоятельств конкретного образовательного процесса способом поддержки веб-портфолио следует считать GitHub и GitHub Pages. По ссылке https://github.com/GossDemos/gossjs-sem2 доступен пример веб-портфолио в виде репозитория. При использовании GitHub Pages мы сопоставляем репозиторию автоматически построенный веб-сайт:
https://github.com/GossDemos/ITMO_labs_2 — репозиторий;
https://gossdemos.github.io/ITMO_labs_2/ — веб-сайт.

ГЛАВА 1. СИНТАКСИС И ОСНОВНЫЕ ПОНЯТИЯ JAVASCRIPT

Подготовка к лекции

Откройте консоль браузера и выполните console.log('Hello, world!');

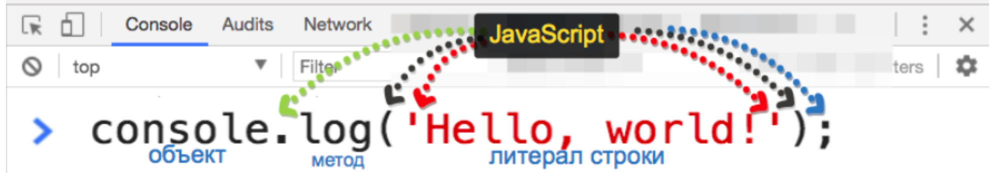


Рис. 2. Составные части инструкции на языке JavaScript

(См. примеры по адресам https://kodaktor.ru/js01_intro_lr.pdf и https://github.com/GossJS/js01_intro_lr.)

К языку JavaScript в этой строке относятся:

- точка;
- скобки;
- кавычки;
- точка с запятой.

Кавычки представляют в тексте программы **литерал** строки. Скобки в данном случае представляют **оператор** вызова функции. **Функция** представлена **именем** log. Эта функция является **методом**, потому что её имя указано путём **уточнения объекта** console.

Всё вместе — это **инструкция** JavaScript, которая в данном случае представляет собой **выражение** вызова функции с символом «точка с запятой».

Далее запустите консоль (терминал) и выполните в ней

```
node -v
```

Вы должны будете увидеть примерно следующее:

```
v10.2.0
```

В случае невозможности установить node на своём компьютере, можно воспользоваться одним из онлайн-инструментов, например, средой c9.io.

Итак, запустите node и убедитесь, что видите приглашение. Чтобы обрести полную уверенность в том, что это именно node, введите process.versions.v8 и нажмите Enter. Вы должны будете увидеть версию движка наподобие той, что указана выше.

Упражнение 1-1

Используя справку от node по объекту console и его методу log (https://nodejs.org/api/console.html#console_console_log_data_args), преобразуйте введённую инструкцию так, чтобы слово world оказалось в кавычках после запятой, а внутри первой пары кавычек был подстановочный символ («гнездо» для строки).

Упражнение 1-2

Используя справку по командной строке вашей операционной системы, найдите способ передать в запускаемый файл временное значение какой-нибудь переменной окружения.

Упражнение 1-3

Найдите способ передать в запускаемый файл параметр(ы) командной строки.

Упражнение 1-4

Создайте исполнимый файл на JavaScript, который можно запускать по его имени из любого места файловой системы, как если бы это была встроенная команда, которая выводит в консоль текущую дату.

Решение (*для MacOS*)

1. Создаём папку для хранения файла

```
mkdir ~/jsexe
```

2. Узнаём, где находится исполнимый файл node

```
which node
/usr/local/bin/node
```

3. Создаём сам файл:

(В этом сценарии используются синтаксические конструкции и идеи, обсуждение которых происходит далее в тексте пособия. Пожалуйста, рассматривайте его как образец материала, который станет понятен в будущем, после изучения нужных разделов.)

```
nano ~/jsexe/dater.js
```

```
1 #!/usr/local/bin/node
2 (async r => {
3         const execFile = require('util').promisify(require('child_process').execFile);
4         const { stdout } = await execFile(`date`);
5         console.log(stdout);
6     }
7 )();
```

(см. https://kodaktor.ru/g/dater.js).

Полученную в предыдущем пункте информацию о расположении исполнимого файла node мы используем в первой строке исполнимого файла dater.js в форме так называемого шебанга (shebang) — специального указания командному интерпретатору на то, с помощью какого исполнимого файла следует запускать данный файл.

4. Даём этому файлу право на выполнение

```
chmod +x ./dater.js
-rwxr-xr-x
```

5. Заносим в файл ~/.bash_profile добавление к PATH

```
export PATH=$PATH:/Users/eliasgoss/jsexe
```

6. Перезагружаем этот файл

```
. ~/.bash_profile
```

7. Запускаем наш файл из любой папки

```
dater.js
Sun Jan 28 01:39:03 MSK 2018
```

Можно также теперь быстро найти путь к этому файлу

```
which dater.js
/Users/eliasgoss/jsexe/dater.js
```

Обзор языка JavaScript

JavaScript — это язык программирования, который работает на разных платформах, самой популярной из которых в конце 2017 г. является браузер. На клиентской стороне веб-разработки он долгие годы был почти безальтернативен (если использовался другой язык, например, Dart, то происходила компиляция в JavaScript). С введением WebAssembly эта ситуация стала несколько сложнее и разнообразнее (это новый тип кода, подобный языку ассемблера, который может быть запущен в современных браузерах и позволяет разрабатывать программы на таких языках, как C с компиляцией в бинарный формат, а не в JavaScript, хотя JavaScript и WebAssembly могут успешно взаимодействовать).

Язык был создан программистом по имени Brendan Eich под первоначальным названием Mocha в 1995 г., затем был переименован в LiveScript и впоследствии получил название JavaScript. JavaScript не имеет отношения к Java, хотя имеет общие с Java части синтаксиса.

Его стандартизованная версия (спецификация) называется ECMAScript.

По опросам StackOverflow на JavaScript программируют свыше 60% ответивших. На момент написания этого текста был зафиксирован тренд роста этого числа.

Длительное время общепринятым стандартом был ES5, но в 2015 г. начал распространение ES6. Начиная с этого момента количество версий увеличивается. Они именуются по годам: редакция ES6 теперь также известна, как ES2015, а ES7 формально называется ECMAScript 2016. Нужно уточнять для версии браузера и nodejs, какой стандарт в каком объёме поддерживается (https://caniuse.com, https://kangax.github.io/compat-table/es6/ и т. д.).

Стандарт реализуется в так называемых движках, таких как V8 (Chrome, Node.js), Spidermonkey, Chakra (названия движков актуальны на декабрь 2017 г.).

JavaScript длительное время был известен как интерпретируемый скриптовый язык, однако современные движки используют сочетание компиляции и интерпретации, известное как just-in-time (JIT) компиляция.

Устройство движка V8 эволюционировало от сложной многокомпонентной схемы к простой двухкомпонентной (в конце 2017 г.):

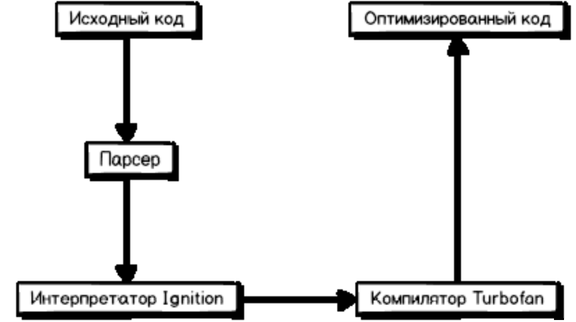


Рис. 3. Принципиальная схема работы движка V8

На момент написания этого текста стало фактическим стандартом веб-разработки написание кода на какой-либо опережающей версии с использованием функциональности, находящейся в стадии предложения или рассмотрения, и перевод этого кода на ES5. Процесс этого перевода называется транспиляцией, а наиболее популярным транспилятором является Babel. Транспиляции подвергается код, который пишется на TypeScript, Elm, ES2016 и т. п.

Ядро JavaScript определяет минимальный прикладной интерфейс для работы с текстом, массивами, датами, регулярными выражениями, но в нём отсутствуют операции ввода/вывода. Ввод/вывод, а также работа с графикой, медиа, сохранение данных – всё это реализуется окружающей средой, и, как правило, это — браузер. Но всё чаще это серверная среда NodeJS.

Вы можете использовать JavaScript чтобы писать скрипты автоматизации действий в операционной системе: создать файл .js прямо на рабочем столе Windows и управлять файлами, папками, ярлыками и прочим.

Существует огромное количество библиотек и фреймворков, расширяющих возможности так называемого ванильного JavaScript, которые возникают и исчезают, претерпевают изменения от версии к версии, сливаются друг с другом и т. д. Всё это в совокупности с такими инструментами, как Babel, Webpack, npm, nodemon и другими подобными образует огромную и постоянно изменяющуюся экосистему.

Консоль

Исторически первой средой исполнения JavaScript является браузер и DOM (Document Object Model, объектная модель документа). Созданный для «оживления» веб-страниц, этот язык изначально предоставлял средства для реагирования на события, в первую очередь возникающие в пользовательском интерфейсе в результате действий пользователя (например, щелчка мышью). В 2017 г. фирма Adobe объявила сроки прекращения поддержки платформы Flash, после чего нагрузка на JavaScript только возрастёт.

На рисунке ниже представлен документ с историческим первым способом подключения сценариев JavaScript к веб-странице — с помощью так назы-

ваемых событийных атрибутов типа onclick. Код на JavaScript при этом помещается в ограниченное кавычками (двойными по соглашению) значение этого атрибута.

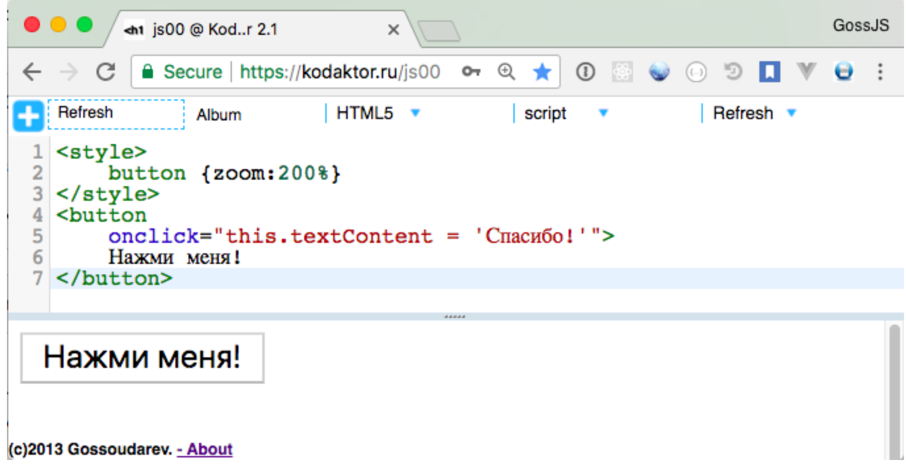


Рис. 4. Подключение JavaScript
к событийному атрибуту (https://kodaktor.ru/js00)

В этом фрагменте **this** — слово, которое является частью JavaScript и которое в разных ситуациях обозначает разное. Здесь — обращение к источнику события, т. е. к элементу button — кнопке.

Итак, мы используем атрибут элемента button, который называется onclick и содержанием которого является фрагмент кода на языке JavaScript.

В данном случае случае этим фрагментом является присваивание текстового значения (литерала строки) свойству textContent объекта, обозначенного словом this и по сути являющегося элементом button.

К языку JavaScript здесь принадлежат:

- this;
- точка;
- знак равенства;
- одинарные кавычки.

Остальное предоставлено средой браузера, DOM (браузер предоставляет и объект console и некоторые другие объекты). Объект — это сложный тип данных, структура, которая объединяет в себе доступные через имена значения разных типов. Они известны как свойства.

Если значением свойства является функция, то такое свойство называется методом.

Браузер реагирует на событие «щелчок» и передаёт управление коду на JavaScript. Подобное взаимодействие между языком и окружающей средой реализуется через API (Application Programming Interface, интерфейс прикладного программирования или интерфейс программирования приложений). DOM представляет собой первейший из таких интерфейсов, которых в распоряжении

JavaScript на сегодняшний день очень много. Забегая вперёд, отметим, что к ним, в частности, относятся:

- fetch;
- canvas;
- DOMParser.

Операторы, выражения и инструкции

Выражение и инструкция соотносятся как фраза и законченное предложение: результатом обработки выражения является вычисленное значение, но оно не обязано менять состояние программы.

Операторы связывают первичные выражения, которые более ни к чему не сводимы: **2 * 3** является примером выражения с **оператором умножения**, которое не имеет побочных эффектов.

Первичные выражения — это литералы, некоторые ключевые слова, ссылки на переменные. Под ссылкой на переменную понимается её упоминание, т. е. упоминание имени, т. е. непосредственное возникновение имени в составе выражения. Чтобы нечто могло быть воспринято как имя переменной (валидный идентификатор), оно должно начинаться с буквы, символа подчёркивания _ или знака доллара $.

Примеры допустимых имён:

- i;
- $PHPStyle;
- camelCaseVariableName;
- имяПеременнойBUnicode;
- λ.

Инструкция меняет состояние программы: она как бы **делает**, в то время как выражение **описывает** порядок вычислений над тем, что входит в его состав. Инструкция императивна, выражение функционально.

Примеры выражений:

- литералы null, false, 113, 0xBAD, 'hello' — литералы простых типов;
- литерал массива [null, false, 113, 0xBAD, 'hello'];
- литерал функции function veryTrue(){return true;} или просто ()=>true;
- имена переменных myName, arrClients;
- выражения с операторами
  a * 9 / 16
  ++b;
- выражения с вызовами функций, в том числе композиция;
- this.textContent;
- arrClients[0];
- console.log(1);
- new Date().

Полный список операторов ES5: https://kodaktor.ru/g/es5ops.

ECMAScript 2015 (ES6) добавил оператор rest/spread, который выглядит как многоточие. О нём подробнее в главе 6.

ECMAScript 2016 добавил оператор возведения в степень, который выглядит как две звёздочки. Ему сопутствует оператор, объединяющий это действие с присваиванием (см. https://kodaktor.ru/g/2_power):

```
let two = 2;
two **= 8; // two = two ** 8
console.log(two); // результат 256
```

Также разработчики используют операторы, добавляемые с помощью плагинов транспиляции (например, на момент написания этого текста, оператор bind, который выглядит как два двоеточия). Об этом подробнее в главе 9.

Программа на языке JavaScript пишется как последовательность инструкций, которые интерпретатор просматривает сверху вниз. Но фактическое выполнение часто происходит в другом порядке, в другой временнóй последовательности. Дело не только в обычной для программирования нелинейности ветвления, циклов и подпрограмм. Для JavaScript характерен нелинейный код, который выполняется в асинхронных функциях или коллбэках (функциях обратного вызова). Об этом подробнее в главе 5.

Попробуем ответить с разных точек зрения на вопрос: что такое переменная?

- Это символическое имя некоторого значения.
- Это то, что возникает при объявлении с помощью ключевого слова let (ранее var) или const.
- Это, в общем-то, отсылка к чему-то, находящемуся в памяти, например к функции.
- Это характерная часть программы, работающей за счёт сохранения состояний, т. е. манипулирования значениями, которые изменяются по ходу работы.

Именование переменных является частью принятого стиля кодирования. Рекомендуется выбрать некоторый удобный стиль и придерживаться его, например, camelCaseStyle, при котором имя переменной состоит из написанных латиницей слов без пробелов, каждое из которых начинается с заглавной буквы, кроме первого.

Любой дискурс переменных должен начинаться с объявления. Для этого используются слова let и const (в рамках legacy и для решения специфических задач — var).

Объявление можно рассмотреть как фиксацию имени в пространстве имён программы, блока или функции.

Объявление тесно связано с присваиванием. Возможны три ситуации:

1. Объявление с присваиванием (это начало жизненного цикла переменной).

2. Объявление без присваивания (для let и var; аналогично первому, но встречается реже, так как редко является полезным).

3. Присваивание без объявления (для let и var; это случай переприсваивания).

Третий вариант не должен встречаться до объявления, хотя и не вызывает ошибки (без включения строгого режима 'use strict'). Присваивание без объявления создаёт не столько переменную, сколько конфигурируемое (удаляемое) свойство глобального объекта:

```
a = 5;
console.log(global.a); // 5
console.log(a); // 5
delete a;
console.log(global.a); // undefined
console.log(a); // ReferenceError: a is not defined
```

В строгом режиме ошибка возникнет раньше, потому что налицо попытка оперировать незафиксированным именем.

Попытка упоминания необъявленной переменной приводит к ошибке **ReferenceError: … is <u>not defined</u> ...** в случаях, кроме упоминания после оператора typeof. Этот оператор возвращает undefined не только для объявленных без присвоения переменных, но и для необъявленных переменных...

```
console.log(typeof b);
let b;
```

... кроме случая, представленного в примере. Если убрать содержимое второй строки, будет выдано undefined, иначе программа прервётся с ошибкой ReferenceError. Такое поведение называется temporary dead zone (TDZ), т. е. вре́менной мёртвой зоной. Возможно, оно кажется не очень логичным, но логика проясняется, если знать, что слово var в таких ситуациях обеспечивает подъём (см. ниже).

Объявление с присваиванием — это, по-видимому, наиболее частый и корректный случай. Рассмотрим его (и само присваивание) подробнее.

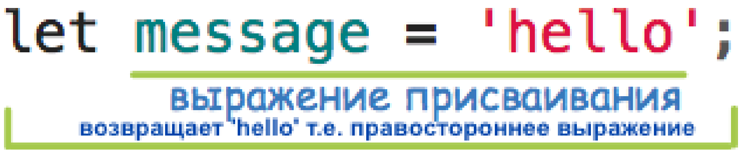


Рис. 5. Объявление с присваиванием

Присваивание — это выражение, получаемое с помощью оператора присваивания, который выглядит как знак равенства. Слева от знака равенства находится левостороннее выражение, а справа правостороннее. Примеры левосторонних выражений: имя, выражение с уточнением, выражение с индексацией, выражение с деструктуризацией (фигурными и/или квадратными скобками). Вычисление выражения присваивания возвращает результат вычисления правостороннего выражения. То есть оператор присваивания правоассоциативен (см. ниже о тернарном операторе!).

Если в правостороннем выражении применяется оператор «запятая», который объединяет в список несколько выражений присваивания, то возвращается результат самого правого.

```
let
  a = 3,
  b = 4;
// const result = a = 4, b = 3;
// SyntaxError: Identifier 'b' has already been declared
const result = (a = 44, b = 33);

console.log(`${a} ${b} ${result}`); // 44 33 33
```

В формальных документах для определения таких понятий используются грамматические конструкции типа формы Бэкуса — Наура и соответствующие им синтаксические диаграммы, например в стиле Д. Крокфорда:

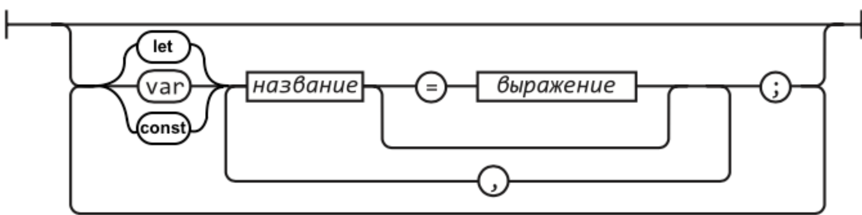


Рис. 6. Объявление с присваиванием: форма Бэкуса — Наура

Тот факт, что в результате вычисления выражения происходит влияние на состояние программы, называется **побочным эффектом выражения**. Побочный эффект присваивания заключается в том, что меняется значение переменной, т. е. получается, что главное назначение присваивания оказывается его побочным эффектом.

Выражение присваивания может рассматриваться как самостоятельная инструкция (expression statement), в этом случае предполагается, что ранее имело место объявление, а теперь происходит переприсваивание.

Объявление без присваивания создаёт имя, с которым связывается значение **undefined**. Условно можно считать, что это объявление с присваиванием значения undefined. В современном JavaScript оно доступно только с помощью слова let.

Присваивание можно рассматривать как изменение в той области памяти, которая ассоциирована с данным именем при объявлении, но если правостороннее выражение есть null, то уместнее считать, что происходит разрыв связи имени и этой области.

https://kodaktor.ru/undefined

Что такое область видимости?

Это границы (scope), в пределах которых доступна объявленная переменная. JavaScript всегда имел тенденцию рассматривать функцию (глава 4) как

основной строительный блок программы. Вообще блок — это фрагмент между открывающей и закрывающей фигурными скобками. Функция как блок в JavaScript традиционно играла особую роль изолятора имён, т. е. переменные, *объявленные с помощью слова var в блоке, объявленном с помощью слова function, не видны за пределами этого блока* (хотя они видны для вложенных фунцкциональных блоков, разумеется). Ситуация с тех пор сильно изменилась в смысле усиления роли блока как такового независимо от слова function.

Глобальные переменные — те, что объявлены на верхнем уровне (в рамках рассматриваемого кода). Количество глобальных переменных должно быть сведено к минимуму, в идеале — к нулю.

Объявления переменных с помощью слова var интерпретируются, как если бы они находились в начале функции (или глобальной области видимости, если объявление находится за пределами функции), независимо от того, где фактически находится объявление; этот эффект носит название поднятие переменных (hoisting: значение, если присвоено, остаётся в той строке (и ниже), где присвоено, а вот имя вместе с undefined поднимается наверх).

Объявления на уровне блока создают привязки (переменные), недоступные за пределами блока. Область видимости блока (block scopes), которую также называют лексической областью видимости (lexical scopes), — благодаря словам let и const — уравнена в правах с функциональной областью видимости. Пример «армия функций» (https://kodaktor.ru/army) хорошо показывает работу этой концепции, но задействует понятия, которые рассматриваются в главах 4, 5 и 6, поэтому мы вернёмся к нему позднее.

Операторы и их побочные эффекты

Рассмотрим присваивания:

```
let one = 1,
   five = 5;
```

Чему будет равно значение выражения:

```
++one + five++;
```

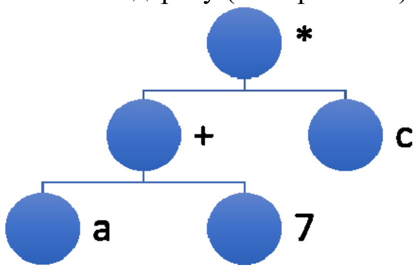
?

Побочный эффект префиксного унарного инкремента ++one (увеличение значения переменной) проявляется «сразу»: он возвращает новое (увеличенное на 1) значение переменной (2), а побочный эффект постфиксного унарного инкремента five++ проявляется «позже», возвращает он «старое» значение (5). Это играет роль только когда подобные операторы используются в составе выражений (т. е. это не важно, когда инкремент является инструкцией, например, в сигнатуре цикла).

Иными словами, префиксный оператор ++ или -- изменяет свой операнд перед выполнением любой другой операции, а постфиксные версии ничего не изменяют, пока все выражение не будет вычислено.

Упражнение 1-5

Запишите выражение по дереву (без пробелов):



О лексической структуре

Что такое синтаксис? Это правила построения слов в языке. Он задаётся грамматикой. Например, упомянутые выше BNF (формы Бэкуса — Наура). Здесь мы не будем их цитировать, а рассмотрим отдельные аспекты.

Лексическая структура языка — это низкоуровневый синтаксис: вид имён переменных, символы, используемые для обозначения комментариев и то, как одна инструкция отделяется от другой. JavaScript использует Unicode. И он чувствителен к регистру: например, слово **let** следует писать именно **let**, а не Let и не LET и т. п. Это касается имён переменных и функций и т. д. Набирать значения Unicode можно с использованием аппаратных возможностей данного компьютера и операционной системы (например, с использованием особых сочетаний клавиш и программ типа «Таблица символов») или с помощью Unicode-экранирования, при котором знак обратного слэша \ играет особую роль метазнака (чтобы получить буквально сам слэш, его литерал, нужно в таких случаях написать знак слэша дважды).

```
> let om = '\u0f68\u0f7c\u0f7e';

> om
'ཨོཾ'
```

Рис. 7. Получение символа, закодированного с помощью Unicode

JavaScript игнорирует пробельные символы \u0020, \u0009, \u00A0 и др. Пробелы широко используются для улучшения читаемости кода (см. рис. 18 и рис. 26 в главе 4, рис. 39 в главе 7).

Многие символы используются в разных ролях в зависимости от контекста. Например, квадратные скобки — массив, вычислимые свойства, деструктуризация. Фигурные скобки — блок, объект, деструктуризация, импорт-экспорт.

Сравните три случая использования синтаксиса:

литеральный объект	блок	импорт объекта и деструктуризация объекта
{ name: 'Elias' }	{ const name = 'Elias'; }	import { name } from './Name'; const { name } = obj;

Эти три примера синтаксиса используют фигурные скобки очень похожим образом. Первый вариант представляет синтаксис литерала объекта, второй — блока кода (в том числе как часть функции или конструкции наподобие цикла), третий — импорта имени из модуля или ключа из объекта путём деструктуризации. Фигурные скобки — весьма часто используемые символы низкоуровневого синтаксиса JavaScript.

Комментарии желательно использовать в процессе обучения и при совместной работе над кодом.

```
// это однострочный комментарий
 /* это комментарий
в нескольких
строках
*/
```

JavaScript игнорирует комментарии, но в некоторых специфических случаях, типа помещения литерала регулярного выражения в многострочный комментарий, могут возникнуть трудноуловимые ошибки.

Следует снабжать свой код комментариями, и помнить, что особым образом оформленные комментарии могут служить метаданными для построения документации (JSDoc и др.).

К числу зависимостей, подобных eslint, позволяющих более грамотно организовать работу с кодом, относится jsdoc (см. упражнение в главе 2).

Зарезервированные слова тоже могут стать источником затруднительных ситуаций. Некоторые идентификаторы играют роль ключевых слов самого языка и не могут быть идентификаторами в программах. Так, слово class было зарезервировано много лет назад, но стало использоваться в программах JavaScript только с 2015 г.

Следует стараться, чтобы используемые имена как можно меньше пересекались с любыми «служебными» словами.

Точки с запятой формально необязательны, но их игнорирование может привести к неоднозначным ситуациям. Рассмотрим код:

```
let a
a
=
3
```

Это будет интерпретировано без ошибок как

```
let a; a = 3;
```

…ибо фрагмент let a a нельзя безошибочно интерпретировать без точки с запятой.

НО в следующем примере ситуация не так однозначна:

```
let y, x = 'b'
y = 'a' + x;
(y + 'cd').split('').forEach( x => console.log(x))
```

Здесь точка с запятой обязательна, и если её убрать, то интерпретатор истолкует часть кода как

```
x(y + 'cd')
```

и мы получим ошибку **x is not a function**, потому что в x хранится строка, а интерпретатор пытается вызвать x как функцию. И здесь хотя бы возникает сообщение об ошибке. В несколько искусственной ситуации типа

```
let y, x = () => 'b'
```

программа работала бы без ошибок и с точкой с запятой, и без, но понять, как она должна работать, было бы ещё труднее.

Иногда встречается такая черта стиля кодирования, фокусирующая внимание на точках с запятой и запятых, как постановка их в начале строк:

```
const
   a = 1
 , b = 2
 , c = 3
 ;
```

И, как крайний случай, может быть вариант стиля кодирования без точек с запятой. Выбор за читателем: приверженцы каждого стиля могут привести весомые аргументы в пользу своих предпочтений.

Методическое примечание

В тексте пособия встречаются скриншоты из редакторов онлайн-кода с разной подсветкой, а также из консоли. В последнем случае следует иметь в виду, что происходит пошаговое выполнение команд, которые отображаются со знаком «больше». Однако этот знак не следует вводить перед набором команды, он нужен при отображении только чтобы отличать команды от результатов их исполнения. Кроме того, в этом режиме для вычисления выражения достаточно набрать его и нажать Enter, вместо того чтобы запускать метод log объекта console:

```
$ node
> let a = 1 + 2;
undefined
> a
3
```

ГЛАВА 2. ЗАВИСИМОСТИ

Зависимости являются неизбежно возникающим феноменом при решении любой достаточно сложной задачи, приводящем к написанию кода, который используется более одного раза в практически неизменном виде. Зависимости в простейшем случае возникают примерно так: мы получаем повторный код, выделяем его в функцию, функция отчуждается в отдельный файл (модуль), модуль выгружается в какой-либо репозиторий, и далее, чтобы использовать такой модуль уже, вполне вероятно, другие программисты должны будут его оттуда загрузить («затребовать», «импортировать», «подключить»). Тогда модуль становится зависимостью (dependency) для проекта. На самом деле зависимостью становится целый проект, который с точки зрения файловой структуры может быть устроен сколь угодно сложно и сам включать много «этажей» зависимостей.

В зависимостях заключается сила и слабость проектов. Например, браузер FireFox приобрёл огромную популярность благодаря своим плагинам. Но плагины существуют достаточно самостоятельно, у них свои разработчики, которые вольны распоряжаться ими по своему усмотрению. Плагин может измениться, исчезнуть, стать вредоносным... В мире Node и JavaScript похожая ситуация, которую превосходно иллюстрирует **история 11 строк кода** (см. более подробно по адресу https://github.com/GossJS/11lines).

В современном JavaScript всё, чем мы пользуемся для создания кода (иногда включая даже редактор кода!), является зависимостью. Их диапазон распространяется от удобных функций типа той, о которой идёт речь в упомянутой выше истории, до мощных инструментов сборки проекта и проверки синтаксиса.

Обычно сначала рассматривают уже имеющиеся зависимости, а затем приступают к созданию своих собственных. Здесь есть ряд сложностей.

1. Программное манипулирование зависимостями требует использования специфических конструкций, длительное время отсуствовавших в JavaScript.

2. Исторически сложилось так, что браузерный и консольный JavaScript манипулируют зависимостями по-разному, и эта ситуация только сейчас (начало 2018 г.) начинает выравниваться.

Мы начнём с зависимостей, которые можно использовать не в программе, а в командной строке. Они не требуют использования этого специфического синтаксиса.

Менеджмент зависимостей

Менеджер зависимостей — это программное обеспечение, позволяющее загружать зависимости из репозиториев и в них отслеживать версии, разрешать конфликты и так далее. На момент написания этих строк в основном используются программы npm и yarn. Возможно, к лету 2018 г. уже приобретёт распространение новое решение — turbo, претендующее на то, чтобы ещё больше

уравнять в правах браузерную и небраузерную платформы. Но возможно и возникновение других, сейчас трудно представимых, ситуаций.

Как происходит работа с современным JavaScript-проектом?

Она начинается с создания папки, в которой возникает файл package.json. Формат JSON мы кратко рассмотрим здесь, а затем вернёмся к более детальному рассмотрению, когда речь пойдёт об объектах JavaScript в форме литерала (plain JavaScript objects).

JSON (JavaScript Object Notation) — формат текстовых файлов, получивший широкое распространение, в том числе как заместитель XML. С его помощью осуществляется взаимодействие между приложениями, отдача информации от сервера клиенту и обратно, хранение настроек в манифествах типа package.json. Это декларативный формат, части которого объявляют или описывают свойства или требования к работе программного обеспечения.

По виду эти файлы напоминают объекты JavaScript в литеральной форме (см. выше и в главе 7 — в виде пар «ключ-значение» через запятую, помещённых между фигурными скобками). На информацию, представляемую как JSON, накладываются более жёсткие ограничения.

Файл package.json нужен в первую очередь для описания проекта, который в потенциале может стать зависимостью в других проектах. Самые важные разделы этого файла есть name и version, без которых система npm не станет устанавливать пакет. Но даже если проект и не будет зависимостью, этот файл в любом случае полезен и удобен для хранения данных о проекте.

Документация по package.json (https://docs.npmjs.com/files/package.json) базируется на рекомендациях системы модулей CommonJS (глава 9), и она на момент написания этого текста не носит формального характера. Вообще файлы в формате JSON, которые используются для настройки работы какого-либо программного обеспечения (package.json, .babelrc, .eslintrc и т. п.) могут быть определены соответствующей схемой (схемы при этом сами представляют собой JSON-документа). Вот пример схемы для package.json: https://kodaktor.ru/j/package.json.schema (см. также, например, https://habrahabr.ru/company/otus/blog/350116/).

Схему можно спроектировать как бы умозрительно, до появления фактических документов («дедуктивно») или сгенерировать по существующему документу, используя его как прототип («индуктивно»). Например, если у нас есть документ в борде https://kodaktor.ru/j/users, то сгенерировать схему для него можно с помощью запроса https://kodaktor.ru/api/schema/users.

Соответственно, отвалидировать документ относительно (agains) схемы можно с помощью таких зависимостей, как jsonschema, который используется на сайте kodaktor.ru следующим образом. Рассмотрим документ по адресу https://kodaktor.ru/j/**userserr** и схему по адресу https://kodaktor.ru/j/**e69ac4e**, тогда запрос https://kodaktor.ru/api2/schema/**userserr**/**e69ac4e** позволит узнать, соответствует ли этот документ этой схеме. Мы увидим сообщение об ошибке (users[0].login is not of a type(s) string), перейдя по этому адресу, потому что в первой записи значение logina заявлено как число, а в схеме указано, что это должна быть строка.

Мы рассмотрим такие примеры, не требующие программной манипуляции:

- вывод различий в коде с подсветкой;
- запуск сценариев с возможностью автоматического перезапуска при внесении изменений кода;
- отслеживание кода на предмет потенциально опасных фрагментов (линтинг).

Также в режиме инструментов командной строки, как правило, работают транспилятор babel и бандлер webpack, широко применяемые для получения готового к работе в продакшн кода, и многие другие зависимости. В первую очередь это актуально при использовании таких фреймворков как Vue или Angular, таких библиотек, как React и других решений с собственной инфраструктурой и языковыми расширениями.

Методическое примечание

Выше в тексте пособия было упражнение на изменение способа работы с методом console.log. При изложении этого материала в лекционной форме следует создать опрос, например, в Гугл, и задать это упражнение для интерактивного выполнения. https://kodaktor.ru/g/logform

Воспользуемся Git и инструментом diff2html-cli для наглядного отображения разницы между первым вариантом кода и вторым.

Для этого:

1. Создадим отдельную папку для работы с зависимостью diff2html-cli и, войдя в неё, инициализируем проект

```
mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && yarn init -y
```

результатом чего станет создание файла package.json — в таблице ниже приведены варианты исполнения соответствующих команд инициализации проект файлом по умолчанию, когда npm или yarn заполняют разделы файла самостоятельно.

<table>
<tr><th>npm init -y</th><th>yarn init -y</th></tr>
<tr><td><pre>{
  "name": "diff",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}</pre></td><td><pre>{
  "name": "diff",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT"
}</pre></td></tr>
</table>

2. Добавим diff2html-cli как девелоперскую зависимость

```
npm i -D diff2html-cli
```

(самостоятельно выясните значение флагов i и -D и соответствующий эквивалент при использовании yarn)

Когда зависимость успешно установлена, то запись об этом добавляется в файл:

```
"devDependencies": {
  "diff2html-cli": "^2.5.4"
}
```

3. Поместим каждую из строк

console.log('Hello, world!');
console.log('Hello, %s!', 'world');

в отдельный файл (1.js и 2.js) и выполним команду:

4.

git diff --no-index 1.js 2.js | diff2html -d word -s line -i stdin

Результат уже был показан выше, он также помещён в пример ***diff1***.

Мы видим, как детально отображается информация об изменениях. Зелёным фоном помечены результаты добавления (строки и фрагменты строк), красным — удаления.

Если инструмент используется достаточно часто и, кроме того, мы осознаём, что его версии меняются и необходимо следить за ними, то можем установить зависимость не в данную папку, а глобально:

npm install -g diff2html-cli
yarn global add diff2html-cli

Мы также можем извлечь информацию о хранении глобальных зависимостей:

yarn global dir
npm list -g --depth=0 --silent

Первая команда (по состоянию на май 2018-го) показывает расположение файла package.json (который нужно потом посмотреть отдельно), а вторая выводит список.

Или вот пример использования yarn для выяснения того, какие зависимости установленны в данном проекте:

yarn list --depth=0

Выше говорилось, что даже текстовый редактор может рассматриваться как зависимость. Примером такого редактора является slap.

Рис. 8. Редактор slap полностью написан на JavaScript

Когда та или иная зависимость не установлена на данном компьютере, её можно как бы временно установить с помощью команды npx: npx slap package.json.

Запуск зависимостей

Локально зависимости устанавливаются в папку node_modules. Для запуска подлежащих исполнению файлов обычно нужно указывать полный путь к ним. Альтернативой является создание раздела scripts, где файлы можно указывать без пути. Что же касается команды npx, то с её помощью можно обращаться к зависимостям, установленным в папке node_modules, без полного пути, только по имени.

Следует также обратить внимание на инструмент nodemon, с помощью которого можно запустить код так, что он будет перезапускаться при сохранении изменений.

Здесь нужно ещё отметить, что различие между обычными (dependencies) и девелоперскими (devDependencies) зависимостями оказывается ничтожным в случае использования таких инструментов сборки проектов (бандлеров) как Webpack. Изначальная идея состоит в том, что девелоперские зависимости нужны только на этапе разработки, в девелоперской среде. Вне всяких сомнений, линтер является типичным примером девелоперской зависимости. Некоторый отдельный случай представляют фреймворки тестирования. Но для бандлера все зависимости фактически являются обычными (dependencies), и он самостоятельно решает, какие зависимости должны быть включены в состав результирующего набора файлов.

Здесь приведены некоторые настройки и инструкции по использованию eslint: https://github.com/GossJS/eslint.

Используя линтинг, можно избежать таких ситуаций, которые приведены ниже:

```
semicolon.js ×
1  const a = 1;
2  const b = 2 + a;
3  (() => console.log(b))(); // 3
4  const c = 4 + b        Интерпретатор видит b(...)()
5  (() => console.log(c))();
6      Unexpected newline between function and ( of function call. (r

$ node semicolon.js
3
/Users/eliasgoss/Yandex.Disk.localized/docs-1111111/js-2017/js_samples/
const c = 4 + b
              ^

TypeError: b is not a function
```

Рис. 9. Нахождение трудноуловимой ошибки с помощью линтера

А вот пример возникновения упомянутой выше потенциально трудноотслеживаемой ошибки при совпадении использования регулярных выражений и многострочных комментариев.

```
1   /*
2   хороший комментарий
3   */
4
5   let reg = /.*/;
6   console.log('a'.match(reg));
7
8   /*
9   let reg = /.*/;
10  console.log('a'.match(reg));
11  */
12      Parsing error: Unexpected token * (Fatal)
```

Рис. 10. Неоднозначность с комментарием и регулярным выражением

Приобрести привычку использовать инструмент типа eslint крайне необходимо, по возможности это должно стать рутинной частью жизненного цикла разработки. Но при этом нужно помнить, что подобные инструменты не являются панацеей. Например, eslint в вышеприведённом примере просто показывает, что символ звёздочки в этом месте не ожидается. Программисту остаётся самому догадаться, что в строке 9 в составе регулярного выражения оказалась последовательность, которая интерпретируется как завершение комментария.

Инструкции по установке и настройке eslint для редактора Atom приведены в главе 4, посвящённой функциям.

Что касается перезапуска сценариев, то для этого в режиме разработки хорошо подходит инструмент nodemon (см. главу 7).

Сборка проекта обсуждается в главе 9.

На платформе Node.js мы (по крайней мере пока) можем использовать существующие модули, включая оформленный соответствующим образом собственный код, с помощью директивы require. Она относится к этапу развития небраузерного JavaScript, на котором возникла необходимость создавать изолированные блоки кода на уровне файлов, предоставляющие строго определённую функциональность и инкапсулирующие реализацию остального.

Функция (глава 4) require работает синхронно, т. е. на время, необходимое для обработки модуля, выполнение импортирующего сценария приостанавливается. При этом происходит обёртывание импортирующего сценария в функцию, что и обеспечивает изоляцию кода от остальных частей сценария.

```
(function (exports, require, module, __filename, __dirname) {
  // сюда помещается импортируемый код
});
```

Затем эта функция исполняется.

Упражнение 2-1

Сравните работу require с require и require_once языка PHP.

Методическое примечание

Это упражнение формулируется здесь с целью создать «ссылку на будущее», к главам 4, 5 и 6.

index.js
```
const { qv } = require('./funcs');
console.log(qv(5)); // 25

```

funcs.js
```
console.log('Good!');
const qv = x => x * x;
const cb = x => x ** 3;
module.exports = {qv, cb};
```

index.php
```
<?php
  $qv = require_once(getcwd() . '/src/funcs.php');
  echo( $qv(5)); // 25
  echo( $cb(5)); // 125  это тоже экспортировалось
```

funcs.php
```
<?php
  echo('Good!');
  $cb = function($x){ return $x ** 3; };
  return function($x){ return $x * $x; };
```

Синхронность исполнения обсуждается ниже в главе 5 при переходе к теме коллбэков и таймеров.

Упражнение 2-2

Используя встроенный модуль crypto, получите md5-хэш какой-либо строки.

Упражнение 2-3

Установите зависимость jsdoc, оформите любой пример (func-jsdoc.js) по соглашениям jsdoc и сгенерируйте для него документацию.

```
/** @const {string} */ const hiWord = 'Hello';


/**
* возвращает приветствие
* принимает строку или ничто, в случае ничто возвращает безымянное приветствие
*/
function sayHello(name = 'Nameless') {
  return `${hiWord}, ${name}!`;
}
console.log(sayHello());
console.log(sayHello());
```

Выполнение npx jsdoc func-jsdoc.js приведёт к созданию папки out с некоторым сайтом, включающим страницу:

file:///Users/eliasgoss/Yandex.Disk.localized/docs-1111111/js-2017/js_samples/posobie/out/global.html#hiWord

Global

Members

(constant) hiWord :string

Type:

- string

Source: func-jsdoc.js, line 1

Methods

sayHello()

возвращает приветствие принимает строку или ничто, в случае ничто возвращает безымянное приветствие

Source: func-jsdoc.js, line 7

Documentation generated by JSDoc 3.5.5 on Tue May 01 2018 20:56:38 GMT+0300 (MSK)

Home

Global

hiWord
sayHello

ГЛАВА 3. ПЕРЕМЕННЫЕ И ТИПЫ ДАННЫХ

JavaScript характеризуется (слабой) динамической типизацией: переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной; в различных участках программы одна и та же переменная может принимать значения разных типов. Типизация в JavaScript также характеризуется как неявная (утиная) — проверка типов выполняется во время выполнения программы.

Чтобы обеспечить более строгий контроль типов можно использовать диалект TypeScript или зависимость Flow, но это требует дополнительной настройки проекта в аспекте транспиляции (глава 9).

Хотя мы не объявляем типы, JavaScript следит за значениями и хранит их по-разному. В этом смысле типы, конечно, есть. Рассмотрим две категории типов: примитивные и ссылочные. В современном JavaScript 6 примитивных типов:

1) null;
2) undefined;
3) Boolean;
4) Symbol;
5) Number;
6) String.

Ссылочный тип в некотором смысле один, это объекты. Но объекты (глава 7) могут строиться по-разному. Это могут быть собственно объекты (литералы объектов), массивы и функции. Кроме того, существуют специфические типы объектов, такие как типизированные массивы, Date, Set или Map со своим «профессиональным» поведением.

Если объявить переменную и связать с ней значение ссылочного типа, а затем объявить другую переменную и связать её с именем первой переменной, то другая переменная станет ссылаться на то же значение, на которое ссылается первая.

Примитивные значения иммутабельны. Нельзя изменить часть числа (например, один бит в его двоичном представлении) или строки. Можно создать новое число или строку и переприсвоить уже существующему имени (если это имя не объявлено с помощью const). При оперировании примитивными значениями, кроме null и undefined, интерпретатор создаёт вокруг них прозрачные объектные обёртки с конструкторами (глава 8), имена которых совпадают с именами типов: Number, String.

К типу Number принадлежит специфическое значение NaN (not a number).

Объекты сравниваются по ссылке (по ссылкам, т. е. проверяется, ссылаются ли имена в выражении сравнения на один и тот же объект), а примитивные значения сравниваются друг с другом.

Преобразование типов

Осуществляются явные и неявные преобразования типов. Явное преобразование — это вызов функций Number(…), String(…), Boolean(…), Object(…) без слова new (т. е. не в роли конструктора обёрток).

Все значения, кроме null/undefined имеют метод toString(), обычно дающий то же, что возвращает функция String(…). Существует алгоритм представления значений с помощью toString() и ещё одного метода valueOf(), который в большей степени связан с представлением в виде числа, но чаще всего возвращает то, относительно чего был вызван.

Неявные преобразования осуществляются, например, при попытках сравнивать значения с помощью двойного равенства или использовании бинарного инфиксного плюса. Бинарный инфиксный плюс отдаёт предпочтение строкам. Таблица правил преобразования приведена по адресу https://kodaktor.ru/g/types.

- Если один из операндов инфиксного + является строкой, то второй тоже преобразуется в строку.
- Унарный + преобразует свой операнд в число аналогично функции Number().
- Унарный ! преобразует операнд в логическое значение (см. 6 значений, дающих false) и инвертирует его. Двойное применение унарного восклицательного знака преобразует в булев тип, т. е. !!x есть то же что Boolean(x).

Почему +!![] даёт результат 1, а +[] преобразуется в 0? Потому что !![] есть преобразование пустого массива в булев тип, и по правилам это true, а +true это преобразование истины в число, и по правилам это 1. А +[] есть преобразование пустого массива в число, и по правилам это 0.

+!![]	+[]
Number(Boolean([]))	Number([])
1	0

Во что преобразуется в логическом выражении new Boolean(false)? В true, так как получится объект (обёртка), а любой объект преобразуется в true в контексте логического выражения.

Метод toString(..) класса Number может преобразовать в представление в системе счисления с основаниями от 2 до 36; функция Number() ждёт литерала целого или вещественного числа; а parseInt/parseFloat пытаются разобрать максимально возможное количество символов числа и игнорируют всё, что следует за ними.

```
console.log(
  parseInt('BAD', 16),           /* 2989 */
  Number('BAD'),                 /* NaN */
  Number('0xBAD'),               /* 2989 */
  (2989).toString(16).toUpperCase() /* 'BAD' */
);
```

В JavaScript не делается различий между целыми и вещественными значениями. Все числа представляются вещественными значениями (с плавающей точкой в 64-битном формате, определяемом стандартом IEEE 754), но способ хранения числа, которое может быть интерпретировано как целое, отличается от способа хранения числа, которое не может быть интерпретировано таким образом. В ES2015 появился метод Number.isInteger(), который может определить, является ли указанное значение целым числом. Когда в вызов этого метода передается значение, по его внутреннему представлению движок JavaScript определяет, является ли оно целым числом. В результате числа, которые выглядят как вещественные (например, 7.0), в действительности могут храниться как целые, и для них этот метод будет возвращать true.

Если представить число, умещающееся в 8 двоичных разрядов, как обычное число JavaScript, напрасно будет потеряно 56 бит. Для более эффективного использования памяти и, возможно, для расширения возможностей метапрограммирования в ES2015 были введены типизированные массивы, о которых мы говорим в главе 6.

Неявное преобразование типов приводит к ситуациям, которые многим наблюдателям кажутся странными или парадоксальными («wat?!») — например, мы наблюдаем нарушение коммутативности оператора + при применении его к таким операндам, как пустой массив и пустой объект в разном порядке:

```
> {}+[]
0
> []+{}
'[object Object]'
>
Node.js CLI
```

```
> {}+[]
< 0
> []+{}
< "[object Object]"
>
Chrome console
```

Рис. 11. Мнимая некоммутативность оператора +

Тем временем интерпретатор руководствуется чётко обозначенными правилами, и эти два случая только кажутся похожими — в первом случае действует **унарный** плюс, а во втором **бинарный**.

Анализируя строку {}+[] интерпретатор не воспринимает первую пару фигурных скобок как литерал объекта. Вместо этого он «видит» пустой блок и игнорирует его. Далее он исполняет унарный плюс перед пустым массивом и согласно правилам представляет его как число нуль.

Чтобы увидеть нормальную коммутативную работу бинарного плюса, мы должны оба выражения заключить в скобки:

```
> ({}+[])
'[object Object]'
> ([]+{})
'[object Object]'
>
```

```
> ({}+[])
< "[object Object]"
> ([]+{})
< "[object Object]"
>
```

Рис. 12. Снятие неоднозначности

В главе 7 объясняется, что такое [object Object].

На собеседованиях часто задают каверзные вопросы, приводя примеры типа

https://habrahabr.ru/company/ruvds/blog/347530/

Поскольку речь идёт о языке программирования — формальной системе, которая определяется аксиомами и правилами вывода, в конечном счёте после всех чисто механических преобразований дело сводится к двум предпосылкам:

(а) результат следует из применения правила преобразования;

(б) так буквально сказано в спецификации языка ECMAScript или другого используемого стандарта (наподобие IEEE 754 для дробных чисел).

Например, почему оператор typeof возвращает number (число) для значения **NaN** (Not-a-Number, не-число)? Стандарт ECMAScript-262 говорит обо всех не-числах как о числах, поэтому их и называют числами. Примеры NaN:

- +undefined;
- ++undefined;
- Math.sqrt(-1);
- Number('one');
- parseInt('two').

По этой же причине значение NaN не равно (как строго, так и нестрого) **ничему**, включая себя.

В JavaScript понятие сравнения может относиться к операторам строгого (===) и нестрогого (==) равенства, к методу Object.is и к операторам >, <, <=, >=. Хотя в двух последних операторах фигурирует знак равенства, они являются просто отрицаниями, соответственно, первых двух.

Рассмотрим значения:

- undefined;
- null;
- 0;
- -0;
- NaN;
- "" (пустая строка).

Все они при преобразовании в логическое значение «работают» как **false**, а остальные — как **true**.

По адресу
https://developer.mozilla.org/enUS/docs/Web/JavaScript/Equality_comparisons_and_sameness
приведены длинные таблицы, показывающие результаты применения к различным операндам различных операторов и алгоритмов сравнения.

На практике можно дать простую рекомендацию: использовать только строгое сравнение и внимательно следить за тем, какие значения сравниваются и почему.

Кроме того, надлежит чётко осознавать природу сравнения значений: например, сравнивать два литерала массива или объекта заведомо бессмысленно.

[1, 2] == [1, 2]
false
[1, 2] === [1, 2]
false
	const a = [1, 2];
	const b = [1, 2];
	console.log(a == b); // false

В то же время

```
const a = [1, 2];
const b = a;
console.log(a == b); // true
```

Чем объясняется такое поведение?

Тем, что, когда мы присваиваем имя a имени b, мы не создаём новый массив и не копируем массив из области памяти, ассоциируемой с именем a в область памяти, ассоциируемую с именем b. Просто теперь оба имени (a и b) ассоциированы с одной и той же областью памяти, они стали синонимами. И это касается всех ссылочных типов (т. е. не-примитивных). Запись a === b означает проверку синонимичности операндов, и в данном случае эта проверка успешна.

Тернарный оператор и ветвление

Пусть у нас есть алгоритм, генерирующий случайное число от 0 до 100. Используем тернарный оператор для того, чтобы идентифицировать, выпало ли число 50, находящееся в середине этого диапазона:

```
const
  a = 0,
  b = 100,
  res = Math.floor(a + Math.random() * (b - a + 1));

const middle = res === 50 ? 'center' : 'not center';

console.log(middle);
```

res === 50 ? 'center' : 'not center'

Операнд 1 условие

Операнд 2 значение, возвращаемое при истинном условии

Операнд 3 значение, возвращаемое при ложном условии

Рис. 13. Операнды тернарного оператора

Тернарный оператор называется так потому, что имеет три операнда, между которыми располагаются его части (вопросительный знак и двоеточие).

Упражнение 3-1

Рассмотрите код в строках 11-13 на странице https://kodaktor.ru/ternary

```
const age = 17;
const restricted = ( age < 18 ) ? 'yes' : 'no' ;
Out.log( restricted );
```

и доработайте его так, чтобы переменная restricted принимала не одно из двух, а одно из трёх различных значений: (а) значение yes при значении переменной age меньше 18; (б) значение notsure при значении переменной age, равном 18; (в) значение no в противном случае.

Это нужно сделать вкладыванием одного тернарного оператора в другой.

Упражнение 3-2

Так как в JavaScript существуют значения, которые нестрого равны друг другу при неявном приведении типов к логическому (они приводятся к false и называются falsy, «ложностные»), а одно из этих значений ещё и не равно самому себе, то нужен способ отличать их друг от друга.

Функция isNaN тоже занимается неявным приведением. Так, значение true возвращается вызовами isNaN() и isNaN('e').

(При этом отметим, что Math.sqrt(-1) *не приводится* к NaN, а в точности **есть** NaN, так же как литерал значения NaN, выглядящий в программе как **NaN**.)

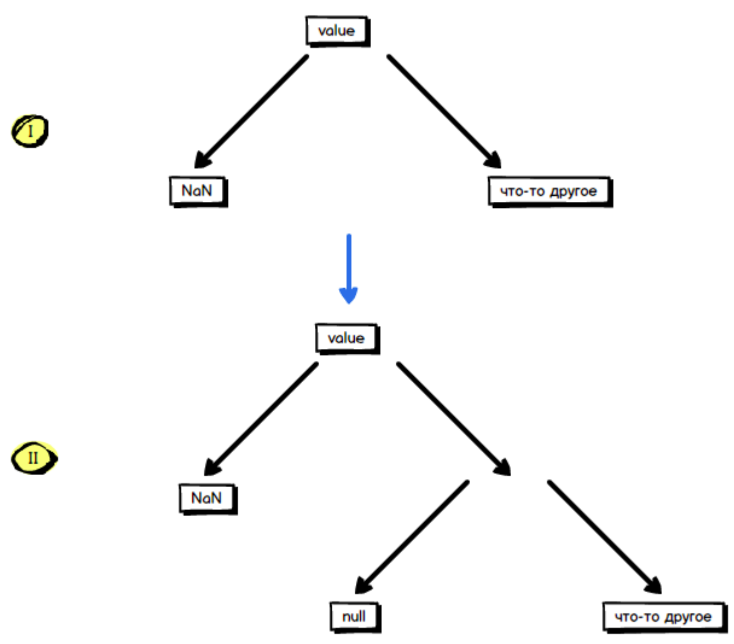
С использованием *операторов* напишите тернарный оператор, возвращающий:

'=NaN', если тестируемое значение **в точности есть** NaN,
'=null, если если тестируемое значение в точности есть null,
'=undefined, аналогично,
'=0', аналогично.
'=""', в случае пустой строки

и

'=false' в случае значения Boolean False.

Для этого поэтапно спроектируйте дерево вида



Примечание 1

Метод Object.is позволяет осуществлять проверки типа тех, о которых идёт речь в упражнении, но он относится к другой теме (глава 7), поэтому здесь мы ограничиваемся разговором об операторах.

Примечание 2

Как и оператор присваивания (см. выше) тернарный условный оператор в JavaScript правоассоциативен. То есть выражение a = b = c эквивалентно a = (b = c), а не (a = b) = c.

В то же время в таких языках как PHP можно встретиться с примером левоассоциативности, и именно в случае тернарного оператора. Чтобы добиться такой же работы условного оператора, как в JavaScript, в PHP необходимо явно расставлять порядок выполнения скобками.

Императивным аналогом тернарного оператора является традиционная инструкция ветвления if, а её специфическим частным случаем является инструкция выбора одного из вариантов switch.

Императивная инструкция не возвращает значения сама по себе, она только переключает поток выполнения в ту или иную ветку.

В зависимости от постановки задачи нам может оказаться нужной только одна ветка или обе. Если нам важно, чтобы какие-то действия выполнялись только при выполнении условия (либо только при его невыполнении, равнозначном выполнению отрицания этого условия), и не важно, что будет происходить в противоположной ситуации, мы можем обойтись только одной веткой, т. е. сокращённой формой инструкции. Тогда (только) при выполнении условия произойдёт указанное действия и то, что следует дальше.

if1.js
```
const a = 19; и больше
if (a > 18) {
  console.log('permit');
} else {
  console.log('restrict');
}
// permit
```

if2.js
```
const a = 19; и больше
if (a > 18) {
  console.log('permit');
}
console.log('restrict');

// permit
// restrict
```

условие выполняется

только одно действие | оба действия

if1.js
```
const a = 18; и меньше
if (a > 18) {
  console.log('permit');
} else {
  console.log('restrict');
}
// restrict
```

if2.js
```
const a = 18; и меньше
if (a > 18) {
  console.log('permit');
}
console.log('restrict');

// restrict
```

условие не выполняется
только одно действие

Рис. 14. Варианты структурирования ветвления

Упражнение 3-3 (полнота)

Исполнитель [алгоритмов] называется полным по Тьюрингу, если с его помощью можно реализовать любую вычислимую функцию, т. е. если он совпадает «по диапазону» задач с машиной Тьюринга (и прочими формализмами).

Строго говоря, ни один вычислитель не будет полным без бесконечного объема памяти, однако этим требованием можно пренебречь, если есть средства для обращения к памяти, не ограниченные ее объемом, например, «бесконечное» наращивание доступной памяти по мере требования.

Рассмотрим «шесть волшебных символов»: []()+! (пара квадратных скобок, пара круглых скобок, плюс и восклицательный знак).

Можно ли создать полный по Тьюрингу язык, используя только эти символы?

Пусть нам известно, что:

- ![] есть false;
- !false есть true;
- +true есть число 1;
- x+y есть сумма чисел x и y;
- пробел может иметь значение.

Теперь запишите, пожалуйста, число 2 с помощью символов [ ] + !

Упражнение 3-4 (внешние переменные)

Определите, как с помощью объекта process и его свойств argv и env получать доступ к переменным окружения и параметрам вызова командной строки. Чем являются переданные значения с точки зрения типа данных?

Глава 4. Функции

Функция — это в общем случае параметризованный код, т. е. код, в котором некоторые изначально фиксированные значения заменяются на «подстановочные гнёзда», в которые тем или иным способом можно помещать конкретные значения — потенциально новые при каждом вызове такого кода. Функция — это, таким образом, шаблонный код. Как и тело цикла, код функции появляется из желания избежать дублирования одинаковых фрагментов кода. Но простейший цикл — это просто замена N совершенно одинаковых фрагментов на один такой фрагмент, «обёрнутый» командой типа «Повтори N раз». А функция — это **именование** фрагмента кода, которое делает возможным его вызов по этому имени, т. е. замена N фрагментов на N упоминаний этого имени. Сам фрагмент при этом располагается некоторым образом отдельно и может вообще не выполниться, если вызовов не последует.

Параметризация цикла — это введение переменной типа счётчика, которая может упоминаться в теле цикла и использоваться для вычислений. Параметризация кода функции происходит аналогично. В каждом конкретном случае её может и не быть. Функция может выполнять какие-то действия независимо от параметров. Ещё одним отличием от цикла является возврат значения.

Выражение вызова функции в классическом случае вычисляется (выполнение последовательности шагов программы приостанавливается, начинает выполняться код функции с передачей в её параметры указанных в операторе вызова значений) и затем возвращённое функцией значение подставляется вместо выражения вызова, а выполнение шагов продолжается с шага, указанного после выражения вызова. В JavaScript функция всегда возвращает значение, даже если оно не определено автором функции с помощью слова **return** и никак не используется. Возвращаемое по умолчанию значение функции — **undefined**. Слово return обозначает досрочное (в том смысле, что до достижения завершающей фигурной скобки тела) завершение работы функции и замену выражения вызова на возвращённое значение — это как бы вычисление функции. Существует слово yield, которое применяется в особом виде функций — генераторах — и позволяет «выдать» очередное значение без завершения работы функции. Если возвращаемое значение зависит только от самой функции (в том числе от аргументов) и никак не задействует другие данные за пределами тела, то такая функция — чистая. По возможности следует стремиться к чистоте функций и к тому, чтобы каждая функция решала свою собственную задачу.

Линтинг

Дальнейшие действия будем осуществлять, пользуясь помощью линтера и редактором Atom. Нам необходимы пакеты linter-eslint и linter.

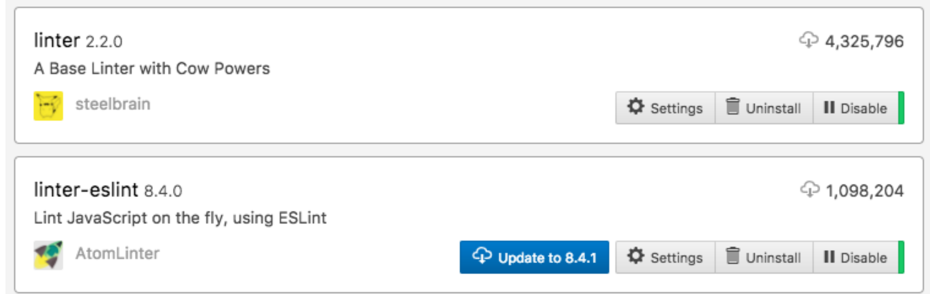


Рис. 15. Настройка линтера в Atom

Настройка линтера, вообще говоря, раскладывается на три составляющих:

- инвариантную;
- специфичную для конкретного редактора;
- специфичную для данного проекта.

Настройку линтера для отличных от Atom сред (Webstorm и т. д.) оставим читателю в качестве упражнения.

Инвариантная часть состоит в том, что линтер может быть использован как самостоятельный инструмент, осуществляющий часть процесса тестирования проекта. Если есть необходимость использовать его отдельно от редактора (в данном случае Atom), то следует установить npm-пакет eslint (это как бы CLI) или вызывать его с помощью команды npx.

Что же касается специфики проекта, то это совокупность правил, которых очень много и которые определяют детали стиля кодинга.

Они сосредоточены в npm-пакетах, именуемых eslint-config-X, где X — название настройки, например:

- eslint-config-rallycoding;
- eslint-config-airbnb.

В проекте должен присутствовать файл .eslintrc, который содержит ссылку на используемые правила. Правила после установки располагаются в папке node_modules.

Вот примеры файла .eslintrc:

<table>
<tr>
<td>

```
{
        "extends": "rallycoding",
        "rules": {
          "no-alert": 0,
        }
}
```

</td>
<td>

```
{
        "extends": "airbnb"
}
```

</td>
</tr>
</table>

На момент написания этого текста были распространены правила, которыми с общественностью поделились разработчики Airbnb.

Пусть у нас есть проект в папке, открытый в Atom.

Если создать файл .eslintrc с настройкой, указанной выше, но не установить правила, будет выдано сообщение об ошибке.

Мы можем посмотреть, какими зависимостями пользуются правила:

npm info "eslint-config-airbnb@latest" peerDependencies

... и увидеть, что среди них присутствует линтер, поэтому после установки правил он сам будет находиться в папке node_modules.

На странице пакета eslint-config-airbnb предлагается команда, работающая в Linux/macOS, которая производит все нужные действия:

```
(
  export PKG=eslint-config-airbnb;
  npm info "$PKG@latest" peerDependencies --
json | command sed 's/[\{\},]//g ; s/: /@/g' | xargs npm install --save-
dev $PKG@latest
)
```

(https://www.npmjs.com/package/eslint-config-airbnb)

Если файл со сценарием уже был открыт в редакторе, его, возможно, придётся закрыть и открыть снова, чтобы настройки применились.

Возьмём работающий код и улучшим его с помощью линтера:

```
1  function sayHello(name){return `Hello, ${name}!`}
2                                  Missing space before opening brace. (space-before-blocks)
```

$

Linter

Severity	Provider	Description	Line
Error	ESLint	'sayHello' is defined but never used. (no-unused-vars)	1:10
Error	ESLint	Requires a space after '{'. (block-spacing)	1:24
Error	ESLint	Missing space before opening brace. (space-before-blocks)	1:24
Error	ESLint	Requires a space before '}'. (block-spacing)	1:49
Error	ESLint	Missing semicolon. (semi)	1:49

Рис. 16. Подлежащий улучшению код

Первое сообщение говорит о том, что функция не используется, а последнее — о том, что после return нет точки с запятой.

После исправления этих замечаний линтера появилось ещё одно (no-console).

Чтобы линтер не протестовал против использования вывода в консоль, в раздел правил файла .eslintrc следует добавить отключение правила no-console:

Settings | .eslintrc

```
1  {
2    "extends": "airbnb",
3    "rules": {
4        "no-console": 0,
5      }
6  }
```

Рис. 17. Разрешение использовать ссылку на объект console

После этого замечаний останется всего три:

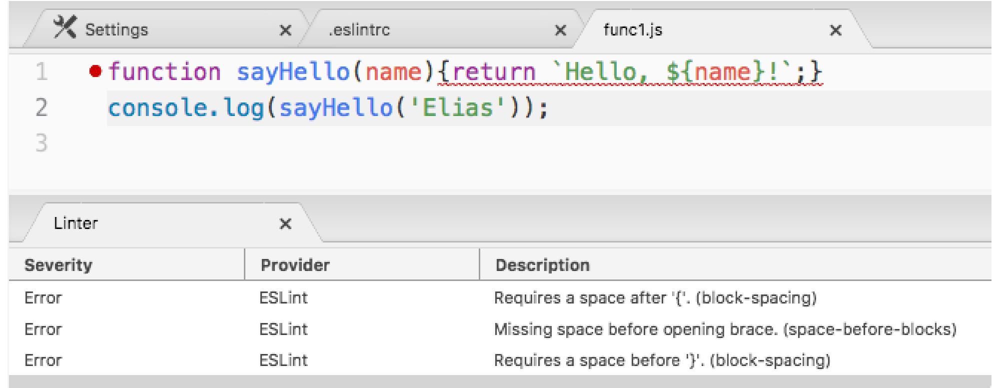


Рис. 18. Замечания линтера о пробелах

Расставив нужные пробелы, получим код, не вызывающий претензий линтера:

```
Settings  .eslintrc  func1.js
1  function sayHello(name) { return `Hello, ${name}!`; }
2  console.log(sayHello('Elias'));
3
```

Рис. 19. Полностью исправленный код

Правило comma-dangle линтера (https://eslint.org/docs/rules/comma-dangle) посвящено постановке последней запятой в списке аргументов. Оставляем его читателю для самостоятельного изучения. Количество формальных параметров доступно у функции как у объекта через свойство length (в примере выше sayHello.length === 1).

Теперь несколько слов о написании кода в теле функции. Существуют различные аргументы за и против тех или иных стилей его оформления, которые касаются расположения фигурных скобок. Сторонники стиля ядра Linux (Linux kernel coding style) ставят открывающую фигурную скобку блока на одной строке с началом блока во всех случаях кроме тела функции:

<table>
<tr><th colspan="2">Linux kernel coding style</th></tr>
<tr><td>if (x is true) {
    we do y
}</td><td>int function(int x)
{
    body of function
}</td></tr>
</table>

С точки зрения Airbnb (и автора этого текста), открывающую фигурную скобку блока тела функции следует ставить в той же строке, где находится слово function. Отступление от этого правила вызывает сообщение об ошибке:

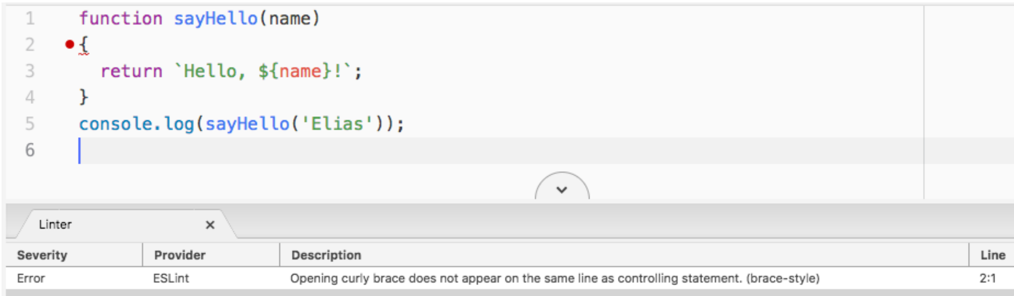


Рис. 20. Замечание об открывающей фигурной скобке

Окончательно, вот не вызывающий претензий линтера работающий код для объявления стандартной функции и её вызова:

цветовая схема One Light	цветовая схема Atom Light
Settings × func1.js 1 function sayHello(name) { 2 return `Hello, ${name}!`; 3 } 4 console.log(sayHello('Elias'));	Settings × func1.js × 1 function sayHello(name) { 2 return `Hello, ${name}!`; 3 } 4 console.log(sayHello('Elias'));

Рис. 21. Полностью корректный код объявления функции

Работая в редакторе, в исследовательских целях удобно совмещать выполнение сценария целиком, запуская его по имени:

```
func1.js ×
1    function sayHello(name = 'Elias') {
2      return `Hello, ${name}!`;
3    }
4    console.log(sayHello());
5
```

```
$ node func1
Hello, Elias!
```

Рис. 22. Запуск файла со сценарием на выполнение по имени и выполнение отдельных команд, связанных с этим сценарием, в отдельной консоли, для чего можно воспользоваться директивой .load следующим образом:

```
func1.js
1  function sayHello(name = 'Elias') {
2    return `Hello, ${name}!`;
3  }
4  console.log(sayHello());
```

загрузка файла в CLI

```
$ node
> .load func1.js
function sayHello(name = 'Elias') {
  return `Hello, ${name}!`;
  }
  console.log(sayHello());

Hello, Elias!
undefined
> sayHello('People');
'Hello, People!'
```

Рис. 23. Загрузка сценария внутрь среды исполнения

Возвращаясь к раскраске кода: по адресу https://kodaktor.ru/g/func1 приведён ещё вариант, сгенерированный с помощью diff2html.

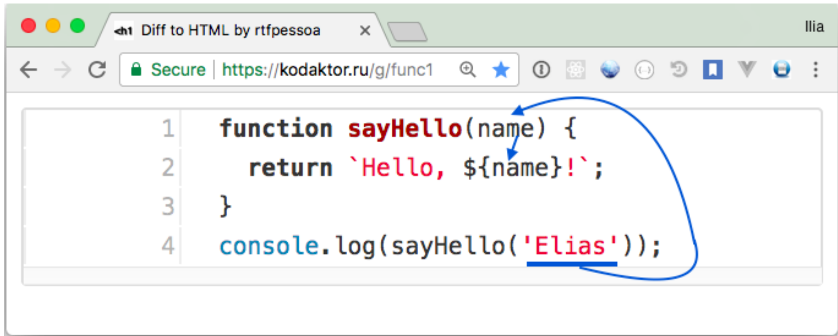


Рис. 24. Код функции с подсветкой на сайте kodaktor.ru

Теперь рассмотрим подробнее этот фрагмент с точки зрения JavaScript. После имени функции указывается в круглых скобках её сигнатура, т. е. список имён через запятую, которые, будучи упомянуты в теле функции, служат гнёздами для подстановки значений при вызове функции. Это формальные параметры (аргументы), в примере выше он один.

Тело функции в совокупности с другими частями (слово function или стрелка, имя, аргументы) образует ***литерал*** функции.

При вызове функции, т. е. упоминании её имени с оператором «круглые скобки» внутри круглых скобок указываются фактические параметры, т. е. то, что передаётся в эти гнёзда.

Фактическим параметром в случае JavaScript может быть и литеральное значение строки, как в примере в строке 4, и переменная, и функция.

В приведённом примере функция возвращает строку, заключённую в одинарные обратные кавычки. Такие строки (template strings) позволяют помещать внутри себя имена переменных, заключённых в фигурные скобки, предварённые знаком доллара. Так получается шаблонизационная интерполяция, т. е. вставка переменных внутрь строки без её разрыва (как это происходило ранее в случае применения оператора «бинарный плюс» для конкатенационной интерполяции).

В JavaScript обращение с аргументами довольно гибкое:

- аргументы можно не упоминать по именам в сигнатуре;
- можно присваивать значения по умолчанию в сигнатуре и не передавать при вызове

```
function sayHello(name = 'Elias') {
  return `Hello, ${name}!`;
}
console.log(sayHello());
```

Если у функции несколько аргументов в сигнатуре, то нужно так спроектировать сигнатуру, чтобы аргументы со значениями по умолчанию начинались с какого-то момента и до конца, потому что нельзя, например, вызвать функцию с пропущенным первым фактическим параметром, но указанным вторым.

В экосистеме Node сложились соглашения по поводу того, чем должны быть аргументы функции по порядку в некоторых стандартных случаях. Это касается ситуаций с ошибками и коллбэками, о чём речь пойдёт позже.

Считается неудачным подходом проектировать функцию с большим числом аргументов, вместо этого функции лучше передавать литерал объекта (паттерн «options»), в котором параметры оказываются именованными.

Созданное Алонзо Чёрчем лямбда-исчисление (λ-calculus) описывает поведение сущностей «функционального типа» в наиболее общем виде. Выражение — это то, что вычисляется, а функцию можно рассмотреть как некоторое обособленное выражение, допускающее в себя объекты для подстановки. Например, конструкцию $((x^2)3)$ можно рассмотреть как применение выражения «икс в квадрате» к объекту «три», результатом чего является подстановка числа 3 в это выражение и его вычисление с получением результата 9. Если выделить те части выражения, которые допускают подстановку, получим безымянную (анонимную) функцию, или лямбду. В JavaScript они наиболее компактно и «прозрачно» представлены стрелочными функциями.

Перепишем предыдущую функцию, которая возвращает строку с приветствием по имени, в стрелочной форме, и добавим стрелочную функцию, которая возвращает квадрат числа.

```
1    const sayHello = (name = 'Elias') => `Hello, ${name}!`;
2
3    const qv = x => x * x;
4
5    console.log(sayHello());
6
7    console.log(qv(3));
```

безымянная стрелочная функция

Рис. 25. Лямбды

Знак =>, состоящий из двух символов «равенство» и «больше» читается как «fat arrow» и иллюстрирует математическое толкование функции как сопоставляющего правила: каждое значение x функция переводит в квадрат этого значения.

Правильная с точки зрения выбранного стиля запись функции определяется такими правилами линтера, как arrow-spacing.

```
const qv = x =>x * x;
Missing space after =>. (arrow-spacing)
```

Рис. 26. Подсказка линтера о записи лямбды

К сожалению, в тексте данного пособия в некоторых случаях пришлось допустить отклонения от этого стиля, в основном, ради экономии места в строке. Это касается и пробелов между инфиксными операторами и их операндами, записи объектов (см. глава 7) и т. п.

Вызов стрелочной функции ничем не отличается от вызова стандартной функции, а объявления отличаются довольно сильно.

Главное отличие в том, что стандартные функции объявляются с помощью слова function, которое связывает имя функции с кодом её тела и является некоторым вариантом инструкции объявления let/const/var. Упоминать имена стандартных функций можно где угодно в коде, потому что всё, касающееся функции (начиная со слова function и до завершающей фигурной скобки), поднимается вверх (hoisting).

Безымянные стандартные функции с появлением стрелочных функций стали несколько менее актуальны, но при необходимости работать с this могут быть полезны.

Правило линтера func-names (https://eslint.org/docs/rules/func-names) указывает, должен ли линтер требовать наличия формального имени.

```
const qv2 = function (x) { return x * x; };
```

Если правило func-names не указано со значением ["error", "never"], то линтер пометит вышеприведённое объявление как ошибочное с пояснением: unexpected unnamed function. В этом случае корректным вариантом будет, например, такой:

```
const qv2 = function q2(x) { return x * x; };
```

Это «второе имя» (в примере совпадающее с «первым») оказывается доступно только внутри тела этой функции и, следовательно, может быть использовано для организации рекурсии.

IIFE и стрелки

Безымянная функция существует как обособленное выражение, как тело без головы, и практически использовать её можно двумя способами:

- присвоив её (тело) имени переменной, как в примерах выше, и затем вызвав;
- создав IIFE (immediately invoked function expression), выражение, которое совмещает объявление функции и её вызов (самоприменение).

Вот пример IIFE:

```
console.log((x => x * x)(3));
```

Рис. 27. Выражение немедленного исполнения функции

IIFE является именно выражением, не инструкцией, т. е. само по себе оно ничего не должно «делать», оно просто возвращает результат вычисления. Чтобы, например, вывести его для наблюдения, нужно передать его коду, который может это сделать, в примере выше — методу log объекта console. Разумеется, это не касается случаев, когда какие-то действия осуществляются внутри тела функции. Тогда IIFE может фигурировать как инструкция.

Варианты создания IIFE разнообразны, например: -function(){}();.

Стрелочные функции отличается от стандартных ещё и тем, что:

- стрелка явно что-то возвращает (а слово return в стандартной функции может отсутствовать);
- у них нет собственного *контекста*, обозначаемого словом this.

В стандартных функциях контекст this — это что-то вроде неявного аргумента, который присутствует всегда. Но передача фактического значения в него происходит иным способом. Чтобы понять это, рассмотрим модели (паттерны) вызова функций:

- вызов функции;
- вызов метода;
- методы call, apply;
- вызов конструктора.

Первый паттерн мы уже рассмотрели. Второй паттерн был ещё раньше продемонстрирован на примере console.log — указывается объект, оператор уточнения, а затем имя функции в составе этого объекта. Конструктор — это специальная функция, которая вызывается при создании нового объекта (инстанциации, экземплификации с помощью оператора new). Вызов конструктора рассматривается в главе 8.

Тем не менее пока просто скажем, что оператор new в любом случае *вызывает* функцию.

Упражнение 4-1

Объясните поведение кода (см. также главу 8):

```
new function () { console.log(111); };
// 111 и возврат {}
(function () { console.log(111); })();
// 111 и возврат undefined
```

Рассмотрим кратко вызов с помощью call и apply.

```
const a = qv(3);
const b = qv.call(null, 3);
const c = Reflect.apply(qv, null, [3]);

console.log(a === b && b === c && c === 9); // true
```

Как видно из примера, все три варианта вызова возвращают одно и то же. Метод call находится в прототипе любой функции (если его не переопределить!) и позволяет передать в функцию её контекст, а также фактические параметры в формате списка через запятую. В примере выше мы передаём null (хотя могли передать что угодно), поскольку это стрелочная функция без собственного контекста.

Также в прототипе функции находится метод bind, который позволяет привязать либо контекст, либо аргументы, либо то и другое вместе, к функции. В отличие от call он возвращает новую функцию, но *не вызывает* её. У этой новой функции на место её формальных параметров и контекста привязаны переданные с помощью bind сущности.

Метод apply находится не только в прототипе функции, но и в специальном непереопределяемом объекте Reflect. Мы можем передать аргументы в виде массива, а не через запятую отдельными значениями.

Если функция определена как метод одного объекта (об этом далее при обсуждении темы объектов), а мы хотим вызвать её как метод другого объекта (относительно другого объекта), то нам нужно передать ей этот другой объект в качестве контекста.

Использование apply выводит нас к теме массивов, равно как и рассмотрение способов обращаться со списком аргументов внутри тела функции:

- массивоподобный объект arguments;
- оператор rest.

Поэтому мы вернёмся к этой теме при обсуждении массивов в главе 6, коснувшись остаточного параметра функции (rest parameter).

Упражнение 4-2

Вызовите функцию без использования круглых скобок.

Это типичный вопрос «с подвохом» для собеседования по JavaScript. Вот примеры ответа:

- new (без передачи аргументов, только побочный эффект);
- console.log`1` // ['1'];
- как геттер – см. главу 7.

Упражнение 4-3

Создайте функцию, которая возвращает трёхкомпонентный цвет CSS (например, 'rgb(123, 32, 12)' по трём переданным ей числам; по умолчанию для каждого компонента генерируется случайный цвет от 0 до 255.

Упражнение 4-4 «Миксины»

Примеси (mixins) — один из популярных шаблонов компоновки объектов в JavaScript [2]. В примеси один объект получает свойства и методы другого объекта.

Примесью или миксином называют объект JavaScript, содержащий коллекцию методов и свойств [5]. Примеси предназначены не для самостоятельного использования, а для включения (подмешивания) в свойства другого объекта. *В терминологии SASS слово mixin означает включаемый параматризованный код, по виду напоминающий класс, но отмеченный директивой @mixin. (В LESS ничем не отличается от класса по виду.) Миксины в SASS изначально похожи на функции, а в LESS они могут быть сделаны похожими на функции если параметризуются. В SASS параметры миксина указываются со знаком $, а в LESS со знаком @.*

Пусть функция задана следующим образом:

```
const qv = function () { return this * this; };
```

Для этой функции подходит определение миксина, потому что она легко превращается в конкретный метод-миксин, который можно подмешать в объект (глава 7).

Вызовите её с помощью метода call так, чтобы она вернула квадрат переданного числа.

Упражнение 4-5

Для этой же функции осуществите привязку контекста методом bind и затем вызовите её для получения возвращаемого значения.

Упражнение 4-6

Для стрелочной функции, заданной как

```
const qv = x => x * x;
```

осуществите привязку числа на место аргумента x с помощью метода bind и вызовите полученную функцию для получения возвращаемого значения.
(Также см. оператор bind (https://github.com/GossJS/bind) в главе 9.)

Примечание

bind привязывает по значению, т. е. если привязывать не конкретный литерал, а имя, будет привязано то, с чем связано это имя в момент исполнения bind.

Упражнение 4-7

Создайте рекурсивную функцию, которая использует тернарный оператор для вычисления факториала в стрелочной форме.

Упражнение 4-8

Создайте рекурсивную функцию, которая получает два числа (x, y) и вычисляет сумму этих двух чисел, рассматривая второе число как количество увеличений первого числа на единицу.

Глава 5. Функция как тип данных

Функции в JavaScript являются объектами. То есть это данные ссылочного типа.

У функций как у объектов есть свойства. Свойство length позволяет получить количество аргументов, которые она принимает. Свойство name содержит то, что считается именем функции.

```
const f = function () { console.log(f.name); };
f(); // f
```

В этом примере интересны два момента:

1. Переменная f доступна как объект со свойством name внутри тела функции как бы сразу, хотя чисто механически следуя слева направо интерпретатор мог бы удивиться обращению к свойству name — ведь в этот момент объявление const f уже «состоялось», но анализ функции ещё не завершен, ибо завершающая фигурная скобка её тела не закрылась.

2. Имени нет справа от слова function, оно есть у переменной, с которой связывается безымянная функция.

Аналогично, но с некоторыми отличиями это работает в объявлениях классов (глава 8).

Мы уже упоминали метод toString (подробнее он обсуждается в главе 7), здесь же просто скажем, что по умолчанию каждая функция как объект обладает методом toString, выводящим её текстовое представление.

JavaScript поддерживает функциональное программирование, так как его функции относятся к сущностям первого класса. Это значит, что функции могут демонстрировать такое же поведение, как переменные.

Рассматривая функции как сущности первого класса, мы говорим о функциях высшего порядка (high order functions): они способны манипулировать другими функциями и могут получать функции в качестве аргументов, или возвращать функции, или делать и то и другое.

В простом безтиповом λ-исчислении между функциями и данными нет разницы (применить к функции можно функцию и вернуть можно функцию), но у этих функций один аргумент. Раскладывание функции от нескольких аргументов на функции, возвращающие функции от меньшего числа аргументов получило название каррирования в честь Хаскелла Карри.

```
const rep = n => s => String.prototype.repeat.call(s, n);
console.log(rep(3)('w')); // www
```

Здесь с именем rep связывается преобразование вида Number -> String -> String (т.е. числам сопоставляется то, что сопоставляет строкам строки).

Выражение rep(3) в примере выше означает, что функция, сопоставляющая числам другую функцию, отработала, вернула функцию и прекратила жизненный цикл. Однако значение её аргумента, число 3, не растворилось в пусто-

те, а оказалось в теле возвращённой функции под именем n. То есть в некотором смысле значение переменной n привязалось или примкнуло к внутренней функции. Здесь можно увидеть аналогию с работой привязки bind и приём частичного применения функций для создания своего рода фабрики функций:

```
const pow = (x, y) => x ** y;

const exp = pow.bind(null, Math.exp(1));

console.log(exp(2)); // 7.3890560989306495

const curryPow = x => y => x ** y;
const partiPow = y => x => x ** y;
console.log(curryPow(2)(10)); // 1024
console.log(partiPow(10)(2)); // 1024
const qv = partiPow(2);
const cb = partiPow(3);

console.log(qv(5)); // 25
console.log(cb(5)); // 125
```

Привязав значение в виде числа e к первой функции, мы получили новую функцию уже от одного аргумента. Это будет функция, возводящая число e в указанную степень. Аналогично, мы создаём функции, вычисляющие квадрат и куб указанного числа. То есть с помощью байндинга мы сделали переменным показатель, зафиксировав степень, а с помощью каррирования — наоборот.

В предыдущей главе мы говорили о том, что обычная модель поведения функции — передача управления «в точку вызова», при которой выражение вызова как бы заменяется на значение, возвращённое функцией.

При использовании модели **коллбэков** в общем случае не предполагается никакого использования значений, *возвращённых* функциями. Тело коллбэка ("callback function", «функции обратного вызова») может образовать некоторое «параллельное» ответвление выполнения программы, которое **никогда** не воссоединится с основным вычислительным процессом.

Коллбэк есть функция, которая передаётся в другую функцию, т. е. в функцию высшего порядка. Эта другая функция может вызвать коллбэк немедленно (синхронно) или через неопределённое время (асинхронно).

Рассмотрим следующий пример:

```
const sayHello = (f = x => x, name = 'Elias') => `Hello, ${f(name)}!`;

console.log(sayHello());                    // Hello, Elias!

console.log(sayHello(x => x.toUpperCase())); // Hello, ELIAS!
```

В функции sayHello теперь есть формальный параметр f, значением по умолчанию для которого записана тождественная функция x => x (которая возвращает свой аргумент, ничего с ним не делая). Будем называть f функциональным аргументом.

Если функцию sayHello вызвать без фактических параметров, то значениями по умолчанию становятся эта функция x => x и слово 'Elias', т. е. фактически происходит применение

```
(x => x)('Elias')
```

Что происходит, если на место f передать функцию x => x.toUpperCase()?

Эта функция вызывается, тут же возвращает значение, и далее это значение (строка в верхнем регистре) встраивается в шаблонную строку и возвращается уже как результат работы «главной» функции sayHello.

Другим примером являются, например, методы массивов типа map (о которых говорится в другой части пособия).

```
[1, 2, 3].map(x => x * x)
```

Мы вызываем здесь метод map массива и передаём ему в качестве функционального аргумента безымянную функцию x => x * x — и она последовательно вызывается для каждого элемента массива, а результатом всего действия является новый массив, составленный из квадратов элементов исходного массива.

Здесь все действия выполняются строго друг за другом и все возвращаемые значения используются. То есть мы имеем дело с синхронным коллбэком.

В случае асинхронного коллбэка инструкция, которая располагается вслед за вызовом асинхронной функции, никак не может рассчитывать на результаты её работы. Чисто теоретически при очень быстром выполнении может так произойти, что асинхронный код выполнится до начала выполнения этой следующей инструкции, но вероятность этого мала. Пренебрежение этим фактом вызывает множество ошибок у начинающих пользователей.

В современном JavaScript можно организовать код таким образом, что асинхронность в нём будет сведена к минимуму. Но в любом случае природа JavaScript асинхронна и её следует хорошо понимать.

Асинхронная функция – это функция, которая может запускать код в очереди, вне прямой связи с последовательностью команд другого кода. Аннотация **async** не делает функцию таковой.

Это не то же самое, что параллельность (параллелизм). При проектировании параллельных процессов проектируется их связь между собой. Асинхронный же код после своего запуска отправляется в непредсказуемое самостоятельное плавание.

Классическим материалом для демонстрации асинхронного коллбэка является запуск таймера.

Различные среды JavaScript предоставляют реализацию метода setTimeout, который позволяет передать ему функцию и число в тысячных долях секунды. Метод делает паузу указанной длительности и после этой паузы вызывает указанную функцию. Реальная пауза получается на небольшой промежуток времени больше, чем указано.

```
t0.js
1    console.time('S');
2    setTimeout(() => console.timeEnd('S'), 7000);
3    console.log('Начало!');

 $ node t0
Начало!
S: 7007.072ms
```

Рис. 28. Асинхронный коллбэк

Методы console.time и console.timeEnd служат началом и завершением измерения времени, которое занято происходящим между ними. Мы вызываем метод setTimeout и передаём ему лямбду, вызывающую метку конца измерения времени. Мы видим, что команда вывода надписи «Начало» находится **после** вызова setTimeout, но выполняется **до** выполнения этой лямбды. Мы также видим, что если бы эта лямбда возвращала какое-то полезное значение (payload), его нельзя было бы использовать для вывода, потому что к моменту возвращения этот вывод уже произошёл.

https://kodaktor.ru/timerhell2

Воспользоваться значением, которое возникает после «паузы», можно только в теле этой функции обратного вызова. Наиболее общий принцип управления асинхронным кодом состоит в том, что мы передаём в асинхронную функцию лямбду, формальные параметры которой служат приёмником полезной информации.

Приведём пример. Напишем сценарий, который под управлением платформы node.js выводит в консоль свой собственный текст (квайн).

```
fs.js
1    require('fs').readFile(process.argv[1], (err, x) => console.log(String(x)));
```

Рис. 29. Простой квайн на JavaScript

Наш коллбэк — лямбда (err, x) => console.log(String(x)) передаётся в асинхронный метод readFile объекта «Файловая система», который здесь представлен как require('fs').

Реализация этого метода вызовет эту лямбду тогда, когда содержимое файла будет успешно считано целиком, и **передаст это содержимое в формальный параметр x**.

Соответственно, когда лямбда будет выполняться, внутри её тела искомое содержимое файла будет доступно под именем x.

Соглашение «Error first» означает, что первый аргумент коллбэка резервируется для передачи в него информации об ошибке.

Этот код можно улучшить, включив обработку ошибки и реализовав стиль написания «цепочки методов» («method chain») путём помещения метода на новую строку:

```
fs.js

1  require('fs')
2    .readFile(process.argv[1], (err, x) => console.log(err || String(x)));
```

Рис. 30. Стиль цепочки методов

Здесь также используется сокращённая схема вычисления логического ИЛИ (short circuitting). Ситуация такова, что либо возникает ошибка, либо из файла считывается содержимое. Если возникает ошибка, то в переменной err будет сообщение о ней вида «Error: ENOENT: no such file or directory», которое будет оценено в рамках дизъюнкции как true. Чтобы дизъюнкция вернула значение true, достаточно, чтобы первый операнд дизъюнкции был true, в этом случае второй и прочие операнды не вычисляются. Если же ошибки нет, то реализация метода readFile поместит в err значение null, которое будет оценено как false и вся операция в целом вернёт второй операнд (и уже неважно, как он будет оценен).

Итак: в модели коллбэка лямбда не возвращает значение, а предоставляет свои формальные параметры как контейнеры для полезных данных.

Это не значит, что return внутри коллбэка не имеет смысла. Поскольку return досрочно завершает работу функции, в том числе и коллбэка, он вполне может возникнуть как составляющая часть условия чтобы прервать вычисления, хотя и не вернуть ничего.

Вернёмся теперь к таймеру. Допустим, мы хотим, чтобы последовательно были отсчитаны несколько временных промежутков.

Учитывая сказанное выше, мы должны скопировать текущий код вызова setTimeout вместе с лямбдой и отметкой начала измерения и вставить его в тело лямбды.

Так получаем три последовательных паузы длиной в 3 и 4 секунды соответственно.

Мы видим, что код сдвигается вправо и «вглубь», а количество пар вложенных скобок увеличивается. Это называется адом обратных вызовов (callback hell). Самое последнее действие самое глубокое по вложенности.

Если нужно обратиться к базе данных, прочитать из неё адрес файла, после чего извлечь содержимое этого файла, из которого получить данные, то это будет примером минимального реалистичного двухэтажного ада коллбэков.

Работа JavaScript основана на идее цикла событий (Event Loop), позволяющей исполнять неблокирующие операции ввода/вывода, несмотря на то, что JavaScript является однопоточным в классическом понимании потоков. Многопоточным является низший уровень, т. е. ядро системы. Движок может отдавать некоторые операции на выполнение ядру системы в фоновом режиме, и после завершения такой операции ядро сообщает движку, что соответствующий такой операции коллбэк может быть добавлен в очередь.

См. также https://medium.com/devschacht/event-loop-timers-and-nexttick-18579cd122e0

Программисту предоставляется объект process, содержащий такие методы как nextTick, помещающие нужный коллбэк в эту очередь. О нём говорилось выше в главе 3 при обсуждении переменных.

Можно также создавать дочерние процессы, что позволяет исполнять внешние команды.

```
t1.js

1   console.time('S1');
2   setTimeout(() => {
3     console.timeEnd('S1');
4     console.time('S2');
5     setTimeout(() => {
6       console.timeEnd('S2');
7       console.time('S3');
8       setTimeout(() => {
9         console.timeEnd('S3');
10      }, 5000);
11    }, 4000);
12  }, 3000);
13  console.log('Начало!');
14
```

```
$ node t1
Начало!
S1: 3003.341ms
S2: 4005.728ms
S3: 5003.825ms
```

Рис. 31. Ад вложенных обратных вызовов

Упражнение 5-1

Напишите код для обращения node к node с целью выяснить собственную версию вызовом себя как внешней команды.

Решение (с использованием промисификации, см. ниже)

```
const execFile = require('util').promisify(require('child_process').execFile);

(async () => {
  const { stdout } = await execFile('node', ['--version']);
  console.log(stdout); // v9.4.0
})();
```

См. также главу 7 о контексте выполнения сценария.

Вложенные асинхронные вызовы затрудняют восприятие и отладку кода.

У этой проблемы есть как минимум два приемлемых решения:

- промисы;
- синхронное ожидание асинхронных функций.

Промисификация

Рассмотрим рефакторинг кода, приведённого на предыдущем рисунке:

```
const pr1 = x => new Promise(res => setTimeout(() => res(), x));

console.time('S1');
pr1(3000)
  .then(() => {
    console.timeEnd('S1');
    console.time('S2');
    return pr1(4000);
  })
  .then(() => {
    console.timeEnd('S2');
    console.time('S3');
    return pr1(5000);
  })
  .then(() => {
    console.timeEnd('S3');
  });
```

Рис. 32. Промисификация

Обратите внимание, что действия не сдвигаются в коде вправо/вглубь. Это достигается благодаря тому, что в конце каждого предыдущего действия происходит возврат объекта одной и той же структуры — промиса. Это позволяет создать цепочку методов с характерной вертикальной записью. Её можно длить сколь угодно долго. Создание промиса на основе таймера заключается в следующем. Мы оборачиваем асинхронную функцию объектом Promise, а конкретный код обработки правильной и ошибочной ситуаций заменяем на вызовы особых коллбэков res и rej. Коллбэк, известный под именем res, — это функция, передаваемая как функциональный аргумент метода then. Что касается ошибочной ситуации, то об этом подробнее на странице https://kodaktor.ru/g/promisify

На момент написания этого текста в браузерной среде нельзя, а на платформе node можно было воспользоваться вторым способом промисификации, который даёт совершенно эквивалентный результат:

```
require('util').promisify(setTimeout)
```

Хотя промисы делают код как бы плоским и более читаемым (увеличение количества последовательных действий сводится к почти механическому копипасту блоков кода), в них всё же присутствуют коллбэки.

Синхронное ожидание асинхронных функций позволяет уменьшить значимость коллбэка и избавиться от ада. Коллбэк сам может **быть** асинхронной функцией, что часто наблюдается в серверных фреймворках типа Express, которые построены на коллбэках. В этом случае при необходимости делать обращения к отложенным действиями внутри коллбэка мы сможем обойтись без коллбэка внутри коллбэка.

Для этого код с асинхронными вызовами следует обернуть следующим образом, сделав его телом асинхронной функции, аннотированной словом **async**:

1. Создаём IIFE
(() => {})();

2. Дописываем слово async:
(async () => {})();

3. Вписываем в тело лямбды асинхронные вызовы, предваряя их словом await

```
(async () => {
  console.time('S1');
  await pr2(3000);
  console.timeEnd('S1');
  console.time('S2');
  await pr2(4000);
  console.timeEnd('S2');
  console.time('S3');
  await pr2(5000);
  console.timeEnd('S3');
})();
```

Инструкции с await возвращают полезные данные, если они существуют. Это те самые данные, которые помещаются в согласованный аргумент коллбэка или в аргумент функции, передаваемой как функциональный аргумент метода then.

Упражнение 5-2а

Используйте IIFE function-функции с именем self и метод toString, чтобы написать программу длиной в 1 строку, которая выводит в консоль свой собственный текст.

Упражнение 5-2б

Промисифицируйте функцию fs.readFile и модифицируйте код из примера так, чтобы работали методы then и catch.

Решение

fspr.js

```
1   require('util').promisify(require('fs').readFile)(process.argv[1])
2     .then(x => console.log(String(x)))
3     .catch(e => console.log(e));
```

Упражнение 5-3

Рефакторизуйте код из упражнения 5-2 так, чтобы он запускался с await в асинхронной лямбде.

Упражнение 5-4

Сравните код из упражнения 5-3 с кодом, в котором используется fs.readFIleSync.

fsync.js

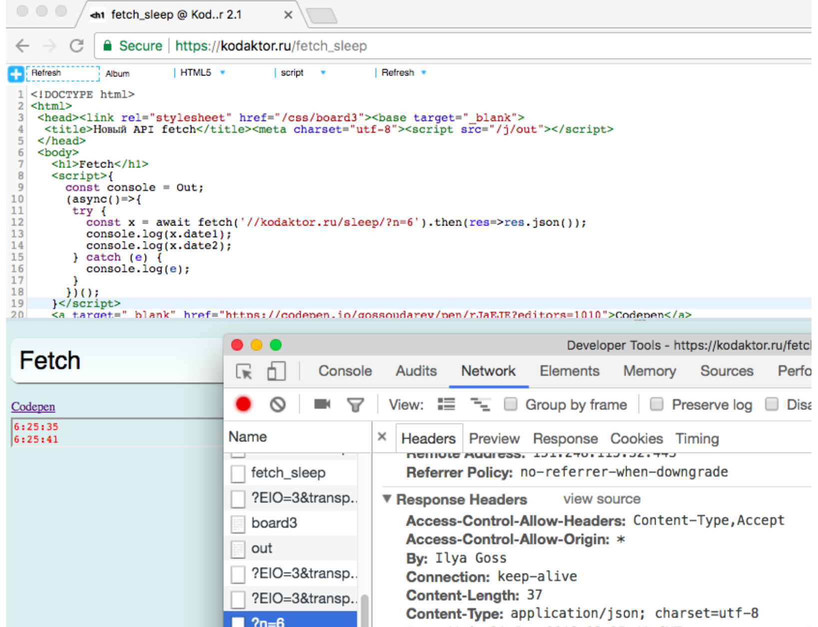
```
1  try {
2    const x = require('fs').readFileSync(process.argv[1]);
3    console.log(String(x));
4  } catch (e) {
5    console.log(e);
6  }
```

Упражнение 5-5

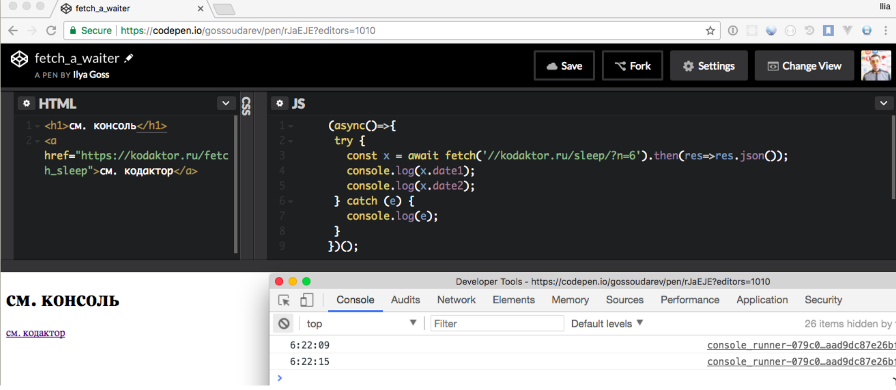
Реализуйте упражнение 5-3 в браузере с помощью fetch.

Решение

https://kodaktor.ru/fetch_sleep



или https://codepen.io/gossoudarev/pen/rJaEJE?editors=1010



Замыкание

В некотором широком смысле любая функция в JavaScript является замыканием в силу определения: замыканием будем называть удержание или способность к удержанию функцией значений переменных из внешней по отношению к ней области видимости. Речь идёт не о статических переменных, которые, например, в PHP объявляются внутри функции с помощью ключевого слова static. Речь идёт о ситуации, в которой одна функция («внешняя»), имеющая переменные в своём теле, возвращает другую («внутреннюю»).

Выше мы уже видели такой пример (с каррированием).

Значения переменных внешней функции замыкаются вокруг внутренней функции несмотря на то, что внешняя функция прекратила работу. В обычном случае после завершения работы функции связанные с ней переменные уничтожаются. (Это не касается специального вида переменных, существующих в ряде языков программирования, в том числе PHP и объявляемых с помощью слова static.)

Приведём пример работы замыкания (https://kodaktor.ru/closure). Мы должны создать функцию, которая возвращает другую функцию и сохранить её вызов в состоянии (т. е. переменной). Результатом этого вызова является та самая возвращённая функция, но у неё имеется замкнутая вокруг неё переменная n с первоначальным значением 0. При каждом вызове этой функции значение переменной инкрементируется:

```
const cc = ((n = 0) => () => ++n)();

console.log(cc()); // 1
console.log(cc()); // 2
console.log(cc()); // 3
```

Рассмотрите пример https://kodaktor.ru/army — в нём демонстрируется, как слово let позволяет создать «армию» функций с независимыми друг от друга внутренними состояниями.

Модели реализации коллбэка

Передача коллбэка в функциональный аргумент является базовой для JavaScript моделью обращения с асинхронным кодом. У неё существует важная разновидность — назначение обработчика события:

- в форме object.onvent = callback;
- в форме object.addEventListener('event', callback).

В качестве event может фигурировать щелчок мыши, нажатие кнопки на клавиатуре, приход данных из сети и т. д.

Исторически сложилось так, что передача коллбэка в setTimeout стоит особняком, но в более широкой перспективе это частный случай регистрации слушателя события (в данном случае события типа «timeout»). И наоборот: ска-

зать, что нечто должно выполниться по щелчку мыши есть примерно то же, что сказать, что нечто должно выполниться при срабатывании таймера, срок которого определён не конкретным заранее известным количеством единиц времени, а сигналом другого происхождения.

Более подробно этот вопрос рассматривается в разделах, посвящённых извлечению данных из сети, потокам и эмиттерам событий, а также DOM и кастомным событиям.

Здесь рассмотрим ещё интересные случаи применения коллбэков:

- в качестве регистрируемых слушателей события — веб-воркеры;
- в качестве обёртываемых проксируемых вызовов — декораторы.

https://kodaktor.ru/worker_user

Веб-воркеры

Как уже говорилось выше, JavaScript выполняется в одном потоке. Браузер никогда не будет выполнять два обработчика событий одновременно, и таймер никогда не сработает, пока выполняется обработчик события. Параллельные обновления переменных приложения или документа невозможны. Однако «фоновые потоки», определяемые спецификацией веб-воркеров, фактически являются параллельными потоками выполнения. Эти потоки выполняются в изолированной среде, не имеют доступа к объектам Window и Document и могут взаимодействовать с основным потоком выполнения только посредством передачи асинхронных сообщений. Таким образом, параллельные изменения дерева DOM по-прежнему невозможны, но можно писать функции, выполняющиеся длительное время, которые не будут останавливать работу цикла событий и подвешивать браузер. Создание нового фонового потока выполнения не является такой тяжеловесной операцией, как открытие нового окна броузера, но также не является легковесной операцией, поэтому создавать новые фоновые потоки следует для выполнения адекватных им задач.

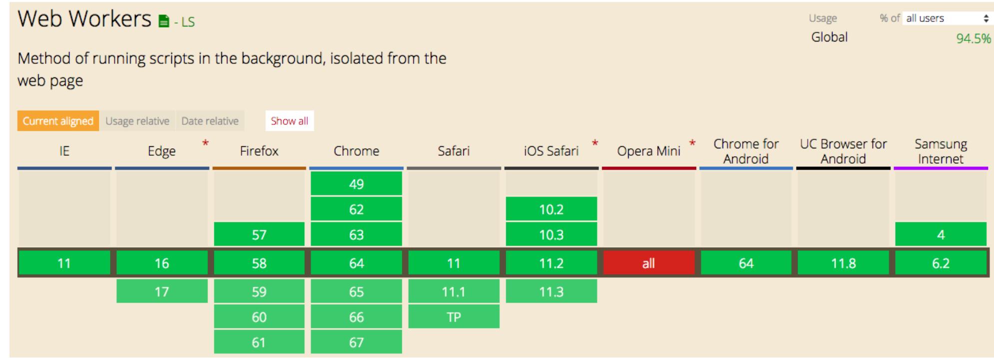


Рис. 33. Распространение веб-воркеров

Веб-воркер можно рассмотреть как удалённый компонент, которому можно послать сигнал и подписаться на его обновления. Это модель «подписчик — издатель», предполагающая, что издатель играет некоторым образом ак-

тивную роль и посылает уведомления подписчику, когда что-то происходит (например, меняется статус соединения, считываются данные и т. п.).

С помощью worker.postMessage(info) мы посылаем сигнал воркеру, а с помощью worker.onmessage = e => { ... } мы подписываемся на обновления, предполагая, что реализация воркера вызовет этот коллбэк и поместит в его формальный параметр полезные данные (payload).

Внутри воркера тоже находится такая подписка:

```
onmessage = e => { ... postMessage(info)... }
```

Она срабатывает в ответ на сигнал, посланный из основного сценария. Особенностью реализации является то, что внутри воркера имеется некое пространство имён по умолчанию, в котором находятся метод postMessage и свойство onmessage, которое можно рассматривать как «местный» глобальный объект воркера.

С рассматриваемой темой тесно связаны атрибуты defer и async элемента script [2, с. 335]. Они оба сообщают браузеру, что сценарий не использует метод document.write() и не генерирует содержимое документа и что браузер может продолжать разбор и отображение документа, пока сценарий загружается. Атрибут defer указывает браузеру отложить выполнение сценария до момента, когда документ будет загружен, проанализирован и станет готов к выполнению. Атрибут async предписывает выполнить сценарий, как только это станет возможно, но не блокирует разбор документа на время загрузки сценария. Если представлены оба атрибута, браузер отдаст предпочтение атрибуту async и проигнорирует атрибут defer. Отложенные сценарии исполняются в порядке их следования в документе, а асинхронные — как только будут загружены, т. е. они могут выполняться в произвольном порядке.

Упражнение 5-6

Спроектируйте способ создания IIFE, при котором выражение вызова писалось бы не справа от выражения функции, а слева.

Решение

```
x => x * x
(x => x * x)(3) // 9
```

→

```
(f => f(3))(x => x * x) // 9
(f => f())((x = 3) => x * x) // 9
```

Упражнение 5-6 «Декораторы»

Декоратор — приём программирования, который позволяет взять существующую функцию и изменить/расширить ее поведение. Декоратор в нашем случае есть функция высшего порядка: получает функцию фактически в виде коллбэка и возвращает обертку, которая делает что-то своё «вокруг» вызова основной функции.

Декорируйте с помощью функции

```
const logged = function (f) {
  return function () { console.log(arguments); return f.apply(this, arguments); };
};
```

функцию

```
const qv = x => x * x;
```

и вызовите так, чтобы значение первого аргумента было выведено в консоль.

Примечание

Используйте то же имя, что и у декорируемой функции. Также см. декоратор @ как babel-плагин в главе 9.

ГЛАВА 6. СТРОКИ И МАССИВЫ

Строки и массивы имеют некоторые совпадающие характеристики и множество различающих. Строка представляет собой последовательность символов (см. замечания о Unicode), а массив — последовательность значений произвольного типа. И строки, и массивы позволяют обращаться к своим частям с помощью оператора индексации, индексы отсчитываются от 0. Строки и массивы имеют свойство length, содержащее длину строки или количество элементов массива. Литерал строки ограничивается кавычками, а литерал массива — квадратными скобками.

```
const first = 'Elias';
const last = 'Goss';
const person = [first, last];
console.log(person[0]); // Elias
console.log(first[0]); // E
console.log(person.length); // 2
console.log(first.length); // 5
```

Индексируемые структуры — традиционный материал для обработки с помощью циклов со счётчиком:

```
arr0for.js
1   const first = 'Elias';
2   const last = 'Goss';
3   for (let i = 0; i < first.length; ++i) console.log(first[i]);
4   for (let i = 0; i < last.length; ++i) console.log(last[i]);
5

$ node arr0for.js
E
l
i
a
s
G
o
s
s
```

Рис. 34. Поэлементная работа со строкой с помощью цикла for

В то же время строки и массивы, в отличие от объектов, реализуют протокол итерации (перебираемости), о чём подробнее см. в главе 7.

Это значит, что их значения можно перебрать, не обращаясь к индексам, с помощью инструкции for .. of:

```
const first = 'Elias';
const last = 'Goss';
for (const t of first) console.log(t);
for (const t of last) console.log(t);
```

Цикл for .. in в этом контексте оказывается излишним (подробнее в главе 7).

Вместо императивных структур, где возможно удобно пользоваться функциональными:

```
const person = [first, last];
person.forEach((x, i) => console.log(`${i} ${x}`));
// 0 Elias
// 1 Goss
```

В отличие от строк, массивы наследуют метод forEach, который принимает коллбэк, вызываемый для каждого элемента массива. Этот метод возвращает undefined, не поддерживает таким образом цепочку методов и не мутирует (сам по себе) перебираемый массив.

В отличие от иммутабельных строк (нельзя изменить какую-то часть строки, как бы подставив вместо одного символа другой, можно только создать новую строку по исходной), массивы, естественно, поддерживают возможность изменять себя по частям. В ряде случаев это как раз нежелательно, поэтому нужно обращать внимание на то, как работает тот или иной метод.

В отличие от строк, являющихся примитивным типом данных, массивы относятся к ссылочному типу. Массивы являются подвидом объектов, о которых см. в главе 7 (о конструкторе, прототипах см. в главе 8).

Какова связь ссылочных типов данных и так называемой реактивности?

```
1   let arrayOne = [1, 2];
2   // создаём синоним:
3   const refToArrayOne = arrayOne;
4   console.log(refToArrayOne[0]); // 1
5   console.log(arrayOne[0]); // 1
6   arrayOne[0] = 3;
7   console.log(refToArrayOne[0]); // 3
8   console.log(arrayOne[0]); // 3
9   // разрываем синонимичность:
10  arrayOne = [11, 12];
11  console.log(refToArrayOne[0]); // 3
12  console.log(arrayOne[0]); // 11
```

Рис. 35. Ссылочность

Начиная со строки 3 и далее до строки 10, имена arrayOne и refToArrayOne ассоциированы с одной и той же областью памяти. Поэтому когда мы из-

когда мы изменяем содержимое этой области в строке 6, имя refToArrayOne отражает это изменение.

Но в строке 10 связь имён arrayOne и refToArrayOne нарушилась, так как правостороннее выражение [11, 12] создаёт новую область памяти и связывает имя arrayOne с ней. А имя refToArrayOne остаётся связанным со старой областью памяти. Если бы имя refToArrayOne не хранило ссылку на эту область, то после строки 10 у интерпретатора появился бы повод передать массив [1, 2] в руки коллектора мусора.

Если бы в строке 3 создавалась «реактивная» переменная, то она была бы не синонимом имени для той же области памяти, а как бы полным дубликатом, повторителем оригинальной переменной (переменной-источника) и изменялась бы вместе с ней. Можно сказать и иначе: синонимы статичны, они привязывается к области памяти в момент присвоения, а реактивные переменные динамичны, они привязываются не к области памяти, а к связи оригинальной переменной и привязанной к ней области памяти.

Свойство length у массивов является динамическим, оно может быть изменено и оказывает влияние на структуру массива. Оно содержит значение, большее, чем самый большой индекс.

```
const arr = ['a', 'b', 'c'];
console.log(arr.length); // 3
arr.length = 4;
console.log(arr); // [ 'a', 'b', 'c', <1 empty item> ]
console.log(arr[3]); // undefined
console.log(arr[33]); // undefined
```

Разрежённым (sparse) называется массив, в котором есть «пустые» ячейки. Такое достигается применением delete к элементу массива:

```
const arr = ['a', 'b', 'c', 'd'];
delete arr[3];
console.log(arr); //  'a', 'b', 'c', <1 empty item> ]
```

или созданием массива с пропусками:

```
const arr = ['a', 'b', 'c', ,];
Unexpected comma in middle of array. (no-sparse-arrays)
console.log(arr); //  'a', 'b', 'c', <1 empty item> ]
```

Рис. 36. Разреженный массив

Линтер в примере выше выражает недовольство наличием разрежённого массива.

Обратите внимание на две запятые в конце. Именно отсутствие чего бы то ни было между ними создаёт эффект дыры в массиве.

Кстати, интересно, что, если оставить в конце одну запятую, линтер недоволен как наличием пробела в конце, так и отсутствием:

```
const arr = ['a', 'b', 'c', ];
There should be no space before ']'. (array-bracket-spacing)
console.log(arr); //  [ 'a', 'b', 'c' ]
const arr = ['a', 'b', 'c',];
A space is required after ','. (comma-spacing)
console.log(arr); //  [ 'a', 'b', 'c' ]
```

Рис. 37. Проблема пробела в синтаксисе литерала массива

Уменьшение свойства length вызывает усечение массива.

```
const arr = ['a', 'b', 'c'];
console.log(arr.length); // 3
arr.length = 2;
console.log(arr); // [ 'a', 'b']
```

Если мы присвоим значение

```
const arr = ['a', 'b', 'c'];
console.log(arr.length); // 3
arr[5] = 'λ';
console.log(arr); // [ 'a', 'b', 'c', <2 empty items>, 'λ' ]
console.log(arr.length); // 6
```

Некоторые особенности строк могут вызвать вопросы. Рассмотрим примеры таких вопросов.

Например: если строка есть примитивный тип, откуда может возникнуть свойство length, если свойства (ключи) бывают только у объектов?

Ответ: при обращении типа 'abc'.length создаётся объектная *обёртка*, которая позволяет обратиться к свойству length.

Другой вопрос: как получить подстроку строки? Чем отличаются методы substr и substring?

Ответ:

```
const text = 'abcdefgh';
console.log(text.substring(2, 4)); // cd
console.log(text.substring(2, 5)); // cde

console.log(text.substr(2, 4)); // cdef
console.log(text.substr(2, 5)); // cdefg
```

В первом случае возвращается интервал [a,b), т. е. от первого указанного номера включительно (считая от 0) до второго указанного номера невключительно.

Во втором случае мы указываем номер и количество символов, отсчитывая от него.

Если нужно просто ответить на вопрос, есть ли вхождение подстроки, то для этого можно использовать регулярное выражение или нововведённый метод includes.

Чем объясняется такое значение свойства length в нижеследующем примере?

```
const emacron = 'ē';
console.log(emacron.length); // 2
```

Ответ: хотя отображается одним видимым символом, эта строка состоит из двух Юникод-символов: U+0065 и U+0304, т. е. строчная латинская буква 'e' и COMBINING MACRON, знак черточки над символом, который как бы сдвигает вывод на одну позицию назад. Этот знак можно получить с помощью **Юникод-экранирования**: '\u0065\u0304'.

Часто некоторые проблемы возникают с взаимными превращениями чисел и строк. Здесь полезны явные преобразования строки в число, такие как Number (для их работы нужно, чтобы аргумент мог быть интерпретирован как число) или более «всеядные» методы парсинга, такие как parseInt.

Упражнение 6-1

Используя метод String.fromCharCode и Юникод-экранирование, получите вывод тибетской составной буквы Ом:

Упражнение 6-2

С помощью цепочки методов toString и toUpperCase переведите десятичное число десять в шестнадцатеричную систему счисления.

Упражнение 6-3

Решите обратную задачу с помощью метода parseInt и с помощью литерала шестнадцатеричного числа.

Решение

```
parseInt('A', 16) // 10
0xA // 10
```

Задачи с массивами

Конвертация из массива в строку и обратно может быть осуществлена с помощью взаимно-обратных методов Join и Split.

Если поиск в строке очень удобно осуществлять с помощью регулярных выражений (см. ниже), то поиск в массиве благодаря es2015 стал удобнее с помощью методов find и findIndex. Ранее многие задачи было удобнее решать с помощью аналогичных методов сторонних библиотек, таких как lodash. Ниже

мы рассмотрим конкретную задачу поиска в массиве на примере массива, полученного из JSON-документа.

Деструктуризация массива

Рассмотрим операцию деструктуризации массива. Она есть немутирующее извлечение данных из массива, некоторая форма запроса к массиву, и выглядит как выражение присваивания, в котором левостороннее выражение само есть массив. Однако мы не можем ничего присвоить массиву как таковому. То, что выглядит как массив в левой части, является своего рода *шаблоном*, который в своей «предельной» форме просто повторяет массив в правой части, только вместо литералов-значений на соответствующих местах поставлены какие-то имена:

```
const colors = ['red', 'green', 'blue'];
const [firstColor, secondColor] = colors;
console.log(firstColor);  //  red
console.log(secondColor); //  green
```

В этом примере операцией деструктуризации из массива colors извлекаются значения «red» и «green» и сохраняются в переменных firstColor и secondColor. Значения извлекаются по номерам их позиций в массиве, а имена переменных могут быть любыми. Любые элементы, явно не упомянутые в шаблоне деструктуризации, игнорируются.

В шаблоне можно пропускать элементы и указывать имена переменных только для элементов, представляющих интерес. Например, если потребуется извлечь из массива только третий элемент, можно не указывать переменные для сохранения первого и второго элементов:

```
const colors = ['red', 'green', 'blue'];
const [, , thirdColor] = colors;
console.log(thirdColor); // blue
```

В этом фрагменте используется операция присваивания с деструктуризацией, извлекающая третий элемент из массива colors. Запятые, предшествующие в шаблоне имени переменой thirdColor, отмечают поля, соответствующие предшествующим элементам массива. Используя такой прием, можно пропустить значения из произвольных элементов в середине массива без необходимости указывать имена переменных.

```
/* eslint no-sparse-arrays: 0 */
const [, , c] = [, , 3];
console.log(c); // 3
```

Выше мы уже говорили о разрежённых массивах. Пример выше иллюстрирует два момента:

- шаблонность выражения деструктуризации (замена литерала на имя);
- подсказку линтера, протестующего против разрежённости.

Операция присваивания массивов с деструктуризацией обладает уникальным свойством, упрощающим обмен значений между двумя переменными.

```
1   let a = 3;
2   let b = 7;
3   const c = a;
4   a = b;
5   b = c;
6   console.log(a); // 7
7   console.log(b); // 3
```

```
1   let a = 3;
2   let b = 7;
3   a += b;
4   b = a - b;
5   a -= b;
6   console.log(a); // 7
7   console.log(b); // 3
```

```
1   let a = 3;
2   let b = 7;
3   [a, b] = [b, a];
4   console.log(a); // 7
5   console.log(b); // 3
6
```

Рис. 38. Обмен значений переменных

В правой части примера присваивание массива с деструктуризацией выглядит как зеркальное отражение. Слева от знака «равно» находится шаблон деструктуризации. Справа находится литерал массива, временно созданный для обмена значений. Деструктуризация применяется к временному массиву, в первый и второй элементы которого копируются значения из b и a. В результате возникает эффект обмена переменных значениями. Если правая часть выражения присваивания вернёт null или undefined, интерпретатор сгенерирует ошибку. Поддерживается возможность деструктуризации вложенных массивов.

В главе 9 в разделе о транспиляции вам предлагается сравнить приведённые выше варианты с тем, который предложит babel при переводе крайнего правого фрагмента на ES5.

Синтаксис присваивания массивов с деструктуризацией поддерживает возможность определения значений по умолчанию для любых элементов массива. Значение по умолчанию используется в случае отсутствия элемента в данной позиции или элемент имеет значение undefined. Синтаксис напоминает значения аргументов функции по умолчанию.

Упражнение (https://kodaktor.ru/arrform)

```
// напишите для следующего массива
const arr = [[1, 3, 5], [2, 4, 6]];
// деструктуризацию, извлекающую второе чётное число
// в переменную even2
```

В главе 7 мы вернёмся к деструктуризации и рассмотрим пример совместной деструктуризации массивов и объектов.

Оператор rest/spread

Остаточный параметр функции (rest parameter) отмечается троеточием (...), предшествующим именованному параметру. Такой именованный параметр превращается в массив Array, содержащий все остальные параметры, переданные в вызов функции, откуда и взялось это название — остаточные. В функции может иметься только один остаточный параметр, и он должен объявляться последним. До появления rest/spread чтобы работать с произвольно переданными параметрами использовался массивоподобный объект arguments. На сегодняшний день объект arguments действует параллельно остаточным параметрам и хранит все аргументы, переданные в вызов функции. Он корректно отражает параметры, переданные в функцию, независимо от наличия остаточного параметра.

Рассмотрим функцию, которая подсчитывает количество нечётных чисел среди переданных ей в виде списка через запятую:

```
const odd = (...nums) => nums.filter(x => x % 2).length;

console.log(odd(4, 6, 0, 5, 10, 3)); // 2
```

Мы видим, что объявление такой функции показывает, что она может обрабатывать произвольное количество параметров. Синтаксис деструктуризации массивов имеет схожее понятие, которое называется остаточные элементы (rest items). Для обозначения остаточных элементов используется синтаксис многоточия (...), который служит для присваивания оставшихся элементов в массиве конкретной переменной. Например:

```
const colors = ['red', 'green', 'blue'];
const [firstColor, ...restColors] = colors;
console.log(firstColor);       //  'red'
console.log(restColors.length); // 2
console.log(restColors[0]);     // 'green'
console.log(restColors[1]);     // 'blue'
```

Первый элемент массива colors присваивается переменной firstColor, а остальные копируются в новый массив restColors. В результате restColors получает два элемента: «green» и «blue». Остаточные элементы удобно использовать для извлечения определенных элементов из массива и сохранения остаточных элементов доступными, однако этот подход имеет еще одно полезное применение.

Мы можем использовать синтаксис выбора остаточных элементов для решения задачи клонирования массива:

```
const colors = ['red', 'green', 'blue'];
const [...clonedColors] = colors;
```

Мы ещё вернёмся к вопросу о природе оператора rest/spread в главе 7, в том числе при обсуждении итерабельности.

Далее здесь рассмотрим несколько задач в форме упражнений с решениями.

Упражнение 6-4

Отсортируйте одномерный числовой массив по возрастанию числовых значений, используя метод sort и лямбду-предикат.

Упражнение 6-5а

Передайте в функцию
const summer = (a, b, c) => a + b + c;
массив, содержащий три числа для подстановки вместо a, b, c, используя метод apply.

Решение

В главе 4 мы обсуждали вызовы функций с передачей им аргументов разными способами. Метод apply позволяет передать функции массив так, чтобы его элементы «встали» на места формальных параметров («распределились» ("spread")).

Старый вариант: summer.apply(null, [1, 2, 3])
Новее: Reflect.apply(summer, null, [1,2,3])
Новый вариант: summer(...[1, 2, 3])

Упражнение 6-5б

Найдите максимальный элемент в массиве.

Решение

У массивов нет метода max, но он есть у объекта Math, причём метод max принимает аргументы через запятую.

Используя *оператор spread/rest* мы можем передать массив в метод, который ожидает списка аргументов через запятую: Math.max(...[3, 44, 5])

Мы можем написать небольшой код, благодаря которому каждый массив получит метод max. Для этого воспользуемся прототипом (о прототипах см. в главе 7 и далее в тексте пособия). Модификация прототипа в общем случае не является очень хорошей идеей, но здесь мы покажем технику, не претендуя на лучшую практику.

Как добавить в прототип всех массивов метод мах, взяв его из Math?

Способ 1

```
1  /* eslint no-extend-native: 0 */
2  const arr = [3, 44, 5];
3  console.log(arr.max); // undefined
4  Array.prototype.max = function () { return Math.max(...this); };
5  console.log(arr.max()); // 44
```

Способ 2

```
1    const arr = [3, 44, 5];
2    console.log(arr.max); // undefined
3    Reflect.defineProperty(Array.prototype, 'max', {
4      get: function () {
5        return Math.max(...this);
6      }
7    });
8    console.log(arr.max); // 44
```

Пошаговые объяснения: см. https://kodaktor.ru/g/array.prototype

Если бы мы хотели обеспечить все массивы сортировкой, как в предыдущем упражнении, то могли бы написать примерно такой код:

```
(f=>Array.prototype.sort=function(){return f.call(this, (x,y)=>x-
y);})(Array.prototype.sort);
```

Объяснение: мы можем сохранить функцию в отдельной переменной

const f = function(){return Array.prototype.sort.call(this, (x,y)=>x-y);};

и далее вызвать её, передав массив:

f.call([10,2,13,9,5]) (или apply — здесь это неважно)

По аналогии с Math.max не получится присвоить новую реализацию метода имени, так как окажется бесконечная рекурсия — sort будет сохранять сам себя, поэтому нужно сохранить старый sort и вызвать его по отношению к массиву.

Чтобы сделать это в одну строку переназначения, мы пользуемся паттерном внедрения зависимости и передаём Array.prototype.sort во внутреннее локальное состояние f.

```
console.log([10, 2, 13, 9, 5].sort()); // [10, 13, 2, 5, 9]
(f => Array.prototype.sort = function () { return f.call(this, (x, y) => x - y); })(Array.prototype.sort);
console.log([10, 2, 13, 9, 5].sort()); // [ 2, 5, 9, 10, 13 ]
```

Упражнение 6-6

Дан массив IP-адресов пользователей, которые посещали сайт.
https://kodaktor.ru/j/ips

Необходимо создать частотную таблицу (массив, в котором нет повторяющихся адресов и напротив каждого адреса указана частота посещения сайта пользователем с этого адреса), после чего отсортировать по убыванию частот так, чтобы в верхней части таблицы были самые активные посетители сайта.

Отдельно ответьте на вопросы:

(а) Сколько всего различных адресов присутствует в массиве?

(б) Со скольких адресов сайт посещался по одному разу?

(в) Какова максимальная частота посещения?

Упражнение 6-7

Используя оператор rest и метод reduce, постройте функцию, в которую можно передать сколько угодно числовых значений через запятую, возвращающую сумму этих значений.

Решение

```
(...nums) => nums.reduce((x, y) => x + y);
```

Упражнение 6-8

Постройте функцию, которая по числу n возвращает массив из n возрастающих значений.

Упражнение 6-9

Используя решение предыдущего упражнения и метод fromCharCode, выведите первые 10 заглавных букв английского алфавита.

Упражнение 6-10а

Решите обратную задачу вывода кодов букв, если дан массив из 10 букв. Используйте метод charCodeAt.

Упражнение 6-10б

Сгенерируйте массив из 10 строк, полученных путём дополнения до 10 знаков нулями слева случайных чисел в диапазоне от двузначных до семизначных. Используйте значения функции по умолчанию для хранения диапазонов чисел, а также Array.from, map и метод padStart (https://github.com/GossJS/11lines).

До ES6 существовало два основных способа создания массивов:

- литералы;
- конструктор Array.

Оба подхода требовали перечислить элементы будущего массива по отдельности. Чтобы упростить создание массивов JavaScript, в ECMAScript 6 были добавлены методы Array.of() и Array.from().

Упражнение 6-11

Используя метод Array.from и одну лямбду, сгенерируйте массив возрастающих значений от 1 до n.

Решение

```
Array.from({ length: 5 }, (v, i) => ++i)
```

Также ES6 добавляет новые методы в каждый массив (т. е. в Array.prototype):

- find и findIndex (для работы над массивами с любыми значениями);
- fill и copyWithin (по аналогии с типизированными массивами).

Ещё ES6 добавил метод String.prototype.includes для проверки вхождения подстроки в строку. Чтобы обеспечить единообразие функциональных возмож-

ностей строк и массивов, метод Array.prototype.includes был добавлен в редакции ES7 (ECMAScript 2016).

Иными словами, после ES5 в 2015 и 2016 г. работа с массивами в JavaScript обогатилась пятью методами.

В ES2015 также введены типизированные массивы. Обычно принцип инкапсуляции при реализации типов данных не позволяет обращаться к отдельным частям значения данного типа. Например, мы не можем обратиться отдельно к 3-му байту из 8 в значении типа Double (IEEE754). Типизированные массивы предоставляют такую возможность. Наряду с собственно типизированными массивами в клиентский JavaScript привносится понятие буфера, которое ранее было введено в платформе Node.

Упражнение 6-12

Представьте число двойной точности в виде последовательности 8 байтов, как они хранятся в памяти под управлением JavaScript: https://kodaktor.ru/types.pdf (https://kodaktor.ru/types_out).

Если создать переменную uint8 как new Uint8Array(1), то она будет ссылаться на массив с единственной ячейкой uint8[0], хранящей значение без знака, которое умещается в единственный байт. Диапазон таких значений от 0 до 255. Попытка записать в такую ячейку число, большее 255 или меньшее 0, приведёт к циклическому сдвигу. Например, 257 превратится в 1, а -1 превратится в 255. Аналогично, при представлении со знаком, т. е. Int8Array, 128 превратится в -128, а -129 превратится в 127. Диапазон такого представления — от -128 до -127 включительно, так как один бит такого байта используется как индикатор знака, а оставшиеся 7 бит хранят модуль числа, и чтобы избежать неоднозначности нуля (-0 и +0) последовательность 10000000 представляет отрицательное число -128, и отрицательных чисел оказывается на одно больше, чем положительных.

В рамках этого пособия мы не будем касаться темы регулярных выражений, поскольку она раскрывается в отдельных учебниках. Скажем только, что стандарт ES2015 улучшил поддержку регулярных выражений языком JavaScript, сделав их более совместимыми с Unicode. Интересным примером применения регулярных выражений является сопоставление маршрутов (https://kodaktor.ru/g/route.tester). Результатом сопоставления регулярного выражения строке является JavaScript-массив. Этот массив имеет свойства и элементы, предоставляющие информацию о сопоставлении. Подобные массивы возвращаются методами RegExp.exec, String.match и String.replace.

Глава 7. Литеральные объекты

Литеральный объект (plain JavaScript object) представляет собой обычный объект, ссылочный тип данных. В современном JavaScript согласно ES2015 (раздел 4.3) существуют 4 категории объектов:

1. Обычные (ordinary) объекты (обладают всеми чертами поведения объектов, которые поддерживаются в JavaScript).
2. Необычные (exotic) объекты (обладают чертами поведения, отличающими их от обычных объектов).
3. Стандартные (standard) объекты (определяются в ES2015, такие как Array, Date, Map и т. д., могут быть обычными и необычными).
4. Встроенные (built-in) объекты (определяются средой выполнения JavaScript, в которой действует сценарий; все стандартные объекты одновременно являются встроенными).

Например, объекты, которые создаются с помощью конструктора Array — необычные. Их свойство length отслеживает и способно изменять элементы массива.

Синтаксис литерального объекта напоминает синтаксис блока (см. загадку {}+[] и []+{} в главе 3). Это одна или более пар «ключ»:«значение», представленных в виде списка через запятую. В «родном» синтаксисе допускается последняя запятая (trailing comma), а в JSON нет. Правило comma-dangle линтера eslint подробно описывает различные установки, при которых считаются допустимыми те или иные варианты постановки запятой, по аналогии с списком аргументов фунции, списком элементов в массиве.

Литеральный объект напоминает тип-запись в Паскале или структуру в C, но близок и так называемым ассоциативным массивам. В ES2015 ассоциативные массивы введены как самостоятельный тип Map.

Простейший способ создать литеральный объект для дальнейшей работы с ним — присваивание, в котором слева находится переменная, а справа сам объект:

```
const person = { first: 'Elias', last: 'Goss' };
console.log(person.first); // Elias
```


Рис. 39. Литерал объекта и уточнение

При дотошном следовании стилю кодирования возникает вопрос о «правильном» написании объекта. Такие правила линтера, как object-curly-spacing, key-spacing, comma-spacing определяют этот стиль:

```
const person = {first: 'Elias', last: 'Goss'};
A space is required after '{'. (object-curly-spacing)
```

Рис. 40. Использование пробела в синтаксисе объектов

К сожалению, в тексте данного пособия в некоторых случаях пришлось допустить отклонения от этого стиля в основном ради экономии места в строке.

В «родном синтаксисе» ключи могут быть написаны «как есть», без кавычек, для чего они должны быть валидными идентификаторами (см. главу 1). В JSON ключи должны заключаться, как и строковые значения, в двойные кавычки. Кроме того, значениями в литералах могут быть, например, функции, а в JSON мы ограничены строками, числами, массивами и другими объектами. Ключи также называют свойствами, но эти термины употребляются обычно в разных контекстах. Когда речь идёт о внутреннем устройстве объекта, с позиции разработчика объекта, говорят о ключах. Когда речь идёт об обращении к объекту извне, с позиции пользователя объекта, говорят о свойствах.

Если ключ не является валидным идентификатором, например, будет содержать пробел, то его нельзя использовать с оператором уточнения, но можно использовать с оператором индексации.

Ключ литерального объекта представляет собой переменную внутри объекта. Это имя можно сохранить в другой переменной, в виде строки. Тогда его можно использовать в объекте опосредованно через эту другую переменную, указав её имя в квадратных скобках, что называется вычисляемыми ключами или свойствами.

```
const first = 'firstName';
const person = { [first]: 'Elias', last: 'Goss' };
console.log(person.firstName); // Elias
```

Но оказывается, что вычислимые ключи могут делать более интересные вещи. Рассмотрим следующий пример:

```
const
  first = x => x,
  second = first;
const person = { [first]: 'Elias', last: 'Goss' };
console.log(person[second]); // Elias
```

В первых трёх строках этого примера с именем first связывается функция, а имя second становится синонимом имени first, т. е. указывает на ту же область памяти, где хранится код функции x => x.

Таким образом, мы можем зафиксировать некоторое значение (в данном случае это функция) и далее оперировать не им самим, а ссылками на него. Развитие этой идеи привело к появлению особого типа данных Symbol, представители которого доступны только в виде ссылок.

Что *реально* является ключом, значением которого является строка 'Elias'? Ответ: то значение, которое доступно как first.toString(). То есть ключом является представление значения, связанного с именем first, в виде строки.

```
const first = x => x;
const person = { [first]: 'Elias', last: 'Goss' };

console.log(person[Object.keys(person)[0]]); // Elias
console.log(person[first.toString()]); // Elias
console.log(person['x => x']); // Elias
console.log(person[x => x]); // Elias
```

Последний вариант обращения может показаться странным. Может показаться, что мы используем в качестве ключа функцию. Но это не так. Ведь если бы это было так, то в последней строке мы бы имели дело с другой функцией (вновь созданным литералом), а не с той же самой, которая фигурирует в первой строке, хотя выглядят они совершенно одинаково. Просто интерпретатор тут же преобразует «литерал функции» в строковое представление, и оно оказывается совпавшим с тем, которое хранится в качестве ключа объекта person.

Понятно, что обращение через точку в таких случаях невозможно. А обращение person.first даст нам undefined, так как ключа 'first' не существует.

Можно ли использовать в качестве ключа функцию?

Да, если мы используем объект типа Map.

```
const
  first = x => x,
  second = first;
const person = new Map();
person.set(first, 'Elias');
person.set('last', 'Goss');

console.log(person.get(second)); // Elias
console.log(person.get(first)); // Elias
console.log(person.get(x => x)); // undefined
console.log(person.get(first.toString())); // undefined
```

Ключом в Map является именно область памяти, а не её строковое представление. А двух одинаковых областей памяти не бывает, даже если в них содержится побитово совпадающее содержимое, адреса всё равно будут разными.

Наименование «объект» может, вообще говоря, быть несколько дизориентирующим, поскольку в классических объектно-ориентированных языках объект есть экземпляр класса. Так, в PHP чтобы получить объект, у которого есть свойства и методы, нужно объявить класс и экземплифицировать его. Это создание объектов от общего к частному.

Когда объект создаётся литералом, то на самом деле создаётся ещё один объект, связанный с ним, который называется прототипом. Это создание объектов от частного ... к другому частному или, возможно, к общему — если необходимо, можно создать новый объект на основе уже существующего, использо-

вав уже существующий объект как прототип. Подробнее о прототипах см. в главе 8. Здесь же пока ограничимся ещё двумя замечаниями:

- чтобы создать «схему генерации объектов», в традиционном JavaScript используется функция-конструктор, которая, как правило, заполняет значения ключей объекта, а отдельно методами заполняется прототип будущих объектов, доступный через свойство prototype конструктора; с некоторыми ухищрениями можно реализовать наследование; классом *в этом случае* является множество объектов, созданных с помощью этого конструктора;
- в ES2015 мы получили возможность объявлять классы и создавать объекты с помощью классов, хотя «под капотом» используется механизм прототипов.

Литеральные объекты — это в некотором смысле одиночки, они не предполагают своего «размножения» в явной форме, создают своего рода пространство имён и изолируют свои переменные от внешнего мира. В этом на них похожи блоки, введённые в ES2015. Но блоки изолируют свои переменные абсолютно — обратиться к переменным внутри блока снаружи невозможно, — а к переменным (ключам) объекта обратиться можно.

```
const sayLast = ({ last }) => last;
console.log(sayLast({ first: 'Elias', last: 'Goss' })); // Goss
```

В этом примере мы передаём безымянный объект с двумя ключами и двумя значениями в функцию. Функция использует паттерн деструктуризации объекта, извлекая из него значение по ключу, указанному в фигурных скобках, и возвращая его. То есть это функция извлечения значения по ключу. У её формального параметра нет явного имени, обращение к переданному объекту происходит неявно.

Изменим немного вышеприведённый пример. Что делает эта функция в модифицированном виде?

```
const sayLast = ({ last }) => ({ last });
console.log(sayLast({ first: 'Elias', last: 'Goss' })); // ?
```

Ответ: она формирует новый объект, у которого будет только один ключ (last), значением которого будет значение, доступное по имени last благодаря работе деструктуризатора в её сигнатуре. В данном случае этим значением будет 'Goss'.

В правой части функции используется сокращённый синтаксис создания литеральных объектов, который можно пояснить таким изображением:

```
const sayLast = ({ last }) => ({ last: last });
```

Рис. 41. Объяснение сокращённого синтаксиса объекта

Чтобы обратиться к значению внутри объекта, необязательно, чтобы объект был ассоциирован с каким-либо именем:

```
console.log({ first: 'Elias', last: 'Goss' }.last); // Goss
console.log(Object.values({ first: 'Elias', last: 'Goss' })[1]); // Goss
console.log(Reflect.get({ first: 'Elias', last: 'Goss' }, 'last')); // Goss
```

С помощью объектов мы можем создавать структуры, напоминающие списки или очереди, в которых каждый элемент имеет ссылку на последующий или предыдущий. Рассмотрим, как с помощью цикла for и коллбэка мы можем осуществить перебор такой структуры:

```
const o5 = { name: 'Obj5' };
const o4 = { name: 'Obj4', next: o5 };
const o3 = { name: 'Obj3', next: o4 };
const o2 = { name: 'Obj2', next: o3 };
const o1 = { name: 'Obj1', next: o2 };
function tail(o, whatToDo) {
  for (; o.next; o = o.next) whatToDo(o);
  return o;
}
const res = tail(o1, x => console.log(x.name)).name;
console.log(res);
```

Деструктуризация объектов

В предыдущей главе мы рассмотрели деструктуризацию массивов. С объектами ситуация выглядит схоже, но здесь нужно добавить, что синтаксис деструктуризации объектов (или весьма схожий) мы обнаруживаем в такой фундаментальной области, как импорт сущностей из нативных модулей (следующая глава). С другой стороны, при изучении языка запросов GraphQL мы также обнаруживаем очень похожую форму записи:

```
const colors = {
  sample: 'rgb(33,44,55)',
  rgb: {
    names: ['red', 'green', 'blue'],
    numbers: ['rgb(255,0,0)', 'rgb(0,255,0)', 'rgb(0,0,255)']
  }
};
const { sample } = colors;
console.log(sample); // rgb(33,44,55)
const { rgb: { names } } = colors;
console.log(names); // [ 'red', 'green', 'blue' ]
const { rgb: { numbers: [, , blueNumber] } } = colors;
console.log(blueNumber); // rgb(0,0,255)
```

Методическое примечание
При изложении этого материала в лекционной форме следует создать опрос, например, в Гугл, и задать упражнение для интерактивного выполнения (на основе последних двух строк вышеприведённого кода): https://kodaktor.ru/g/objform

Эта схема должна пояснить, как формируется «запрос к объекту»:

```
const colors = {                                              const {
  sample: 'rgb(33,44,55)',
  rgb: {                                                        rgb: {
    names: ['red', 'green', 'blue'],
    numbers: ['rgb(255,0,0)', 'rgb(0,255,0)', 'rgb(0,0,255)']     numbers: [, , blueNumber]
  }                                                             }
};                                                            } = colors;
              console.log(blueNumber); // rgb(0,0,255)
```

Рис. 42. Схема деструктуризации объекта

При деструктуризации массивов мы можем указывать в левой части (деструктуризационном шаблоне) любые имена переменных, важен только их порядок. При деструктуризации объектов мы можем переименовать извлекаемый ключ с помощью нового имени после двоеточия (удобно при импорте с помощью require):

```
const { sample } = colors;
console.log(sample); // rgb(33,44,55)
const { sample: myColor } = colors;
console.log(myColor); // rgb(33,44,55)
```

На лекции студент задал вопрос, как быть, если в массиве много элементов, и извлечь хотелось бы, например, 500-й (https://kodaktor.ru/g/destr). Вряд ли удачной идеей было бы писать 499 запятых перед соответствующей переменной.

Ответ можно получить в том числе интуитивно, вспомнив о том, что индексы массивов есть просто свойства объектов.

```
1   const colors = {
2     sample: 'rgb(33,44,55)',
3     rgb: {
4       names: ['red', 'green', 'blue'],
5       numbers: ['rgb(255,0,0)', 'rgb(0,255,0)', 'rgb(0,0,255)']
6     }
7   };
8
9   const { rgb: { numbers: [, , blue] } } = colors;
10  const { rgb: { numbers: { 2: bloo } } } = colors; √
11  console.log(blue); // rgb(0,0,255)
12  console.log(bloo); // rgb(0,0,255)
```

Рис. 43. Использование индекса как ключа при деструктуризации
https://kodaktor.ru/g/destruct_resp

Передавая функции объект вместо аргументов через запятую, мы делаем сигнатуру функции более читаемой, реализуем возможность управлять именами и порядком аргументов. Например, встроенная в JavaScript функция defineProperty (метод класса Reflect), определяет новое свойство объекта с помощью конфигурационного объекта, в котором есть такие ключи как value и enumerable:

```
const person = { firstName: 'Elias' };
Reflect.defineProperty(person, 'lastName', {
  value: 'Goss', enumerable: true
});
```

Ещё пример: в фреймворке Express функции (методу) sendFile передаётся путь к папке и точка монтирования этого пути (в этом примере текущая папка):

```
sendFile('docs/index.html', { root: '.' })
```

Сформировался даже паттерн RO[RO] (Receive Object[Return Object]), подразумевающий передачу функции параметров в виде конфигурационного одиночного объекта (и, возможно, возвращать данные в аналогичной форме). Сочетая передачу объектов с деструктуризацией, мы можем добиваться элегантного и компактного кода:

```
// RORO
const sayName = function ({ first: f, last: l = 'Goss' } = {}) {
  return `${f} ${l}`;
};
console.log(sayName({ first: 'Elias' })); // Elias Goss
```

Когда мы передаём функции объект с целью сгруппировать несколько аргументов (в том числе это касается функций-конструкторов), то для краткости можем говорить, например, «объект „methods“», и под этим подразумевается объект, который является значением ключа «methods», который, в свою очередь, сам принадлежит какому-то объекту. Вот пример применения указанной выше ситуации:

```
new Vue({
  el: '#app',
  data: {
    title: 'Приложение Vue',
    name: 'Elias'
  },
  methods: {
    sayHello(time) {
      return `Hello and good ${time}, ${this.name}!`;
    }
  }
});
```

В фреймворке Vue мы создаём экземпляр объекта с помощью конструктора Vue и передаём этому конструктору объект конфигурации, который включает который включает настройки приложения.

В Vue, React и других библиотеках и фреймворках на системном уровне поддерживается работа с состоянием приложения. Оно обычно представлено литеральным объектом и рассматривается как иммутабельное: изменения не вносятся в существующий объект, а вместо этого создаётся новая версия состояния. Например, в библиотеке Redux архитектура определяет функции-редьюсеры, которые возвращают эти версии:

```
const initialState = { list: ['первое', 'второе'] };
export default function taskReducer(state = initialState, action) {
  const list = [...state.list, action.payload];
  return { ...state, list };
}
```

Мы видим здесь, что rest/spread применяется как к массивам, так и к объектам. Применение к объектам в контексте React является транспилируемым (по состоянию вопроса на момент написания этого текста используется плагин transform-object-rest-spread).

См. также https://kodaktor.ru/objectspread

Рассматриваемому вопросу близка тема слияния объектов и создания глубоких копий (клонов). Встроенные в JavaScript средства (Object.assign()) имеют свои ограничения в этом аспекте (работа только с собственными перечислимыми (см. ниже) ключами, вызов геттеров и сеттеров, что приводит не только к копированию структуры, но и присваиванию значений, актуальных на момент выполнения этой операции). Использование более соответствующих этой задаче методов (Object.getOwnPropertyDescriptor() and Object.defineProperty()) мы оставляем для исследования читателю. Равным образом предлагается исследовать понятие сильной и слабой ссылки (https://kodaktor.ru/weak1, https://kodaktor.ru/weak2) и нативной реализации наблюдаемых объектов (Observable, https://kodaktor.ru/observe).

Мы обсудим несколько ниже второй аспект rest/spread, связанный с понятием итерабельности.

Методы

Каким образом создаются ***методы***?

```
const person = {
  first: 'Elias',
  sayHello() {
    return `Hello, ${this.first}!`;
  }
};
console.log(person.sayHello()); // Hello, Elias!
```

Простейший вариант включения функции внутрь объекта показан выше. Внутри объекта слово this указывает на этот объект. Но это касается только стандартно объявленных функций (слово function в примере отсутствует вместе с двоеточием в силу сокращённого синтаксиса!), а у стрелочных функций нет такой возможности.

Методы, объявленные с помощью сокращённого синтаксиса, могут использовать super для доступа к прототипу (ниже в разделе «Простой доступ к прототипу с помощью ссылки super»), а методы, объявленные с помощью слова function, такой возможности не имеют.

Поскольку ключи создаются простым обращением к ним (уточнением или индексацией), то создать метод можно, присвоив ключу функцию (именованную, безымянную, стрелочную).

Предположим, что у нас есть два объекта: person1 и person2, в первом из которых есть метод sayHello, а во втором нет. Может ли person1 одолжить свой метод объекту person2?

Да, это возможно благодаря таким методам, как call (см. глава 4). Первым аргументом мы передаём контекст, который доступен как this.

```
const person1 = {
  first: 'Elias',
  sayHello() {
    return `Hello, ${this.first}!`;
  }
};
const person2 = { first: 'Victor' };
console.log(person1.sayHello()); // Hello, Elias!
console.log(person1.sayHello.call(person2)); // Hello, Victor!
```

На случай переопределения метода call можно предусмотреть вызов с помощью Reflect.apply:

```
person1.sayHello.call = '';
// console.log(person1.sayHello.call(person2));
// TypeError: person1.sayHello.call is not a function
console.log(Reflect.apply(person1.sayHello, person2, []));
```

Упражнение 7-1

Создайте объект с двумя методами, хотя бы один из которых поддерживает паттерн цепочки методов.

Примечание

Метод, поддерживающий цепочку методов, возвращает this. Следовательно, он никак не может возвращать значение, связанное с его аргументами, следовательно, не может быть чистой функцией. Цепочка методов всегда означает эксплуатацию побочного эффекта.

Менеджмент ключей (свойств и методов) имеет свои тонкости, которые связаны с:

- подходом к созданию (литералы или классы);
- метасвойствами (атрибутами).

К классам мы перейдём несколько позже, в главе 8, а сейчас рассмотрим метасвойства.

Метапрограммирование представляет собой уровень программирования, при котором программа оперирует не только данными предметной области, но и своими собственными частями и характеристиками. Сюда включается и генерация программного кода. Сущности первого класса — это некоторый шаг к метапрограммированию. Но рассматриваемый термин может иметь и темпоральное значение в следующем смысле. На каком-то этапе развития языка объекты описывали данные, на обработку которых нацелена программа. Добавление ключей в объект осуществлялось просто точечной или скобочной нотацией (т. е. операторами уточнения и индексации). Эти ключи можно было перебрать в цикле. Затем ситуация стала меняться.

При выводе объекта (с помощью valueOf или сериализацией), а также при использовании таких методов как Object.keys, видны не все ключи (не обязаны быть видны), которые становятся видны при более детальном рассмотрении. С другой стороны, при переборе с помощью цикла for..in в число ключей попадают те, которые принадлежат к цепочке прототипов.

Энумерабельность и итерабельность

Это свойства ключей и значений объекта (метасвойства, свойства свойств). Вот каким образом можно представить из различие:

Ключи и значения в объекте соответствуют **именам** и **сущностям**.	
Энумерабельность: доступны **имена**, а через них — сущности (литеральный объект, энумерабелен и **не** итерируем, есть for..in нет for..of) задаётся метасвойством (атрибутом ключа)	***Итерабельность***: доступны **сущности**, и не через имена, а путём продвижения/обхода (обычный массив: энумерабелен **и** итерируем есть и for..in и for..of) задаётся специальным методом Symbol.iterator

Если создать объект литералом:

```
const o = { first: 'Elias', last: 'Goss' };
```

то мы можем перечислить его ключи (first и last) с помощью for..in:

```
for (let k in o) console.log(k);
```

Чтобы создать ключ методом метапрограммирования, мы должны явно указать, хотим ли мы, чтобы он был перечислим:

Reflect.defineProperty(o, 'age', {value:44, enumerable:true});

В консоли node мы можем получить все (собственные, т. е. содержащиеся не в прототипе объекта, а в нём самом) ключи: require('util').format('%o', o).

В рассматриваемом случае, если мы хотим увидеть и несобственные ключи, то должны написать require('util').format('%o', Object.prototype).

При изучении темы ключей объектов может возникнуть сложность из-за того, что слово in по-разному ведёт себя:

- как оператор (показывает не энумерабельный ключ);
- как часть цикла (НЕ показывает не энумерабельный ключ, хотя может и показывать, если несобственное; поэтому ключ length показывается в for..in у структуры NodeList — оно не сделано неэнумерабельным и оно несобственное, см. https://kodaktor.ru/enum1_arrays).

Следовательно, на эту особенность нужно обратить особое внимание.

Ещё одним источником путаницы является разнообразие способов обращения к ключам и их разное отношение к собственным/несобственным и перечислимым/неперечислимым ключам.

Метод Object.getOwnPropertyNames показывает все собственные ключи (перечислимые и неперечислимые).

С другой стороны, цикл for..in показывает все перечислимые ключи (собственные и несобственные).

В то же время метод Object.keys показывает как бы пересечение этих множеств: собственные перечислимые ключи.

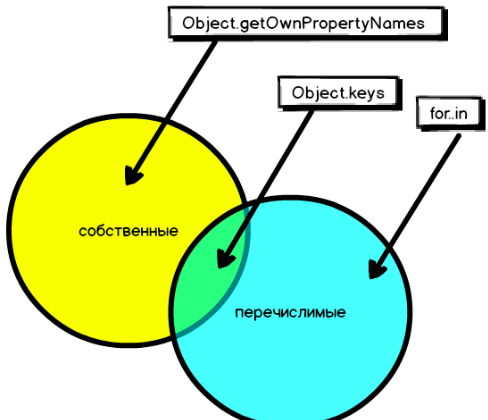


Рис. 44. Методы получения ключей относительно их метасвойств

Оставим для исследования читателю следующие четыре соприкасающиеся с вышерассмотренным содержанием темы (ниже приведены ссылки на борды кодактора с объяснениями и упражнениями по этим вопросам):

1. Мутабельность и расширяемость объектов. Object.preventExtensions() делает объект нерасширяемым (но не его прототип). Object.seal() и Object.freeze() с разной степенью жёсткости необратимо запрещают изменения объекта.

2. Обращение к свойствам, обозначенным как недоступные для записи или удаления (writable, configurable). Поведение языка может показаться странным в некоторых случаях: казалось бы, при попытке изменения значения свойства, созданного только для чтения, должно было бы быть выдано сообщение об ошибке, но (https://kodaktor.ru/g/proxy) этого не происходит. Прокси-объекты и ловушки (trap) позволяют решать задачу создания обработчиков таких ситуаций.

3. Выполнение сценария в контексте некоторого объекта ([4, с. 83], платформа node, функции runInThisContext, script.runInNewContext()или vm.runInNewContext()). Собственные ключи такого объекта становятся областью видимости такого сценария.

4. Символы, итераторы и генераторы (https://kodaktor.ru/iterable).

Итератор — это объект с особым интерфейсом, спроектированным для итераций [2]. Все объекты-итераторы имеют метод next(), возвращающий объект результата. С итераторами тесно связаны итерируемые (iterable) объекты — объекты, обладающие свойством Symbol.iterator. Стандартный символ Symbol.iterator определяет функцию, возвращающую итератор для данного объекта. Массивы, множества и ассоциативные массивы, а также строки являются в ES2015 итерируемыми, поэтому для них определены итераторы по умолчанию. Упомянутый выше Object.keys возвращает итератор, значениями которого являются собственные перечислимые ключи. Оператор расширения rest/spread работает со всеми итерируемыми объектами и использует итератор по умолчанию. Он позволяет преобразовать множество в массив. Генератор (https://kodaktor.ru/gen0) — это функция, возвращающая итератор. Функции-генераторы отмечаются звездочкой (*) после ключевого слова function и используют новое ключевое слово yield. Определение коллекции элементов NodeList в DOM (спецификации HTML, а не ES2015) включает итератор по умолчанию, который действует, подобно итератору по умолчанию массива. Это означает, что NodeList можно использовать в цикле for..of или в любом другом контексте, где применяется итератор по умолчанию. Символы — введённый в ES2015 новый примитивный тип данных — не имеют литерального представления. Они могут быть интересным образом использованы как значения констант, про которые достаточно знать, что они различны, но которые невозможно прочитать из памяти явным образом.

Вернёмся к формату JSON. В JavaScript существует объект JSON, который позволяет применить метод stringify к объекту и получить JSON-корректное представление (сериализацию). Это подобно сжатию с потерями: если, к примеру, в объекте были литералы функций, они окажутся выброшенными. Некоторые объекты вообще невозможно корректно сериализовать, так как они содержат циклические ссылки.

В node с помощью метода format из пакета 'util' можно получить в некотором смысле наиболее полное, глубокое с точки зрения вложенности представление объекта.

https://kodaktor.ru/g/util.format

В этом месте нужно обратить внимание на то, что представление, доступное нам при взаимодействии с программой, отличается от того, с чем имеет дело интерпретатор. Мы всегда имеем дело с той или иной формой текста, даже когда используем дизассемблер или бинарный режим редактора. Иными словами, видим не сами байты, а их изображение. Это тем более касается объектов как сущностей сложной структуры.

В разных средах текстовое представление выглядит по-разному:

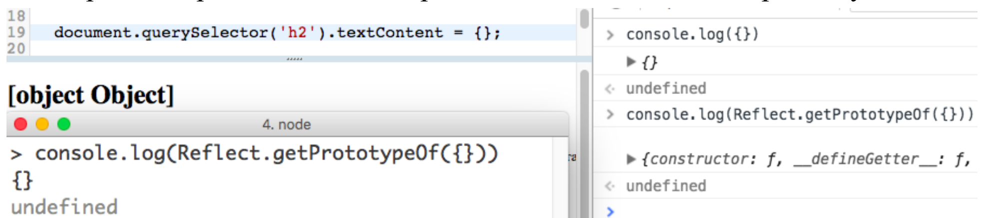


Рис. 45. Визуальное представление объекта

Как должна быть интерпретирована строка [object Object]? Она напоминает MIME-типы ('text/html' и т. п.), где правая часть более специфична, чем левая: в данном случае она отражает конструктор.

Методы toString и valueOf находятся в Object.prototype. Метод toString возвращает строковое представление объекта, а метод valueOf возвращает *сам объект* (хотя, повторимся, поскольку он выводится в консоль или на экран браузера, то так или иначе преобразуется в текст).

Поведение методов toString и valueOf можно переопределить или уничтожить:

```
> ({})
{}
> ({}+'')
'[object Object]'
> delete Object.prototype.toString
true
> ({}+'')
TypeError: Cannot convert object to primitive value
```

Рис. 46. Удаление метода toString

В данном примере мы удалили toString у всех объектов сразу. При попытке выполнить действие, требующее преобразования в строку, интерпретатор попытался вызвать toString и не нашёл. Можно сказать, что это своего рода неявный коллбэк. Метод valueOf мы не удаляли, но и не переопределили, а он возвращает сам объект (т. е. не примитивное значение, не строку). Если мы переопределим его, то если он возвращает строку, она будет использована, если же иное примитивное значение, то оно будет преобразовано в строку и использовано:

```
> Object.prototype.valueOf = ()=>5;
[Function]
> ({}+'')
'5'
```

Рис. 47. Переопределение метода valueOf

Здесь мы используем бинарный плюс, который, если хоть один из операндов является строкой, преобразует и другие операнды в строки. Обратившись к valueOf объекта, он получает 5, преобразованное в строку, конкатенирует с пустой строкой и получает '5'.

В случае необходимости создания клона (полной копии, дубликата) объекта могут возникнуть дополнительные вопросы. Поскольку объект может иметь много уровней вложенности, а значения могут иметь ссылочные типы, вот пример такого вопроса: следует ли создавать копию функции, особенно если она делает нечто, никак не связанное с данным объектом? В простых случаях мы можем осуществить сериализацию и восстановление объекта, использовать Object.assign и, на будущее, оператор распространения. В сложных случаях придётся осуществлять рекурсивный обход. Ситуация осложняется тем, что у ключей самих есть свойства, рассмотренные выше. У объекта может иметься цепочка прототипов. Клонировать ли и её также? Однозначного ответа на эти вопросы нет.

Такие инструменты как https://jsonlint.com/ позволяют проверить корректность формы JSON-документа. Если же мы хотим зафиксировать определённую структуру и потребовать, чтобы те или иные ключи могли иметь те или иные типы значений, мы должны создать схему. Сверку документа с его схемой будем называть валидацией.

Упражнение 7-2 (https://kodaktor.ru/g/json_intro)

Рассмотрим документ JSON по адресу kodaktor.ru/j/users. Мы видим, что значением ключа users является массив объектов вида {"login":..., "password":...}.

Он соответствует схеме, расположенной по адресу kodaktor.ru/api/schema/users:

Secure | https://kodaktor.ru/api/schema/users

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "description": "",
    "type": "object",
    "properties": {
        "users": {
            "type": "array",
            "uniqueItems": true,
            "minItems": 1,
            "items": {
                "required": [
                    "login",
                    "password"
                ],
                "properties": {
                    "login": {
                        "type": "string",
                        "minLength": 1
                    },
                    "password": {
                        "type": "string",
                        "minLength": 1
                    }
                }
            }
        }
    },
    "required": [
        "users"
    ]
}
```

Находящиеся в массиве объекты можно в некотором смысле описать как «безымянные», потому что в той структуре, которая представлена выше, у них нет какого-либо явного различающего признака, гарантирующего уникальность. В массиве могут оказаться объекты с полностью совпадающими ключами и значениями.

```
1 <script src="/j/out"></script>
2 <script>{
3   (async()=>{
4      const res = await fetch('//kodaktor.ru/j/users').then(x=>x.json());
5      Out.log(res.users);
6    })();
7 }</script>
```

```
[object Object],[object Object],[object Object],[object Object]
```

Но так как это массив, то мы, разумеется, можем обратиться к его элементам по индексам

```
const res = await fetch('//kodaktor.ru/j/users').then(x=>x.json());
Out.log(res.users[0].login);
```

```
student
```

или с помощью итерации:

```
const {users} = await fetch('//kodaktor.ru/j/users').then(x=>x.json());
Out.log(users.forEach(({login})=>Out.log(login)));
```

```
student
myuser
teacher
myking
```

Однако что, если мы хотим устроить поиск по такой совокупности данных? Как ответить на вопросы: «Есть ли такой логин?» и «Какой пароль у пользователя с таким логином?»?

Предположим, что ключ login является первичным ключом и однозначно определяет каждый объект, а пароли могут и совпадать.

Мы можем сравнивать перебираемые логины:

```
const {users} = await fetch('//kodaktor.ru/j/users').then(x=>x.json());
Out.log(users.forEach(({login}, i)=>{
    if (login==='teacher') Out.log(i);
}));
```

```
2
```

Недостатком такого подхода является то, что перебор продолжается и после нахождения логина, т. е. независимо от того, найден он или нет.

Здесь мы можем либо

(а) использовать **функциональный** подход на примере метода *find*

const user = users.find(({login}) => login==='teacher');

(получим найденный элемент, user ? user.password: 'Не найдено')

или *filter*:

(получим массив найденных элементов, в том числе пустой при ненахождении)

либо
(б) перебирать **императивно**, т. е. циклом; этот подход небезопасен: если искомого логина не окажется, мы выйдем за пределы размерности массива, т. к. счётчик будет увеличен и достигнет значения, на единицу большего, чем последний индекс (или равного длине массива, что то же самое):

```
7    const {users} = await fetch('//kodaktor.ru/j/users').then(x=>x.json());  users = (4) [{…}, {…}, {…}, {…}]
8    let i = 0;  i = 4
9    for (;users[i].login !== 'teacdher'; ++i) {console.log(i)};
10   Out.log(i);
```

```
Uncaught (in promise) TypeError: Cannot read property 'login' of undefined
```

Эту проблему можно разрешить, включив обработку исключения: https://kodaktor.ru/json_f0bfc, но лучше сделать так, чтобы исключение не возникало.

В нижеследующем фрагменте может быть выведен только один из двух вариантов: корректный индекс найденного элемента или вердикт о ненайденности: https://kodaktor.ru/json_57e2b.

Цикл продолжается, пока верны оба условия.

- если элемент присутствует, то первое условие (l > i) выполняется автоматически, а второе условие перестаёт выполняться при нахождении элемента и приращение ++i тоже не выполняется, т. е. индекс верный.
- если элемент отсутствует, то второе словие выполняется автоматически, а первое перестаёт выполняться, когда весь массив перебран и i становится равным его длине. Иными словами, если цикл закончился и при этом неверно что i < l, значит элемент не найден, в противном случае выводится верный индекс.

Массив перебирается целиком только в одном случае — когда элемента в нём нет.

Мы можем осуществить поиск с помощью lodash. Эта библиотека появилась до того, как в массивах JavaScript был реализован метод find. Если в нативном методе используется функция-предикат, то в lodash критерий поиска задаётся объектом
https://kodaktor.ru/json_ceb5b

```
const login = 'teacher';
const user = _.find(users, { login });

Out.log(user ? user.password: 'Not found');
```

```
qq
```

Чтобы удалить объект с логином teacher, например, нужно выполнить:

```
const res = _.remove(users, u => u.login === 'teacher');
```

Стрелочная лямбда играет роль предиката-отсеивателя, или фильтра: если она возвращает true, то такой объект удаляется.

При этом метод _remove возвращает массив удалённых объектов, который окажется пустым, если подходящих для удаления объектов в массиве не нашлось.

Если мы рассматриваем ключ как первичный, т. е., например, считаем, что логины уникальны, то можем сгенерировать объект, состоящий из объектов, у которых будут уникальные ключи, имена которых будут равны значениям этого первичного ключа:

```
const newusers = _.mapKeys (users, 'login');

console.log(newusers);
```

```
▼{student: {…}, myuser: {…}, teacher: {…}, myking: {…}}
  ▶myking: {login: "myking", password: "myqueen"}
  ▶myuser: {login: "myuser", password: "mypas"}
  ▶student: {login: "student", password: "tneduts"}
  ▶teacher: {login: "teacher", password: "qq"}
```

Мы также можем построить решение на чистом JavaScript с использованием метода reduce:
https://kodaktor.ru/json_c1656

ГЛАВА 8. ПРОТОТИПЫ И КОНСТРУКТОРЫ

Как уже говорилось в главе 7, при создании литерального объекта автоматически создаётся ещё один объект, который называется прототипом. Но это касается всех сущностей, которые относятся к объектам. Более того, это относится и к примитивным типам данных.

Например, когда мы создаём переменную, которой присваиваем строковое значение, мы можем сразу обратиться к её свойству length или методу includes. Но свойства есть только у объектов. Но при обращении к строке как к объекту происходит создание объекта-обёртки. И у этого объекта есть прототип, содержащий метод includes:

```
const first = 'Elias';
console.log(first.includes('li')); // true
console.log(first.includes('ea')); // false
// вызов через прототип
console.log(String.prototype.includes.call(first, 'li')); // true
console.log(String.prototype.includes.call(first, 'ea')); // false
console.log(Reflect.apply.call(null, String.prototype.includes, first, ['li'])); // true
console.log(Reflect.apply.call(null, String.prototype.includes, first, ['ea'])); // false
```

Выше показаны три способа обращения к методу includes:

- прямой;
- косвенный;
- метапрограммистский.

По способу указания в справочных материалах можно быстро понять, находится ли ключ в прототипе всех объектов данного класса или только в самом классе:

Рис. 48. Обозначения методов через прототипы

Запись типа Object.prototype.propertyIsEnumerable означает, что вызывать метод propertyIsEnumerable нужно уточнением конкретного объекта. Запись типа Object.getOwnPropertyNames означает, что вызывать метод нужно ровно как он представлен, передавая ему нужные аргументы (включая конкретный объект).

Объекты обёртки создаются, как уже было сказано выше, автоматически. Основой для их создания являются конструкторы (String, Number, Boolean — это функции, спроектированные как конструкторы). Всегда можно обратиться к ним в явной форме: просто вызвав эти функции, передав им в качестве аргумента литерал строки или значение, которое будет преобразовано в строку. Поскольку это именно конструкторы, их можно вызвать с помощью оператора new (т. е. экземплификацией, см. модели вызова функций в главе 4). Однако конкретно в случае с конструкторами обёрток это считается нерекомендованной практикой (https://eslint.org/docs/rules/no-new-wrappers).

Конструктор — это функция, объявленная с помощью слова function (не стрелочная), про которую предполагается, что она будет основой для создания объектов некоторой одинаковой структуры. Соответственно, чтобы это имело смысл, такая функция должна эту структуру создавать, а следовательно, она должна быть заранее спроектирована в аспекте свойств (и методов).

Вызов конструктора осуществляется с оператором new. Если конструктор предполагает наличие аргументов, то вызов сопровождается скобками, но рекомендуется использовать скобки в любом случае (https://eslint.org/docs/rules/new-parens).

Функция может определить факт вызова с ключевым словом new, и для этого ES2015 определяет метасвойство new.target. Метасвойство — это необъектное свойство с дополнительной информацией о его цели (такой, как new). Целью обычно является конструктор вновь созданного экземпляра объекта, который будет присвоен ссылке this в теле функции. То есть new. target — это ссылка на функцию, вызванную с ключевым словом new. При вызове с помощью call или apply new.target получает значение undefined. Метасвойство new.target можно также использовать в конструкторах классов, чтобы определить, как вызывался класс. В простейшем случае new.target содержит ссылку на функцию-конструктор класса.

Ничто не мешает использовать произвольную function-функцию в роли конструктора, но это в целом не особенно полезно:

```
const Cube = function (x) { return x ** 3; };
const cb = new Cube(5);
console.log(cb); // Cube {}
console.log(cb.constructor); // [Function: Cube]
```

Свойство constructor, доступное в новом объекте (оно содержится в его прототипе), позволяет определить, какая функция была вызвана как конструктор для этого объекта.

В частности, это может быть анонимная функция:

```
/* eslint new-parens: 0 */
const cb = new function () { return { first: 'Elias' }; };
console.log(cb); // { first: 'Elias' }
console.log(cb.constructor); // [Function: Object]
```

Как видно из примера выше, если конструктор возвращает *объект*, то он и является тем, что возвращает оператор экземплификации.

Если конструктор не возвращает объект, то создаётся новый объект, который должен был бы заполняться свойствами в рамках конструктора, и в следующем примере он получается пустым:

```
/* eslint new-parens: 0 */
const cb = new function () {  };
console.log(cb); // {}
console.log(cb.constructor); // [Function]
```

Каким же образом конструктор должен заполнять структуру вновь создаваемого объекта?

Когда объект создаётся (т. е. когда конструктор не возвращает объект), он доступен внутри конструктора через слово this. Он тогда будет тем объектом, который возвращает оператор экземплификации.

```
const Cube = function (x) { this.x = x; };
const cb = new Cube(5);
console.log(cb); // Cube { x: 5 }
console.log(cb.constructor); // [Function: Cube]
```

Таким образом, у объекта cb теперь есть свойство x с некоторым значением.

Чтобы объект оправдывал своё название объекта, у него должны быть методы.

Методы принято добавлять не в теле конструктора, а отдельно, записывая их в качестве свойств объекта, доступного через свойство prototype функции-конструтора.

```
const Cube = function (x) { this.x = x; };
Cube.prototype.volume = function () { return this.x ** 3; };
const cb = new Cube(5);
console.log(cb); // Cube { x: 5 }
console.log(cb.constructor); // [Function: Cube]
console.log(cb.volume()); // 125
```

Такое обращение с методами объясняется тем, что значения свойств у каждого конкретного объекта предполагаются своими, а методы нет никакой нужды копировать в конкретный объект.

Фактически можно смотреть на прототип как на расшаренный между объектами набор методов, на который у них у всех есть ссылка. Естественно, изменения в прототипе мгновенно отражаются на всех объектах, имеющих этот прототип.

Свойство prototype является особенным. Для обычного объекта оно ничем не выделяется среди всех прочих возможных свойств. Присвоение значения этому свойству для них не играет никакой специфической роли. Иное дело — конструкторы. Array, String и т. п. имеют прототипы, в которых находятся методы, которые можно вызвать в контексте каждого литерала массива, строки и т. п.

При этом следует помнить, что объект-прототип доступен через свойство prototype только в одной ситуации, приведённой выше: когда по конструктору создаётся новый объект. В примере выше метод volume будет находиться в прототипе объекта cb, но не будет доступен через его свойство prototype.

Каким же образом мы можем убедиться в том, что свойство или метод находится в прототипе объекта?

Собственные ключи

В главе 7 мы уже обозначили это, но повторимся: собственным (own) называется ключ (свойство) объекта, который находится в этом объекте, но не в его прототипе. Мы можем получить массив собственных ключей как минимум двумя способами:

- Object.getOwnPropertyNames;
- Reflect.ownKeys.

Если мы можем написать cb.constructor и получаем некоторый результат, но этого свойства нет в массиве собственных ключей, вывод только один: это свойство находится в прототипе. Но в примере выше мы определили только метод volume. Откуда же берётся свойство constructor?

Когда мы писали Cube.prototype, то подразумевали, что объект, скрывающийся за этим выражением, уже существует. Любая функция получает свойство prototype, потому что любая функция может быть использована как конструктор. Свойство prototype ссылается в этом случае на объект, у которого установлено свойство constructor.

```
const Cube = function (x) { this.x = x; };

console.log(Cube.prototype); // Cube {}
console.log(Object.getOwnPropertyNames(Cube.prototype)); // [ 'constructor' ]
console.log(Cube.prototype.constructor); // [Function: Cube]
```

Рис. 49. Связь свойств constructor и prototype

Это свойство constructor ссылается на саму функцию, которой принадлежит свойство prototype, содержащее это свойство constructor.

Мы использовали метод Object.getOwnPropertyNames чтобы убедиться, что свойство constructor находится в объекте, доступном как Cube.prototype.

При экземплификации этот объект станет прототипом экземпляра класса Cube и именно поэтому мы сможем получить его значение. Соответственно, это не будет его собственным свойством. Единственным собственным свойством будет x.

```
const cb = new Cube(5);
console.log(cb); // Cube { x: 5 }
console.log(cb.constructor); // [Function: Cube]
console.log(Object.getOwnPropertyNames(cb)); // [ 'x' ] === Reflect.ownKeys(cb)
```

Однако почему в предыдущем примере при выводе в консоль Cube.prototype мы не увидели там свойства constructor?

Ответ: потому что оно обозначено как неперечислимое, а такие свойства выводятся только с помощью специальных методов вроде методов получения собственных ключей.

Чтобы получить доступ к прототипу объекта, мы не можем обратиться к **его** свойству prototype. Свойство prototype, если вообще имеет значение, отличное от undefined, относится не к объекту, которому принадлежит, а к объектам, создаваемым на основе конструктора. Для получения доступа к прототипу объекта o, мы используем:

- Reflect.getPrototypeOf(o);
- o.__proto__.

Причем второй способ, хотя и часто работает по факту, но является deprecated.

Любой объект может стать прототипом для другого объекта. Для этого можно предложить как минимум три варианта действий:

- присвоить этот объект свойству прототип конструктора;
- использовать метод Object.create;
- использовать метод Reflect.setPrototypeOf.

Цепочка прототипов

Соотвественно существует цепочка прототипов, т. е. когда объект является чьим-то прототипом и при этом сам имеет прототип.

В JavaScript определена сущность, которая становится «изначальным прототипом», т. е. находится в начале этой цепочки, когда объекты создаются автоматически. Она доступна через выражение Object.prototype

```
console.log(Reflect.getPrototypeOf({}) === Object.prototype); // true
const o = Reflect.getPrototypeOf(Object.prototype);
console.log(o); // null
console.log(require('util').format('%o', o)); // 'null'
```

То есть у того, что известно как Object.prototype, нет и не может быть прототипа.

В нашем примере прототипом прототипа экземпляра класса Cube тоже является Object.prototype.

```
const Cube = function (x) { this.x = x; };
Cube.prototype.volume = function () { return this.x ** 3; };
const cb = new Cube(5);
console.log(cb.constructor); // [Function: Cube]

const cbProt = Reflect.getPrototypeOf(cb); // Cube { volume: [Function] }
console.log(cbProt === Cube.prototype); // true
const cbProtProt = Reflect.getPrototypeOf(cbProt);
console.log(cbProtProt === Object.prototype); // true
```

Cube есть конструктор класса Cube. Поэтому объект cb приобрёл в качестве прототипа то, что находится в Cube.prototype.

Object есть конструктор класса Object. Поэтому, собственно, все объекты, созданные с помощью этого конструктора, и приобретают в качестве прототипа то, что находится в Object.prototype.

Object.prototype не является единственной сущностью, у которой нет прототипа.

```
const empty = Object.create(null);
const o = Reflect.getPrototypeOf(empty);
console.log(o); // null
```

Возвращаясь к заданному выше вопросу (каким же образом мы можем убедиться в том, что свойство или метод находится в прототипе объекта?), мы теперь можем дать на него такой ответ:

1. Убедиться, что это свойство или метод не выводится в массиве собственных ключей.
2. Получить прототип и убедиться, что это свойство или метод, наоборот, выводится в массиве собственных ключей прототипа (но это может касаться объектов выше в цепочке прототипов вплоть до Object.prototype).

Подытоживая, уточним, каково всё-таки взаимоотношение между конструктором и прототипом. В главе 7 мы начали отвечать на этот вопрос, здесь же закончим.

Класс — это множество объектов, наследующих свойства от общего объект-прототипа. Это по определению: два объекта являются экземплярами одного класса, только если они наследуют один и тот же прототип.

Но создать несколько объектов, наследующих один и тот же прототип, можно по-разному: фабрично или с помощью конструктора. Фабричное создание объекта не оперирует словом new и, следовательно, не задействует конструктор.

Если два объекта наследуют один и тот же прототип, обычно (но не обязательно) это означает, что они были созданы и инициализированы с помощью одного конструктора.

Прототип первичен потому, что

- можно объявить несколько функций и присвоить их свойству prototype один и тот же объект, тогда при их вызове как конструкторов они будут создавать объекты одного класса — по определению;
- можно создать несколько объектов фабрично по одному и тому же прототипу, и это опять же сформирует класс объектов.

Вот пример фабричного создания объекта по прототипу, определённому в примерах выше:

```
const cb2 = Object.create(Cube.prototype, {
  x: {
    writable: true,
    configurable: true,
    value: 4
  }
});
console.log(cb2.x); // 4
console.log(cb2.volume()); // 64
```

Здесь мы указали прототип и собственное свойство (x) нашего объекта во втором аргументе метода Object.create. Это так называемый propertiesObject (объект с описанием свойств другого объекта). Если он определён, то его собственные перечислимые свойства (т. е. те, что определены в нём самом, а не перечислимые свойства в рамках его цепочки прототипов) определяют дескрипторы свойств, которым предстоит быть добавленным и во вновь создаваемый объект с соответствующими именами. Эти свойства соответствуют второму аргументу метода Object.defineProperties.

Однако имя конструктора обычно используется в качестве имени класса, т. е. если прототип определяет поведение объекта, то конструктор определяет, так сказать, его фасад. В JavaScript существует оператор instanceof, правый операнд которого представляет собой имя конструктора.

Оператор instanceof проверяет, наследует ли объект cb свойство Cube.prototype.

Он не проверяет, был ли объект cb инициализирован конструктором Cube.

Тем не менее, правым операндом указывается именно конструктор.

Мы можем сначала создать объект без прототипа, потом назначить ему прототип (Reflect.setPrototypeOf), и результат будет таким же.

Упражнение 8-1

С помощью директивы require импортируйте объект o3 под именем obj из модуля, код которого находится по адресу https://kodaktor.ru/j/protochain.

Определите цепочку прототипов этого объекта и экспортируйте результат в качестве массива имён этого прототипа.

Классы ES2015

Стандарт ES2015 позволил перейти к созданию объектов с помощью синтаксиса классов. На самом деле «под капотом» всё равно происходит использование прототипов, но реализация наследования стала более удобной (мы не будем приводить рассказ о том, как это делалось в ES5, но подробности можно найти в материалах Д. Крокфорда и в книге Д. Флэнагана [1]). В книге Н. Закаса [2] приводится детальное объяснение того, как создание объекта с помощью объявления класса и его экземплификации соответствует созданию объекта с помощью конструктора.

```
1   const Cube = function (x) { this.x = x; };
2   Cube.prototype.volume = function () { return this.x ** 3; };
3   const CubeClass = class {
4     constructor(x) {
5       this.x = x;
6     }
7     volume() {
8       return this.x ** 3;
9     }
10  };
11  const cb = new Cube(5);
12  const cbCl = new CubeClass(5);
13
14  console.log(cb.constructor); // Cube
15  console.log(cbCl.constructor); // CubeClass
16
17  const cbClProt = Reflect.getPrototypeOf(cbCl); // CubeClass {}
18  const cbProt = Reflect.getPrototypeOf(cb); // Cube { volume: [Function] }
19  console.log(cbClProt === CubeClass.prototype); // true
20  console.log(cbProt === Cube.prototype); // true
```

Рис. 50. Связь конструкторов классов и прототипов

В этом примере показано, что классы просто собирают вместе то, что ранее писалось раздельно: свойства и методы. Методы по-прежнему находятся в прототипе, только при объявлении с помощью класса они неперечислимы. Впрочем, это несложно переопределить:

```
const cbClProt = Reflect.getPrototypeOf(cbCl); // CubeClass {}
Reflect.defineProperty(cbClProt, 'volume', { enumerable: true });
console.log(Reflect.getPrototypeOf(cbCl)); // CubeClass { volume: [Function: volume] }
```

В примере выше определения класса в 1–2 строках и в 3–10 работают очень похоже, но не полностью эквивалентно, поскольку в 1–2 строках не ут-

верждается неперечислимость метода volume. Есть и ещё несколько отличий синтаксиса классов, в том числе:

- весь код в объявлении класса автоматически выполняется в строгом режиме и внутри классов нет никакой возможности выйти из строгого режима;
- вызов конструктора класса без ключевого слова new завершается ошибкой;
- попытка затереть имя класса внутри метода завершается ошибкой.

Если вызвать конструктор Cube без new, ничего страшного не произойдёт: просто будет вызвана функция Cube. Так как в ней нет return, этот вызов возвратит undefined.

В случае с синтаксисом классов попытка вызвать без new приведёт к ошибке: TypeError: Class constructor CubeClass cannot be invoked without 'new'.

Соответственно, чтобы учесть эти особенности, можно написать такую реализацию класса Cube на основе конструктора:

```
let Cube2 = (() => {
  'use strict';

  const Cube2 = function (x) {
    if (typeof new.target === 'undefined') {
      throw new Error('Use new!');
    }
    this.x = x;
  };
  Object.defineProperty(Cube2.prototype, 'volume', {
    value() {
      if (typeof new.target !== 'undefined') {
        throw new Error('Use new!');
      }
      return this.x ** 3;
    },
    enumerable: false,
    writable: true,
    configurable: true
  });
  return Cube2;
})();
```

Рис. 51. Эквивалентная реализация класса на основе функции-конструктора

Использование IIFE позволяет собрать части класса воедино и защитить имя Cube2 от изменений внутри тела IIFE (благодаря const), оставив возможность внешнему коду переприсвоить переменную Cube2 (благодаря let).

Что касается затирания имени класса, то здесь вспомним об именовании функций в главе 5.

```
1   /* eslint no-new: 0 */
2   /* eslint new-parens: 0 */
3   /* eslint prefer-const: 0 */
4   /* eslint no-func-assign: 0 */
5
6   let F = function F() {
7     F = 4;
8     console.log(F.name);
9   };
10  new F; // F
11
12  let G = class G {
13    constructor() {
14      // G = 4; // TypeError: Assignment to constant variable.
15      console.log(G.name);
16    }
17  };
18  new G; // G
```

Рис. 52. Попытка присвоения значения имени класса

В случае с функцией, если убрать F справа от function, результатом будет не F, а undefined. Это потому что инструкция let позволила переприсвоить значение, а у числа нет свойства name. Свойство name у функции примет значение, указанное справа от function, если оно будет указано. Если после let стоит одно имя, а после function другое, то предпочтение будет отдано тому, которое стоит после let.

В строгом режиме попытка присвоить значение имени, указанному после function внутри функции, вызывает ошибку, потому что внутри функции объявление function рассматривается как объявление константы.

```
'use strict';

let F = function FF() {
  // FF = 4; // TypeError: Assignment to constant variable.
  console.log(F.name);
};
new F; // FF
// new FF; FF is not defined
```

Рис. 53. Попытка присвоения значения имени функции

В классах, как говорилось выше, действует строгий режим. Поэтому попытка изменить имя, заданное словом class (если оно задано) вызовет ту же ошибку.

Но снаружи класса изменить его вполне возможно, если использовать let.

```
 let          function
───── X =  ────────── [Y]
const          class
```

Рис. 54. Схема наименования функций и классов

Суммируя вышесказанное, слова function/class могут выступать в роли объявителей переменных самостоятельно, и в этом случае они создают имя, ви-

димое как внутри функции/класса, так и снаружи, при этом class работает как const всегда, а function работает как const только в строгом режиме.

Если использовать let/const, то возможен случай указания имени после function/class и неуказания. В случае указания оно становится доступно только внутри функции/класса, становясь значением X.name.

Упражнение 8-2

Пусть класс Doubler определяет объекты, у которых есть внутреннее свойство n, доступное через геттер result, а также метод double, вызов которого без всяких аргументов удваивает значение этого свойства n. Что должен возвращать метод double чтобы реализовать паттерн «цепочка методов» («method chain»)? Напишите этот класс и пример вызова.

Упражнение 8-3

Объясните появление следующих результатов использования оператора instanceof:

```
  Array instanceof Object
true
  Array instanceof Function
true
  Function instanceof Array
false
  Array instanceof Array
false
  Object instanceof Object
true
  Function instanceof Function
true
  Function instanceof Object
true
```

super

Здесь мы оставим читателю для самостоятельного исследования **super** — указатель на текущий прототип объекта, фактически значение выражения Object.getPrototypeOf(this), который был введён в ES2015. Как указывает Н. Закас [2, с.107], с помощью ссылки super можно вызвать любой метод прототипа объекта при условии, что вызов производится в методе, объявленном с применением сокращенного синтаксиса. Можно обращаться к конструктору базового класса, вызывая метод super(); производные классы обязаны использовать super() в конструкторе (не производные классы не имеют права это делать). Если не определить конструктор в производном классе, функция super() будет вызвана автоматически со всеми аргументами, указанными в инструкции создания нового экземпляра класса. Функция super() должна вызываться перед любыми попытками использовать this в конструкторе, так как она отвечает за инициализацию this. Вызов super() может быть опущен только если конструктор класса возвращает объект.

Упражнение 8-4

Дано описание класса Father с пропущенным описанием класса Son:

```
class Father {
  constructor(name = 'Darth') {
    this.name = name.toUpperCase();
  }
  get myName() {
    return this.name;
  }
}
const father1 = new Father();
console.log(father1.myName);

class Son extends Father {




}
const son1 = new Son();
console.log(son1.myName);
```

Напишите код класса Son так, чтобы выполнялось следующее:

```
const son1 = new Son('Luke'); // DARTH
console.log(son1.myName); // Luke DARTH's son
```

ГЛАВА 9. МОДУЛЬНОСТЬ И ТРАНСПИЛЯЦИЯ

Методическое примечание

Эту тему можно рассмотреть сразу после темы функций на материале импорта и экспорта констант и функций, а затем перейти к прототипам и классам.

Модульность подразумевает выделение в приложении независимо реализованных частей, которые зависимы друг от друга в контексте всего приложения. Они изолированы (silo) и обладают неким механизмом управления зависимостями на достаточно абстрактном уровне (например, DI — dependency injection). Когда речь идёт о веб-интерфейсе, мы также говорим о компонентах, делая акцент не только на независимой разработке частей, но и повторном использовании кода, позволяющем избежать его прямого дублирования.

Задача управления компонентами в браузере возникала не раз, и такие её решения, как React, обладают неоспоримыми достоинствами. Но часто бывает, что сторонние решения как бы подстёгивают развитие стандарта, ядра, нативной реализации. В реализацию DOM встроен способ нативной компонентизации (https://kodaktor.ru/compo2), позволяющий определить шаблоны структуры/контента кастомных элементов, которым управляет JavaScript. По ряду аспектов (например, темплейты и слоты) эта реализация напоминает Vue.

Начиная с ES2015, JavaScript позволяет определять изолированный код на уровне языка. Если раньше для этого приходилось:

- использовать IIFE;
- транспилировать код, содержащий CommonJS-директиву require, в код, совместимый с браузерами,

то теперь в современных браузерах поддерживается нативная модульность на директивах import и export (атрибут type="module" в элементе script, см. https://kodaktor.ru/module_import и https://kodaktor.ru/module_export).

Методическое примечание

В этом месте можно вернуться к главе 2 (её завершению и упражнениям 2-1 и 2-2 на работу require и сравнение с работой функции require_once в языке PHP).

В новом синтаксисе, напоминающем деструктуризацию объектов, мы можем экспортировать, например, такие сущности, как переменные, ассоциированные с функциями.

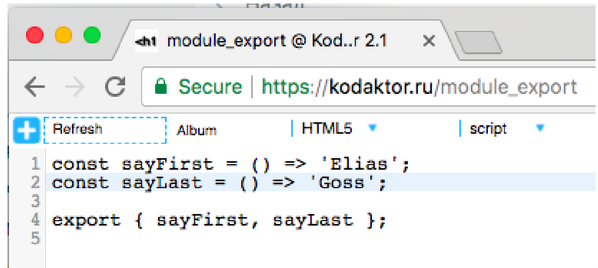


Рис. 55. Экспорт сущностей из модуля

Можем импортировать эти сущности в отдельной веб-странице, используя атрибут module элемента script:

```
1 <script src="/j/out"></script>
2 <script type="module">
3   import  { sayFirst, sayLast } from '/j/module_export';
4
5   Out.log(sayFirst());
6   Out.log(sayLast());
7
8 </script>
```

```
Elias
Goss
```

Рис. 56. Экспорт сущностей в модуль

Обычные скрипты не теряют, впрочем, своей актуальности, поскольку позволяют пользоваться переменными, глобальными для всех скриптов веб-страницы, к чему разработчики пока остаются привычными.

На момент написания этого текста версия 10.2.0 платформы node поддерживала нативную модульность в экспериментальном режиме:

- файлы должны иметь расширение .mjs ("Michael Jackson Script");
- вызов файла осуществляется с флагом:
  node **--experimental-modules** module_import.mjs.

module_export.mjs

```
1 const sayFirst = () => 'Elias';
2 const sayLast = () => 'Goss';
3
4 export { sayFirst, sayLast };
5
```

module_import.mjs

```
1 import  { sayFirst, sayLast } from './module_export.mjs';
2
3 console.log(sayFirst());
4 console.log(sayLast());
5 console.log('That is all for now!');
6
```

```
 $ node -v
v10.2.0
→ eliasgoss@ ~/Yandex.Disk.localized/docs-1111111/js-2017/js_samples
 $ node --experimental-modules module_import.mjs
(node:72042) ExperimentalWarning: The ESM module loader is experimental.
Elias
Goss
That is all for now!
```

Рис. 57. Импорт и экспорт в node v.10.2 (май 2017)

На момент написания этого текста, опять-таки, наиболее распространённой практикой было писать код с import и export в обычных файлах .js и не запускать их нативно, а пропускать через транспилятор с плагином babel-preset-env, автоматизируя этот процесс с помощью бандлера webpack.

Если мы хотим запускать такие сценарии в режиме командной строки с помощью nodemon, то может понадобиться его специальная настройка:

https://github.com/GossJS/express_starters1/blob/step2esm/**nodemon.json**

{
 "execMap": {
 "mjs": "node --experimental-modules"
 }
 }

Тогда автоматизировать запуск сценария можно с помощью секции scripts и main файла package.json:

```
{
  "scripts": {
    "start": "nodemon --watch"
  },
  "engines" : {
    "node" : ">=8.5.0"
  },
  "main": "index.mjs",
  "devDependencies": {
    "nodemon": "^1.12.1"
  }
}
```

Рис. 58. Настройка nodemon

Это позволит запускать файл index.mjs просто командой yarn start или npm start

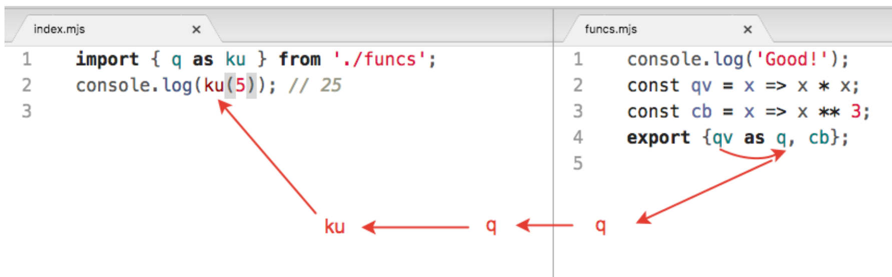


Рис. 59. Переименование при импорте и экспорте

Примеры экспорта и импорта класса см. по адресам:

- https://kodaktor.ru/personclass.simple.mjs
- https://kodaktor.ru/personobject.simple.mjs

Упражнение 9-1

Создайте модуль с классом для системы npm и задокументируйте его.

Транспиляция

Под транспиляцией понимается процесс преобразования кода, написанного с помощью следующих версий языка или на диалектах JavaScript в некий

стандартный вариант, понимаемый всеми браузерами. Для его осуществления предназначен Babel (https://babeljs.io/).

Первая версия Babel (ранее 6to5) была выпущена в 2014 г. и представляла собой инструмент, с помощью которого можно было преобразовать синтаксис ES6 в синтаксис ES5. В ходе развития проекта стала появляться платформа для поддержки всех самых последних изменений в ECMAScript.

Когда в спецификацию ECMAScript предлагают включить новую функцию (tc39), она проходит несколько стадий одобрения, от стадии 0, Strawman (только что предложенная и исключительно экспериментальная), до стадии 4, Finished (одобренная в качестве составной части стандарта). Babel предоставляет пресеты (заготовки) для каждой из этих стадий:

- babel-preset-stage-0: Strawman;
- babel-preset-stage-1: Proposal;
- babel-preset-stage-2: Draft;
- babel-preset-stage-3: Candidate.

Пресеты состоят из плагинов, каждый из которых отвечает за совершение того или иного преобразования.

На момент написания данного текста **babel-preset-env** является пресетом для транспиляции из кодов ES2015, ES2016, ES2017, обобщающим предыдущие пресеты.

Мы можем использовать babel для исследования языка и, в частности, анализа того, как создатели транспилятора реализовали те или иные алгоритмы. Для этого удобен REPL по адресу babeljs.io/repl или инструменты командной строки.

Упражнение 9-2

Используя babel-preset-env, транспилируйте следующий сценарий:

BABEL Learn ES2015

```
1 let
2   a = 3,
3   b = 7;
4 [a, b] = [b, a];
```

И сравните результат с теми реализациями обмена переменных, которые рассматривались в главе 6 при обсуждении деструктуризации массивов.

Для этого:

1. Создайте проект в папке:

```
mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && yarn init -y
```

2. Добавьте babel-cli и babel-preset-env в раздел девелоперских зависимостей

yarn add --dev babel-cli babel-preset-env

(в случае автоматизированной сборки проекта вместо babel-cli скорее всего будет достаточно усечённого варианта **babel-core**).

3. Создайте простейшую настройку babel в файле .babelrc

echo '{"presets":["env"]}' > .babelrc

4. Создайте файл с описанным выше контентом (index.js)

```
   20 Feb 12 00:20 .babelrc
   38 Feb 12 00:23 index.js
 8432 Feb 12 00:19 node_modules
  185 Feb 12 00:19 package.json
58779 Feb 12 00:19 yarn.lock
```

```
{
  "name": "20180212_001921",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "devDependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-env": "^1.6.1"
  }
}
```

5. Выполните вызов транспилятора, указав его исполнимый файл, исходный файл и результирующий файл:

npx babel index.js -o result.js

Опционально можно создать соответствующий сценарий в package.json (без npx).

Выше мы уже обсудили нативную модульность и отсутствие её поддержки в стабильной версии node (вне экспериментального режима).

В babel-cli входит инструмент babel-node, с помощью которого можно запускать файлы, находящиеся в отношении нативной модульности. При этом происходит как бы мгновенная транспиляция и передача на выполнение результата движку node. Но это способ, который годится только для девелоперского этапа.

Для продакшн мы всегда можем преобразовать эти файлы пофайлово так, чтобы их можно было выполнить с помощью node, не прибегая к транспиляции.

Добавим к нашему проекту библиотеку moment для работы с датами/временем.

yarn add moment

Разместим файлы, находящиеся в отношении нативной модульности, в папке ./src

Исходные файлы в папке ./src

файл	содержимое

main.js	import moment from 'moment';
	import name from './name';
	console.log(name);
	console.log(moment.unix('1500514362').format('DD.MM.YYYY HH:mm:ss'));

name.js	export default 'E. B. Goss';

Для наглядности здесь одна зависимость (moment) берётся из node_modules (yarn add moment), а другая из той же папки ./src

```
$ node ./src/main
/Users/eliasgoss/20180212_001921/src/main.js:1
(function (exports, require, module, __filename, __dirname) { import moment from 'moment';
                                                               ^^^^^^
                                        исполнение с помощью node - неудачно
SyntaxError: Unexpected token import
    at new Script (vm.js:51:7)
    ...
    at startup (bootstrap_node.js:193:16)
-▶ eliasgoss@ ~/20180212_001921
$ npx babel-node ./src/main   исполнение с помощью babel-node - успешно
E. B. Goss
20.07.2017 04:32:42
```

Выполнив команду

npx babel ./src -d ./lib

мы получим папку lib с транспилированными файлами, в которых вместо import и export будут require и module.exports

```
$ npx babel ./src -d ./lib
src/main.js -> lib/main.js
src/name.js -> lib/name.js      пофайловое преобразования
-▶ eliasgoss@ ~/20180212_001921
$ ls
index.js     lib     node_modules     package.json     result.js     src
-▶ eliasgoss@ ~/20180212_001921
$ ls ./lib
main.js name.js
-▶ eliasgoss@ ~/20180212_001921
$ node ./lib/main
E. B. Goss
20.07.2017 04:32:42     успешное исполнение транспилированного кода
```

файл	содержимое

main.js	'use strict';
	
	var _moment = require('moment');
	
	var _moment2 = _interopRequireDefault(_moment);
	
	var _name = require('./name');
	
	var _name2 = _interopRequireDefault(_name);
	
	function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { default: obj }; }
	
	console.log(_name2.default);
	con-
	sole.log(_moment2.default.unix('1500514362').format('DD.MM.YYYY HH:mm:ss'));

name.js	'use strict'; Object.defineProperty(exports, "__esModule", { value: true }); exports.default = 'E. B. Goss';

Итак, на самом базовом уровне транспиляция отображает файлы с нативной модульностью в файлы с CommonJS-модульностью 1:1.

Но задача часто состоит ещё и в объединении получившихся файлов.

Сборка проекта

Мы можем построить модульный проект из многих файлов, в том числе изолированных друг от друга модулей. Но исторически так сложилось, что в браузере появилась тенденция интегрировать все файлы в один (бандл, bundle.js). Это объясняется поведением протокола HTTP, которое вынуждает браузер создавать множественные соединения, некоторые одновременно, большинство последовательно. Чтобы ускорить этот процесс, файлы собираются в один.

Есть и ещё работа для сборщика: при использовании технологий, требующих обработки кода (таких как SASS, TypeScript, JSX и т. п.) удобно соединить вместе процесс обработки и организации результата в некоторое целое. Можно поручить транспиляцию с помощью babel одному звену сборки, создание CSS другому и так далее. В случае webpack этим занимается загрузчик (loader).

При программировании с использованием следующих версий языка (как бы с обгоном современности по встречной полосе) транспиляция оказывается безальтернативной. В 2018 мы можем запускать программу с модулями без транспиляции, но всё равно надёжнее получить код, работающий без включения экспериментальных ключей и настроек.

На момент написания этого текста разработчики webpack представили версию 4. В качестве killer feature представлена нуль-конфигурация: до версии 4 для запуска даже самого простого проекта требовалось написать достаточно неуклюжий файл типа webpack.config.js или webpack.config.babel.js, но теперь самые базовые настройки (такие как точки входа и выхода) включены по умолчанию. Пример см. по адресу https://github.com/GossJS/webpack.

Продемонстрируем бандлинг как таковой

```
mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && yarn init -y
```

Установим webpack и инструмент для его вызова из командной строки

```
yarn add --dev webpack webpack-cli
```

Создадим папку src, в которой webpack по умолчанию начнёт поиск с файла index.js

```
mkdir ./src
echo "console.log(require('./name'));" > ./src/index.js
echo "module.exports = 'Elias';" > ./src/name.js
```

Выполним первый файл, который обращается ко второму:

```
node ./src
Elias
```

Теперь запустим сборку с параметрами по умолчанию, т. е. без всяких конфигурационных файлов, при которой webpack начнёт анализ с точки входа ./src/index.js и создаст итоговый файл, включающий в себя используемые зависимости:

```
npx webpack
```

и получим файл ./dist/main.js
... который и является бандлом и может быть использован независимо от исходных файлов.

Сборщик не осуществляет транспиляцию. Однако он собирает бандл из модулей. И это значит, что для сборки бандла в том числе из нативных модулей транспиляция не нужна:

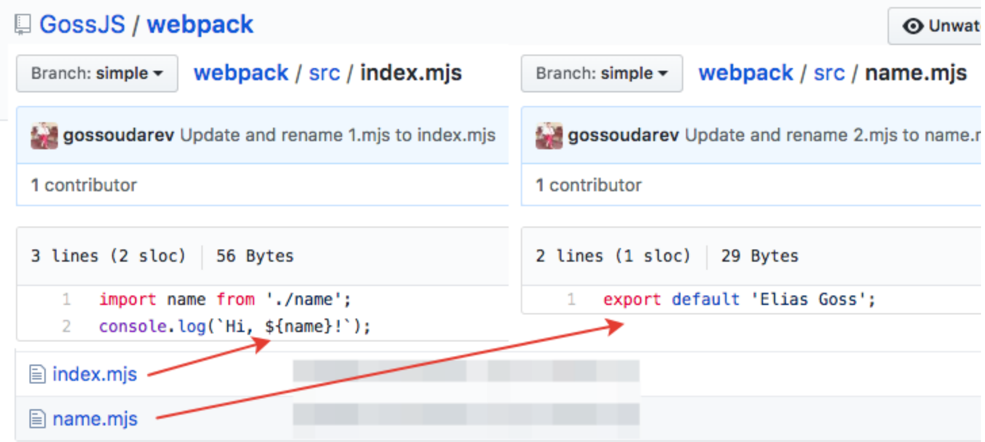


Рис. 60. Исходные файлы для работы webpack

Мы можем создать бандл из уже транспилированных файлов: например, используем в качестве index.js и name.js файлы, соответственно, main.js и name.js, полученные в результате работы babel на предыдущем этапе. При этом нужно будет добавить moment к проекту (yarn add moment).

Мы получим файл, который включает в себя ещё и библиотеку moment.js.

Результат можно наблюдать по адресу https://kodaktor.ru/moment_simple (файл с бандлом переименован из main.js в moment_simple.js).

То есть иными словами webpack собирает как из нативных mjs-модулей, так и из обычных файлов js, независимо от того, используется ли в них нативная модульность или CommonJS. Самое главное, чтобы от точки входа можно было переходить по директивам импорта/затребования к анализу других файлов.

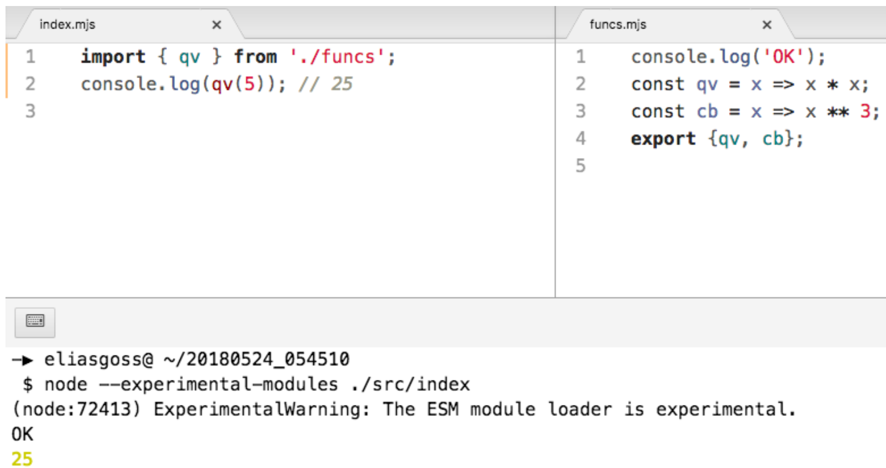


Рис. 61. Выполнение сценария
с включением экспериментальной поддержки модулей

```
index.mjs
1   import { qv } from './funcs';
2   console.log(qv(5)); // 25
3
```

```
funcs.mjs
1   console.log('OK');
2   const qv = x => x * x;
3   const cb = x => x ** 3;
4   export {qv, cb};
```

```
yarn run v1.7.0
warning package.json: No license field
$ webpack
Hash: 68c1330d19ab7fe23b19
Version: webpack 4.8.3
Time: 514ms
Built at: 2018-05-24 05:58:11
    Asset       Size  Chunks             Chunk Names
bundle.js  609 bytes       0  [emitted]  main
Entrypoint main = bundle.js
[0] ./src/index.mjs + 1 modules 139 bytes {0} [built]
    | ./src/index.mjs 56 bytes [built]
    | ./src/funcs.mjs 83 bytes [built]
✨  Done in 2.59s.
→ eliasgoss@ ~/20180524_054510
 $ node bundle
OK
25
```

Рис. 62. Выполнение сценария, собранного из модулей в бандл

Для решения более сложных задач нужны, конечно, конфигурационные файлы, которые указывают загрузчики, и плагины, и прочие подробности сборки. Здесь полем для исследования становится оптимизация собранного файла по размеру и другим характеристикам.

Видимо, наиболее частой сферой применения webpack являются сборки проектов, написанных с использованием не только модульности или новых особенностей ES, но и специальных языков типа TypeScript или JSX. То есть проекты на Angular и React очевидно собираются с помощью webpack; при этом используются достаточно изощрённые конфигурационные файлы, создание которых превращается в один из видов программирования. Это относится как к клиентской, так и серверной стороне. Например, при использовании React Server Side Rendering (SSR) язык JSX применяется на серверной стороне, что требует создания даже нескольких конфигурационных файлов и, возможно, некоторой стратегии управления ими.

Упражнение 9-3

Импортируйте модули moment и faker в сценарий, использующий их для вставки даты-времени и случайного адреса электронной почты в программно созданный и вставленный в body элемент pre, так, чтобы при каждом обновлении страницы появлялась новая дата-время и новый случайно сгенерированный адрес электронной почты:

```
1 .
2 <script src="/.                    "></script>
```

```
.
01.03.2018 06:34:51
Whitney_Witting90@hotmail.com
```

Здесь собирается проект, который будет работать в браузере. Поэтому будут использоваться обращения к методам и свойствам DOM:

- document.createElement;
- document.body.appendChild;
- setAttribute;
- textContent.

Этот проект является по существу моделью того, как осуществляется взаимодействие с приложением, разработанным с использованием таких библиотек/фреймворков как React. *Исходная идея*: JavaScript-сценарий выполняет всю работу, включая генерацию необходимых тегов. Это значит, что от веб-страницы требуется минимум тегов (собственно говоря, только body). Отчасти эта идея лежит в основе работы SPA (Single Page Application).

Часть I. Создайте проект

mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && yarn init -y

0. Добавьте в проект webpack и webpack-cli в качестве девелоперских зависимостей:

yarn add --dev webpack webpack-cli

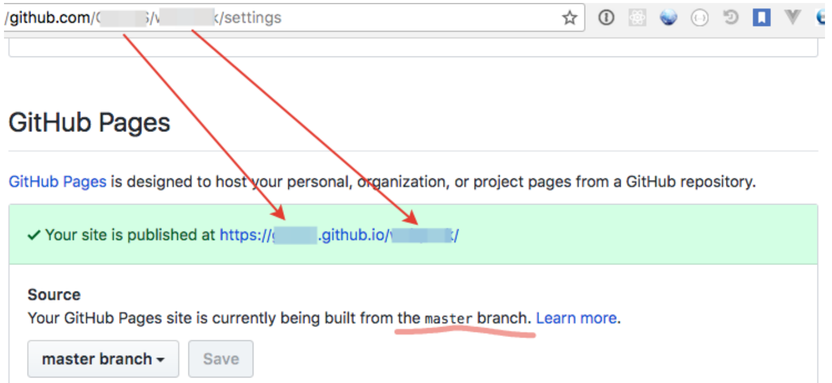
1. Добавьте в проект модули moment и faker
yarn add moment faker.
2. Импортируйте moment из moment и internet из faker в файле ./src/index.js.
3. Выражение document.createElement('pre') позволит вам создать элемент pre для упрощения вывода данных в тело веб-страницы.
4. Его следует подать на вход вызову метода document.body.appendChild и получившийся результат поместить в переменную (назовём её el).
5. Добавляйте содержимое к элементу pre инструкцией вида el.textContent += '\n';
Вам нужно добавить результат вызова moment().format с соответствующим форматом даты-времени и вызов internet.email().
6. По аналогии нужно создать и вставить элемент h4 для выполнения условия z7a (https://kodaktor.ru/g/z7a) — используйте метод setAttribute для установки свойств title и id.
7. Запустите сборку, выполнив npx webpack или внеся команду webpack в секцию scripts файла package.json.

```
"scripts" : {
  "build": "webpack"
},
```

с дальнейшим запуском вида yarn run build

Часть II. Обеспечьте интернет-доступ к проекту (здесь приведён вариант с GitHub Pages, но главное найти возможность хостить бандл)

8. Создайте репозиторий в GitHub. Загрузите файл main.js в главную ветку вашего репозитория.
9. Включите поддержку GitHub Pages



10. В файле README.MD вставьте теги <script src="main.js"></script> и далее перейдите по адресу
https://имя.github.io/репо/
убедитесь, что
(а) выполняется условие, показанное на рисунке, т. е. сценарий работает, как предполагается;
(б) выполняется условие z7a.

Упражнение 9-4

Примените транспилятор для использования декораторов, обсуждавшихся в главе 5 и оператора bind. В борде https://kodaktor.ru/babel_deco показано, как это можно сделать непосредственно в браузере, подключив транспилятор babel через элемент script и его атрибут type со значением "text/babel". Создайте аналогичный проект с помощью webpack и разместите на GitHub Pages.

В качестве развития этой темы полезно поупражняться в добавлении новых возможностей ES.Next, таких как свойства класса или оператор pipe. Уровнем высшего пилотажа здесь, вероятно, стало бы написание и тестирование собственного плагина, реализующего какое-нибудь синтаксическое усовершенствование языка.

index.js

```
class Person {
    name = 'Elias';
}

const e = new Person();

console.log(e.name);
```

old.js

```
class Person {
  constructor() {
    this.name = 'Elias';
  }
}

const e = new Person();

console.log(e.name);
```

.babelrc

```
{
  "presets": [
    "env"
  ],
  "plugins": [
    "transform-class-properties"
  ]
}
```

ЗАКЛЮЧЕНИЕ

Представленная вашему вниманию версия пособия призвана обеспечить введение в разработку на языке JavaScript. Здесь намеренно излагалось содержание, которое мало зависит от того, на какой платформе запускаются сценарии. Приложения, которые одинаково работают в браузере и вне его, принято называть изоморфными. Специфике разработки на фронтэнде и бэкэнде следует посвятить отдельные пособия. Их подготовкой автор занимается уже сейчас.

В приложениях же к этому тексту располагаются примеры (псевдо)тестовых заданий, курсовых проектов (они же темы для выступлений на практических занятиях) и некоторых лабораторных работ.

ПРИЛОЖЕНИЕ 1

Примеры вопросов для итоговой аттестации

№	Формулировка	Ответ
1.	В ES2015 появился метод Number. _____ (), который может определить, является ли указанное значение целым числом. Когда в вызов этого метода передается значение, по его внутреннему представлению движок JavaScript определяет, является ли оно целым числом. В результате числа, которые выглядят как вещественные (например, 7.0), в действительности могут храниться как целые, и для них этот метод будет возвращать true	isInteger
2.	Какой класс из JavaScript API используется для создания парсера DOM, содержащего, в числе прочего, метод parseFromString?	DOMParser
3.	Какое свойство ключа объекта отвечает за видимость этого ключа в цикле for...in?	enumerable
4.	Согласно списку предложений в ECMAScript (tc39), на какой стадии находится по состоянию на 04.05.2018 предложение Observable? Введите одну цифру	1
5.	В инструкции инициализации литерала массива квадратными скобками определите и выпишите символ, завершающий литерал массива. let r = ['first', 'second', 'third']; // массив r	]
6.	Назовите имя переменной-счётчика в инструкции цикла for (let c=0;c<d;c++)	c
7.	Назовите встроенный объект языка JavaScript, который инкапсулирует математическую функциональность (его методы sqrt() или floor())	Math
8.	Какое значение должно быть возвращено при вычислении данного выражения: (x=>y=>x<<y)(5)(3)	40

Продолжение табл.

№	Формулировка	Ответ
9.	Выберите примитивные типы из списка. Выберите один или несколько ответов: Function Array Number Symbol Boolean	Number; Symbol; Boolean
10.	Назовите метод класса Function, позволяющий вызвать функцию, переопределив её this-контекст и передав её массив аргументов	apply
11.	Назовите бинарный оператор метапрограммирования JavaScript, позволяющий проверить, что левый операнд является экземпляром второго операнда как класса	instanceof
12.	Какое значение в JavaScript не равно само себе?	NaN
13.	Какой тип структуры возвращает метод document.querySelectorAll?	NodeList
14.	Назовите статический класс (объект) современного JavaScript, позволяющий переопределять в рамках метапрограммирования операции по отношению к объектам, используя Proxy в качестве обёртки вокруг каждого конкретного объекта	Reflect
15.	Пусть массив m есть [3,2]; Сколько различных элементов в массиве [1,2, ...m, 1]?	3
16.	Какое значение возвращает функция в JavaScript, если в определении функции не указано явно, что она возвращает?	undefined
17.	Назовите класс DOM/ JavaScript для создания собственных пользовательских событий (сообщений, обрабатываемых слушателями событий типа 'click')	CustomEvent

Продолжение табл.

№	Формулировка	Ответ
18.	Что является прототипом объекта, созданного инициализатором литерала объекта?	Object.prototype
19.	Что в языке JavaScript является результатом извлечения квадратного корня из отрицательного числа?	NaN
20.	Сколько раз в консоль будет выведена единица? `for(let i=0;i<3;++i){for(let i=0;i<4;++i){console.log(1)}}`	12
21.	Какова сумма элементов массива, получаемого в результате вычисления выражения `[1,2,3].map(x=>x*x)`	14
22.	Назовите унарный оператор JavaScript, который, будучи поставлен перед выражением (в том числе выражением вызова функции), превращает его значение в undefined	void
23.	Какое значение будет вычислено при исполнении этого выражения: `((a=2,b=10)=>Math.pow(a,b))()`	1024
24.	Назовите имя синхронной функции подключения внешних файлов в Node.js, включая зависимости и файлы JSON	require
25.	Назовите стандартное имя файла (с расширением) дескрипторов пакетов CommonJS, в котором задаются зависимости для современного веб-приложения на JavaScript	package.json
26.	Какое ключевое слово ES2015 позволяет обозначить экспортируемую сущность как доступную для импорта по умолчанию?	default

Продолжение табл.

№	Формулировка	Ответ
27	Какая папка содержит в Node.js-проекте зависимости, загруженные с помощью менеджера пакетов?	node_modules
28	Как называется метод жизненного цикла React-компонента, вызов которого означает, что компонент получил своё место в DOM, и в теле которого мы можем сделать асинхронный запрос?	componentDidMount
29	Назовите значение переменной x после выполнения кода: const { rgb: { numbers: { 2: x } } } = { sample: 'rgb(33,44,55)', rgb: { names: ['red', 'green', 'blue'], numbers: ['rgb(255,0,0)', 'rgb(0,255,0)', 'rgb(0,0,255)'] }};	rgb(0,0,255)
30	Как называется протокол, позволяющий отладить Node-приложение с помощью браузера Chrome/Chromium?	DevTools

Приложение 2

Примеры тем курсовых проектов / самостоятельных исследований

1. Исследование консоли разработчика Chrome vs инструменты FireFox.
2. Исследование типа данных Symbol.
3. Исследование типизированных массивов и буферов.
4. Исследование ассоциативных массивов и множеств.
5. Исследование LocalStorage по сравнению с cookies.
6. Исследование хранилищ данных «ключ — значение».
7. Исследование паттернов проектирования в JavaScript (Одиночка, MVC, RORO и др.).
8. Исследование потоков и observable.
9. Исследование WebRTC и вещания с веб-камеры, а также рабочего стола.
10. Сравнительный анализ интерфейсов, доступных JavaScript (Web Audio API, Bluetooth и др.).
11. Исследование WebGL.
12. Исследование TypeScript.
13. Исследование WebAssembly.
14. Исследование JavaScript в устройствах (контроллерах) и управления ими (например, Raspberry или Arduino).
15. Исследование системы JSDoc.
16. Исследование Отладчика NodeJS и асинхронного тестирования веб-приложений, в том числе с помощью безголового браузерного движка.
17. Сравнительное исследование библиотеки React и языка JSX и фреймворка Vue.
18. Исследование стратегий управления состоянием приложения (Flux/Redux, MobX и др.).
19. Исследование функционально-реактивного программирования и Rx.JS.
20. Исследование компилятора JavaScript V8.

Приложение 3

Примеры заданий для лабораторных работ

I. Атомарные задания

Каждое из таких заданий выполняется отдельно, независимо от других и может быть использовано для самопроверки или уточнения отметки при промежуточной или итоговой аттестации.

1. Ответьте на вопрос, создав форк борда: https://kodaktor.ru/pseudocheckin_js01_1.

2. Исправьте ошибку в коде: https://kodaktor.ru/bind02032018.

3. Осуществите импорт из репозитория npm зависимости под названием goss_concat и выясните, с помощью какого ключевого слова определяется объект, который экспортируется этим модулем.

4. Дополните код https://kodaktor.ru/on так, чтобы слушатель события запускался только единожды.

5. Возьмите строку zyx и последовательно передайте микросервисам (каждый раз передавая следующему результат работы предыдущего) https://kodaktor.ru/api/up/, https://kodaktor.ru/api/ps/, https://kodaktor.ru/api/rp/ и, наконец, https://kodaktor.ru/api/rv/ — каким будет результат работы последовательных выполнений fetch?

6. Напишите код JavaScript одной строкой для разбиения трёхбайтового шестнадцатеричного значения цвета на три десятичных числа.

7. Рассмотрите код в строках 64–66 на странице https://kodaktor.ru/js23012018 и доработайте его так, чтобы переменная restricted принимала не одно из двух, а одно из трёх различных значений: (а) значение yes при значении переменной age меньше 18; (б) значение notsure при значении переменной age, равном 18; (в) значение no в противном случае (это нужно сделать вкладыванием одного тернарного оператора в другой).

8. Сгенерируйте таблицу логинов пользователей согласно заданию по адресу https://kodaktor.ru/g/web06112017.

9. Используйте методы массивов from и map для формирования цепочки Unicode-символов согласно заданию по адресу https://kodaktor.ru/startask.

10. Реализуйте конкатенацию строк с помощью композиции функции согласно заданию по адресу https://kodaktor.ru/compo3.
11. Основываясь на интуитивном понимании кода, написанного с помощью библиотеки callbag, перепишите решение https://kodaktor.ru/rx3 на операторах Rx.JS.
12. Используйте код-заготовку по адресу https://kodaktor.ru/ed2f5ef для разработки асинхронного компонента, обращение к которому могло бы осуществляться с помощью инструкции await new MyButton('click').
13. Получите извне кода асинхронной функции возвращаемое ею значение: https://kodaktor.ru/06042018.
14. Осуществите последовательное посещение интернет-адресов с помощью генератора: https://kodaktor.ru/06042018_2.
15. Дополните код так, чтобы замыкание работало как счётчик: https://kodaktor.ru/closure_async_task.

II. Лабораторные задания

Каждое из таких заданий (А, Б, и т. д.) занимает приблизительно от 1 до 3 академических часов и должно завершиться публикацией результата на ресурсе, позволяющем опубликовать статические страницы (например, GitHub Pages или кодактор) и/или на ресурсе, позволяющем запустить серверное приложение (например, Heroku). Ссылки на работающие ресурсы вместе с краткими рекомендациями по проверке их работоспособности должны быть включены в состав веб-портфолио.

А. Создание изображения с помощью JavaScript Canvas API.

1. Создайте веб-страницу (борд) на Кодактор.ру или ином онлайн-редакторе.
2. Вставьте элемент script и определение функции

```
function makeCanvas(x, y) {
    const canvas = document.createElement('canvas'),
          ctx = canvas.getContext('2d');
    canvas.setAttribute('width', x);
    canvas.setAttribute('height', y);
    return { canvas, ctx };
}
```

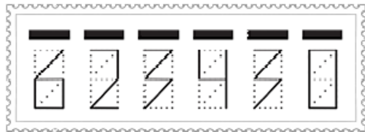
3. Вызовите функцию для создания переменных, содержащих холст и контекст

```
const { canvas, ctx } = makeCanvas(300, 120).
```

4. Добавьте холст в дерево DOM

```
document.body.appendChild(canvas).
```

5. Используя методы контекста, изобразите свои инициалы, использовав в качестве образца написание в стиле почтового индекса:



6. Используя кривую Безье, нарисуйте под инициалами дугу другого цвета:

7. Используя аффинные преобразования (https://kodaktor.ru/affine), осуществите поворот этого изображения на 90 градусов против часовой стрелки.

8. Создайте форк этого борда (поставив на него ссылку в первом и обратно), в котором этот логотип используется как фоновый узор.

Б. Развёртывание простого приложения на основе React с компонентным интерфейсом в браузере с использованием транспилятора babel, получающего данные из JSON-ресурса.

Пример работы React-приложения в браузере: https://kodaktor.ru/min_react_live

JSX — это язык шаблонизации и создания повторно используемых компонентов, позволяющий смешивать инструкции JavaScript и HTML-разметку. React — это библиотека для управления компонентами, использующая идею состояния компонента для реактивного (автоматического) обновления частей компонента, связанных с состоянием.

1. Создайте веб-страницу (борд) на Кодактор.ру или ином онлайн-редакторе, в которой есть только пустой элемент <div></div> в разделе body.
2. Подключите react, react-dom и babel.
3. Пишите код в элементе <script type="text/babel"></script>.
4. Главным компонентом должен быть список **VoteButtonList** кнопок для голосования, в каждую из которых передаётся название фреймфорка и количество актуальных голосов.
Подключите компонент в точку монтирования

```
ReactDOM.render(
  <VoteButtonList url="https://kodaktor.ru/j/react5b_6cbf2"/>,
  document.querySelector('div')
);
```

передав ему в параметре url адрес документа
https://kodaktor.ru/j/react5b_6cbf2

Secure | https://kodaktor.ru/j/react5b_6cbf2


```json
[
  {
    "title": "Angular",
    "votes": 0,
    "id": 0
  },
  {
    "title": "React",
    "votes": 3,
    "id": 1
  },
  {
    "title": "Vue",
    "votes": 2,
    "id": 2
  }
]
```

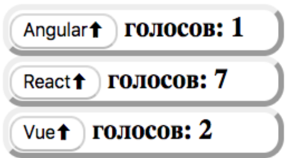

описывающего результаты голосования разработчиков при выборе наиболее перспективного фреймворка (библиотеки) для JavaScript. Необходимо извлечь массив данных в событии жизненного цикла компонента ComponentDidMount и сохранить в состоянии главного компонента

```javascript
componentDidMount() {
  fetch(this.props.url)
    .then(x => x.json())
    .then(frameworks => this.setState({ frameworks }));
}
```

или в асинхронной форме

```javascript
async componentDidMount() {
  const frameworks = await fetch(this.props.url).then(x => x.json());
  this.setState({ frameworks });
}
```

5. Напишите код компонента VoteButton, в состоянии которого хранится количество голосов, отданных за тот фреймворк, которому этот компонент соответствует. Каждый такой компонент должен содержать кнопку, увеличивающую хранимое внутри его состояния количество голосов. Вот пример списка из трёх компонентов VoteButton:



6. Разместите адрес борда (а также его скриншот и лог действий) в репозитории (веб-портфолио) и в специально созданном форуме в соответствующем курсе в СДО Moodle.

В. Создание интерфейса MaterialUI на React-платформе.

Цель: создать клиентский интерфейс с компонентом Date Picker (http://www.material-ui.com/#/components/date-picker) для реализации вычисления разницы между текущей датой и выбранной с помощью компонента.

1. Создайте проект

mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && yarn init -y.

2. Установите линтер и его файл настройки:

bash <(curl -s https://kodaktor.ru/g/eslint_exec)

3. Установите минимально необходимые зависимости, включая девелоперские:

yarn add react react-dom moment react-tap-event-plugin material-ui
yarn add --dev webpack webpack-cli webpack-dev-server babel-core babel-loader babel-preset-env babel-preset-react

4. Создайте файл настройки транспилятора:

echo ' {"presets":["env"]}' > .babelrc

5. Создайте файл настройки webpack:

curl 'https://kodaktor.ru/j/min_react_webpack4' -o 'webpack.config.babel.js'

```
1   const WDS_PORT = 1234;
2
3   export default {
4     mode: 'development',
5     devtool: 'sourcemap',
6     resolve: { extensions: ['.js', '.jsx'] },
7     module: {
8       rules: [{
9         test: /\.jsx$/,
10        loader: 'babel-loader',
11        query: {
12          babelrc: false,
13          presets: [
14            'react', ['env', {
15              modules: false,
16            }],
17          ],
18        },
19      }],
20    },
21    devServer: {
22      port: WDS_PORT,
23      host: '0.0.0.0',
24    },
25  };
```

6. Добавьте запускающий скрипт в файл package.json

```
"scripts": {
  "start": "webpack-dev-server"
},
```

7. В папке ./src создайте файл index.jsx и поместите в него JavaScript-сценарий, созданный в результате выполнения предыдущей лабораторной работы.

8. В корне приложения создайте файл index.html, в котором содержится строка

```
<div></div><script src="./main.js"></script>
```

9. Проверьте работу приложения, запустив его

```
yarn start
```

и перейдя по адресу localhost:1234 в браузере.

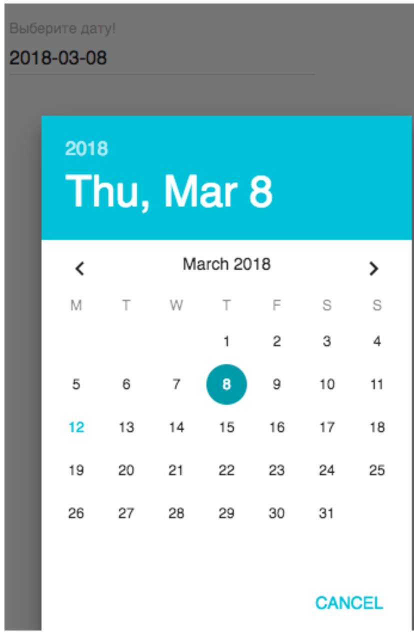
10. Для работы с элементами интерфейса MaterialUI нужно импортировать их

```
import MuiThemeProvider from 'material-ui/styles/MuiThemeProvider';
import DatePicker from 'material-ui/DatePicker';
import injectTapEventPlugin from 'react-tap-event-plugin';
```

и вставить метку, при щелчке по которой элемент будет появляться на веб-странице:

```
<MuiThemeProvider>
    <DatePicker
      onChange={(n = null, date) => alert(date)}
      floatingLabelText="Выберите дату!"
      autoOk={true}
    />
</MuiThemeProvider>
```

При щелчке по надписи «Выберите дату» должен отображаться искомый компонент:



11. Добавьте код, который с помощью moment получает текущую дату и вычисляет разницу в годах, месяцах и дней между ней и той, которая выбрана с помощью установленного компонента.

12. Разместите приложение в Интернете, а его адрес (а также скриншот и лог действий) в репозитории (веб-портфолио) и в специально созданном форуме в соответствующем курсе в СДО Moodle.

Г. Создание простого веб-сервера на основе Node.js.

1. Создайте новый проект:

mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && yarn init –y

или

mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && npm init -y
(https://kodaktor.ru/g/init).

2. Установите инструмент nodemon для автоматизации перезапуска сценария и moment для работы с датой и временем: yarn add --dev nodemon или npm i -D nodemon и yarn add moment или npm i moment

```
"scripts" : {
  "start": "nodemon"
},
```

3. Установите настройки линтера и создайте нужный файл .eslintrc.

4. Создайте в папке проекта файл index.js с содержимым:

```
1    const http = require('http');
2    const moment = require('moment');
3
4    http.createServer((req, res) => {
5      res.end(moment().format('DD.MM.YYYY HH:mm:ss'));
6    }).listen(4321);
```

5. Запустите сценарий yarn start и выполните curl localhost:4321.

6. Убедитесь, что в консоли отображается текущая дата и время.

7. Добавьте к проекту поддержку выдачи данных в формате JSON с выдачей соответствующего заголовка и кодировки UTF-8: http://kodaktor.ru/git-checkout.gif

```
1    const http = require('http');
2    const moment = require('moment');
3
4    http.createServer((req, res) => {
5      res.setHeader('Content-Type', 'application/json; charset=utf-8');
6      res.end(JSON.stringify({ date: moment().format('DD.MM.YYYY HH:mm:ss') }));
7    }).listen(4321);
```

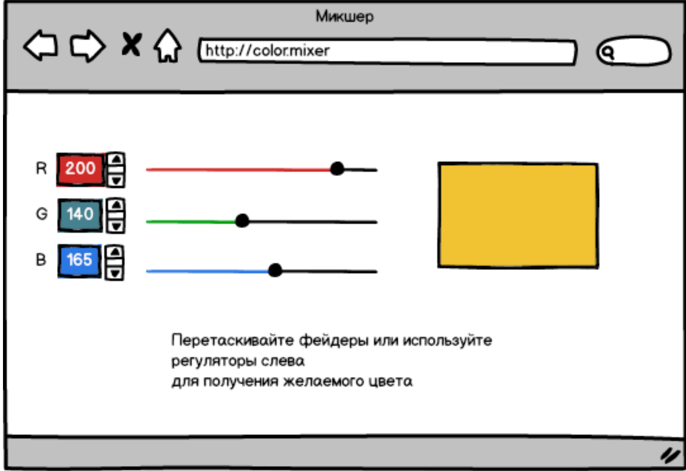
8. Перейдите по адресу localhost:4321 в браузере и убедитесь, что выдаётся ответ в формате JSON.
9. Осуществите рефакторинг кода так, чтобы коллбэк, отвечающий на запросы, явным образом указывался для события request:

```
1  const http = require('http');
2  const moment = require('moment');
3
4  const server = http.createServer();
5  server.listen(4321);
6  server.on('request', (req, res) => {
7    res.setHeader('Content-Type', 'application/json; charset=utf-8');
8    res.end(JSON.stringify({ date: moment().format('DD.MM.YYYY HH:mm:ss') }));
9  });
```

Д. Проектирование и разработка интерфейса микшера цветов — инструмента для подбора цветовых сочетаний для веб-страницы.

Цель: создать интерактивный интерфейс с использованием элементов управления HTML5 Shadow DOM таких как input type="number" и input type="range".

1. Спроектируйте интерфейс, используя средство прототипирования:



2. Создайте веб-страницу (борд) на Кодактор.ру или ином онлайн-редакторе.

3. Посмотрите, как должен работать интерфейс в динамике, используя борд https://kodaktor.ru/g/mixer.

4. Задайте минимальные и максимальные значения элементов управления.

5. Используя обработчики событий элемементов управления, синхронизируйте их между собой так, чтобы изменение значение элемента, отвечающего за конкретный цвет, вызывало реактивное изменение второго, парного ему элемента и наоборот. Например, перемещение фейдера, задающего синий цвет, должно вызывать изменение значения соответствующего числового счётчика в левой части интерфейса.

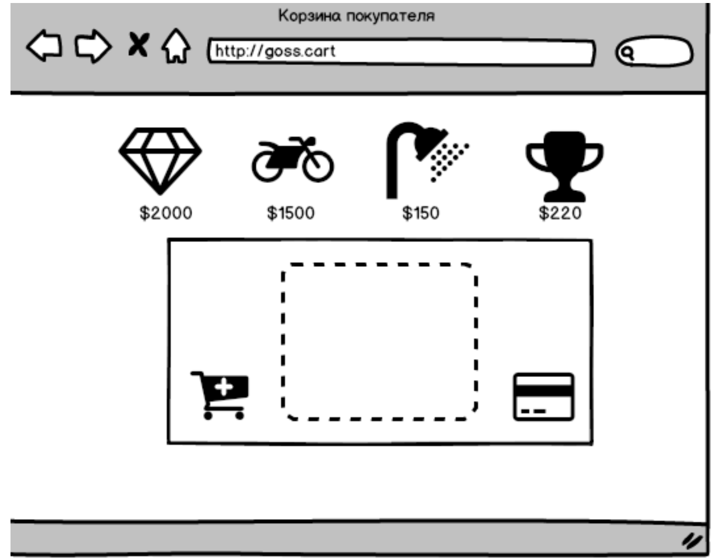
6. Изменение значений элементов управления должно сопровождаться изменением фонового цвета какого-либо прямоугольника на веб-странице либо всей веб-страницы.

6*. Добавьте текстовые поля, в которых отображаются доступные для копирования RGB-представления цветов в десятичном и шестнадцатеричном виде.

7. Разместите отчёт (лог действий) в репозитории (веб-портфолио).

Е. Проектирование и разработка интерфейса корзины покупателя в интернет-магазине.
Цель: создать интерактивный интерфейс с использованием интерфейса перетаскивания, реализованного в HTML5.

1. Спроектируйте интерфейс, используя средство прототипирования:



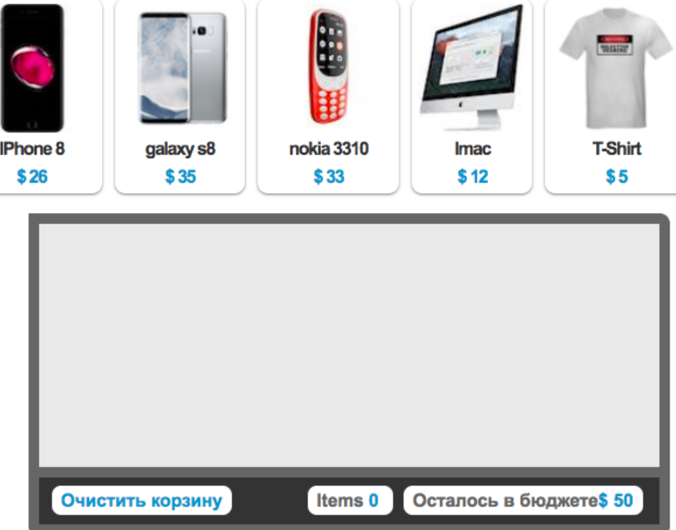
2. В верхней части корзины должна быть представлена галерея доступных товаров с ценниками. Пусть бюджет покупателя ограничен суммой в $2000.

3. Товары должны добавляться в корзину путём перетаскивания их изображений в область перетаскивания, обозначенную пунктирной рамкой. При этом должен вестись подсчёт суммарной стоимости заказанных товаров.

4. При превышении указанного выше лимита перетаскивание не должно завершаться успехом, вместо этого в таком случае должно выдаваться сообщение «Кредит превышен!».

5. Элементы, содержащие товары, и «корзина» должны быть реализованы в виде независимых друг от друга компонентов.
Используйте интерфейс CustomEvent для связывания компонентов между собой путём передачи сообщений и реализации модели событие-слушатель.

6. Используйте атрибуты draggable, ondragstart, ondragover и ondrop (https://kodaktor.ru/drop4) для реализации перетаскивания.



7. Разместите отчёт (лог действий) в репозитории (веб-портфолио).

ПРИЛОЖЕНИЕ 4

Направления самостоятельной работы

Рассмотренное в этом пособии содержание может быть использовано как площадка для переходного этапа к изучению серверного веб-программирования в первую очередь на платформе Node.js. Автору представляется, что удачным способом осуществить этот переход является изучение тестирования клиентских сценариев (в первую очередь своих собственных) и серверных сценариев (про которые известно, хотя бы примерно, как они должны работать).

Например, известно, что при отправке запроса по адресу http://kodaktor.ru/sleep/?n=6 через 6 секунд (или чуть более) должен поступить ответ в формате JSON. Исследуйте, какие инструменты удобно использовать, чтобы проверить корректность работы этого адреса (микросервиса) — проверить длительность паузы между отправкой запроса и получением ответа, проверить тип присланного микросервисом ответа и т. д. Вам могут помочь следующие ресурсы:

- https://kodaktor.ru/testing;
- https://kodaktor.ru/testing_add;
- https://kodaktor.ru/lr_gmailtest;
- https://kodaktor.ru/gmailtest;
- https://kodaktor.ru/g/lr_yandextestclick;
- https://kodaktor.ru/g/testing_promise;
- https://github.com/GossJS/testing.

Далее перейдите к тестированию приложений, использующих компоненты.

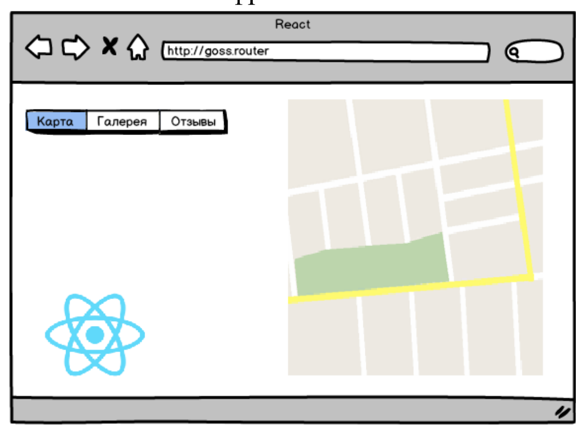
Исследуйте создание React-приложения с клиентским роутингом, переключающим представления из трёх компонентов на основе существующего шаблона. *Для этого:*

1. Создайте и инициализируйте проект в папке
cd && mkdir $(date +%Y%m%d_%H%M%S) && cd $_ && git clone -b heroku https://github.com/GossJS/reactDemo2018.git . && yarn

2. Далее нужно установить зависимость react-router-dom

yarn add react-router-dom

3. Работа роутера похожа на работу набора фреймов: меню, пространство для сменных фреймов и сами сменные фреймы.



В ветке Likers есть модифицированный ./src/index.jsx который содержит класс-компонент и функциональный компонент.
Сейчас в файле index располагается всё вместе. Там же находится главная команда рендеринга

```
r(
  <div><Counter stars="3" /><Counter stars="10" /></div>,
  document.querySelector('.cont'),
);
```

4. Вынесите счётчик в отдельный файл Counter.jsx

```
import React, { Component as C } from 'react';
class Counter extends C {
  constructor(props) {...}
  plus() {...}
  render() {...}
}
const Stars = ({ length }) => <span>...</span>;
export default Counter;
```

Файл index.jsx должен после указанных преобразований иметь вид:

```
import React from 'react';
import { render as r } from 'react-dom';
import Counter from './Counter';
r(
  <div><Counter stars="3" />
    <Counter stars="10" />
  </div>,
  document.querySelector('.cont'),
);
```

5. На следующем этапе создайте корневой компонент, играющий роль приложения.

```
import React from 'react';
import { render as r } from 'react-dom';
import Counter from './Counter';
const App = () => (
  <div><Counter stars="3" />
    <Counter stars="10" />
  </div>);
r(
  <App />,
  document.querySelector('.cont'),
);
```

Это начальный этап обёртывания, выстраивания матрёшечной последовательности вложенных друг в друга слоёв приложения. Так как всё это теги, то здесь проявляется декларативный подход к реализации функциональности.
То, что сейчас выглядит как <App /> будет обёрнуто в <BrowserRouter />.
То есть функциональность роутинга будет реализована за счёт помещения приложения в контекст маршрутизатора.

```
import React from 'react';
import { render as r } from 'react-dom';
import { BrowserRouter } from 'react-router-dom';
import Counter from './Counter';
const App = () => (
  <div><Counter stars="3" />
    <Counter stars="10" />
  </div>);
r(
  <BrowserRouter><App /></BrowserRouter >,
  document.querySelector('.cont'),
);
```

6. Заданная выше в скетче структура может быть реализована так. Сменные «страницы»-«фреймы» представлены компонентами. «Страница» отзывов, т. е. звёздочки-лайки у нас уже есть. Нужно ещё две «страницы», которые можно пока реализовать компонентами-заглушками. Пусть это будут Map.jsx и Gallery.jsx.
Постройте компонент с фрагментом карты (https://yandex.ru/map-constructor/), это будет iframe, который возвращает функциональный компонент.

```
Map.jsx
1  import React from 'react';
2
3  const Map = () =>
4    (<div><h1>Карта</h1><iframe title="Lomo" src="https://yandex.ru/map-widget/v1/?
5
6  export default Map;
7
```

```
Gallery.jsx
1  import React from 'react';
2
3  const Gallery = () =>
4    (<div><h1>Галерея</h1><p>Рисунки</p></div>);
5
6  export default Gallery;
7
```

```
[./node_modules/react/index.js] 190 bytes {main} [built]
[./src/index.jsx] 456 bytes {main} [built]
    + 80 hidden modules
i ｢wdm｣: Compiled successfully.
```

7. Пространство «контейнера» следует поделить на

- ту часть, где находятся пункты меню, они же ссылки-кнопки, позволяющие переходить от фрейма к фрейму, и имеющие условный атрибут target="имя_фрейма";
- ту часть, где располагается сменный контент, т. е., иными словами, происходит рендеринг маршрутов.

Пусть это будут компоненты Menu и Content, соответственно. Простейшая модель построения сменного контента есть выбор одного из нескольких вариантов в зависимости от соответствия условию, т. е. switch case. Поэтому в декларативном роутинге логично расположить маршруты в элементах Route внутри элемента Switch.

```
import { BrowserRouter, Switch, Route, Link } from 'react-router-dom';
const Content = () => (
  <main>
    <Switch>
      <Route exact path="/" component={Map} />
      <Route path="/gallery" component={Gallery} />
      <Route path="/counter" component={Counter} />
    </Switch>
  </main>
);
```

ИТОГО

```
import React from 'react';
import { render as r } from 'react-dom';
import { BrowserRouter, Switch, Route, Link } from 'react-router-dom';
import Counter from './Counter';
import Map from './Map';
import Gallery from './Gallery';

const Content = () => (
  <main>
    <Switch>
      <Route exact path="/" component={Map} />
      <Route path="/gallery" component={Gallery} />
      <Route path="/counter" component={Counter} />
    </Switch>
  </main>
);
const Menu = () => (
  <header>
    <nav>
      <ul>
        <li><Link to="/">Карта</Link></li>
        <li><Link to="/gallery">Галерея</Link></li>
```

```
        <li><Link to="/counter">Отзывы</Link></li>
      </ul>
    </nav>
  </header>
);
const App = () => (
  <div><Menu /><Content />
  </div>);
r(
  <BrowserRouter><App /></BrowserRouter>,
  document.querySelector('.cont'),
);
```

8. Чтобы в режиме WDS можно было вводить маршруты прямо в адресной строке типа

http://localhost:1234/gallery
необходимо включить опцию:

```
 devServer: {
    port: WDS_PORT,
    host: '0.0.0.0',
    historyApiFallback: true,
  },
```

9. Чтобы то же самое было возможно под управлением, например, Express, добавьте код

```
.get('/*', r => r.res.sendFile('docs/index.html', { root: '.' }))
```

10. Используйте форк репозитория-задания по адресу https://github.com/GossJS для размещения результата выполнения работы, который позволяет оценить его работоспособность: загрузите в репозиторий-ответ результат, скриншот и/или скринкаст выполнения и, по необходимости, инструкции для запуска результата.

11. Разработайте план проверки работоспособности приложения и произведите отбор инструментов из экосистемы React для реализации тестирования.

ЛИТЕРАТУРА

1. *Флэнаган*, *Д.* JavaScript. Подробное руководство / пер. с англ.. — 6-е изд. — СПб. : Символ-Плюс, 2012. — 1080 с.

2. *Закас*, *Н.* ECMAScript 6 для разработчиков. — СПб. : Питер, 2017. — 352 с.

3. *Браун*, *И.* Веб-разработка с применением Node и Express. Полноценное использование стека JavaScript. — СПб. : Питер, 2017.

4. *Пауэрс*, *Ш.* Изучаем Node. Переходим на сторону сервера. — 2-е изд., доп. и перераб. — СПб. : Питер, 2017. — 304 с.

5. *Стефанов*, *С.* React.js. Быстрый старт. — СПб. : Питер, 2017. — 304 с.

6. *Холмс*, *С.* Стек MEAN. Mongo, Express, Angular, Node. — СПб. : Питер, 2017. — 496 с.

7. *Кантелон*, *М.* Node.js в действии / М. Кантелон, М. Хартер, Т. Дж. Головайчук [и др.]. — СПб. : Питер, 2015. — 448 с.

8. *Пьюривал*, *С.* Основы разработки веб-приложений. — СПб. : Питер, 2015. — 272 с.

9. *Никсон*, *Р.* Создаем динамические веб-сайты с помощью PHP, MySQL, JavaScript, CSS и HTML5. — 4-е изд. — СПб. : Питер, 2016. — 768 с.

Илья Борисович ГОСУДАРЕВ

ВВЕДЕНИЕ В ВЕБ-РАЗРАБОТКУ НА ЯЗЫКЕ JAVASCRIPT

Учебное пособие

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*
Ответственный редактор *С. В. Макаров*
Корректор *Е. В. Разенкова*
Выпускающий *С. В. Орловский*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел./факс: (812) 336-25-09, 412-92-72
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 10.04.19.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 11,70. Тираж 100 экз.

Заказ № 275-19.

Отпечатано в полном соответствии с качеством
предоставленного оригинал-макета в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.