Эндрю Ринингсланд, Свизек Теллер

Визуализация данных с помощью библиотеки D3.js 4.x

Ændrew Rininsland, Swizec Teller

D3.js 4.x Data Visualization

Learn to visualize your data with JavaScript

Third Edition

Packt>
BIRMINGHAM – MUMBAI

Эндрю Ринсланд, Свизек Теллер

Визуализация данных с помощью библиотеки D3.js 4.x

Учимся визуализировать ваши данные с помощью языка JavaScript

ДМК Пресс
Издательство
Москва, 2017

УДК 004.22:004.92D3
ББК 32.972.1
Р51

Ринисланд Э., Теллер С.
Р51 Визуализация данных с помощью библиотеки D3.js 4.x / пер. с анг. А. А. Слинкина. 3-е изд. – М.: ДМК Пресс, 2017. – 298 с.: ил.

ISBN 978-5-97060-569-1

Книга знакомит с одной из самых распространенных и мощных библиотек визуализации данных – D3.js. Прочтя ее, вы сможете решить любую задачу: от создания визуализации с нуля до запуска ее на сервере и написания автоматизированных тестов.

Издание предназначено разработчикам веб-приложений, специалистам по анализу и обработке данных и всем, интересующимся интерактивным представлением данных в вебе с помощью библиотеки D3.

УДК 004.22:004.92D3
ББК 32.972.1

First published in the English language under the title 'Learning D3.JS 4.x Data Visualization – Third Edition – (9781787120358)

ISBN 978-1-78712-035-8 (анг.)
ISBN 978-5-97060-569-1 (рус.)

Содержание

Об авторах

Эндрю Ринсланд – разработчик и журналист, в последние десять лет он большую часть времени занимался созданием интерактивного контента для таких изданий, как Financial Times, Times, Sunday Times, Economist и Guardian. За три года работы в газетах Times и Sunday Times он участвовал в различных проектах – от написания некрологов о смерти таких личностей, как Нельсон Мандела, до резонансных расследований типа допингового скандала – крупнейшей в истории утечки данных об анализах крови спортсменов. В настоящее время является старшим разработчиком группы интерактивной графики в редакции Financial Times.

Выражаю благодарность своей восхитительной подруге Наоми, которая бог знает сколько раз подбадривала и нахваливала меня, пока я работал над книгой. Спасибо также всем членам группы D3.js Slack, оказывавшим мне всемерную поддержку, а особенно завсегдатаям канала #v4-migration, которые помогли освоиться с бесчисленными изменениями в версии 4.

Свизек Теллер, автор книги «Data Visualization with d3.js» – технарь по призванию. В 21 год он основал свой первый стартап, а теперь, в поисках очередной достойной идеи, трудится в роли специалиста по всем уровням веб-разработки, отдавая предпочтение внештатному сотрудничеству со стартапами на ранних этапах становления. Во время, свободное от кодирования, он ведет блог, пишет книги или выступает на различных мероприятиях в Словении и соседних странах, мечтая сделать доклад на какой-нибудь крупной международной конференции. В ноябре 2012 года он начал писать книгу «Почему программисты работают по ночам» и задался целью улучшить жизнь разработчиков, где бы они ни жили.

Я благодарен @gandalfar и @robertbasic, которые подначивали меня по ходу работы и выступали в роли подопытных свинок для проверки моих примеров. Также выражаю искреннюю признательность всем членам группы @psywerx, которые не дали мне спятить и создали один из лучших в мире наборов данных.

О рецензенте

Герардо Фуртадо – преподаватель биологии, популяризатор науки, автор книги по эволюционной биологии, опубликованной Бразильским университетом, а также нескольких художественных произведений.

Его второе академическое пристрастие – визуализация данных, именно оно в конечном итоге и привело его к D3.js, самой мощной (и комплексной) JavaScript-библиотеке для этой цели.

Предисловие

Здравствуйте, читатели! В этой книге вы познакомитесь с основами одной из самых распространенных и мощных библиотек визуализации данных, но не только. К концу нашего совместного путешествия вы приобретете все навыки, необходимые настоящему специалисту по D3, и сможете решить любую задачу: от создания визуализации с нуля до запуска ее на сервере и написания автоматизированных тестов. Тех, чьи знания JavaScript несколько подзамшели, ждет приятный сюрприз – эта книга располагает к использованию самых последних добавленных в язык средств, и мы всегда объясняем, почему это круто и чем отличается от «старой школы».

Краткое содержание книги

В главе 1 «Первое знакомство с D3, ES2017 и Node.js» рассматриваются новейшие средства для создания визуализации данных с помощью D3.

Глава 2 «Начала DOM, SVG и CSS» содержит обзор базовых веб-технологий, используемых в D3.

Глава 3 «Геометрические примитивы в D3» посвящена определению и созданию базовых геометрических элементов, из которых составляются визуализации данных.

В главе 4 «Извлечение пользы из данных» вы научитесь преобразовывать данные, так чтобы D3 могла их визуализировать.

Глава 5 «Все для удобства пользователя» покажет вам, как дополнить визуализацию средствами анимации и интерактивности.

Глава 6 «Иерархические макеты в D3» посвящена иерархической компоновке, которая поднимет ваши навыки владения D3 на новый уровень, где для создания сложных диаграмм применяются повторно используемые шаблоны.

В главе 7 «Прочие макеты» обсуждаются неиерархические макеты, позволяющие ускорить создание диаграмм других типов.

В главе 8 «Использование D3 на сервере с применением Canvas, Koa 2 и Node.js» описаны создание и развертывание веб-службы на базе сервера Node.js, которая отрисовывает визуализации D3 с помощью библиотек Koa.js и Canvas.

В главе 9 «Обретение уверенности в своих визуализациях» показано, как улучшить качество кода за счет проверки соблюдения стандартов кодирования, статической проверки типов и автоматизированного тестирования проектов.

В главе 10 «Проектирование хорошей визуализации данных» сопоставляются различные подходы к визуализации и предлагается набор рекомендаций.

Что необходимо для чтения книги

Вам понадобится компьютер, на котором можно запустить Node.js. В первой же главе мы обсудим, как его установить, а работать он может практически на любой машине, хотя для разработки несколько лишних гигабайтов памяти не помешает. Для нескольких примеров построения карт нужны солидные вычислительные ресурсы, но большинство компьютеров, выпущенных после 2014 года, с такой нагрузкой справится.

Понадобится также последняя версия браузера; я предпочитаю Chrome и разрабатывал примеры в нем, однако Firefox тоже годится. Можете также попробовать Safari, Internet Explorer/Edge, Opera или любой другой браузер, но, на мой вкус, инструменты разработчика в Chrome самые лучшие.

На кого рассчитана эта книга

На разработчиков веб-приложений, авторов интерактивных новостей, специалистов по анализу и обработке данных и всех, интересующихся интерактивным представлением данных в вебе с помощью библиотеки D3. Предполагается знакомство с основами JavaScript, но предварительный опыт визуализации данных или работы с D3 не потребуется.

Графические выделения

В этой книге тип информации обозначается шрифтом. Ниже приведено несколько примеров с пояснениями.

Фрагменты кода внутри абзаца, имена таблиц базы данных, папок и файлов, URL-адреса, данные, которые вводит пользователь, и адреса в Твиттере выделяются следующим образом: «Если в сообщении говорится что-то типа 'Command not found', проверьте, все ли правильно установлено, и убедитесь, что Node.js включен в переменную среды $PATH».

Кусок кода выглядит так:

```
"babel": {
  "presets": [
    "es2017"
  ]
},
"main": "lib/main.js",
"scripts": {
  "start": "webpack-dev-server --inline",
},
```

Входная и выходная информация командных утилит выглядит так:

```
$ brew install n
$ n lts
```

Новые термины и **важные фрагменты** выделяются полужирным шрифтом. Например, элементы графического интерфейса в меню или диалоговых окнах выглядят в книге так: «Мы в основном будем пользоваться вкладками **Elements** и **Console**: вкладка **Elements** предназначена для просмотра DOM, а **Console** – для ввода JavaScript-кода и анализа ошибок».

Предупреждения и важные примечания выглядят так.

Советы и рекомендации выглядят так.

Отзывы

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или может быть не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг — возможно, ошибку в тексте или в коде — мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательство ДМК Пресс и Packt очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, и помогающую нам предоставлять вам качественные материалы.

Глава 1

Первое знакомство с D3, ES2017 и Node.js

Библиотека **Data-Driven Documents (D3)**, разработанная Майком Бостоком (Mike Bostock) и сообществом D3 в 2011 году, пришла на смену прежней библиотеке Бостока Protovis. Она поддерживает отрисовку данных с точностью до пикселя благодаря абстрагированию вычисления масштабов и осей с помощью простого предметно-ориентированного языка (DSL) и благодаря использованию идиом, понятных всякому, кто работал с популярной JavaScript-библиотекой jQuery. В D3, как и в jQuery, нужно сначала выбрать множество элементов, а затем применить к ним операции с помощью цепочки функций-модификаторов. В контексте визуализации данных такой декларативный подход оказывается гораздо проще многих других имеющихся инструментов. На официальном сайте по адресу https://d3js.org/ приведено много примеров, демонстрирующих мощь D3, но поначалу разобраться в них нелегко. Прочитав эту книгу, вы будете знать D3 в достаточной степени, чтобы понять примеры и приспособить их к своим нуждам. Если вы хотите познакомиться с разработкой D3 поближе, скачайте исходный код с сайта GitHub по адресу https://github.com/d3.

В этой главе мы заложим фундамент для выполнения представленных в книге примеров. Я объясню, как писать код на языке ECMAScript 2017 (ES2017) – последней и самой передовой версии JavaScript – и как с помощью программы Babel транслировать его на язык ES5, который понимает любой современный браузер. Затем мы познакомимся с основами D3 v4 на примере создания простой диаграммы.

Что такое D3.js?

Благодаря точному контролю и элегантности D3 заслуженно считается одной из самых мощных библиотек визуализации с открытым исходным кодом. Но это также означает, что для простых задач – изображения одного-двух линейных графиков – она не слишком подходит; в этом случае лучше взять какую-нибудь библиотеку для рисования графиков. Кстати, многие из них сами основаны на D3. Обширный список опубликован по адресу https://github.com/sorrycc/awesome-javascript#data-visualization.

D3 построена на принципах функционального программирования, которое ныне переживает второе рождение в сообществе JavaScript. Эта книга не о функциональном программировании, но многое покажется знакомым тем, кто уже сталкивался с этими принципами. Если вы не из их числа или привыкли к объектно-ориентированному программированию, не расстраивайтесь, я буду объяснять все необходимое по ходу дела и надеюсь, что раздел о функциональном программировании в начале главы 4 поможет вам понять, почему эта парадигма так полезна, особенно в задачах визуализации данных и конструирования приложений.

Что случилось с классами?

Во втором издании этой книги было много примеров использования механизма классов, появившихся в языке ES2015. Но теперь мы используем *фабричные функции*, а ключевое слово `class` ни разу не встречается. С чего бы это?

В ES2015 классы на самом деле были *синтаксическим сахаром* поверх фабричных функций, т. е. в конечном счете компилировались в фабричные функции. Хотя с помощью классов можно повысить уровень организации сложного кода, в итоге они просто скрывают происходящее внутри. К тому же использование объектно-ориентированных парадигм, в т. ч. классов, идет вразрез с одним из самых действенных и элегантных аспектов языка JavaScript – полноправными функциями и объектами. Ваш код станет проще и изящнее, если вы будете придерживаться функциональных парадигм, да и примеры, созданные сообществом D3, читать будет проще, поскольку в них классы почти никогда не используются.

Против использования классов можно привести еще много аргументов, для изложения которых здесь просто нет места. Рекомендую обратиться к великолепной серии статей Эрика Эллиота «The Two

Pillars of JavaScript» по адресу www.medium.com/javascript-scene/the-two-pillars-of-javascript-ee6f3281e7f3.

Что нового в версии D3 v4?

Одно из главных событий, случившихся после выхода предыдущего издания этой книги, – выпуск версии 4.

Из многочисленных изменений самым важным является полный пересмотр пространства имен D3. Это означает, что ни один пример, приведенный в этой книге, не будет работать в версии D3 3.x, а примеры из второго издания книги «Learning D3.js Data Visualization» не будут работать в D3 4.x. Это, пожалуй, самое страшное, что жестокий м-р Босток мог учинить над такими авторами учебников, как я (шучу, шучу). Но шутки в сторону – из-за этого многие примеры кода, разработанные сообществом D3, потеряли актуальность и могут даже показаться странными тому, для кого эта книга стала первым знакомством с библиотекой. Поэтому так важно проверять, для какой версии D3 написан пример, – если для 3.x, то стоит поискать аналогичный пример для 4.x, чтобы не пасть жертвой когнитивного диссонанса.

Обычно в примерах версия D3 указывается в тегах `script` в начале кода. Если вы видите такой код:

```
<script src="https://d3js.org/d3.v3.min.js"></script>
```

то пример написан для версии 3.x. А если такой:

```
<script src="https://d3js.org/d3.v4.min.js"></script>
```

то это более современный пример, рассчитанный на версию 4, т. е. вы движетесь в правильном направлении.

С этим связано также разбиение D3 на много библиотек меньшего размера (микробиблиотек). Вы можете пойти по одному из двух путей:

- работать с D3 как с единой библиотекой (монобиблиотекой) так же, как в версии 3;
- включать в проект лишь отдельные компоненты D3 (микробиблиотеки).

В этой книге принят первый подход. Использование микробиблиотек заметно усложнило бы изучение D3, хотя при этом уменьшается размер конечного комплекта файлов, который должны будут загрузить пользователи для просмотра вашей графики. Тем не менее я буду отмечать, в каком пакете находится та или иная функциональность,

а вы, когда получше освоитесь с D3, сможете перейти на микробиблиотеки, вместо того чтобы включать все подряд, нужное и ненужное.

Что такое ES2017?

Одно из главных изменений в этой книге с момента ее первого издания – упор на современный JavaScript, в данном случае ES2017. Эта версия, ранее называвшаяся ES6 (Harmony), – крупный шаг в развитии языковых средств JavaScript, позволивший реализовать новые паттерны, которые упрощают восприятие кода и повышают его выразительность. Если вы писали на JavaScript раньше и примеры в этой главе кажутся вам странными, значит, вы имели дело с прежним, более распространенным синтаксисом ES5.

Но не переживайте! На привыкание к новому синтаксису не уйдет много времени, а я уж постараюсь объяснять новые языковые возможности по мере надобности. Хотя поначалу кривая обучения может показаться довольно крутой, в конце вы будете писать код куда лучше и окажетесь на переднем крае современной разработки на JavaScript.

Отличное введение во все новшества, которые принес нам ES2015-17, можно найти в руководстве, составленном авторами программы Babel.js (https://babeljs.io/docs/learn-es2015/), которой мы будем постоянно пользоваться на страницах этой книги.

Прежде чем двигаться дальше, я хотел бы развеять некоторые мифы о том, чем на самом деле является ES2017. Первоначально версии стандартов ECMAScript (сокращенно ES) нумеровались последовательно: ES4, ES5, ES6, ES7. Но начиная с ES6 этот порядок был изменен, поскольку важно было отразить тот факт, что новые стандарты выходят ежегодно, чтобы не отставать от современных тенденций разработки. Так что текущий стандарт относится к 2017 году. Важной вехой стала версия ES2015, которая примерно соответствует ES6. Стандарт ES2016, ратифицированный в июне 2016 года, основывается на предыдущем стандарте с добавлением нескольких исправлений и двух новых возможностей. ES2017 пока находится на стадии проекта, т. е. новые предложения рассматриваются и разрабатываются – до ратификации, которая произойдет в течение 2017 года. Эта книга писалась, когда новые средства еще не были утверждены, и, возможно, они даже не войдут в стандарт ES2017, так что с их официальным включением, быть может, придется подождать до выхода следующей версии стандарта.

Но это не причина для беспокойства, поскольку мы все равно используем Babel.js для компиляции всего кода на ES5, чтобы он одинаково работал в Node.js и в браузере. Для полноты картины я буду указывать, в какой версии стандарта появилось то или иное средство (например, модули были включены в ES2015), но, говоря о JavaScript, я всегда имею в виду современный JavaScript безотносительно к номеру спецификации ECMAScript.

Запускаем Node и Git из командной строки

В этой книге я постараюсь не выказывать предпочтений какому-либо конкретному редактору или операционной системе (хотя сам использую Atom в macOS X), но все-таки какие-то предварительные условия должны быть соблюдены.

Первое из них – Node.js. В настоящее время Node широко применяется для веб-разработки и, по сути дела, представляет собой средство для выполнения JavaScript-кода из командной строки. Ниже я покажу, как написать серверное приложение для Node, а пока займемся просто установкой Node и `npm` (замечательного менеджера пакетов для Node).

Если вы работаете в Windows или macOS X без Homebrew, то воспользуйтесь установщиком по адресу https://nodejs.org/en/. Если в вашей системе на базе macOS X имеется Homebrew, то я рекомендую вместо этого установить пакет `n`, который позволяет без труда переключаться с одной версии Node на другую:

```
$ brew install n
$ n lts
```

Пользователям Windows знак `$` может показаться непонятным. В операционных системах на базе UNIX обычный пользователь видит в командной строке приглашение `$`, а суперпользователь *root* – приглашение `#`. Включая знак `$`, я показываю, что вы должны выполнять команды от имени обычного пользователя, а не суперпользователя.

В любом случае по завершении проверьте результат, выполнив такие команды:

```
$ node --version
$ npm --version
```

Если будут напечатаны версии `node` и `npm`, значит, все хорошо.

Я работаю соответственно с версиями 6.5.0 и 3.10.3. Ваша версия может отличаться, но важно, чтобы версия Node была не ниже 6.0.0.

Если в сообщении говорится что-то типа `Command not found`, проверьте, все ли правильно установлено, и убедитесь, что Node.js включен в переменную среды `$PATH`.

В этой книге мы с помощью Babel и Webpack будем преобразовывать наш изысканный модульный современный код на JavaScript в нечто, что могут выполнить даже самые захудалые, побитые молью браузеры (ау, Internet Explorer 9!).

Для начала создадим файл `package.json`, в котором будут храниться версии всех нужных нам зависимостей. Для этого создадим новую папку и выполним команду `npm init`:

```
$ mkdir d3-projects
$ cd d3-projects
$ npm init -y
```

Флаг `-y` означает, что `npm init` должна использовать параметры по умолчанию, не задавая никаких вопросов.

Затем с помощью `npm` установим все необходимое:

```
$ npm install "babel-core@^6" "babel-loader@^6" "babel-preset-es2017@^6"
"babel-preset-stage-0@^6" "webpack@^2" "webpack-dev-server@^2" css-loader
style-loader json-loader --save-dev
```

Эта команда установит версию 2 Webpack, версию 6 Babel и комплект начальных установок и дополнительных модулей для того и другого. Имена и номера версий программ будут записаны в файл `package.json`, так что для повторной установки понадобится всего лишь ввести команду

```
$ npm install
```

Еще установим D3:

```
$ npm install d3 --save
```

Далее нужно создать конфигурационный файл для Webpack. Не буду объяснять, что означает каждая директива, все комментарии вы сможете найти в репозитории кода к этой книге. Просто сохраните показанный ниже код в файле `webpack.config.js`:

```
const path = require('path');
module.exports = [{
```

```
  entry: {
    app: ['./lib/main.js'],
  },
  output: {
    path: path.resolve(__dirname, 'build'),
    publicPath: '/assets/',
    filename: 'bundle.js',
  },
  devtool: 'inline-source-map',
  module: {
    rules: [{
      test: /.js?$/,
      exclude: /(node_modules|bower_components)/,
      loader: 'babel-loader',
    }, {
      test: /.json$/,
      loader: 'json-loader',
    }, {
      test: /.css$/,
      loader: 'style-loader!css-loader',
    }],
  },
}];
```

Напоследок отредактируем файл `package.json`, добавив несколько аббревиатур, которые упростят нам жизнь. После строки, начинающейся словом `name`, вставьте такие строки:

```
"babel": {
  "presets": [
    "es2017"
  ]
},
"main": "lib/main.js",
"scripts": {
  "start": "webpack-dev-server --inline",
},
```

Все это необходимо, если вы начинаете новый проект с нуля.

Можно вместо этого клонировать репозиторий книги из GitHub. GitHub – это место, где размещается большая часть всего открытого и иного кода в мире. Я не только положил туда примеры и тестовые данные, но и поработал над конфигурацией. Далее я буду исходить из предположения, что вы клонировали репозиторий, создав копию на своей машине. Для этого выполните такие команды:

```
$ git clone https://github.com/aendrew/learning-d3-v4
$ cd learning-d3-v4
```

В результате будут клонированы среда разработки и все примеры в каталоге `learning-d3-v4/`, после чего мы сделаем этот каталог текущим и установим все зависимости с помощью `npm`.

Есть еще один вариант: разветвить репозиторий на GitHub и клонировать свое ответвление вместо моего. Это позволит публиковать результаты своего труда в облаке, упростит получение помощи от коллег, даст возможность отображать завершенные проекты на страницах GitHub и даже отправлять исправления и дополнения в родительский проект. Что, в свою очередь, будет способствовать улучшению будущих изданий этой книги. Чтобы разветвить проект `aendrew/learning-d3-v4`, нажмите кнопку «fork» на сайте GitHub и замените строку `aendrew` в приведенном выше фрагменте кода своим именем пользователя GitHub.

Каждая глава книги хранится в отдельной ветке. Для переключения между главами выполните команду вида

```
$ git checkout chapter1
```

Вместо `1` укажите номер интересующей вас главы. Но пока останемся в главной ветке. Чтобы вернуться к ней, введите такую команду:

```
$ git stash save && git checkout master
```

В главной ветке вы будете писать свой код по мере работы над книгой. Установить зависимости все равно нужно, сделаем это прямо сейчас:

```
$ npm install
```

Весь исходный код, с которым вы будете работать, находится в папке `lib/`. Обратите внимание, что там имеется только файл `main.js`; почти всегда мы будем иметь дело именно с ним, поскольку `index.html` – всего лишь минимальный контейнер, в котором отображается результат нашей работы. Ниже он приведен целиком, и это первый и последний раз, когда в книге встретится HTML-код:

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset="utf-8">
  <title>Learning Data Visualization with D3.js</title>
  </head>
  <body>
  <script src="assets/bundle.js"></script>
  </body>
</html>
```

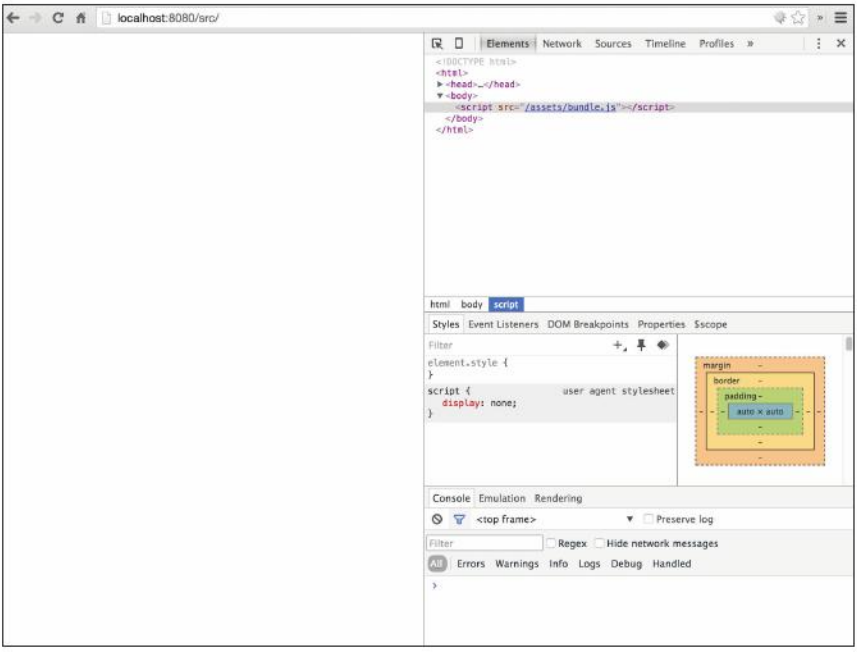
Имеется также таблица стилей в файле `styles/index.css`; пока она пуста, но скоро мы начнем ее заполнять.

Следующий шаг – запустить сервер разработки:

```
$ npm start
```

Эта команда запускает сервер Webpack, который преобразует наш новомодный код на JavaScript стандарта ES2017 в совместимый с ES5 и отправляет его браузеру.

Теперь введите в адресной строке Chrome (или другого браузера, я не привередливый – лишь бы только не Internet Explorer!) строку `localhost:8080` и откройте консоль разработчика (*Ctrl+Shift+J* в Linux и Windows, *Option+Command+J* – в Mac). Появятся пустая страница и пустая консоль JavaScript с приглашением для ввода кода:



Краткое введение в инструменты разработчика Chrome

Инструменты разработчика Chrome – неоценимое средство для разработки веб-приложений. В большинстве современных браузеров

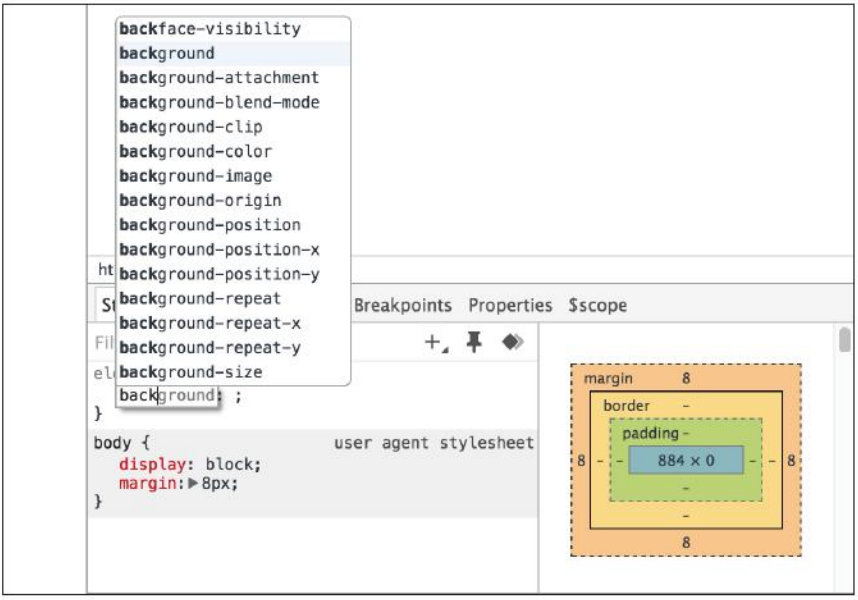
есть нечто подобное, но мы для простоты ограничимся только тем, что предлагает Chrome. Вы вполне можете выбрать другой браузер. Особенно хорош Firefox Developer Edition и говорят – да-да, ребята у меня за спиной, я слышу, – что Opera тоже неплох!

Мы в основном будем пользоваться вкладками **Elements** и **Console**: вкладка **Elements** предназначена для просмотра DOM, а **Console** – для ввода JavaScript-кода и анализа ошибок. Остальные шесть вкладок нужны для более крупных проектов.

С помощью вкладки **Network** можно узнать, сколько времени ушло на загрузку файлов, и изучить Ajax-запросы. Вкладка **Profiles** позволяет профилировать JavaScript-код. Вкладка **Resources** полезна для исследования данных на стороне клиента, а вкладки **Timeline** и **Audits** – когда в программе имеется глобальная переменная, приводящая к утечке памяти, и вы пытаетесь понять, почему вдруг Chrome «отъел» 500 МБ оперативной памяти. Мне доводилось пользоваться всеми этими вкладками в ходе разработки с использованием D3, но, пожалуй, наибольшую пользу они все-таки приносят при создании крупных веб-приложений с применением таких каркасов, как React и Angular.

Но главное, что нас будет интересовать, – это вкладка **Sources**, на которой показаны все исходные файлы, загруженные веб-страницей. Мало того, что мы видим, какой код на самом деле загружается, так еще эта вкладка содержит полнофункциональный отладчик JavaScript, которым решаются воспользоваться немногие простые смертные. Описывать технику отладки программы скучно, и вообще, это не тема данной главы, но все же расстановка точек останова вместо включения бесконечных предложений `console.log`, для того чтобы понять, что делает программа, – умение, которое принесет вам немало пользы в будущем. Хорошее руководство имеется на странице https://developers.google.com/web/tools/chrome-devtools/javascript/breakpoints.

Однако в основном вы будете использовать Инструменты разработчика для исследования CSS-стилей в правой части вкладки **Elements**. Здесь можно узнать, какие правила CSS влияют на стилизацию элемента, это позволяет отыскать зловредные правила, которые все портят. Можно также отредактировать CSS и сразу же увидеть результат своих действий.



Неизбежный пример – столбчатая диаграмма

Никакое введение в D3 нельзя считать полным без примера простой столбчатой диаграммы. В контексте D3 они играют ту же роль, что программа «Здравствуй, мир» в языках программирования, при этом 90% любого рассказа о данных можно свести к демонстрации подходящей диаграммы или графика. Хорошие примеры можно найти в статьях из *Financial Times* или *Economist* – там часто изложение завершается итоговым линейным графиком или гистограммой. Поскольку по роду деятельности я занимаюсь разработками для отдела новостей (раскрою секрет: я работаю в отделе интерактивной графики редакции *Financial Times*), многие мои примеры так или ина-

че связаны с текущими событиями или темами, для подачи которых полезна визуализация данных. Сообщество поддержки новостных редакций приложило немало усилий для создания среды, в которой процветает D3, и для любого целеустремленного журналиста уверенное владение инструментами типа D3 становится все более важным профессиональным навыком.

Нашим первым набором данных будут результаты подсчета голосов на референдуме по брекситу, опубликованные избирательной комиссией Великобритании. Мы построим столбчатую диаграмму, отражающую явку по регионам. Источник данных – http://www.electoralcommission.org.uk/find-information-by-subject/elections-and-referendums/past-elections-and-referendums/eu-referendum/electorate-and-count-information.

Мы создадим по одному столбику для каждого региона Великобритании. Первый шаг – настроить простой контейнер, который затем будет заполнен с помощью нашего прелестного JavaScript-кода. Можно либо сразу перейти к коду, либо кое-что подготовить, чтобы потом было проще. Для начала двинемся по первому пути. Скоро на вас обрушится целая лавина новомодного JavaScript-кода, поэтому пока постараемся не усложнять.

Откройте файл `lib/main.js` и добавьте в него свою первую строку, имеющую отношение к библиотеке D3:

```
const chart = d3.select('body')
  .append('svg')
  .attr('id', 'chart');
```

Здесь мы выбираем HTML-элемент `<body>`, добавляем в его конец элемент `<svg>` и назначаем ему идентификатор `#chart`. Эта схема будет встречаться нам очень часто.

Прежде чем двигаться дальше, обратим внимание на первую конструкцию современного JavaScript:
Ключевое слово `const` используется для определения переменной, которая не будет существенно изменяться. Говоря *существенно*, я имею в виду, что как-то изменяться она все-таки может (например, можно добавить элементы в массив или модифицировать объект), но попытка присвоить ей другое значение приведет к исключению. Исключение возбуждается и при попытке использовать переменную до ее объявления. В отличие от других языков, в JavaScript константы ограничены областью видимости текущей функции (они не являются глобальными, если специально не сделать их таковыми). Это весьма полезно в функциональных программах, поскольку

позволяет предотвратить странные ошибки, вызванные поднятием переменных (необычная особенность JavaScript, заключающаяся в том, что переменные интерпретируются в начале замыкания функции, а не там, где они фактически определены). Дополнительные сведения о `const` имеются по адресу http://mdn.io/const.
Еще один способ определения переменных в JavaScript дает ключевое слово `let`, которое действует как `var`, но подобно `const` имеет блочную область видимости, т. е. видимость переменной ограничена блоком, предложением или выражением, в котором она используется. Это тоже помогает предотвратить трудные для отладки ошибки. Подробнее см. http://mdn.io/let.

Когда употреблять одно, а когда другое? Ключевое слово `const` лучше использовать, если вы не собираетесь присваивать переменной другое значение, а `let` – если собираетесь. Я стараюсь не переприсваивать значения переменным, поэтому обычно использую `const`. Хотя в современном JavaScript употребление ключевого слова `var` не запрещено, делать этого не стоит – используйте `let` или `const`, отдавая предпочтение `let`, если не уверены, что правильно в данной ситуации.

Пора! Откроем браузер, не забыв предварительно запустить сервер разработки (для его запуска выполните `npm start`), и перейдем по адресу `http://localhost:8080`.

Uncaught Error: Cannot find module "d3"

М-да. Нехорошо вышло...

Ошибка связана с тем, что мы еще не импортировали D3. Если вам уже доводилось работать с D3, то, наверное, вы привыкли, что она присоединяется к глобальному объекту `window`. Именно это происходит, когда файл d3.js включается с помощью тега `<script>`. Но мы этого делать не будем, а воспользуемся новой функциональностью ES2015 – импортом модулей!

Вернемся к файлу `main.js`. В самом начале добавьте строку

```
import * as d3 from 'd3';
```

Разберемся, что это значит. Все предложения импорта должны находиться в начале файла (никаких потайных вызовов `require()` внутри функций в стиле Node.js!). Это открывает возможность для статического анализа, так что новые инструменты JavaScript оказываются более эффективными. Каждое такое предложение начинается ключевым словом `import`.

Дальше может следовать фигурная скобка. Модуль ES2015 допускает экспорт двух видов:

- **именованный**: в этом случае в фигурных скобках указываются имена конкретных экспортируемых членов (хотя их можно переименовывать);
- **по умолчанию**: такой член в модуле может быть только один, при импорте его можно назвать как угодно. Пример будет показан ниже.

В предложении выше мы импортировали все входящие в D3 микробиблиотеки в пространство имен `d3`.

Вернитесь в браузер и перейдите на вкладку **Elements**: вы увидите новый элемент SVG с идентификатором `#chart` в конце страницы. Это уже прогресс!

Загрузка данных

Вернемся к `main.js`. Нам нужно как-то получить данные, и ниже я покажу, как это сделать правильно, а пока последуем по старому, трудному и неприятному пути – воспользуемся объектом `XMLHttpRequest`:

```
const req = new window.XMLHttpRequest();
req.addEventListener('load', mungeData);
req.open('GET', 'data/EU-referendum-result-data.csv');
req.send();
```

Здесь мы создаем новый объект `XMLHttpRequest`, просим его загрузить файл из каталога `data` и передаем полученные данные функции `mungeData()`, которую скоро напишем.

Вы обратили внимание на использование уродливого ключевого слова `new`? И на то, что для объявления переменной понадобились четыре строки? И на то, что ответ обрабатывается функцией обратного вызова? Бр-р-р! Но ничего, в последующих главах мы все исправим. Единственное преимущество такого подхода – в том, что он работает практически в любом браузере без всяких полифилов[1]. Но есть множество способов сделать это лучше, и мы рассмотрим их в главе 4.

В загруженном CSV-файле каждому избирательному округу Великобритании соответствует одна строка, содержащая разнообразные данные: процент проголосовавших за каждый вариант, явка избирателей, количество недействительных или испорченных бюллетеней и т. д. Мы преобразуем этот файл в массив объектов, представляющих средний процент проголосовавших за выход в более крупных территориальных образованиях.

---

[1] См. https://ru.wikipedia.org/wiki/Полифил. – *Прим. перев.*

Пора уже написать функцию `mungeData()`. Для преобразования строки CSV-файла в объект мы воспользуемся функцией `d3.csvParse()` (из микробиблиотеки `d3-dsv`), а затем применим функции из микробиблиотеки `d3-array` для обработки этих объектов:

```
function mungeData() {
  const data = d3.csvParse(this.responseText);
  const regions = data.reduce((last, row) => {
    if (!last[row.Region]) last[row.Region] = [];
    last[row.Region].push(row);
    return last;
  }, {});
  const regionsPctTurnout = Object.entries(regions)
    .map(([region, areas]) => ({
      region,
      meanPctTurnout: d3.mean(areas, d => d.Pct_Turnout),
    }));

  renderChart(regionsPctTurnout);
}
```

Да, кстати, вот и еще одно новшество ES2015! Вместо того чтобы раз за разом набирать `function() {}`, мы теперь можем определить анонимную функцию, написав просто `() => {}` . Во-первых, мы сэкономили пять ударов по клавишам, а во-вторых, *двойная стрелка* не привязывает значение `this` к какому-то другому объекту. Поскольку мы применяем функциональный стиль программирования, это не столь важно, но если бы мы использовали классы, то это существенно упростило бы жизнь. Подробнее см. http://mdn.io/Arrow_functions.

Преобразование данных состоит из трех шагов.

1. Сначала преобразуем данные в массив объектов с помощью функции `d3.csvParse()` и присвоим результат переменной `data`.
2. Затем преобразуем массив в объект, в котором ключами являются названия регионов, а значениями – массивы входящих в регион избирательных округов.
3. И наконец, `Object.entries` преобразует объект в многомерный массив, состоящий из элементов, содержащих пары ключ-значение, который затем можно редуцировать в новый объект, содержащий названия регионов и среднюю процентную явку, вычисленную по всем входящим в регион избирательным округам.

Вы, наверное, обратили внимание на необычную сигнатуру функции `Array.prototype.map`:

```
.map(([region, areas]) => {
```

Здесь используется новая возможность ES2015 – *деструктурирующее присваивание*, – чтобы сопоставить элементу массива временное имя. Обычно сигнатура обратного вызова имеет вид:

```
function(item, index, array) {}
```

Но поскольку мы знаем, что `item` – массив с двумя элементами, то можем поименовать каждый из них, упростив тем самым чтение кода (в этот раз мы не использовали аргументов `index` и `array`, но если бы они понадобились, то мы поместили бы их после деструктурирующей конструкции).

И напоследок преобразованные данные передаются еще не написанной функции `renderChart()`, которую мы добавим ниже.

Можно также просто добавить к написанному выше коду такие строки:

```
const regionsPctTurnout = d3.nest()
  .key(d => d.Region)
  .rollup(d => d3.mean(d, leaf => leaf.Pct_Turnout))
  .entries(data);
```

Функция `d3.nest()` входит в состав микробиблиотеки `d3-collection`, которая будет рассмотрена в главе 4. D3 – беспристрастная библиотека в том смысле, что одну и ту же задачу можно решить разными способами, и зачастую среди них нет единственно правильного. Я буду демонстрировать различные подходы, а вы уж сами выбирайте, какой вам больше нравится.

Двенадцать (плюс-минус) столбиков

Настало время изобразить данные на экране.

Добавим в файл `main.js` новую функцию `renderChart()`:

```
function renderChart(data) {
  chart.attr('width', window.innerWidth)
    .attr('height', window.innerHeight);
}
```

Она всего лишь устанавливает размеры ранее созданной переменной `chart` равными размерам окна. Мы почти у цели, держитесь крепче!

Но сначала нужно определить масштабы, которые описывают, как D3 отображает данные на значения пикселей. Иначе говоря, масштаб – это просто функция, которая отображает входную область определения в выходную область значений. На случай, если это труд-

но запомнить, я беззастенчиво украл упражнение из великолепного пособия Скотта Мюррея по масштабам, приведенного в книге «Interactive Data Visualization for the Web»:

Когда я говорю «входная», ты говоришь «область определения».
Когда я говорю «выходная», ты говоришь «область значений».
Понятно? Поехали.
Входная! Область определения!
Выходная! Область значений!
Входная! Область определения!
Выходная! Область значений!
Запомнил? Ну и отлично.

Глупо, конечно, но я часто ловлю себя на том, что бормочу эти строки, когда сроки поджимают, а я поздно ночью работаю над какой-нибудь диаграммой. Попробуйте сами!

Далее добавим в `renderChart()` такой код:

```
const x = d3.scaleBand()
  .domain(data.map(d => d.region))
  .rangeRound([50, window.innerWidth - 50])
  .padding(0.1);
```

Теперь масштаб `x` – функция, которая отображает область определения, состоящую из названий регионов, в область значений от `50` до ширины окна просмотра минус `50` с промежутками, которые определяются функцией `.padding()` с аргументом 0.1. В результате мы создали зонный масштаб, который похож на обычный с тем отличием, что выходная область разделена на секции. Мы еще поговорим о масштабах ниже.

В этом примере задана фиксированная ширина полей 50, которая передается масштабам и другим функциям. Произвольное число, зашитое в код, часто называют *магическим числом*, понимая под этим, что всякий читающий код видит какое-то непонятное значение, при котором код работает, как по волшебству. Это плохо, никогда так не поступайте – такой код трудно читать, а кроме того, если вы захотите изменить значение, то придется найти все места, где оно встречается. Я лишь хотел привлечь ваше внимание к этому факту. Далее мы будем определять подобные величины более разумным способом, следите за новостями!

Все еще внутри `renderChart()` определим еще один масштаб `y`:

```
const y = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.meanPctTurnout)])
  .range([window.innerHeight - 50, 0]);
```

Масштаб `y` отображает линейную область определения (от нуля до максимального значения данных, которое возвращает функция `d3.max`) в область значений от `window.innerHeight` (минус поле высотой `50` пикселей) до `0`. Инвертировать диапазон обязательно, потому что D3 считает, что верхней точке графика соответствует значение `y=0`. Обнаружив, что диаграмма D3 нарисована вверх ногами, попробуйте инвертировать область значений в одном из масштабов.

Теперь определим оси. Добавьте в конец `renderChart` следующий код:

```
const xAxis = d3.axisBottom().scale(x);
const yAxis = d3.axisLeft().scale(y);
```

Мы сказали, какой масштаб использовать на каждой оси при нанесении делений и по какую сторону от оси располагать метки. D3 сама решит, сколько нарисовать делений, где они должны находиться и как их пометить. Поскольку большинство элементов D3 одновременно является объектами и функциями, мы можем изменить внутреннее состояние обоих масштабов, не присваивая результат никакой переменной. Область значений `x` – дискретный список, а область значений `y` – диапазон от `0` до максимального значения в наборе данных.

Теперь нарисуем оси:

```
chart.append('g')
  .attr('class', 'axis')
  .attr('transform',
    &grave;translate(0, ${window.innerHeight - 50})&grave;)
  .call(xAxis);
```

Еще одно новшество ES2015! Аргумент атрибута `transform` заключен в обратные апострофы (`), это шаблонная литеральная строка. Она отличается от обычной строки в двух отношениях: внутри допускаются символы новой строки, а благодаря синтаксической конструкции `${}` мы можем вычислить и подставить в строку произвольное JavaScript-выражение. В данном случае просто выводится значение `window.innerHeight`, но, в принципе, это может быть любое выражение, возвращающее нечто, похожее на строку, например `Array.prototype.join`, – тогда будет подставлено содержимое всего массива. Удобно, ничего не скажешь.

Мы поместили в диаграмму элемент `g`, сопоставили элементу `axis` CSS-класс и с помощью атрибута `transform` перенесли его в левый нижний угол диаграммы.

Наконец, мы вызвали функцию `xAxis`, предоставив D3 возможность сделать все остальное.

Вторая ось рисуется так же, только аргументы другие:

```
chart.append('g')
  .attr('class', 'axis')
  .attr('transform', 'translate(50, 0)')
  .call(yAxis);
```

Итак, метки расставлены, осталось только нарисовать данные:

```
chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => x(d.region))
  .attr('y', d => y(d.meanPctTurnout))
  .attr('width', x.bandwidth())
  .attr('height', d => (window.innerHeight - 50) - y(d.meanPctTurnout));
```

Кода много, но смысл его очень прост:

- для каждого прямоугольника (`rect`) в графике загрузить данные;
- обойти данные;
- для каждого элемента добавить прямоугольник;
- задать атрибуты прямоугольника.

Не обращайте внимания, что в начальный момент нет ни одного прямоугольника; здесь мы по существу создаем выделение, привязанное к данным, и производим над ним операции. Могу понять, какое странное чувство возникает, когда оперируешь несуществующими элементами (когда я изучал D3, это стало для меня главным камнем преткновения), но это идиома, полезность которой станет ясна позднее, когда мы начнем добавлять и удалять элементы в соответствии с изменяющимися данными.

Масштаб `x` помогает вычислять горизонтальные позиции, а функция `bandwidth()` дает ширину столбика. Масштаб `y` вычисляет вертикальные позиции, и мы вручную получаем высоту каждого столбика от `y` до нижней оси. Отметим, что когда значения атрибута для каждого элемента разные, атрибут задается в виде функции (`x`, `y` и `height`), в противном случае – в виде значения (`width`).

Теперь добавим «бантик» – заставим каждый столбик вырастать из горизонтальной оси. Добро пожаловать в мир анимации!

Модифицируйте только что добавленный код, как показано ниже; изменившиеся строки выделены полужирным шрифтом:

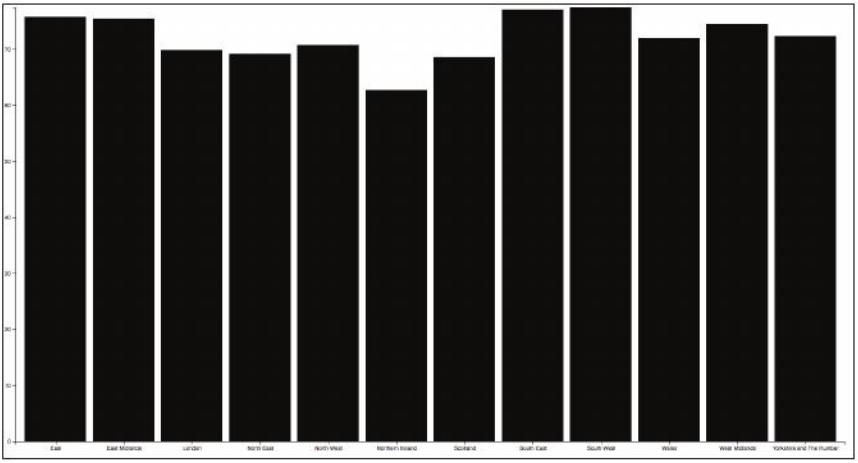
```
chart.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
  .attr('class', 'bar')
  .attr('x', d => x(d.region))
.attr('y', window.innerHeight - 50)
  .attr('width', x.bandwidth())
  .attr('height', 0)
    .transition()
    .delay((d, i) => i * 20)
    .duration(800)
    .attr('y', d => y(d.meanPctTurnout))
    .attr('height', d =>
      (window.innerHeight - 50) - y(d.meanPctTurnout));
```

Разница в том, что мы статически сместили все столбики вниз (`window.innerHeight - 50`), присвоив им нулевую высоту, а затем инициировали анимацию функцией `.transition()`. После этого мы определяем нужный нам переход.

Прежде всего мы хотим отложить переход каждого столбика на 20 миллисекунд относительно предыдущего, задав аргумент `i*20`. Большинство обратных вызовов D3 возвращает некоторое значение (привязанное к элементу, обычно оно обозначается `d`) и индекс (порядковый номер обрабатываемого элемента, обычно обозначаемый `i`), а аргумент `this` при этом ссылается на текущий выбранный элемент DOM. Если бы мы использовали классы, то этот последний факт был бы важен, т. к. в противном случае мы работали бы в контексте объекта `rect` типа `SVGElement`, а не в том, который нам действительно нужен. Но поскольку мы в большинстве случаев пользуемся фабричными функциями, знать, какой контекст связан с `this`, не так важно.

В результате гистограмма постепенно появляется слева направо, а не возникает сразу, как черт из табакерки. Далее мы говорим, что каждая анимация должна длиться чуть меньше секунды – `.duration(800)`. А в конце определяем конечные значения анимированных атрибутов – `y` и `height` – такие же, как в предыдущем коде. Обо всем остальном позаботится D3.

Сохраните файл и обновите окно браузера. Если все пойдет по плану, то должна появиться такая диаграмма:



Как видим, явка на референдуме по вопросу выхода из ЕС была довольно высокой, а самой высокой она оказалась на юго-западе. Прикиньте – мы только что решили задачу из области журналистики данных! На случай, если у вас не получилась такая диаграмма, напомним, что полностью код доступен по адресу http://github.com/aendrew/learning-d3-v4/tree/chapter1.

Но кое-чего нам пока не хватает, главным образом CSS-стилей элементов SVG.

Можно было бы просто добавить CSS в HTML-код, но тогда пришлось бы открывать этот мерзкий файл `index.html`. К тому же кому захочется писать HTML-код, когда изучаешь новомодный JavaScript?

Прежде всего создадим файл `index.css` в каталоге `styles/`:

```
html, body {
  padding: 0;
  margin: 0;
}

.axis path, .axis line {
  fill: none;
  stroke: #eee;
  shape-rendering: crispEdges;
}

.axis text {
  font-size: 11px;
}
```

```
.bar {
  fill: steelblue;
}
```

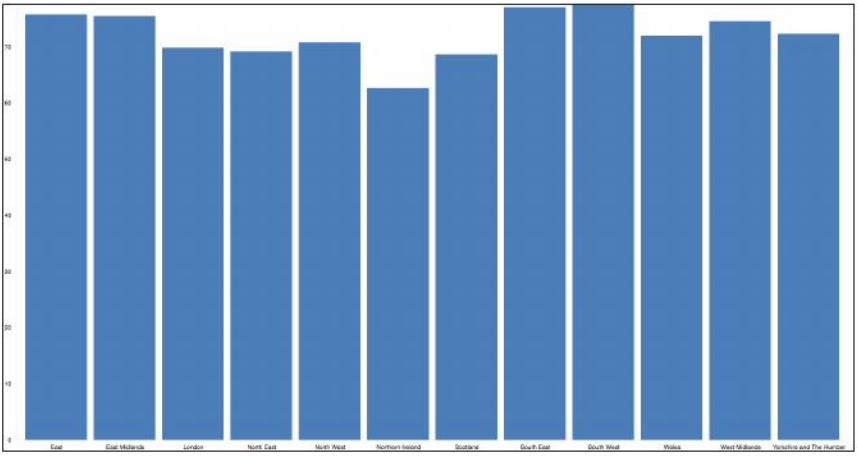
Затем добавим такую строку в начало `main.js`:

```
import * as styles from 'styles/index.css';
```

Чума, правда?! Тегов `<style>` вообще не понадобилось!

Отметим, что если в `require()` или `import` упомянуто что-то, не являющееся JS-файлом, значит, это результат работы загрузчика Webpack. Хотя автор – большой поклонник Webpack, на самом деле мы всего лишь импортируем стили в `main.js` посредством Webpack, а не затребуем их глобально с помощью тега `<style>`. Это хорошо, поскольку вместо загрузки на сервер десятков файлов на этапе развертывания законченного кода мы развертываем только один оптимизированный комплект. Можно также ограничить область действия правил CSS тем местом, где они включаются. Есть и другие полезные трюки, о которых можно прочитать по адресу https://github.com/webpack-contrib/css-loader.

Глядя на этот CSS, мы понимаем, зачем нужно было добавлять классы к фигурам. Теперь в процессе стилизации мы можем ссылаться на элементы напрямую. Мы сделали оси тонкими, раскрасили их светло-серым цветом и уменьшили шрифт меток. Сами столбики стали голубыми. Сохраните файл и обновите страницу. Вот и готова наша первая диаграмма, созданная с помощью D3:



Рекомендую поэкспериментировать со значениями `.width` и `.height`, чтобы составить представление о возможностях D3. Обратите внимание, как все масштабируется под любой размер, хотя больше мы никакого кода не изменяли. Фантастика!

Резюме

В этой главе вы узнали, что такое библиотека D3, и вкратце познакомились с философией ее работы. Вы настроили свой компьютер для создания прототипов и экспериментов с визуализацией. Далее в книге мы будем предполагать, что среда уже настроена.

Мы проработали простой пример и, пользуясь простейшими средствами D3, создали анимированную гистограмму. Мы узнали о масштабах и осях, о том, что вертикальная ось направлена вниз, что любое свойство, заданное в виде функции, заново вычисляется для каждого элемента данных. Для придания привлекательности изображению мы воспользовались комбинацией CSS и SVG. Мы также познакомились с возможностями ES2017, Babel и Webpack и установили Node.js. Вот какие мы молодцы!

Но главное – в этой главе мы получили в свое распоряжение базовые инструменты, с помощью которых можно самостоятельно поиграть с D3.js. Пробуйте – и вам воздастся! Не бойтесь что-то сломать – вы всегда сможете восстановить код главы в исходном состоянии, выполнив команду `$ git reset --soft origin/chapter1`, где вместо `1` нужно подставить нужный номер главы.

В последующих главах мы рассмотрим все вышеизложенное более глубоко, обращая особое внимание на взаимодействие DOM, SVG и CSS между собой. В этой главе был представлен обширный материал; если что-то осталось непонятным, не переживайте. Просто наберитесь сил для следующей главы, и постепенно все станет на свои места.

Глава 2

Начала DOM, SVG и CSS

Возможно, вы уже имели дело с манипуляцией DOM и CSS средствами таких библиотек, как jQuery; если так, то многое в этой главе покажется знакомым, поскольку в D3 входит полный набор инструментов манипуляции. Если же нет, то не расстраивайтесь – эта глава введет вас в курс дела.

На HTML DOM очень похоже пространство имен SVG, с которым мы будем сталкиваться в большинстве примеров. SVG – основа построения визуализаций, поэтому мы уделим этой технологии особое внимание. Мы начнем с рисования фигур вручную, а затем в главе 3 повторим весь путь с применением генераторов фигур D3.

В этой главе будут рассмотрены базовые технологии, на которых стоит D3, а именно:

- объектная модель документа (Document Object Model – DOM);
- масштабируемая векторная графика (Scalable Vector Graphics – SVG);
- каскадные таблицы стилей (Cascading Style Sheets – CSS).

DOM

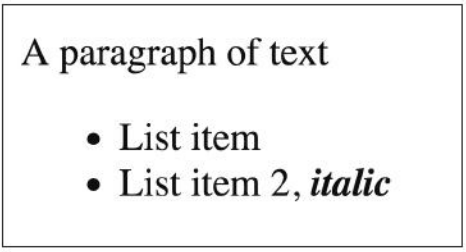
Объектная модель документа (DOM) – не зависящая от языка программирования модель для представления структурированных документов на языках разметки HTML, XML и им подобных. Модель можно рассматривать как дерево узлов, напоминающее результат разбора документа браузером.

В корне дерева находится неявный узел документа, соответствующий тегу `<html>`; браузеры создают этот узел, даже если тег явно не за-

дан, и строят под ним дерево, описывающее, как выглядит документ. Рассмотрим простой HTML-документ:

```
<!DOCTYPE html>
<title>A title</title>
<div>
  <p>A paragraph of text</p>
</div>
<ul>
  <li>List item</li>
  <li>List item 2, <em><strong>italic</strong></em></li>
</ul>
```

Обратите внимание на отсутствие тегов `<html>`, `<head>` и `<body>`. Chrome этот документ разбирает следующим образом:



Вы увидите это дерево, если введете на консоли JavaScript слово `document`. Двойным щелчком мыши дерево можно раскрыть, тогда Chrome будет подсвечивать часть страницы, соответствующую тому элементу на консоли, над которым находится курсор мыши.

Можете также протестировать выбор произвольного элемента, набрав на консоли команду `$('.some-selector')`. Даже если на страницу не импортирована jQuery, эта конструкция все равно будет работать, потому что на консоли Chrome она является встроенным синонимом для `document.querySelector('.some-selector')` (но переопределяется jQuery, если она включена в состав страницы). Аналогично `$$('.some-selector')` является аббревиатурой метода `document.querySelectorAll('.some-selector)`, полезного, когда требуется вернуть все выбранные элементы, не только первый.

Манипулирование DOM с помощью d3-выборки

У каждого узла в дереве DOM имеется целый набор методов и свойств, позволяющих манипулировать документом.

Для примера рассмотрим HTML-код из примера выше. Чтобы сделать слово ***italic*** подчеркнутым, полужирным и курсивным (результат применения тегов `<em>` и `<strong>`), следовало бы написать такой код:

```
document.querySelector('strong').style
  .setProperty('text-decoration', 'underline')
```

Метод `document.querySelector` находит первый элемент с тегом `<strong>` и возвращает его. Затем мы подчеркиваем найденный элемент, задав для него свойство `text-decoration`.

Применение современного DOM API гораздо элегантнее, чем в прошлые времена, но чревато неприятностями, если не дополнить функциональность браузера с помощью *полифилов*. Можно было бы использовать для этой цели библиотеку jQuery, которая предлагает изящный API для такого стиля разработки, или воспользоваться D3, в которую включен аналогичный набор средств для выборки и манипулирования элементами DOM (все они находятся в пакете `d3-selection`).

Благодаря *d3-выборке* мы можем рассматривать HTML просто как еще один тип визуализации данных. Запомните и запишите – старый добрый HTML можно использовать для визуализации данных.

На практике это означает, что, применяя одни и те же приемы, мы можем представить данные в виде таблицы или интерактивного изображения. А главное – для этого понадобятся одни и те же данные.

Перепишем пример выше с помощью d3-выборки:

```
import * as d3 from 'd3';
d3.select('strong').style('text-decoration', 'underline')
```

Гораздо проще! Мы выбрали элемент с тегом `strong` и задали для него свойство с помощью метода `style`. И все!

Кстати говоря, любое свойство, заданное средствами D3, может быть динамическим, т. е. в качестве его значения может фигурировать функция. Ниже мы увидим, почему это полезно.

Описанная только что операция называется выборкой. Она лежит в основе всей работы D3 – метод `d3.select` выбирает один элемент, а метод `d3.selectAll` создает похожую на массив выборку, содержащую несколько элементов. Познакомимся с выборками поближе.

Выборки

Выборка – это массив элементов текущего документа, отобранных с помощью некоторого CSS-селектора. Строить выборку можно по

классу, по идентификатору или по имени тега. Можно даже использовать псевдоселекторы, позволяющие, например, выбрать каждый второй абзац: `p:nth-child(n+1)`.

Псевдоселекторы в сочетании с D3 – весьма эффективный механизм, зачастую они позволяют заменить цикл или сложные математические вычисления. Синтаксически они отличаются наличием двоеточия в начале и описывают состояние элемента. Например, псевдоселектор `:hover` активен, если курсор мыши находится над элементом.

Применяя CSS-селекторы для создания цепочки манипуляций, мы получаем простой язык для определения множества элементов документа. По существу, это тот же язык, к которому мы привыкли при работе с jQuery и CSS.

Чтобы получить элемент с идентификатором `graph`, мы пишем `d3.select('#graph')`. Чтобы получить все элементы с классом `blue` – `d3.selectAll('.blue')`. Для получения всех абзацев документа – `d3.selectAll('p')`.

Селекторы можно комбинировать для создания более сложных условий отбора. Селектор `.llama.duck` реализует логический оператор `AND`: он отбирает все элементы, для которых одновременно заданы классы `.llama` и `.duck`. А чтобы отобрать элементы, для которых задан хотя бы один из классов `.llama` или `.duck`, т. е. реализовать оператор `OR`, нужно перечислить классы через запятую: `.llama, .duck`. А как быть, если надо выбрать дочерние элементы? Тут пригодятся вложенные выборки!

Это можно сделать с помощью простого селектора вида `tbody td` или сцепить два вызова `d3.selectAll()`: `d3.selectAll('tbody').selectAll('td')`. В обоих случаях будут выбраны все ячейки в теле таблицы. Имейте в виду, что во вложенных выборках сохраняется иерархия выбранных элементов, и это открывает некоторые интересные возможности.

А не создать ли нам таблицу?

Пришло время привести пример. Создайте новую папку `chapter2/` в каталоге `lib/`, а в ней пустой файл `table-factory.js`. Начиная с этого места, мы почти не будем модифицировать файл `main.js`, а станем создавать модули, загружаемые из него. Это позволит разбить код на легко управляемые части. Умение работать с модулями исключительно важно для организации и повторного использования кода, при-

вычка разбивать проект на меньшие компоненты сослужит отличную службу любому, кто хочет стать разработчиком на JavaScript.

Поскольку мы не хотим замусоривать кодовую базу HTML-кодом, то напишем ряд функций для построения таблиц. Мы собираемся создать такую таблицу:

```
<table class="d3-table">
  <thead>
    <tr>
      <td>One</td>
      <td>Two</td>
      <td>Three</td>
      <td>Four</td>
      <td>Five</td>
      <td>Six</td>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>q</td>
      <td>w</td>
      <td>e</td>
      <td>r</td>
      <td>t</td>
      <td>y</td>
    </tr>
  </tbody>
</table>
```

Можно было бы просто скопировать ее в `index.html`. Но ведь мы хотим избежать написания HTML, правда? Значит, построим таблицу с помощью D3.

Откройте файл `table-factory.js` и добавьте в него такую фабричную функцию:

```
import * as d3 from 'd3';

export default function tableFactory(_rows) {
  const rows = Array.from(_rows);
  const header = rows.shift();    // Первая строка – заголовок
  const data = rows;              // Все остальное – обычные данные

  const table = d3.select('body')
    .append('table')
    .attr('class', 'table');

  return {
    table,
```

```
    header,
    data,
  };
}
```

Это дает нам внешний контейнер. Сначала мы с помощью метода `Array.from` создали копию аргумента `_rows`, ее имя начинается со знака подчеркивания, чтобы не получилось двух переменных с одним именем. Технически можно было бы проделать все необходимые действия и прямо с переменной `_rows`, но это означало бы изменение данных внутри функции, т. е. *побочный эффект*. При написании программ в функциональном стиле мы стараемся всячески избегать побочных эффектов, поскольку они затрудняют отладку.

Затем мы вырезаем первую строку, которая интерпретируется как заголовок, создаем таблицу, после чего возвращаем заголовок, таблицу и оставшиеся строки в виде одного объекта.

Еще одно новшество ES2015! Вы, вероятно, заметили, что возвращенный объект не является привычным множеством пар ключ-значение, – в ES2015, когда в создаваемом объекте имя ключа совпадает с именем переменной, присваиваемой этому ключу в качестве значения, указывать ключ (и тем самым дважды писать одно и то же имя) необязательно. Иначе говоря, Babel понимает, что `{"someVariable": someVariable}` – то же самое, что `{someVariable}`.

Чтобы увидеть, как это работает, нужно попросить `main.js` загрузить наш новый класс. Открыв файл, мы найдем в нем оставшийся от предыдущей главы код. Ну беспорядок же! Давайте-ка приберемся, воспользовавшись модулями ES2015. Перенесите весь код в файл `chapter1/index.js`, оставив `main.js` пустым.

Теперь файл `main.js` пуст, а весь код из главы 1 находится в отдельном файле. Так мы дальше и будем организовывать код, и пусть это войдет у вас в привычку. Модульность не только позволяет использовать старый код повторно, но и приучает думать о расширяемости.

Если вы ранее извлекли ветку `origin/chapter1` из Git-репозитория, то в ней вы сейчас и находитесь, и рассмотренный выше код хранится в отдельном файле. Если мы по пути где-то разминулись, то, чтобы встретиться снова, выполните команду `git stash save && git checkout origin/chapter1`, находясь в каталоге `learning-d3-v4/`.

Вернемся к файлу `index.js` и создадим уже таблицу.

Добавьте в файл main.js такой код:

```
import tableFactory from './chapter2/table-factory';
```

```
const header = ['one', 'two', 'three', 'four', 'five', 'six'];
const rows = [
  header,
  ['q', 'w', 'e', 'r', 't', 'y'],
];

const table = tableFactory(rows);
```

В командной строке выполните команду:

```
$ npm start
```

В браузере перейдите по адресу `http://127.0.0.1:8080`. Щелкните правой кнопкой мыши по странице, выберите из меню команду **Inspect Element** (Просмотреть код) – и вы увидите элемент `table`:

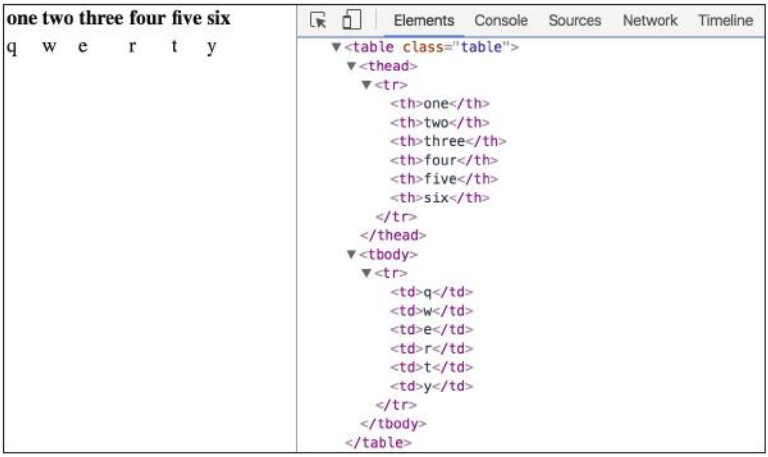
Надо же, действительно есть таблица!

Вернемся к функции `tableFactory` и добавим код формирования таблицы:

```
export default function tableFactory(_rows) {
  const rows = Array.from(_rows);
  const header = rows.shift();    // Первая строка - заголовок
  const data = rows;              // Все остальное - обычные данные

  const table = d3.select('body')
    .append('table')
    .attr('class', 'table');

  const tableHeader = table.append('thead')
    .append('tr');

  const tableBody = table.append('tbody');

  // Каждый элемент "header" - строка.
  header.forEach(value => {
    tableHeader.append('th')
    .text(value);
  });

  // Каждый элемент "data" - массив
  data.forEach(row => {
    const tableRow = tableBody.append('tr');

    row.forEach(value => {
      // Каждый элемент "row" - строка
```

```
      tableRow.append('td')
        .text(value);
    });
  });

  return {
    table,
    header,
    data,
  };
}
```

Теперь таблица должна выглядеть так:



Так что же мы сделали?

Ключ к ответу на этот вопрос – три вызова метода `forEach`. В первом из них в цикле обходится массив строк заголовка и в строку `thead` добавляются ячейки (элементы `td`), содержащие переданные значения. Далее следуют два вложенных вызова `.forEach()`, которые делают то же самое для строк тела таблицы. В данном случае в теле таблицы всего одна строка, поэтому можно было и обойтись без двух вложенных `.forEach()`, зато теперь, если понадобится добавить в таблицу еще одну строку, то достаточно будет просто расширить массив данных в переменной `rows`. В следующей главе мы еще поговорим о методе `Array.prototype.forEach` и других методах массива.

Может показаться, что для столь простой таблицы слишком много работы, но у такого подхода масса преимуществ. Вместо того чтобы

тратить время на ввод статической таблицы, которая нигде больше не используется, мы написали простую JavaScript-библиотеку, которая будет порождать таблицу всякий раз, как нам понадобится. Можно даже расширить функцию `tableFactory`, чтобы она делала что-то иное, не изменяя уже написанного кода.

Только вот эти вложенные `forEach` выглядят некрасиво. К счастью, d3-выборка предлагает механизм добавления, обновления и удаления данных. Вместо того чтобы следить за тем, где что находится в проекте, мы просто должны попросить D3 позаботиться о наших данных, и она будет самостоятельно обновлять все присоединенные элементы. В основе этого механизма лежат выборки и функции, которые ими манипулируют.

Пример выборки

Создание прототипа JavaScript-кода в значительной мере сводится к экспериментам в браузере с помощью консоли JavaScript, где можно изменять значения. Вместо того чтобы раз за разом обновлять страницу и протоколировать результаты, мы можем выполнить операцию в браузере и сразу же увидеть результат. Для этого нужно присоединить функции к глобальному объекту `window`.

Замените все содержимое файла `main.js` таким кодом:

```
import tableFactory from './chapter2/table-factory';
import * as d3 from 'd3';

window.d3 = d3;
window.tableFactory = tableFactory;
```

Тем самым мы присоединили функцию `tableFactory` и библиотеку D3 к глобальному объекту `window`, так что теперь то и другое доступно из консоли.

На консоли разработчика в Chrome введите такие строки:

```
d3.selectAll('.table').remove();
tableFactory([
  [1,2,3,4,5,6],
  ['q', 'w', 'e', 'r', 't', 'y'],
  ['a', 's', 'd', 'f', 'g', 'h'],
  ['z', 'x', 'c', 'v', 'b', 'n']
]);
```

Кстати, чтобы ввести на консоли символ новой строки, нужно одновременно нажать клавиши **Shift** и **Enter**. Но это и необязательно: можете набрать весь этот текст в одной строке, если вам так проще. Я его отформатировал только для наглядности.

В результате старая таблица удаляется и добавляется новая.

Теперь выкрасим текст во всех ячейках в красный цвет:

```
d3.selectAll('td').style('color', 'red')
```

Текст сразу же становится красным. Следующие два сцепленных обращения к `d3.selectAll()` делают текст в заголовке таблицы полужирным:

```
d3.selectAll('thead').selectAll('th').style('font-weight', 'bold')
```

Отлично! Еще поупражняемся с вложенными выборками и сделаем зелеными ячейки во втором, четвертом и шестом столбцах:

```
d3.selectAll('tbody tr').selectAll('td')
  .style('color', (d, i) => { return i%2 ? 'green' : 'red'; })
```

Эта два обращения дают нам все экземпляры `<td>` в теле, разбитые на строки, т. е. массив из трех объектов типа `NodeList` по шесть элементов в каждом: `[ NodeList[6], NodeList[6], NodeList[6] ]`. Они находятся в массиве `_groups`, возвращенном внешним `d3.selectAll()`, тогда как выборки `<tr>` находятся в массиве `_parents`. Затем мы воспользовались методом `.style()`, чтобы изменить цвет выбранных элементов.

Использование функции вместо статического свойства дает нам желаемую точность контроля. При вызове функции передаются элемент данных `d` (мы обсудим этот аргумент ниже) и индекс столбца `i`, в котором этот элемент находится. Нам остается просто вернуть строку *green* или *red* в зависимости от четности индекса.

Следует помнить, что в больших документах сцепление селекторов может оказаться эффективнее их объединения оператором `OR`, поскольку каждая следующая выборка просматривает только элементы, найденные ранее.

Манипулирование содержимым

D3 позволяет гораздо больше, чем просто выборку и изменение свойств элементов. Мы можем также изменять содержимое элементов, добавлять новые элементы и удалять существующие.

Добавим столбец в таблицу из предыдущего примера:

```
const newCol = d3.selectAll('tr').append('td')
```

Мы выбрали все строки таблицы (получив в ответ объект выборки) и добавили в конце каждой строки новую ячейку методом `.append()`. Все методы D3 возвращают текущую выборку, поэтому их можно сце-

пить или присвоить результат переменной (в данном случае `newCol`) для использования в будущем.

Сейчас мы имеем пустой невидимый столбец. Поместим в него текст, чтобы результат проявился:

```
newCol.text('a')
```

Теперь мы хотя бы видим, что новый столбец присутствует. Однако заполнить все ячейки в нем одним и тем же значением `a` неинтересно, поэтому поступим так же, как с другими столбцами:

```
newCol.text((d, i) => ['Seven', 'u', 'j', 'm'][i])
```

Динамическое задание содержимого с помощью функции позволяет выбрать нужную строку из списка значений в зависимости от индекса столбца `i`. Схема понятна?

Метод `.remove()` позволяет удалить элементы. Например, удалить последнюю строку таблицы можно следующим образом:

```
d3.selectAll('tbody tr:last-child').remove()
```

Соединение данных с выборками

Мы подошли к интересной части наших забав с DOM. Помните, я говорил, что HTML – это визуализация данных? Так вот, соединение данных с выборками и есть интересующий нас механизм.

Для соединения данных с выборкой предназначена функция `.data()`. Она принимает в качестве аргумента функцию или массив и необязательную функцию, которая сообщает D3, как отличить одни части данных от других.

В процессе соединения данных с выборкой может произойти одно из трех:

- данных больше, чем выбранных элементов (длина данных больше длины выборки). Для ссылки на новые данные служит функция `.enter()`;
- данных ровно столько, сколько и раньше. Для обновления состояния элементов можно воспользоваться выборкой, возвращенной самой функцией `.data()`;
- данных меньше, чем раньше. Для ссылки на отсутствующие данные служит функция `.exit()`.

В версии D3 v4 появилась еще функция `.merge()`, которая позволяет выполнять операции над выборками, содержащими как новые, так и обновленные элементы.

Функции `.enter()` и `.exit()` нельзя сцеплять, потому что они возвращают просто ссылки, не создавая новую выборку, а вот функции `.enter()` и `.merge()` сцеплять *можно*. Это означает, что обычно наше внимание обращено на `.enter()` и `.exit()`, и все три случая нужно обрабатывать отдельно.

Возникает вопрос: «Как возможно, что данных одновременно больше и меньше, чем раньше?» Дело в том, что элементы выборки связаны с конкретными элементами данных, а не с их номерами. Если вы сначала сдвинули массив (т. е. удалили первый элемент), а затем добавили новый элемент в конец массива, то старый первый элемент будет доступен по ссылке `.exit()`, а новый – по ссылке `.enter()`. Это невероятно мощный механизм, который позволяет D3 отслеживать изменение данных, не заставляя нас писать однообразный код.

Элемент данных (datum) передается функциям обратного вызова в аргументе `d`. Отметим также, что существует метод `selection.datum()`, который принимает единственный объект и применяет его ко всем выбранным элементам.

Давайте сделаем что-нибудь интересное с помощью соединений и HTML.

Пример визуализации с помощью HTML

Мы визуализируем данные Всемирной организации здравоохранения по ожидаемой продолжительности жизни в разных странах, представив их в виде простой таблицы.

Набор данных можно получить по адресу http://apps.who.int/gho/data/node.main.688, щелкнув по ссылке **JSON simplified structure**. Сохраните его в каталоге `data/` (если вы в начале главы выполнили команду `git checkout`, то набор данных уже там).

Создайте файл `index.js` в каталоге `lib/chapter2` и замените код в файле `main.js` таким:

```
import lifeExpectancyTable from './chapter2/index';
lifeExpectancyTable();
```

Добавьте показанный ниже код в файл `chapter2/index.js`:

```
import tableFactory from './table-factory';

export default async function lifeExpectancyTable() {
  const getData = async () => {
    try {
      const response = await fetch('data/who-gho-life-expectancy.json');
```

```
      const raw = await response.json();
    } catch (e) {
      console.error(e);
      return undefined;
    }
  };
}
```

Здесь мы воспользовались одной из лучших, на мой взгляд, возможностей ES7, которую мне не терпится вам представить.

Взглянув на определение функций `lifeExpectancyTable()` и `getData()`, вы заметите новое ключевое слово `async`. Оно означает, что функция асинхронная, т. е. возвращает объект типа `Promise`. О том, что такое `Promise` (обещание), мы подробно поговорим в главе 3, а пока можете считать, что это переменная специального вида, которая примет значение в будущем. Это позволяет использовать в асинхронных функциях еще одно ключевое слово, `await`, означающее «*дождаться, когда переменная примет значение, а затем продолжить*». Мы также воспользовались новым Fetch API, который и сам возвращает обещание, а это значит, что мы можем с помощью `await` ждать *всего, чего угодно*!

По большей части мы будем использовать для загрузки данных именно эту схему, хотя в главе 3 рассмотрим и другие варианты. Вон с наших глаз, дурно пахнущий `XMLHttpRequest` из предыдущей главы!

Располагая данными, мы можем приступить к построению Таблицы Жизни (в смысле, ожидаемой ее продолжительности). Добавьте следующий код между `const raw` и блоком `catch`:

```
return raw.fact.filter(d => d.dim.GHO === 'Life expectancy at birth (years)'
  && d.dim.SEX === 'Both sexes' && d.dim.YEAR === '2014')
  .map(d => [
    d.dim.COUNTRY,
    d.Value,
  ]);
```

Этот код заменяет странный формат, принятый ВОЗ, гораздо более простым, оставляя только ожидаемую продолжительность жизни при рождении в 2014 году для обоих полов. Пришло время воспользоваться нашим построителем таблиц `tableBuilder`.

Добавьте показанный ниже код в конец функции `lifeExpectancyTable`, прямо перед закрывающей фигурной скобкой:

```
const data = await getData();
data.unshift(['Country', 'Life expectancy (years from birth)']);
return tableFactory(data);
```

Он формирует строку заголовка таблицы и передает ее нашей функции `tableFactory`. В браузере должна появиться такая скучная неотсортированная таблица:

Country	Life expectancy (from birth)
Comoros	63.2
Cuba	79.0
Cyprus	80.3
Burundi	59.1
Bosnia and Herzegovina	77.2
Switzerland	83.2
Australia	82.7
Belgium	80.9
Bahrain	76.8
Bolivia (Plurinational State of)	70.4
Central African Republic	50.8
Angola	51.7
Cabo Verde	73.0
Austria	81.4
Chile	80.3
Cote d'Ivoire	52.8

Это уже прекрасный результат, хотя данной таблице пока недостает самого главного свойства, которое мы ожидаем от таблиц, – возможности быстро находить нужную информацию.

Мы хотим отсортировать ее по алфавиту. Однако для этого функция `tableFactory()` должна быть чуть поумнее. Попробуем воспользоваться соединениями, о которых узнали выше, и перепишем функцию следующим образом:

```
export default function tableFactory (_rows) {
  const rows = Array.from(_rows);
  const header = rows.shift(); // Вырезать первый элемент, это заголовок

  const table = d3.select('body')
    .append('table')
    .attr('class', 'table');

  table.append('thead')
    .append('tr')
    .selectAll('td')
    .data(header)
```

```
    .enter()
      .append('th')
      .text(d => d);

  table.append('tbody')
    .selectAll('tr')
    .data(rows)
    .enter()
      .append('tr')
      .selectAll('td')
      .data(d => d)
        .enter()
        .append('td')
        .text(d => d);

  return {
    table,
    header,
    rows,
  };
}
```

Мы использовали метод `selection.data` для создания строк и ячеек таблицы. Поскольку теперь D3 отслеживает данные и их связь с тем, что изображено на экране, мы можем выполнять различные операции над данными, а их результаты будут отражаться в таблице.

Заменим последнюю строку функции `lifeExpectancyTable` таким кодом:

```
return tableFactory(data).table
  .selectAll('tr')
  .sort();
```

Теперь таблица будет перерисована в другом порядке, хотя мы не обновляли страницу и не добавляли и не удаляли элементов вручную. Поскольку данные соединены с HTML, нам даже не понадобилась ссылка на исходную выборку элементов `tr` или на данные. На мой взгляд, прелестно.

Функция `.sort()` принимает функцию сравнения, а та, в свою очередь, получает два аргумента `a` и `b` и возвращает `-1`, если `a` меньше `b`, `0` – если `a` равно `b`, и `1` – если `a` больше `b`. Если `a` и `b` – числа, то для

сортировки в порядке возрастания достаточно просто вычесть `b` из `a` (а для сортировки в порядке убывания – вычесть `a` из `b`). Можно также воспользоваться функциями сравнениями `d3.ascending` и `d3.descending`, входящими в состав D3, – это, с одной стороны, короче, а с другой – гораздо понятнее передает, чего мы добиваемся.

Но представьте себе человека, который видит нашу таблицу в газете или журнале. Что больше всего интересует читателя? Средняя продолжительность жизни в конкретной стране или страны с наибольшей или наименьшей ожидаемой продолжительностью жизни? Если говорить о новостных СМИ, то читателей обычно интересуют максимальные и минимальные числа: кто самый быстрый, что самое большое, какая компания заработала больше всего денег? В данном случае имеет смысл сортировать данные по ожидаемой продолжительности жизни, а для этого нужно заменить вызов `.sort()` таким кодом:

```
.filter(i => i)
.sort(([countryA, yearsA], [countryB, yearsB]) => yearsA - yearsB);
```

Использованная здесь возможность ES2015 довольно проста. Сначала мы производим фильтрацию по *идентификатору*, чтобы исключить неопределенные строки. Затем задаем функцию сравнения, принимающую два аргумента, – в данном случае нам известно, какие элементы массива содержат название страны и среднюю продолжительность жизни в ней. Мы применили *деструктурирующее присваивание массива* – появившийся в ES2015 механизм, позволяющий присвоить первый элемент первого аргумента переменной `countryA`, первый элемент второго аргумента – переменной `countryB`, второй элемент первого аргумента – переменной `yearsA` и т. д. Затем мы вычитаем `yearsB` из `yearsA`, чтобы отсортировать данные в порядке возрастания. Если бы мы захотели отсортировать в порядке убывания, то надо было бы вместо этого вычитать `yearsA` из `yearsB`.

Country	Life expectancy (years from birth)
Sierra Leone	48.1
Central African Republic	50.8
Angola	51.7
Lesotho	52.1
Chad	52.6
Cote d'Ivoire	52.8
Nigeria	53.6
Somalia	54.3
South Sudan	56.6
Mozambique	56.7
Cameroon	56.7
Malawi	57.6
Mali	57.8
Equatorial Guinea	57.9
Guinea	58.1
Liberia	58.1
Guinea-Bissau	58.4
Swaziland	58.4
Burundi	59.1
Zimbabwe	59.2
Burkina Faso	59.3
Democratic Republic of the Congo	59.3
Benin	59.7
Togo	59.7
Afghanistan	59.9
United Republic of Tanzania	60.7
Gambia	60.8
Zambia	61.1
Niger	61.4
Uganda	61.5
South Africa	62.0

Масштабируемая векторная графика

SVG (Scalable Vector Graphics) – формат векторной графики для описания изображений на языке XML. Он появился в 1999 году и сейчас поддерживается всеми основными браузерами. Векторное изображение можно нарисовать в любом масштабе без потери качества. То есть одно и то же изображение будет одинаково хорошо выглядеть и на дисплее, изготовленном по технологии Retina, и на маленьком экране мобильного телефона.

SVG-изображения составлены из фигур, которые можно создать вручную с помощью путей или собрать из базовых фигур (например, прямых и окружностей), определенных в стандарте. Фигуры представлены в формате элементами XML и их атрибутами. Поскольку это основанный на XML формат, как и HTML, многое из того, что вы знаете о HTML, относится также и к SVG.

Сам по себе SVG-код – это просто текст, который можно вручную редактировать, исследовать с помощью отладочных инструментов браузера и сжимать стандартными алгоритмами сжатия. Поскольку формат текстовый, мы можем создать изображение средствами D3, а затем скопировать получившийся XML-код в `svg`-файл и открыть его в другой программе просмотра SVG, например Preview или Adobe Illustrator.

Кроме того, браузер может рассматривать SVG как обычную часть документа. Для его стилизации можно использовать CSS, к фигурам можно присоединять обработчики событий мыши и даже перемещать их в процессе анимации. Все это делает SVG одним из самых эффективных средств визуализации данных в вебе.

Рисование с помощью SVG

Для рисования в D3 мы можем добавлять фигуры вручную, определяя соответствующие элементы SVG, или пользоваться вспомогательными функциями, которые позволяют без труда создавать сложные фигуры.

В основном мы в этой книге будем работать с SVG. К этому очень быстро привыкаешь. У большинства диаграмм в D3 много общих черт, поэтому мы напишем фабричную функцию, которая будет выполнять подготовительные действия и тем сэкономит нам уйму времени. Создайте папку `common/` внутри `lib/`, а в ней файл `index.js`. Поместите в этот файл код импорта D3:

```
import * as d3 from 'd3';
```

Подготовим прототип, который впоследствии можно будет переопределить:

```
const protoChart = {
  width: window.innerWidth,
  height: window.innerHeight,
  margin: {
    left: 10,
    right: 10,
```

```
    top: 10,
    bottom: 10,
  },
};
```

Это обычная заготовка, содержащая значения, которые нам понадобятся позже: поля шириной 10 пикселей с каждой стороны и ширину и высоту, равные внутренним размерам окна браузера.

Затем создадим фабричную функцию, которая возвращает простую диаграмму с указанными выше параметрами и позволяет их при необходимости переопределить:

```
export default function chartFactory(opts, proto = protoChart) {
  const chart = Object.assign({}, proto, opts);

  chart.svg = d3.select('body')
    .append('svg')
    .attr('id', chart.id || 'chart')
    .attr('width', chart.width - chart.margin.right)
    .attr('height', chart.height - chart.margin.bottom);

  chart.container = chart.svg.append('g')
  .attr('id', 'container')
  .attr('transform', &grave;translate(${chart.margin.left},
                                      ${chart.margin.top})&grave;);

  return chart;
}
```

Здесь определенный выше объект `protoChart` является значением по умолчанию аргумента `proto`. Таким образом, мы можем либо переопределить отдельные свойства в первом аргументе, либо передать совершенно другой прототип, ели нужно нечто необычное. Затем мы с помощью d3-выборки добавляем элемент SVG в тело страницы и задаем его атрибуты `width`, `height` и `margin`, так чтобы элемент имел нужный нам размер и правильные поля справа и снизу. Далее добавляется групповой элемент `g`, который мы сдвигаем вправо и вниз, чтобы сформировать поля сверху и слева.

Все это просто, но сэкономит нам время в последующих главах.

Добавление элементов и фигур вручную

SVG-изображение – это набор элементов, отрисовываемых как геометрические фигуры. В стандарте определены семь базовых элементов, и почти все они – сокращенный способ определения пути:

- отрезки прямых (путь с двумя точками);
- прямоугольники (путь с четырьмя точками и прямыми углами);

- окружности (круговой путь);
- эллипсы (продолговатый путь);
- ломаные (путь, состоящий из отрезков прямых);
- многоугольники (замкнутый путь, состоящий из отрезков прямых);
- текст (единственный элемент, не являющийся путем).

Для построения SVG-изображений мы добавляем эти элементы в документ и задаем атрибуты. Для любого элемента можно задать стиль обводки, определяющий, как надо рисовать границы, и стиль заливки. Все элементы можно поворачивать, наклонять и параллельно переносить с помощью атрибута `transform`.

Текст

Как уже было сказано, текст – единственный элемент SVG, не являющийся путем. Мы рассмотрим сначала именно его, а оставшуюся часть главы посвятим геометрическим фигурам. Добавьте в начало файла `chapter2/index.js` такие строки:

```
import * as d3 from 'd3';
import chartFactory from '../common/index';
```

Затем добавьте новую функцию ниже `lifeExpectancyTable`:

```
export async function renderSVGStuff() {
  const chart = chartFactory();
  const text = chart.container.append('text')
    .text("Ceci n'est pas un trajet!")
    .attr('x', 50)
    .at tr('y', 200)
    .attr('text-anchor', 'start');
}
```

Мы воспользовались нашей функцией `chartFactory` для создания новой диаграммы. Затем мы добавили элемент `text` в свойство `container`, соответствующее групповому элементу, в котором заданы все поля. После этого мы задали сам текст и добавили атрибуты для его позиционирования в точке `(x, y)`. Атрибут `text-anchor` определяет горизонтальное положение текста относительно якорной точки с координатами `(x, y)`. Он может принимать значения `start`, `middle` и `end`.

Мы также можем подстроить положение текста, задав смещение с помощью атрибутов `dx` и `dy`. Это особенно удобно, когда нужно задать поле и базовую линию текста относительно размера шрифта, поскольку размеры можно задавать в единицах `em`.

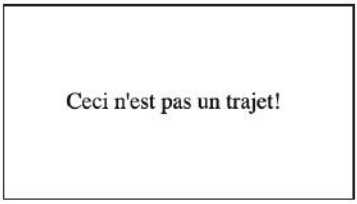
Теперь перейдем к файлу `lib/main.js` и импортируем новую функцию. Замените первое предложение `import` в файле `lib/main.js` таким:

```
import './chapter2/index';
```

Затем добавьте следующую строку в конец файла `chapter2/index.js`:

```
renderSVGStuff();
```

Проверяем.



Функция находится в файле `chapter2/index.js`, так что нам нужно только импортировать ее в `lib/main.js`. Во всех последующих примерах SVG мы будем просто добавлять код в `renderSVGStuff`, чтобы можно было сосредоточиться на библиотеке D3, а не заниматься бесконечным написанием оберток на JavaScript. Ничего особенно полезного мы в этой главе не делаем, так что мои рассуждения о пользе модульности до конца этого раздела не пригодятся.

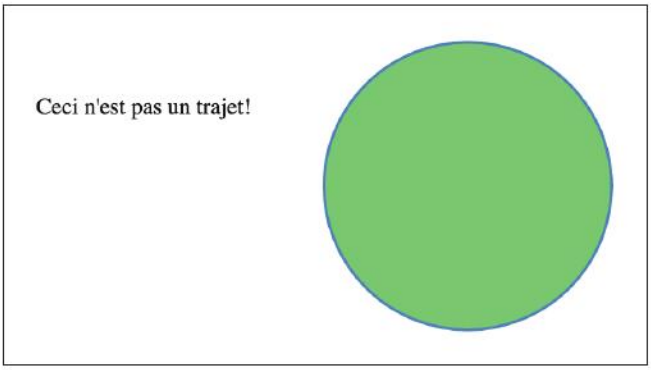
Геометрические фигуры

Разобравшись с текстом, перейдем к более интересной теме – геометрическим примитивам, из которых мы будем строить все, от столбиков столбчатой диаграммы до сложных географических карт.

Начнем со случаев, когда нужна всего одна точка:

```
svg.append('circle')
  .attr('cx', 350)
  .attr('cy', 250)
  .attr('r', 100)
  .attr('fill', 'green')
  .attr('fill-opacity', 0.5)
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 2);
```

Окружность определяется своим центром (`cx, cy`) и радиусом `r`. В данном случае мы получаем зеленый круг с синей границей:



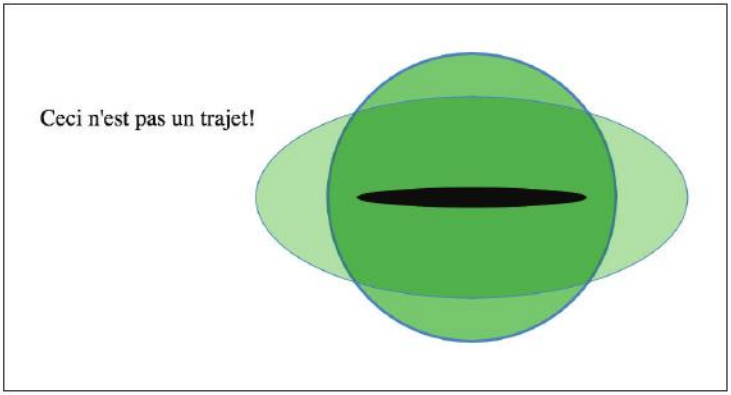
Математически окружность – частный случай эллипса. Чтобы сделать ее вытянутой, нужно задать атрибуты `rx` и `ry`:

```
const ellipses = chart.container.append('ellipse')
  .attr('cx', 350)
  .attr('cy', 250)
  .attr('rx', 150)
  .attr('ry', 70)
  .attr('fill', 'green')
  .attr('fill-opacity', 0.3)
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 0.7);
```

Мы добавили элемент `ellipse` и задали ряд хорошо известных атрибутов. Для определения эллипса нужно задать центр (`cx, cy`) и два радиуса, `rx` и `ry`. Благодаря небольшому значению атрибута `fill-opacity` круг под эллипсом виден. Добавим еще один эллипс:

```
chart.container.append('ellipse')
  .attr('cx', 350)
  .attr('cy', 250)
  .attr('rx', 80)
  .attr('ry', 7)
```

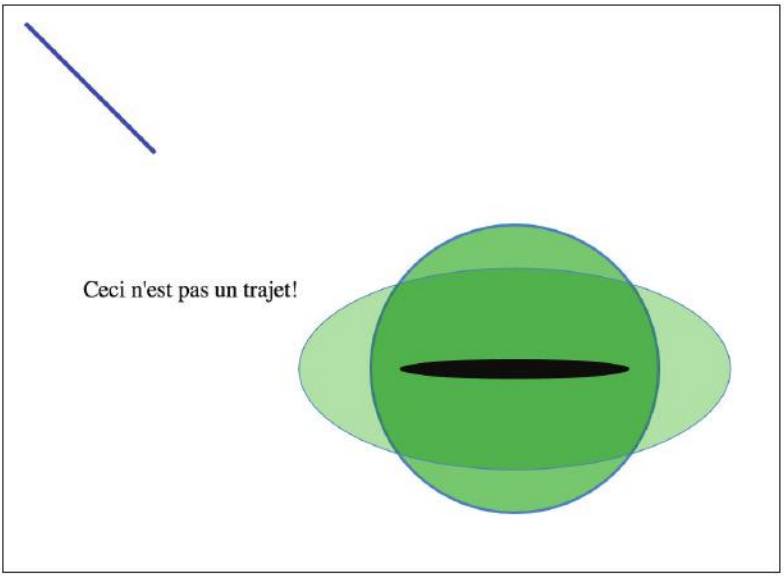
Здесь `rx` значительно больше `ry`, так что эллипс будет сильно сплющенным. И в результате получилось что-то вроде зеленоватого глаза!



Но нам этого мало. Мы хотим еще диагональную синюю прямую. Она, безусловно, придаст завершенность этому произведению искусства. Чтобы нарисовать отрезок прямой, добавьте такой код:

```
const line = chart.container.append('line')
  .attr('x1', 10)
  .attr('y1', 10)
  .attr('x2', 100)
  .attr('y2', 100)
  .attr('stroke', 'blue')
  .attr('stroke-width', 3);
```

Как и раньше, мы добавили в контейнер новый элемент и задали его атрибуты. Мы перешли к элементам, определяемым несколькими точками, – отрезку, соединяющему точки (`x1, y1`) и (`x2, y2`). Чтобы отрезок был виден, нужно задать атрибуты `stroke` (цвет линии) и `stroke-width` (толщину линии), иначе получится черная линия нулевой толщины, она, конечно, будет невидимой.



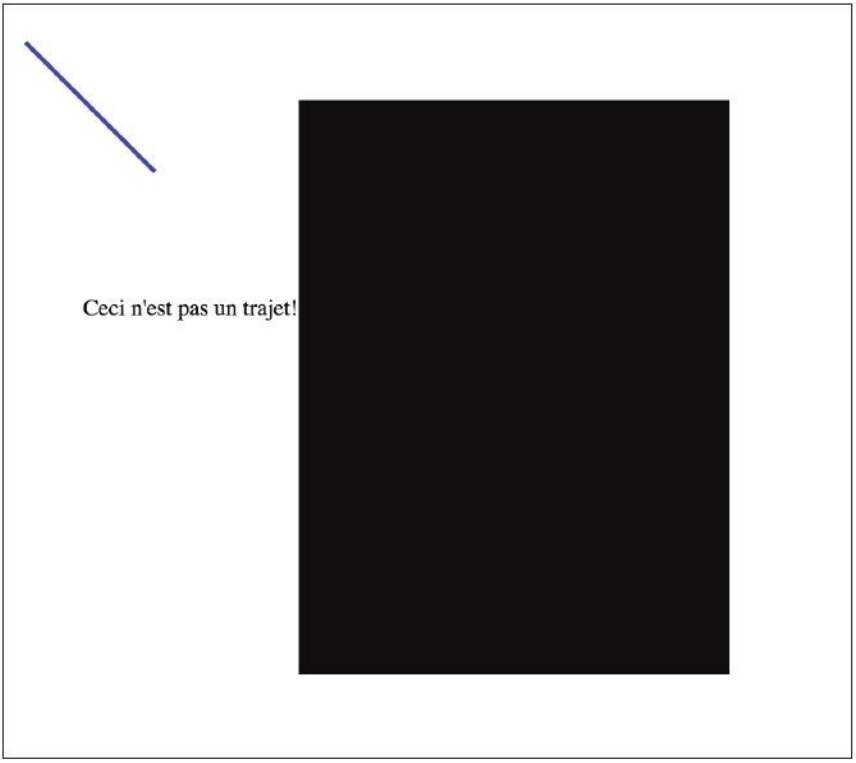
Наш отрезок направлен вниз, хотя `y2` больше `y1`. Объясняется это тем, что в большинстве графических форматов начало координат (`x=0, y=0`) находится в левом верхнем углу.

Гм, чего-то еще не хватает... А давайте-ка рассмотрим элементы с четырьмя вершинами! Для рисования прямоугольника нам понадобится элемент `rect`:

```
const rect = chart.container.append('rect')
  .attr('x', 200)
  .attr('y', 50)
  .attr('width', 300)
  .attr('height', 400);
```

Мы добавили в контейнер элемент `rect` и задали его основные атрибуты: координаты левого верхнего угла (`x, y`), ширину `width` и высоту `height`.

Теперь наше изображение выглядит так:



Получился довольно большой черный прямоугольник. Именно так и выглядят фигуры, если не задать свойств заливки и обводки, – по умолчанию фигуры SVG не имеют границы и заливаются стопроцентно непрозрачным черным цветом. Хуже того, этот прямоугольник еще и загородил наш таинственный зеленый глаз!

Порядок расположения по оси z в SVG совпадает с порядком появления элементов в разметке, и никакого аналога имеющегося в HTML z-индекса, позволяющего изменить этот порядок, нет. Чтобы поместить прямоугольник за глазом или поверх глаза, нужно изменить строку

```
const rect = chart.container.append('rect')
```

Воспользуемся методом `insert()` вместо `append()`:

```
const rect = chart.container.insert('rect', 'circle')
```

Первый аргумент – тип вставляемого элемента, второй – CSS-селектор элемента, перед которым вставляется данный. В данном случае мы хотим поместить прямоугольник за ранее нарисованным кругом, поэтому указываем круг. Но, в принципе, можно указать любой CSS-селектор – например, если задать `:firstchild`, то элемент окажется на вершине стека, т. е. за всеми остальными (включая и самый первый текстовый элемент).

Того же результата можно было бы добиться и другим способом, появившимся в версии D3 v4, – воспользоваться функциями `selection.raise()` и `selection.lower()`. Функция `selection.raise()` пересоздает выборку, делая ее самым нижним элементом в родительском контейнере, а функция `selection.lower()` делает выборку следующим за ее родителем элементом стека. Это очень полезно, когда мы хотим создать несколько элементов внизу некоторой группы SVG, а затем перемещать их наверх в ответ на щелчок или касание.

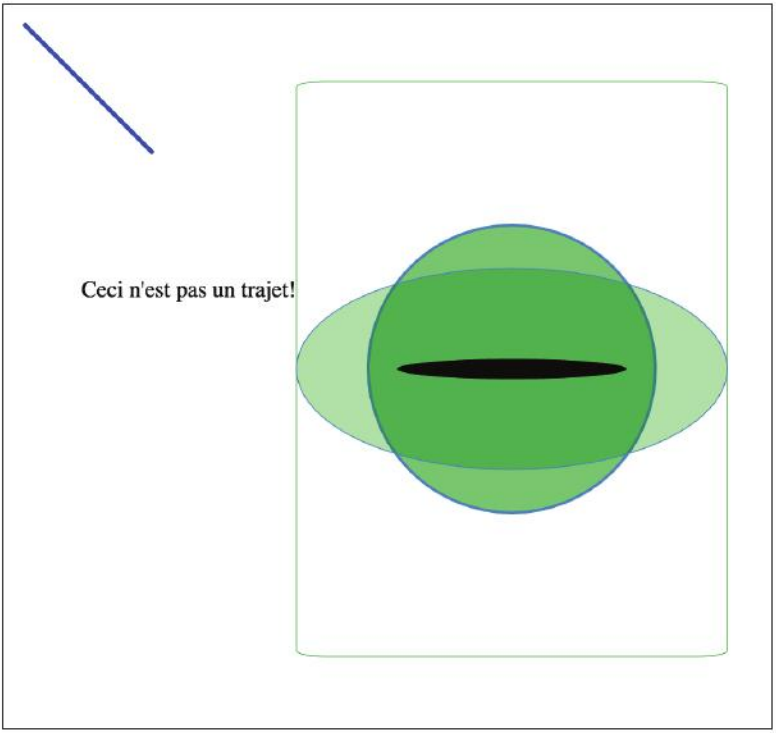
Давайте произведем заливку и обводку прямоугольника, задав еще три свойства:

```
rect.attr('stroke', 'green')
  .attr('stroke-width', 0.5)
  .attr('fill', 'white')
```

И немного скруглим углы:

```
  .attr('rx', 20)
  .attr('ry', 4);
```

Получится вот что:



Так-то лучше, правда? У прямоугольника появилась тонкая зеленая граница. А скругление углов определяется атрибутами `rx` и `ry`, задающими радиусы в направлении осей x и y.

Чтобы увидеть сгенерированный код SVG в формате XML, щелкните правой кнопкой мыши по изображению и выберите из меню команду **Просмотреть код** – она покажет выбранный элемент в окне Инструментов разработчика:

```
<svg id="chart" width="801" height="726">
  <g id="container" transform="translate(10, 10)">
    <rect x="200" y="50" width="300" height="400"/>
    <text x="50" y="200" text-anchor="start">
      Ceci n'est pas un trajet!</text>
    <circle cx="350" cy="250" r="100" fill="green"
      stroke="steelblue" fill-opacity="0.5" stroke-width="2"/>
```

```
    <ellipse cx="350" cy="250" rx="150" ry="70" fill="green"
      fill-opacity="0.3" stroke="steelblue" stroke-width="0.7">
        <ellipse cx="350" cy="250" rx="20" ry="7"/>
    </ellipse>
    <line x1="10" y1="10" x2="100" y2="100" stroke="blue"
      stroke-width="3"/>
  </g>
</svg>
```

Не хотел бы я писать такое вручную.

Но как бы то ни было, все заданные нами элементы и атрибуты налицо. Возможность посмотреть на файл разметки и понять, что происходит, очень полезна для отладки. При работе с чем-то вроде Canvas (мы затронем эту тему в главе 8) результатом отрисовки будет единственный элемент, который может быть представлен неразборчивой строкой в кодировке base64. Это одновременно и важнейший плюс SVG, и его ахиллесова пята – работать с кодом и отлаживать его очень просто, но хранение сложного дерева элементов в памяти с некоторого момента приводит к замедлению браузера (сейчас считается, что прибегать к Canvas следует, когда в состав диаграммы входит более 1000 элементов SVG).

Мы рассмотрели фигуры, определяемые 1, 2, 3 и 4 точками (окружности, прямые, эллипсы и прямоугольники соответственно). Я говорил, что ломаные и многоугольники также являются базовыми элементами SVG, но пока мы с ними не встречались. Хотя такие фигуры можно построить с помощью d3-выборок, как и выше, но гораздо проще воспользоваться генераторами путей, входящими в состав D3. Скоро мы к этому перейдем.

Преобразования

Но прежде чем переходить к более сложным вещам, рассмотрим преобразования. Пока что будем считать, что объектом преобразования являются фигуры SVG целиком. Не вдаваясь в математические детали, скажем, что в SVG под преобразованиями понимаются аффинные преобразования системы координат, в которой рисуются фигуры. Такие преобразования можно описать умножением матриц, которое вычисляется очень эффективно – графическая карта гораздо быстрее вычислит анимацию фигуры, перемещающейся слева направо, если будет применять преобразование, а не пересчитывать положение каждой точки напрямую.

Но если ваши мозги не заточены под линейную алгебру, то работать с преобразованиями как с матрицами довольно сложно. SVG приходит на помощь, предоставляя набор предопределенных преобразований: `translate()`, `scale()`, `rotate()`, `skewX()` и `skewY()`.

Согласно Википедии, аффинным называется преобразование, которое переводит точку в точку, прямую в прямую, плоскость в плоскость, и при этом параллельные прямые остаются параллельными. Расстояния между точками могут изменяться, но отношение расстояний между точками на прямой сохраняется. Следовательно, с помощью аффинного преобразования прямоугольник можно повернуть, увеличить, даже превратить в параллелограмм, но невозможно из прямоугольника получить трапецию.

Компьютеры работают с преобразованиями как с матрицами, потому что любую последовательность преобразований можно представить одной матрицей. А это значит, что при рисовании фигуры нужно будет применить только одно преобразование.

Для применения преобразований служит атрибут `transform`. Мы можем определить несколько последовательно выполняемых преобразований. Результат может зависеть от порядка операций, как будет ясно из примеров ниже.

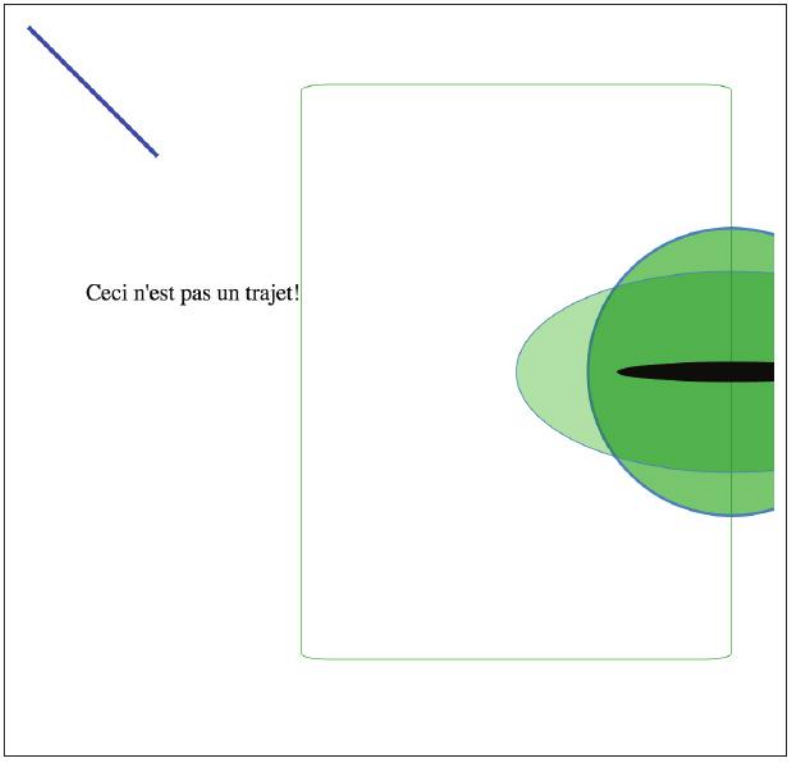
Давайте сдвинем глаз на границу прямоугольника:

```
rt.container.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0)');
```

Мы выбрали все элементы, составляющие глаз (два эллипса и окружность), и применили к ним преобразование `translate`, которое сдвигает начало системы координат фигуры на вектор `(150, 0)`, т. е. перемещает фигуру на 150 пикселей вправо и на 0 пикселей вниз.

Попробовав сдвинуть фигуру еще раз, мы обнаружим, что новые преобразования применяются к исходному состоянию фигуры, т. к. у каждой фигуры может быть не более одного атрибута `transform`.

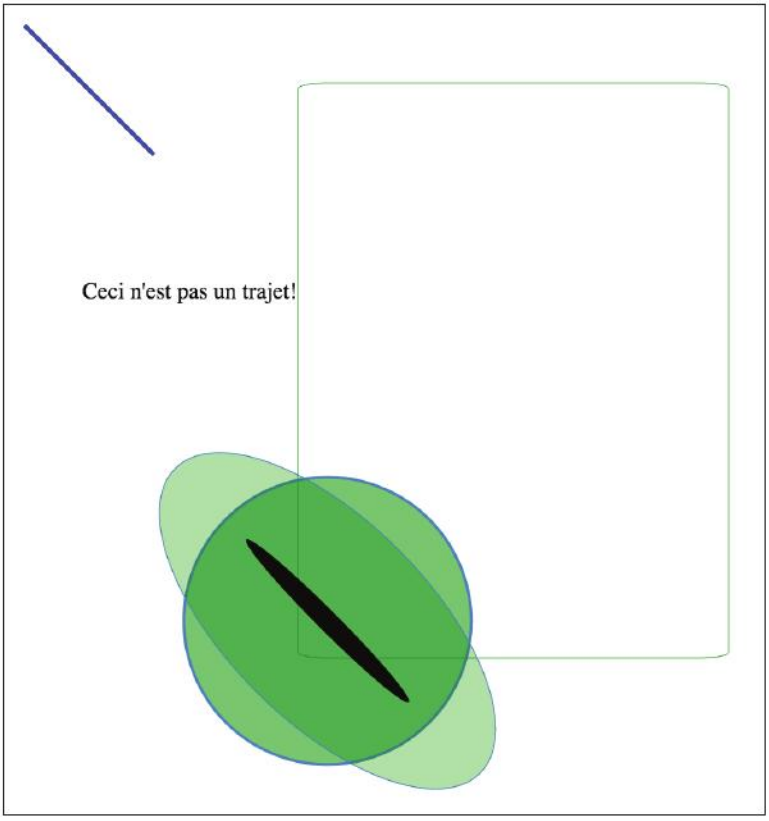
Теперь рисунок выглядит так:



Повернем глаз на 45 градусов:

```
chart.container.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0) rotate(45)');
```

Результат будет таким:

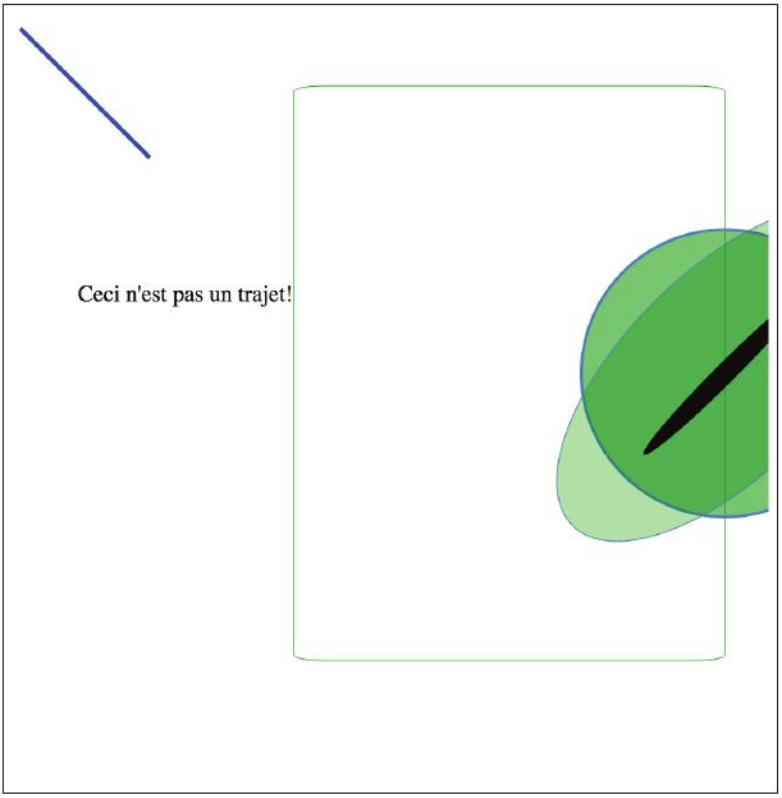


Но это совсем не то, что мы хотели!

Дело в том, что поворот производится относительно начала координат, а не относительно самой фигуры. Мы должны определить центр вращения самостоятельно:

```
chart.container.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0) rotate(-45, 350, 250)');
```

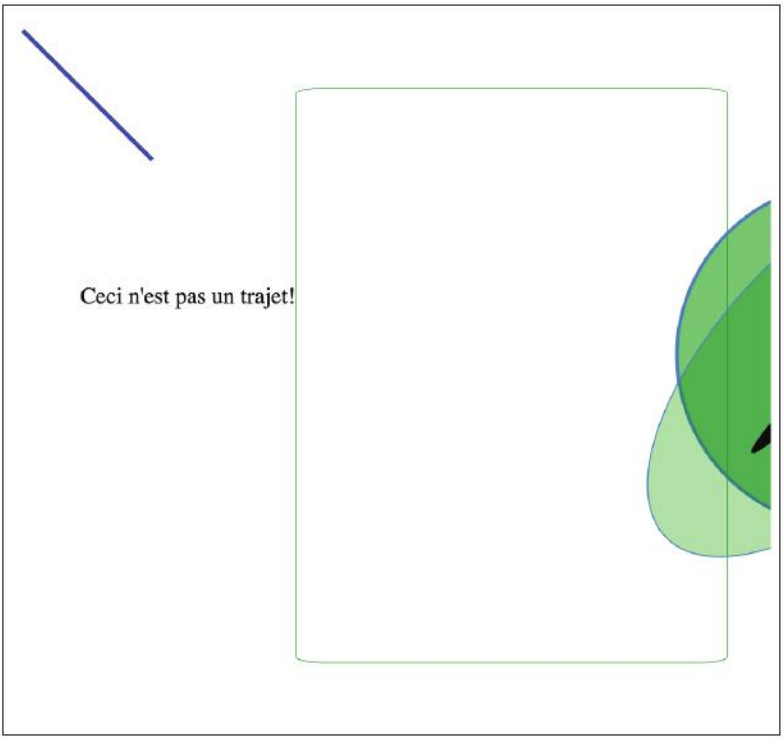
Задав еще два аргумента `rotate()`, мы определили центр вращения и достигли желаемого результата:



Теперь немного увеличим глаз с помощью преобразования `scale()`:

```
chart.container.selectAll('ellipse, circle')
  .attr('transform', 'translate(150, 0) rotate(-45, 350, 250) scale(1.2)');
```

В результате наш объект увеличится в 1.2 раза по обеим осям; если бы мы задали два аргумента, то коэффициенты масштабирования по осям можно было бы сделать разными.

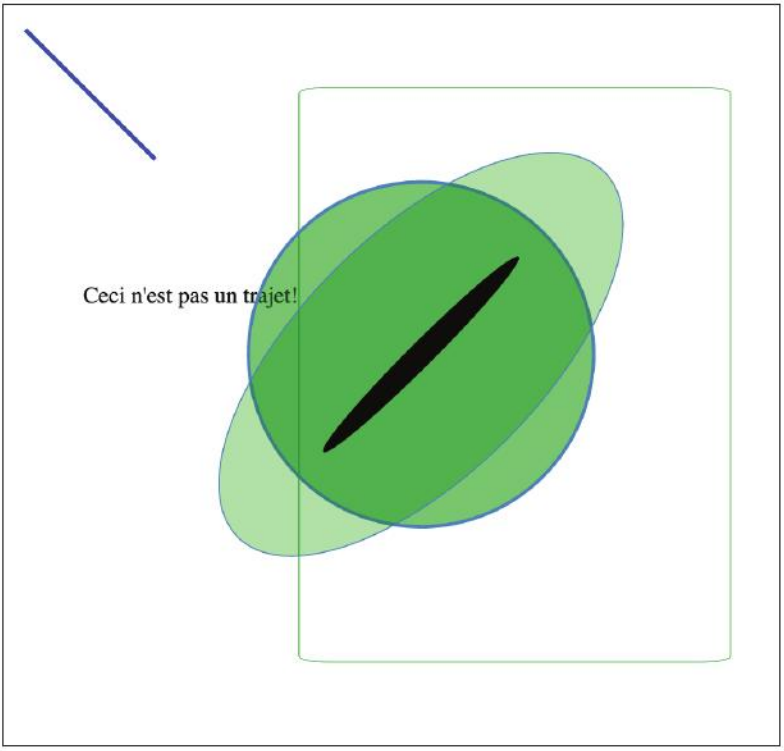


И снова глаз сдвинулся в сторону, потому что центр масштабирования совпадает с началом координат изображения в целом. Чтобы вернуть глаз назад, нужно применить еще один параллельный перенос. Однако система координат, с которой мы работаем, теперь повернута на 45 градусов и масштабирована. Это усложняет нашу задачу. Чтобы сдвинуть глаз на 70 пикселей влево, мы должны произвести параллельный перенос вдоль обеих осей на `70*sqrt(2)/2` пикселей (коэффициент равен синусу и косинусу угла 45 градусов).

Но это как-то запутанно. И само число странное, и работы слишком много для такой простой операции. Да еще рисунок заехал за пределы страницы. Давайте вместо этого изменим порядок операций:

```
chart.container.selectAll('ellipse, circle')
  .attr('transform', &grave;translate(150, 0)
    scale(1.2)
    translate(-250, 0)
    rotate(-45, ${350 / 1.2}, ${250 / 1.2})&grave;);
```

Гораздо лучше! Мы получили ровно то, что хотели.

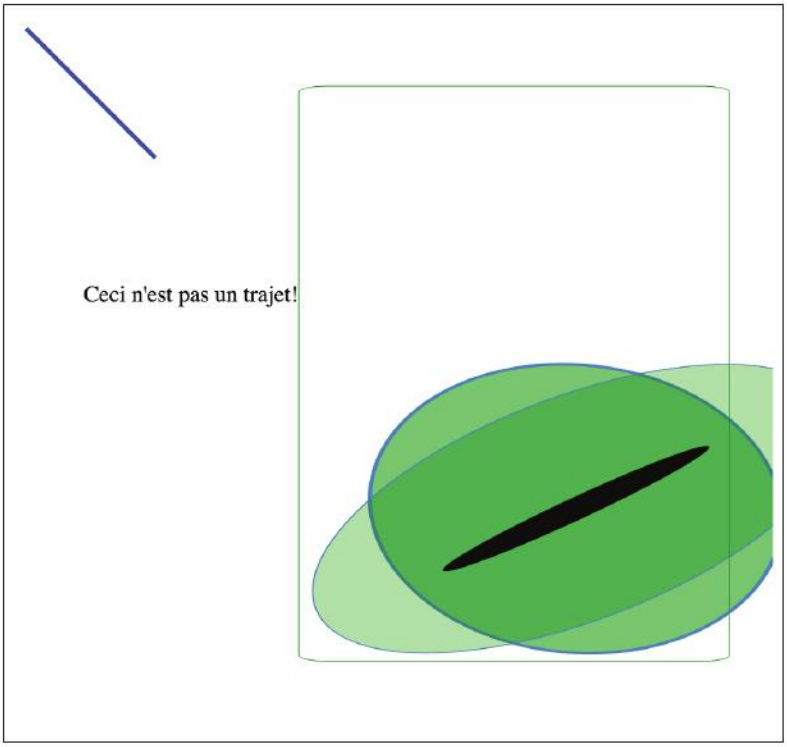


Разберемся, что изменилось. Сначала мы выполнили параллельный перенос `translate` в уже знакомую позицию, а затем масштабирование `scale` с коэффициентом `1.2`, в результате чего глаз сдвинулся в сторону. Чтобы исправить это, мы произвели параллельный перенос влево на 250 пикселей, и, наконец, выполнили поворот на 45 градусов, не забыв поделить координаты центра вращения на `1.2`.

Есть еще одна операция, которой мы еще не подвергали наш бедный глаз: скос. Существуют два таких преобразования: `skewX` и `skewY`, выполняющих скос вдоль соответствующей оси:

```
chart.container.selectAll('ellipse, circle')
  .attr('transform', &grave;translate(150, 0)
    scale(1.2)
    translate(-250, 0)
    rotate(-45, ${350 / 1.2}, ${250 / 1.2})
    skewY(20)&grave;);
```

Мы всего лишь добавили `skewY(20)` в конец атрибута `transform`:



И снова так тщательно настроенное центрирование оказалось нарушено. Исправление оставляем в качестве упражнения для читателя.

Повторим еще раз, что композиция преобразований сводится к перемножению матриц. На самом деле любое преобразование можно определить с помощью функции `matrix()`, а все остальные методы – просто частные случаи этой функции. Чтобы в полной мере овладеть преобразованиями, поинтересуйтесь в спецификации W3C по адресу http://www.w3.org/TR/SVG/coords.html#EstablishingANewUserSpace, какие матрицы соответствуют каждому из рассмотренных выше преобразований.

CSS

Каскадные таблицы стилей (CSS) появились еще в 1996 году, так что являются долгожителями в вебе, хотя широкую популярность обрели

только в ходе войны между таблицами и CSS-стилями в начале 2000-х годов. Скорее всего, вам доводилось использовать CSS для стилизации HTML. Так что этот раздел станет освежающим ветерком после странного на вид SVG-кода.

Что мне больше всего нравится в CSS, так это простота; взгляните сами:

```
selector {
  attribute: value;
}
```

Этот код описывает CSS лучше, чем я мог бы изложить словами: селекторы используются для модификации свойств путем задания значений. Хотя, конечно, к этому многое можно добавить, особенно в части каскадирования свойств при обходе дерева DOM, суть CSS отражена в этих трех строчках достаточно точно.

Мы пользуемся селекторами постоянно. Селектор – это любая строка, описывающая один или несколько элементов DOM.

Хотя селекторы способны на разные хитрые вещи, в последнее время много было написано о том, как разумно выбирать имена классов, так что по большей части CSS – просто декларативный язык. Один из таких подходов, получивший название BEM (Block Element Modifier – блок-элемент-модификатор), предлагает давать классам имена вида `.block__element-modifier`, например: `.somemodule__submit-button--large`. Это позволяет избежать конфликтов CSS и не копаться на панели **Element** в окне Инструментов разработчика, пытаясь понять, почему элемент стилизован именно так, а не иначе. Дополнительные сведения см. по адресу http://getbem.com/.

Немного освежим память:

- `path`: выбирает элементы с путем `<path>`;
- `.axis`: выбирает элементы с атрибутом `class="axis"`;
- `.axis line`: выбирает элементы с тегом `<line>`, являющиеся потомками элементов с атрибутом `class="axis"`;
- `.axis, line`: выбирает все элементы с атрибутом `class="axis"` и элементы с тегом `<line>`.

D3-выборки – подмножество CSS-селекторов, поэтому все, что можно выбрать методом `d3.selectAll`, можно описать и с помощью CSS.

В D3 есть три способа доступа к CSS:

- задать атрибут класса методом `.attr()`, этот способ чреват ошибками в случае изменения разметки;

- воспользоваться методом `.classed()`, это предпочтительный способ определения классов;
- задать стили непосредственно методом `.style()`.

Давайте ненадолго наплюем на удобочитаемость и превратим оси в психоделические видения. Добавьте в конец файла `index.css` такие строки:

```
.trippy {
  animation: wheee 3s infinite;
  fill: green !important;
}
@keyframes wheee {
  0% {stroke: red;}
  25% {stroke: yellow;}
  50% {stroke: blue;}
  75% {stroke: green;}
  100% {stroke: red;}
}
```

Модификатор `!important` используется для того, чтобы переопределить цвет заливки оси (если вы прочли предыдущее примечание, то помните, что он определяется в атрибуте `style`, а значит, имеет приоритет над CSS, если только в объявлении CSS нет модификатора `!important`), а затем мы в бесконечном цикле раз в три секунды меняем цвет обводки (который больше нигде не задан). Эта анимация определена с помощью селектора `keyframes`, определяющего различные этапы анимации. Поверьте, это не последний раз, когда вы почувствуете себя под кайфом, читая эту книгу!

Добавьте в `lib/main.js` строку, которая загрузит CSS-таблицу в HTML-файл (можно было бы воспользоваться и тегами `<link>`, но это так скучно):

```
import '../styles/index.css';
```

И измените прежний цикл рисования следующим образом:

```
axes.forEach((axis, i) =>
  chart.container.append('g')
    .data(d3.range(0, amount))
    .classed('trippy', i % 2)
    .attr('transform', &grave;translate(0,${(i * 50) + chart.margin.top})&grave;)
    .call(axis)
);
```

К чему нам эти глупости – задавать пять раз подряд одно и то же значение? С помощью функции `.classed()` мы добавляем класс

`.trippy` к каждой второй оси. Функция `.classed()` добавляет указанный класс, если второй аргумент равен `true` или отсутствует, и удаляет класс в противном случае. Теперь все числа на второй и четвертой осях меняют цвет, что выглядит очаровательно и жизнеутверждающе.

Надеюсь, вы немного оживились, потому что мы *наконец-то* подошли к концу этой ужасно скучной обзорной главы и скоро займемся вещами невероятной крутизны! А к анимации мы еще вернемся в главе 5.

Резюме

Глава получилась-таки насыщенной. Если сумели дочитать до конца, перехватите стаканчик чего-нибудь, заслужили. Если нет, тоже не переживайте – материала было много, так что никто вас не осудит, если вы попозже вернетесь и что-то перечитаете.

Мы поговорили о манипуляциях DOM и подробно рассмотрели SVG: от рисования фигур вручную до использования преобразований. И напоследок мы видели, как с помощью CSS сделать изображение симпатичным.

Все последующее зиждется на этом фундаменте, но теперь у вас есть средства нарисовать все, что сможете измыслить.

Глава 3

Геометрические примитивы в D3

В этой главе мы займемся рисованием фигур, как и в предыдущей, но будем использовать встроенные в D3 генераторы, чем немало облегчим себе жизнь. Большинство генераторов находится в микробиблиотеке `d3-shape`.

Пути

В главе 2 мы создали несколько унылых фигур, определяемых максимум четырьмя точками. И хотя по этой дороге можно зайти довольно далеко в рисовании простых диаграмм, в более сложных случаях не обойтись без путей. Элемент **path** определяет контур фигуры, которую можно обвести, залить и т. д. Он является обобщением всех остальных элементов фигур и может использоваться для рисования практически всего.

Однако минуточку: а как насчет ломаных и многоугольников? Да, это тоже многоточечные примитивы SVG, но, по существу, они почти идентичны пути.

Возможности пути в основном определяются атрибутом `d`, который задается на некотором мини-языке (в программировании такие языки называются *предметно-ориентированными*, или **DSL**), включающем среди прочего три простые команды:

- `M` – `moveto` (переместить перо в точку);
- `L` – `lineto` (провести отрезок в точку);
- `Z` – `closepath` (замкнуть путь).

Для создания пути проделайте следующие действия. Создайте в папке `lib/` новую подпапку `chapter3/` и в ней файл `index.js`. В файле

`lib/main.js` замените строку

```
import './chapter2/index';
```

такой:

```
import './chapter3/index';
```

Затем создайте новую функцию

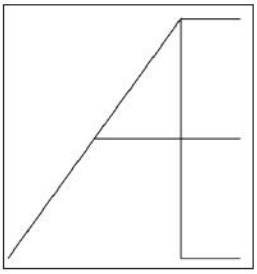
```
export function yayPaths() {
  const path = chart.container.append('path')
  .attr('d', 'M 10 500 L 300 100 L 300 500 M 300 100 l 100 0 M 155 300 l
245 0 M 300 500 l 100 0')
  .attr('stroke', 'black')
  .attr('stroke-width', 2)
  .attr('fill', 'transparent');
}
yayPaths();
```

Мы добавили в SVG-код новый элемент и задали его атрибуты. Особенно интересен атрибут `d`:

```
M 10 500 L 300 100 L 300 500 M 300 100 l 100 0 M 155 300 l 245 0 M 300 500 l
100 0
```

Разберемся, что это такое. Путь SVG всегда начинается с заглавной буквы `M`, означающей «переместить перо в эту точку» – в данном случае в точку `(10, 500)`. Затем мы проводим отрезок прямой `(L)` в точку `(300, 100)` и еще один в точку `(300, 500)`. После этого перемещаем перо в точку `(300, 100)` и проводим отрезок, используя строчную букву `l`, – это означает, что задаются относительные значения; в данном случае конечная точка отстоит на 100 пикселей вправо, т. е. перо остановится в точке `(400, 100)`. Далее перемещаем перо в точку `(155, 300)`, рисуем отрезок длиной 245 пикселей, перемещаем перо в точку `(300, 500)` и рисуем еще один отрезок длиной 100 пикселей.

Перезагрузите страницу в браузере и посмотрите, что получилось. Надо же, мои инициалы.



Но на этом возможности путей не заканчиваются. Помимо `M`, `L`, `l` есть команды для создания кривых и дуг. Но, как вы уже поняли, создавать сложные фигуры вручную чересчур утомительно. По счастью, в D3 имеются полезные функции-генераторы, которые принимают на входе JavaScript-код и преобразуют его в определение пути. Их мы далее и рассмотрим.

Сотрите код рисования дифтонга в файле `chapter3.js` и замените его таким:

```
import * as d3 from 'd3';
import chartFactory from '../common/index';
export function yayPaths() {
  const chart = chartFactory();
}

yayPaths();
```

Это новая функция, которая будет порождать диаграмму с помощью нашей фабрики; мы уже нечто подобное делали раньше. Для начала нарисуем скромную синусоиду. Нам понадобятся данные, которые мы подготовим с помощью встроенной в JavaScript функции `Math.sin`:

```
const sine = d3.range(0, 10)
  .map(k => [0.5 * k * Math.PI, Math.sin(0.5 * k * Math.PI)]);
```

Вызов `d3.range(0,10)` дает нам список целых чисел от 0 до 9. Каждое из них мы преобразуем в массив длины 2, представляющий максимум и минимум кривой. Надо полагать, вы еще не забыли, что синусоида проходит через точки `(0,0)`, `(Pi/2, 1)`, `(Pi, 0)`, `(3Pi/2, -1)` и т. д. Мы загружаем эти данные в генератор пути.

Генераторы пути – та волшебная палочка, которая творит магию D3. Это обширная тема, и мы вернемся к ней в главе 6. По существу, генератор представляет собой функцию, которая принимает некоторые данные (соединенные с элементами) и порождает определение пути на мини-языке SVG. Генератору можно сказать, как использовать наши данные. Кроме того, мы можем модифицировать его конечный результат.

Прямая линия

Для создания отрезка прямой мы используем генератор `d3.line()` и определяем функции-акцессоры `x` и `y`. Акцессоры говорят генератору, как прочитать координаты из данных.

В версии D3 v3 генераторы находились в пространстве имен `d3.svg`. В версии v4 они перемещены в пространство имен верхнего уровня и в `d3-shape`. Ошибка вида **Cannot read property 'line' of undefined** означает, что пример, который вы нашли в сети, скорее всего, рассчитан на версию 3.

Начнем с определения двух масштабов. Напомним (см. главу 1), что масштабом называется функция, отображающая область определения в область значений; подробнее мы поговорим о них в следующей главе:

```
const x = d3.scaleLinear()
  .range([
    0,
    (chart.width / 2) - (chart.margin.left + chart.margin.right),
  ])
  .domain(d3.extent(sine, d => d[0]));

const y = d3.scaleLinear()
  .range([
    (chart.height / 2) - (chart.margin.top + chart.margin.bottom),
    0,
  ])
  .domain([-1, 1]);
```

Теперь воспользуемся ими для определения генератора пути:

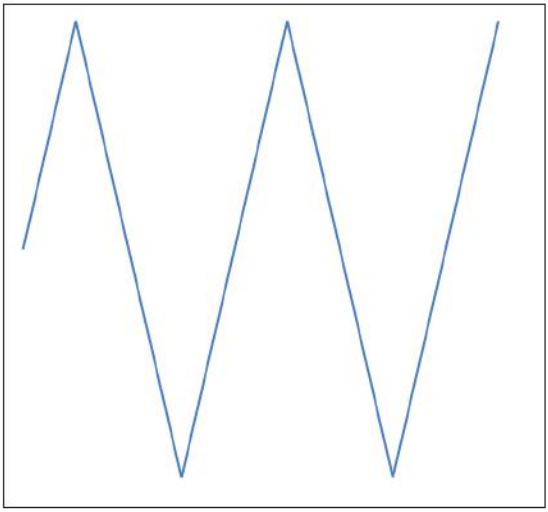
```
const line = d3.line()
  .x(d => x(d[0]))
  .y(d => y(d[1]))
```

Нужно всего лишь взять базовый генератор прямых и присоединить к нему акцессоры. Мы сообщили генератору, что он должен применять масштаб `x` к первому элементу и масштаб `y` ко второму элементу каждого массива. По умолчанию предполагается, что набор данных представляет собой коллекцию массивов, непосредственно определяющих точки, так что `d[0]` совпадает с `x`, а `d[1]` – с `y`.

Осталось только нарисовать линию:

```
const g = chart.container.append('g');
g.append('path')
  .datum(sine)
  .attr('d', line)
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 2)
  .attr('fill', 'none');
```

Этот код создает новый групповой элемент, в который добавляет путь и задает для него данные синусоиды с помощью функции `.datum()`. Использование этой функции вместо `.data()` означает, что мы можем отрисовать функцию в виде единственного элемента, а не создавать новый отрезок для каждой точки. Мы предоставили генератору возможность определить атрибут `d`. Все остальное нужно для того, чтобы путь стал видимым. График выглядит следующим образом:



Взглянув на получившийся код в окне Инструментов разработчика в Chrome, мы увидим примерно такую абракадабру:

```
<path
d="M0,185L42.83333333333333,0L85.66666666666666,184.99999999999997L128.5,37
0L171.33333333333331,185.00000000000003L214.16666666666669,0L257,184.999999
99999991L299.8333333333333,370L342.66666666666663,185.00000000000009L385.5,
0" stroke="steelblue" stroke-width="2" fill="none"></path>
```

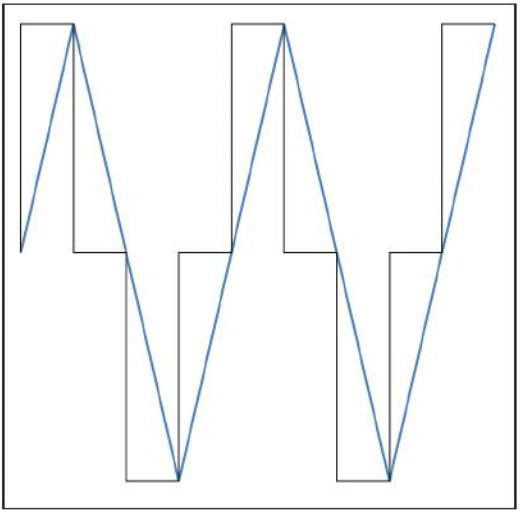
Хоть это и короче, чем наш JavaScript-код, но наш код куда понятнее, и рассуждать о нем гораздо проще.

Синусоида получилась у нас очень угловатой, совсем не такой, как в школе. Дело можно поправить с помощью интерполяции. Интерполяцией называется угадывание отсутствующих точек линии по имеющимся. По умолчанию используется линейная интерполяция, т. е. точки просто соединяются отрезками прямых. Но результат получается не ахти. Исправим это дело!

Добавьте в функцию такой код:

```
g.append('path')
  .datum(sine)
  .attr('d', line.curve(d3.curveStepBefore))
  .attr('stroke', 'black')
  .attr('stroke-width', 1)
  .attr('fill', 'none');
```

Здесь все то же самое, что и раньше, только мы добавили интерполятор `d3.curveStepBefore` и изменили стилизацию. Получилось вот что:



Мы заменили фабрику кривых в нашем генераторе одним из встроенных в D3 интерполяторов. Метод `.curve` принимает функцию интерполяции и возвращает генератор отрезков, который передается атрибуту пути `d`, так что в результате получается другая фигура.

Это прекрасный пример функционального стиля программирования, принятого в D3, мы еще не раз встретимся с ним на страницах этой книги.

В D3 имеются 18 интерполяторов, все они описаны в официальной документации по модулю `d3-shape` на странице https://github.com/d3/d3-shape/blob/master/README.md#curves. Рекомендую попробовать каждый и посмотреть, что получается.

В версии D3 v4 метод `line.interpolate` переименован в `line.curve` и вместо строки принимает функцию из пространства имен `d3` или модуля `d3-shape` в зависимости от того, какой интерполятор используется. Все имеющиеся встроенные интерполяторы перечислены на вышеупомянутой странице документации.

Область

Под областью (area) понимается закрашенная часть плоскости между двумя линиями. Вспомните комбинированную диаграмму, на которой показано пространство под кривой.

Как и в случае линий, мы создаем генератор пути и сообщаем ему, как использовать данные. Если речь идет о простой горизонтальной области, то необходимо определить один акцессор `x` и два акцессора `y`, `y0` и `y1`, потому что у области есть верх и низ. Часто акцессор `y` просто возвращает 0, т. е. снизу область ограничена осью абсцисс. Но это необязательно, D3 предоставляет средства для обработки более общего случая.

Разместим на одном рисунке несколько генераторов для сравнения. Для начала добавим еще один групповой элемент внутри того же элемента SVG:

```
const g2 = chart.container.append('g')
  .attr('transform',
    &grave;translate(${(chart.width / 2) + (chart.margin.left + chart.margin.
right)},
    ${chart.margin.top})&grave;);
```

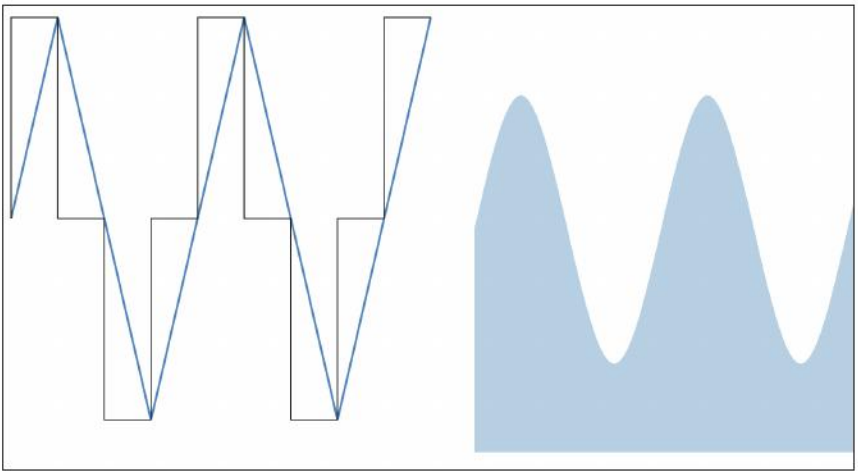
Теперь определим генератор области и нарисуем эту область:

```
const area = d3.area()
  .x(d => x(d[0]))
  .y0(chart.height / 2)
  .y1(d => y(d[1]))
  .curve(d3.curveBasis);

g2.append('path')
  .datum(sine)
  .attr('d', area)
  .attr('fill', 'steelblue')
  .attr('fill-opacity', 0.4);
```

Мы взяли генератор пути `d3.area()` и сказали, что он должен получать координаты с помощью определенных ранее масштабов `x` и `y`. В качестве интерполятора мы указали **B-сплайн**, который строит по данным гладкую кривую.

Изменение в версии D3 v4: как вы уже догадались, модуль `d3.svg.area` теперь стал просто `d3.area` и является частью микробиблиотеки `d3-shapes`.

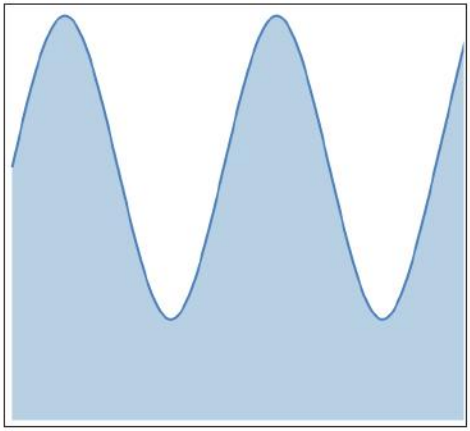
Чтобы нарисовать гладкую кривую, мы задали `y0` в качестве нижней границы области. Получилась такая аппроксимация синусоиды (закрашенная область):



Часто область используется в сочетании с линией. Мы можем либо определить для пути границу, воспользовавшись свойством `stroke`, либо с помощью генератора линий нарисовать линию, совпадающую с верхней границей области. Продемонстрируем второй способ:

```
g2.append('path')
  .datum(sine)
  .attr('d', line.curve(d3.curveBasis))
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 2)
  .attr('fill', 'none');
```

Мы воспользовались тем же генератором, что и прежде, изменив только интерполятор:



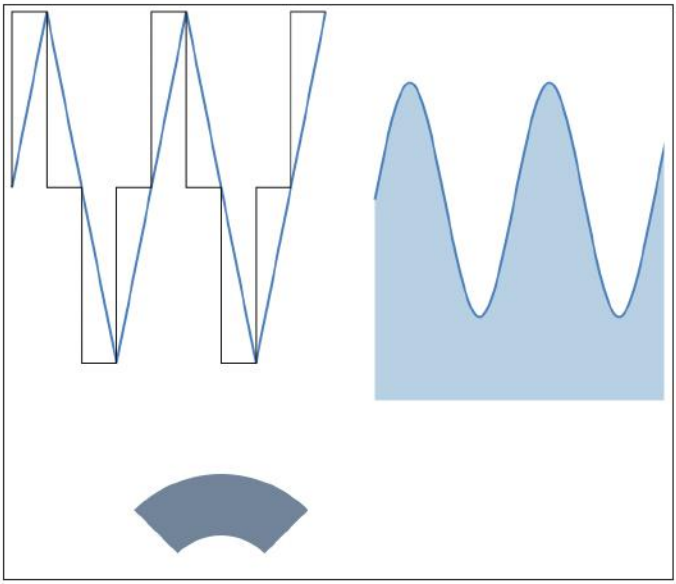
Дуга

Под дугой (arc) понимается круговой путь, заданный внутренним и внешним радиусами, а также начальным и конечным углами. Часто дуги применяются для построения секторных и кольцевых диаграмм.

Работает все так же, как и раньше: нужно только сказать генератору, как использовать данные. Единственное различие состоит в том, что на сей раз акцессоры по умолчанию ожидают получить именованные атрибуты, а не массивы с двумя значениями, как раньше:

```
const arc = d3.arc();
const g3 = chart.container.append('g')
  .attr('transform',
      &grave;translate(${chart.margin.left + chart.margin.right},
      ${(chart.height / 2) +
        (chart.margin.top + chart.margin.bottom)})&grave;);

g3.append('path')
  .attr('d',
    arc({
      outerRadius: 100,
      innerRadius: 50,
      startAngle: -Math.PI * 0.25,
      endAngle: Math.PI * 0.25
    }))
  .attr('transform', 'translate(150, 150)')
  .attr('fill', 'lightslategrey');
```

В этот раз мы смогли обойтись генератором по умолчанию `d3.arc()`. Вместо передачи данных мы вычислили углы вручную и сдвинули дугу ближе к центру.



Завораживающая картинка!

Хотя обычно в SVG используются градусы, начальный и конечный углы задаются в радианах. Нулевой угол соответствует 12 часам, отрицательным считается направление против часовой стрелки, а положительным – по часовой стрелке. При каждом обороте на `2Pi` мы возвращаемся к нулю.

И снова модуль `d3.svg.arc` из версии D3 v3 стал называться `d3.arc` в версии v4 и входит в состав пакета `d3-shapes`. Полагаю, идею вы уловили – все, что находилось в `d3.svg`, переехало в пространство имен верхнего уровня D3.

Символ

Иногда в процессе визуализации данных нам нужен простой способ пометить точки. На помощь приходят символы – миниатюрные значки, используемые как альтернатива кружочкам.

Генератор `d3.symbol()` (из пакета `d3-symbol`) принимает акцессоры типа и размера, а позиционирование оставляет нам. Добавим на нашу комбинированную диаграмму символы, отмечающие, где функция обращается в нуль.

Как всегда, начнем с генератора пути:

```
const symbols = d3.symbol()
  .type(d => (d[1] > 0 ? d3.symbolTriangle : d3.symbolDiamond))
  .size((d, i) => (i % 2 ? 0 : 64));
```

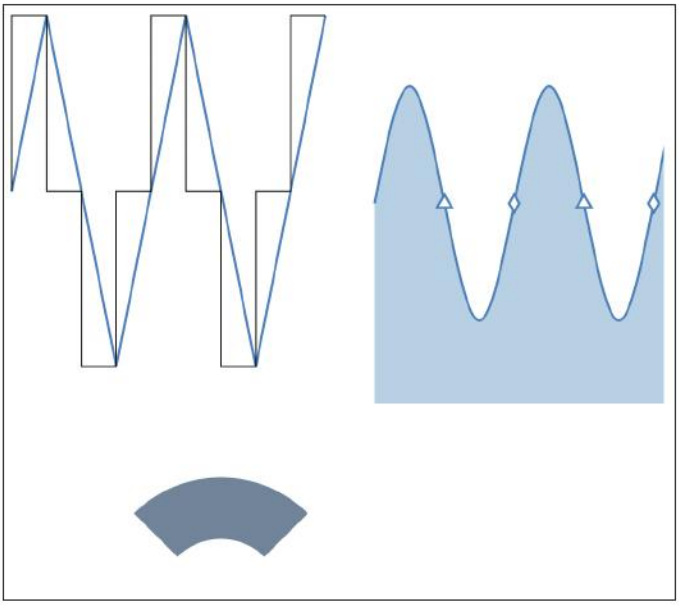
Мы передали генератору `d3.symbol()` акцессор типа, который говорит, что нужно рисовать треугольник, если координата `y` положительна, и ромб в противном случае. Это работает, потому что наши данные для построения синусоиды неидеальны из-за ограниченной точности представления чисел с плавающей точкой, в частности `Math.PI`; мы получаем числа, очень близкие к нулю, знак которых зависит от того, будет ли `Math.sin` чуть больше или чуть меньше нуля в точках, где настоящий синус равен 0.

Акцессор размера сообщает функции `symbol()`, какую площадь должен занимать каждый символ. Поскольку близка к нулю каждая вторая точка данных, остальные мы скрываем, делая площадь равной 0.

Теперь можно нарисовать символы:

```
g2.selectAll('path')
  .data(sine)
  .enter()
  .append('path')
  .attr('d', symbols)
  .attr('stroke', 'steelblue')
  .attr('stroke-width', 2)
  .attr('fill', 'white')
  .attr('transform', d => &grave;translate(${x(d[0])},${y(d[1])})&grave;);
```

Здесь мы перебираем данные, добавляем новый путь для каждой точки и преобразуем его в символ, сдвинутый в нужную позицию. Результат выглядит так:



Посмотреть, какие еще существуют символы, можно, заглянув в код `d3.symbols` или на страницу `https://github.com/d3/d3-shape/blob/master/README.md#symbols`.

Хорда и лента

Хорды чаще всего применяются для отображения связей между групповыми элементами, расположенными по кругу. Для этого используются квадратичные кривые Безье, чтобы создать замкнутую фигуру, соединяющую две точки на дуге. Иными словами, простая хорда выглядит, как разбойничий ус:

По сравнению с версией D3 v3, хорды претерпели значительные изменения. Теперь для них выделен отдельный пакет (`d3-chord`), а создание экземпляра `d3.chord` приводит к созданию нового макета, тогда как сами фигуры рисуются с помощью функции `d3.ribbon`. Сейчас мы просто хотим нарисовать ленту, поэтому не будем обсуждать хордовый макет во всей полноте. Мы передаем `d3.ribbon()` для создания нового генератора лент, который и рисует путь:

```
g3.append('g')
  .selectAll('path')
  .data([{
    source: {
      radius: 50,
      startAngle: -Math.PI * 0.30,
      endAngle: -Math.PI * 0.20,
    },
    target: {
      radius: 50,
      startAngle: Math.PI * 0.30,
      endAngle: Math.PI * 0.30,
    },
  }])
    .enter()
    .append('path')
    .attr('d', d3.ribbon());
```

Здесь мы добавляем новый групповой элемент, определяем набор данных, содержащий один элемент, и добавляем путь, в котором значением атрибута `d` является генератор `d3.ribbon()` без параметров.

Сами данные прекрасно работают с акцессорами по умолчанию, поэтому мы можем препоручить их попечению `d3.ribbon`. Если понадобится изменить положение концов хорды, то можно будет воспользоваться функциями `ribbon.source()` (начальная точка) и `ribbon.target()` (конечная точка). Обеим передаются дополнительные акцессоры, задающие радиус дуги, начальный угол `startAngle` и конечный угол `endAngle`. Как в случае генератора дуг, углы задаются в радианах. Давайте подготовим данные и нарисуем хордовую диаграмму:

```
const data = d3.zip(d3.range(0, 12), d3.shuffle(d3.range(0, 12)));
const colors = ['linen', 'lightsteelblue', 'lightcyan', 'lavender',
'honeydew', 'gainsboro'];
```

Ничего интересного. Мы определили два массива чисел, один из них перетасовали и скомпоновали из них массив пар. Детали мы рассмотрим в следующей главе, а пока просто отметим, что `d3.range` воз-

вращает массив чисел в заданном диапазоне, `d3.shuffle` случайным образом переставляет элементы массива, а `d3.zip` создает массив массивов. Затем мы определили несколько цветов.

```
const ribbon = d3.ribbon()
  .source(d => d[0])
  .target(d => d[1])
  .radius(150)
  .startAngle(d => -2 * Math.PI * (1 / data.length) * d)
  .endAngle(d => -2 * Math.PI * (1 / data.length) * ((d - 1) % data.length));
```

В совокупности эти массивы определяют генератор. Мы хотим разделить окружность на участки равной длины и соединить случайные пары точек хордами.

Акцессоры `.source()` и `.target()` говорят, что первый элемент пары является начальной точкой, а второй – конечной. Что касается `startAngle`, вспомните, что угол всей окружности равен `2Pi`, а мы делим его на число участков. Наконец, чтобы выбрать участок, мы умножаем этот угол на номер участка. Акцессор `endAngle` почти такой же, только сдвинут на один участок.

```
g3.append('g')
  .attr('transform', 'translate(300, 200)')
  .selectAll('path')
  .data(data)
  .enter()
  .append('path')
  .attr('d', ribbon)
  .attr('fill', (d, i) => colors[i % colors.length])
  .attr('stroke', (d, i) => colors[(i + 1) % colors.length]);
```

Чтобы нарисовать диаграмму, мы создаем новую группу, соединяем набор данных с элементами и добавляем путь для каждой точки. Затем используем написанный выше генератор `ribbon` для рисования хорд, соединяющих начальные точки с конечными, и раскрашиваем в веселенькие цвета.

Результат изменяется при каждом обновлении страницы и выглядит примерно так:

Мы вернемся к обсуждению хордового макета в главе 6.

Здесь раньше был раздел о пакете `d3.svg.diagonal`, но после выхода версии D3 v4 он был удален. Скажем лишь, что рисование отдельных диагоналей никогда не было ни простым, ни особенно полезным и что вся эта функциональность теперь обернута иерархическими макетами и включена в состав модуля `d3-hierarchy`. Мы расскажем о нем в главе 6.

Оси

Теперь, научившись рисовать различные фигуры, воспользуемся этими знаниями для создания чего-то действительно полезного, а именно осей графика из прямых линий и текста. Делать это вручную утомительно, поэтому D3, жалея нас, предлагает генераторы осей. Генератор берет на себя рисование прямой, расстановку делений, нанесение меток через равные интервалы и т. д.

Ось в D3 – это просто комбинация генераторов путей, сконфигурированных для решения конкретной задачи. Чтобы нарисовать простую линейную ось, достаточно создать масштаб и сказать оси, что им нужно воспользоваться.

Работая с D3, следует помнить, что **масштаб** – это функция, отображающая входную область определения на выходную область значений, тогда как **ось** – это визуальное представление масштаба. Кроме того, поскольку мы в дальнейшем будем пользоваться осями часто, повторим еще раз запоминалку Скотта Мюррея из главы 1: «Входная! Область определения! Выходная! Область значений!» Честно говоря, она никогда не наскучивает.

Чтобы настроить ось под свои требования, мы могли бы задать нужное число делений и метки. Можно даже создавать круговые оси. Очистим экран, для чего удалим строку `yayPaths();` в конце файла `chapter3/index.js` и заменим ее таким кодом:

```
export function axisDemos() {
  const chart = chartFactory({
    margin: { top: 30, bottom: 10, left: 50, right: 50 },
  });

  const amount = 200;
}
axisDemos();
```

Здесь мы заменяем поля по умолчанию более широкими, чтобы их было лучше видно. Кроме того, мы определили константу, задающую максимальное значение. И еще нам понадобится линейный масштаб:

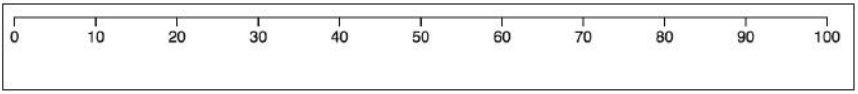
```
const x = d3.scaleLinear()
  .domain([0, amount])
  .range([
    0,
    chart.width - chart.margin.right - chart.margin.left - 20,
]);
```

Максимальное выходное значение задано равным ширине диаграммы (которая, как вы помните, совпадает с шириной всего окна) за вычетов двух 10-пиксельных полей и еще 20 пикселей, поскольку горизонтальной оси нужно примерно столько места для размещения справа последней трехзначной метки.

Ось будет пользоваться следующей функцией для трансляции данных (область определения) в координаты (область значений):

```
const axis = d3.axisBottom()
  .scale(x);

chart.container.append('g')
  .data(d3.range(0, amount))
  .call(axis);
```

Мы сообщили генератору `d3.axis()` о том, что нужно использовать наш масштаб `x`. Затем мы создали новый групповой элемент, соединили данные и вызвали `axis`. Важно вызывать генератор оси сразу для всех данных, чтобы он мог добавить собственный элемент. Теперь ось выглядит так:



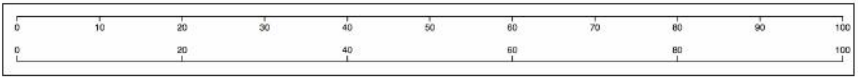
В версии D3 v3 стилизовать оси приходилось самостоятельно, чтобы они выглядели хотя бы минимально пристойно. В версию v4 включен разумный набор умолчаний для осей, так что одной головной болью стало меньше. Эти умолчания задаются в атрибуте `style`, поэтому если вы захотите переопределить их в CSS, не забудьте добавить модификатор `!important`.

Изменив переменную `amount`, вы увидите, что у оси хватает «интеллекта» для выбора числа делений, соответствующего имеющемуся месту. Это окажется очень полезно, когда впоследствии мы займемся анимацией.

Посмотрим, как различные параметры отражаются на внешнем виде осей. Мы построим в цикле несколько осей с одинаковыми данными. Замените обращение `chart.container.append('g')` таким кодом:

```
const axes = [
  d3.axisBottom().scale(x),
  d3.axisTop().scale(x).ticks(5),
];

axes.forEach((axis, i) =>
  chart.container.append('g')
    .data(d3.range(0, amount))
    .attr('transform',
      &grave;translate(0,${(i * 50) + chart.margin.top})&grave;)
    .call(axis)
);
```

Пока ограничимся двумя осями: первая – стандартная, а на второй будет ровно 5 делений:

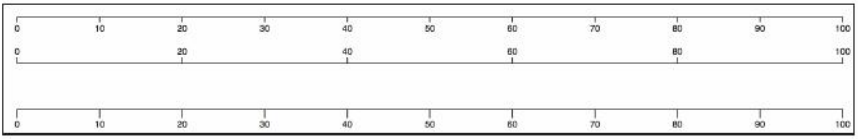


Работает! Генератор оси вычислил, какие деления лучше оставить, и разместил для них метки без всякого вмешательства с нашей стороны.

Добавим в массив еще оси и посмотрим, что получится:

```
d3.axisBottom().scale(x).tickSize(10, 5, 10),
```

Функция `.tickSize()` уменьшает размер мелких делений. У нее три аргумента: `major` (крупные деления), `minor` (мелкие деления) и `end` (концевые деления).



И последний фокус: определим нестандартные деления и поместим их над осью. Добавим в массив еще одну ось:

```
d3.axisTop().scale(x).tickValues([0, 20, 50, 70, 100])
  .tickFormat((d, i) => ['a', 'e', 'i', 'o', 'u'][i]),
```

Здесь функция `.tickValues()` точно определяет, в каких местах должны находиться деления, а функция `.tickFormat()` описывает соответствующие им метки. Функция `axisTop()` размещает деления и метки над осью.

В версии D3 v3 необходимо было задавать ориентацию оси методом `axis.orient()` на этапе ее создания. В версии v4 ориентация оси задается с помощью функций `d3.axisTop`, `d3.axisBottom`, `d3.axisLeft` и `d3.axisRight`. Тем самым подчеркивается тот факт, что после начальной отрисовки ориентацию изменить нельзя.

Резюме

Ну что ж, все идет неплохо! Мы начинаем по-настоящему использовать D3 и рисовать вещи посложнее круга или прямоугольника. То ли еще будет в следующих главах!

Сначала мы нарисовали путь SVG вручную с применением предметно-ориентированного языка рисования. Затем нарисовали несколько прямых и кривых, области, оси и – самое главное – проделали все это с помощью генераторов путей из пакета `d3-shape`.

Очень скоро мы перейдем к построению действительно интересных диаграмм, но прежде обсудим, как преобразовать данные в необходимый для работы формат. Нас ждет глава 4 «Извлечение пользы из данных».

Глава 4

Извлечение пользы из данных

При создании визуализаций для веба часто бывает, что исходный формат данных отличается от того, который нужен для использования D3. Мы обсудим, как сделать набор данных пригодным для D3 и обычных программ на JavaScript.

Сначала ненадолго погрузимся в функциональное программирование, чтобы иметь общую основу для беседы. Многое из сказанного будет очевидно тем, кому доводилось писать на Haskell, Scala, Lisp или даже на JavaScript в функциональном стиле. В настоящее время функциональное программирование – горячая тема среди разработчиков на JavaScript. И тому есть веские причины – такой код проще читать, он развивает правильные привычки, например не изменять значений переменных, и в нем находит применение одна из лучших языковых особенностей JavaScript – *полноправные* функции. Скоро мы узнаем, что это значит.

Затем мы перейдем к загрузке внешних данных разными способами, поближе познакомимся с масштабами и закончим рассмотрением временных и географических данных.

Функциональный подход к данным

Я уже говорил, что библиотека D3 спроектирована в *функциональном* стиле, т. е. в ней используются идиомы, которые JavaScript заимствовал из функционального программирования. И хотя на D3 можно писать программы в классическом объектно-ориентированном стиле, жить нам будет гораздо проще, если мы будем смотреть на код и данные с функциональной точки зрения.

Хорошая новость состоит в том, что JavaScript – почти функциональный язык; в нем достаточно средств для получения преимуществ от функционального подхода, и вместе с тем он предоставляет свободу для программирования в императивном или объектно-ориентированном стиле. Но есть и плохая новость – в отличие от настоящих функциональных языков, среда исполнения не дает никаких гарантий по поводу кода.

В главе 9 мы рассмотрим язык TypeScript, который позволяет компилировать JavaScript со статическими типами, и программу Tern.JS, предназначенную для анализа кода, с тем чтобы сделать его более пригодным для инструментальных средств. Все это позволит более уверенно говорить о ходе обработки данных в процессе визуализации.

В этом разделе мы обсудим основы функционального стиля кодирования и преобразование данных к удобному для работы формату. Если вас заинтересует настоящее функциональное программирование, рекомендую обратиться к книге «Learn You a Haskell for Great Good», которую можно бесплатно почитать и скачать с сайта http://learnyouahaskell.com/.

Идея функционального программирования проста: вычисления могут зависеть только от аргументов функции. Несмотря на простоту, у этой идеи далеко идущие последствия. И главное из них – то, что мы больше не зависим от состояния, что, в свою очередь, обеспечивает прозрачность ссылок. Это значит, что функция, вызванная с одинаковыми параметрами, всегда дает один и тот же результат.

На практике это означает, что мы одновременно проектируем код и поток данных, т. е. получаем данные на входе, применяем к ним последовательность функций и на выходе получаем результат.

Важна также неизменяемость, благодаря которой у функций нет побочных эффектов и данные не изменяются при прохождении через программу. В главе 2 мы использовали функцию `Array.from()` для получения копии аргумента, чтобы функция не могла изменить переданных ей данных. Обычно мы также выполняем присваивание начальных значений переменным, определенным с ключевым словом `const`, так что попытка присвоить другое значение приведет к ошибке. Мы и дальше будем создавать копии переменных, хотя в основном пользуемся концепцией неизменяемости в ES6 в той мере, в какой она защищает от повторного присваивания с помощью ключевого слова `const`. У нас также сохраняется возможность расширять объ-

екты и взаимодействовать с массивами такими методами, как `Array.prototype.pop()` и `Array.prototype.unshift()`.

В примерах ниже я употребляю полные имена встроенных методов, чтобы отличить их от методов `filter` и `map` d3-выборки. Так, `Array.prototype.map` – это метод объекта `prototype` встроенного примитива `Array`.

А кстати, что такое прототип? Язык JavaScript основан на прототипах, т. е. практически все в нем является объектами, наследующими от других объектов – прототипов. Все массивы являются потомками объекта `Array` и, следовательно, наследуют методы его прототипа, в частности `.map()`, `.reduce()` и `.filter()`. Сам прототип `Array` наследует другим прототипам, и эта цепочка простирается до `Object.prototype`. D3-выборки также являются потомками прототипа `Array`, но D3 подменяет некоторые его функции, в т. ч. `.filter()` и `.sort()`, собственными вариантами, а также добавляет еще ряд полезных методов, например `.each`, или полностью `d3.selection.prototype.each` (или просто `selection.each`, как этот метод называется в документации). Даже классы в ES2015, для которых используется совершенно другая модель наследования, пришедшая из объектно-ориентированного программирования, в конечном счете основаны на прототипах.

Мы уже встречались с функциональным подходом в предыдущих примерах, особенно в главе 2. Наш набор данных и в начале, и в конце представлял собой массив значений. Мы выполняли некоторые действия для каждого элемента и, решая, что делать, рассматривали только текущий элемент. Мы также хранили текущий индекс, чтобы можно было немного сжульничать в императивном духе – заглянуть вперед и назад в процессе обработки потока.

Хотя далее в книге я придерживаюсь близкого к функциональному стиля, здесь нет возможности как следует объяснить все интересные и важные нюансы современной разработки на JavaScript. Рекомендую серию статей Эрика Эллиота на эту тему, их можно найти по адресу https://medium.com/javascript-scene/the-rise-and-fall-and-rise-of-functional-programming-composable-software-c2d91b424c8c.

Встроенные функции массива

В JavaScript встроен целый ряд функций для работы с массивами. Мы обратим внимание на те, что по природе своей функциональны: методы итерирования.

Операции отображения, редукции и фильтрации (`Array.prototype.map`, `Array.prototype.reduce` и `Array.prototype.filter`) крайне полезны

для изменения модели данных; на самом деле map-reduce – один из основных паттернов в базах данных NoSQL, а возможность распараллеливания этих операций открывает путь к высокому уровню масштабируемости.

Рассмотрим встроенные функции подробнее. Стоит отметить, что все они *неизменяющие*, т. е. возвращают копию входного массива, оставляя оригинал неизменным.

- `Array.prototype.map` применяет указанную функцию к каждому элементу массива и возвращает массив результатов:

```
> [1,2,3,4].map(d => d+1)
[ 2, 3, 4, 5 ]
```

- `Array.prototype.reduce` применяет функцию-комбинатор к начальному значению и к каждому элементу массива, сворачивая массив в одно значение:

```
> [1,2,3,4].reduce((acc, curr) => acc + curr, 0)
10
```

- `Array.prototype.filter` перебирает весь массив и оставляет только те элементы, для которых предикат возвращает `true`:

```
> [1,2,3,4].filter(d => d%2)
[ 1, 3 ]
```

- Функции `Array.prototype.every` и `Array.prototype.some` возвращают `true`, если соответственно все или хотя бы один элемент массива равен `true`:

```
// Все элементы нечетные?
[1,3,5,7,9].every(elem => elem % 2); // True
[1,2,5,7,9].every(elem => elem % 2); // False

// Есть ли хотя бы один нечетный элемент?
[1,3,5,7,9].some(elem => elem % 2); // True
[1,2,5,7,9].some(elem => elem % 2); // True
[0,2,4,6,8].some(elem => elem % 2); // False
```

Также полезна – особенно в контексте функционального программирования – функция `Array.from`, которая создает копию массива.

Иногда возникает соблазн использовать `Array.prototype.forEach` вместо `Array.prototype.map`, потому что `.forEach` воздействует на исходный массив, не создавая копии. Но поскольку мы стараемся быть верны функциональному стилю программирования, то `.forEach` будем использовать для применения некоторой логики к элементам массива, но не для их изменения. Функции `Array.prototype.map`,

`Array.prototype.reduce` и `Array.prototype.filter` можно применять для преобразования формата данных без изменения самих данных. Это важно, т. к. существенно упрощается отладка, скажем, выяснение вопроса о том, почему некоторое значение выглядит не так, как должно. Одна из целей функционального программирования – написание *идемпотентного* кода, не имеющего *побочных эффектов*, или, иначе говоря, не оказывающего влияния на состояние программы.

Некоторые из этих функций появились в JavaScript сравнительно недавно, но `.map` и `.filter` существуют с версии JavaScript 1.7, а `.reduce` – с версии 1.8. В недобрые старые времена приходилось использовать `es6-shim` или что-то вроде библиотеки `Underscore.js`, если хотелось ими воспользоваться, но поскольку теперь перед нами маячит светлое будущее ES2017, Babel добавляет все необходимое в процессе трансляции нашего кода.

Функции для работы с данными в D3

В состав D3 входит ряд вспомогательных функций для работы с массивами. По большей части они предназначены для обработки данных: вычисления средних, упорядочения, разбиения массива на две части и т. д. В версии D3 v4 они находятся в пакете `d3-array`.

Разделы документации по D3, относящиеся к `d3-array`, определенно принадлежат к числу наиболее полезных. Мы рассмотрим несколько функций из этого пакета, но их гораздо больше, чем можно было бы охватить в одной главе. Иногда во время разработки я открываю в одной из вкладок файл README для пакета `d3-array`, поскольку это отличный справочник по сигнатурам всех функций. Познакомьтесь с ним, перейдя по адресу https://github.com/d3/d3-array.

Ниже перечислены наиболее полезные функции в пакете `d3-array`; большинство из них принимает функцию-акцессор, которая служит для задания свойства элементов массива, используемого в вычислениях.

- `d3.min` и `d3.max` возвращают соответственно наименьшее и наибольшее значения в массиве. `d3.extent` возвращает массив, содержащий оба этих значения.
- `d3.sum`, `d3.mean` и `d3.median` возвращают сумму, среднее и медиану элементов массива. К вычислению среднего и медианы следует подходить ответственно: отображение среднего массива с экстремальными значениями на обоих концах может исказить истинную картину распределения данных. Мы еще вернемся к этому вопросу в главе 10.

- `d3.ascending` и `d3.descending` – предопределенные функции сравнения, позволяющие нам не запоминать, что из чего вычитать при сортировке. Эти функции также гарантируют, что числа будут отсортированы в естественном, а не лексикографическом порядке (последнее является поведением `Array.prototype.sort` по умолчанию, если сортируются числа, введенные как строки; мы еще обсудим этот вопрос, когда будем говорить о статической типизации ниже).
- С функцией `d3.range` мы уже несколько раз встречались. Она получает два целых числа и возвращает массив чисел между ними, включающий первое число и не включающий последнего. Можно задать еще и третий аргумент – шаг между соседними числами, по умолчанию он равен 1. Отметим, что эту функцию можно применять и к нецелым числам, но с ограниченной точностью; детали см. в документации.

В пакете `d3-array` имеется также функция `d3.histogram`, которой мы будем пользоваться в главе 7 для построения гистограмм.

Управление объектами с помощью пакета d3-collection

Я не стану много говорить о пакете `d3-collection`, потому что Babel предоставляет такие объекты, как Map и Set в составе платформенного JavaScript API. Если вы работаете со старым браузером и не пользуетесь Babel для преобразования кода, то `d3-collection` – хороший способ получить в свое распоряжение некоторые современные идиомы без полифилов. Но нельзя не упомянуть об одной чрезвычайно полезной функции, `d3.nest`. Это аналог результата NoSQL-запроса, в котором документы агрегируются в группы.

Допустим, мы загрузили такие данные из CSV-документа:

```
const sensorData = [
  {location: "a", status: "normal", average: "100", date: "2016-11-01"},
  {location: "a", status: "normal", average: "200", date: "2016-11-02"},
  {location: "a", status: "normal", average: "300", date: "2016-11-03"},
  [...]
  {location: "b", status: "normal", average: "400", date: "2016-11-01"},
  {location: "b", status: "alarm", average: "500", date: "2016-11-02"},
  {location: "b", status: "alarm", average: "600", date: "2016-11-03"},
  [...]
];
```

Эта таблица могла быть результатом выгрузки базы данных MySQL, первая строка в ней является заголовком, описывающим

поля: местоположение, среднее значение некоторого датчика и дата снятия показаний. Можно было бы применить несколько сложных обращений к `Map.prototype.reduce` или просто вызвать `nest`, а затем `rollup`:

```
d3.nest()
  .key(d => d.location)
  .rollup(d => d3.mean(d, v => v.average))
  .map(sensorData);

// >> {a: 200, b: 500}
```

Что здесь происходит? Сначала мы создаем отображение – объект, который сопоставляет одно или несколько значений ключу. В данном случае мы решили группировать по местоположению. Затем мы сворачиваем значения, т. е. вызываем функцию, которая присваивает значение ключу – в данном случае вычисляет среднее всех показаний за день (при каждом обращении в `d` находится массив, содержащий все значения, ассоциированные с ключом).

Функции `d3.nest` и `nest.rollup` позволяют делать много интересного, но у меня не хватит места обо всем рассказать. Очень хороший обзор способов использования пакета `d3-collection` есть в пособии на странице http://learnjsdata.com/group_data.html.

А как насчет множеств и отображений ES6?

Множества и отображения – два новых примитива, появившихся в ES2015. Оба контейнера допускают итерирование, т. е. их можно обойти в цикле, как если бы это были массивы.

Зачем они нужны? В основном чтобы лучше структурировать доступ к данным. Они мало чем отличаются от аналогичных примитивов в других языках. Но в этой книге мы не будем их использовать, потому что для d3-выборок они не нужны, проще работать с массивами. В пакете `d3-collection` есть свои варианты объектов `Map` и `Set`, но они применяются главным образом для работы с массивами данных в `d3.nest` и в конечном итоге являются просто полифилами для браузеров, не поддерживающих их на внутреннем уровне (например, Internet Explorer). И что это дает, коль скоро Babel и так реализует Map и Set? Да почти ничего. При работе с D3 все данные так или иначе оказываются в массивах.

Когда-то возникало желание «поженить» объекты `Map` и `Set` из ES6 с d3-выборками, но этот вопрос висит открытым с 2015 года, а значит, никакого движения в этом направлении не наблюдается. Впрочем,

заглядывайте на страницу https://github.com/d3/d3/issues/2584, поскольку положение может измениться, особенно когда поддержка Map и Set в браузерах улучшится.

Масштабы

Мы уже много раз встречались с масштабами и даже запоминалку сочинили. Ну-ка, посмотрим, как вы ее запомнили.

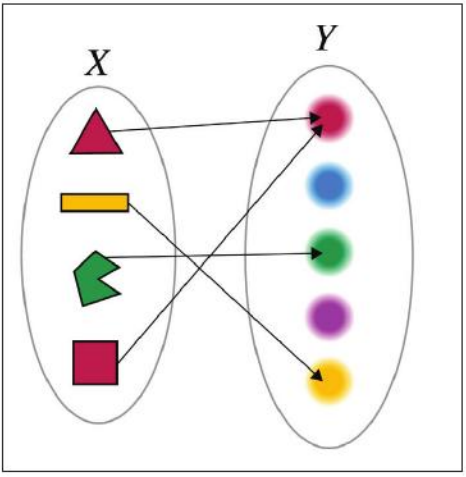
```
ВИКТОРИНА:
ВХОДНАЯ!: [ ] ОБЛАСТЬ ОПРЕДЕЛЕНИЯ! [ ] НЕ ОБЛАСТЬ ОПРЕДЕЛЕНИЯ!
ВЫХОДНАЯ!: [ ] ОБЛАСТЬ ЗНАЧЕНИЙ! [ ] НЕ ОБЛАСТЬ ЗНАЧЕНИЙ!
```

Если вы ответили ВХОДНАЯ! = ОБЛАСТЬ ОПРЕДЕЛЕНИЯ! и ВЫХОДНАЯ! = ОБЛАСТЬ ЗНАЧЕНИЙ!, значит, все отлично!

Масштабы позволяют избежать математики. Они делают код короче, понятнее и надежнее, поскольку элементарные математические ошибки труднее всего отлаживать.

Еще раз повторю то, что не устаю долбить, начиная с главы 1: область определения функции – это ее входные аргументы, а область значений – то, что она возвращает.

Следующий рисунок взят из Википедии:



Здесь X – область определения, Y – область значений, а стрелки обозначают функцию. Чтобы реализовать это вручную, нужно написать довольно длинный код:

```
let shape_color = shape => {
  if (shape == 'triangle') {
    return 'red';
  } else if (shape == 'line') {
    return 'yellow';
  } else if (shape == 'pacman') {
    return 'green';
  } else if (shape == 'square') {
    return 'red';
  }
};
```

То же самое можно сделать с помощью словаря, но функция `d3.scale` дает более элегантный и гибкий способ:

```
const scale = d3.scaleOrdinal()
  .domain(['triangle', 'line', 'pacman', 'square'])
  .range(['red', 'yellow', 'green', 'red']);
```

Куда как лучше!

Масштабы бывают трех типов: у порядковых область определения дискретная, у количественных – непрерывная, а у временных – непрерывная и связанная со временем. Все масштабы находятся в пакете `d3-scale`, это я говорю на случай, если вы предпочитаете использовать микробиблиотеки или хотите найти документацию.

Порядковые масштабы

Порядковые масштабы самые простые, по существу, это словарь, в котором ключи – область определения, а значения – область значений.

В примере выше мы определили порядковый масштаб, явно задав область определения и область значений. Если не задать область определения, то библиотека выведет ее самостоятельно, исходя из способа использования, но результаты могут получиться неожиданными.

Если в порядковом масштабе область значений меньше области определения, то повторяются предыдущие значения. Мы получили бы тот же самый результат, если бы в качестве области значений задали просто `['red', 'yellow', 'green']`.

Потренируемся. Создадим полосные и точечные масштабы – порядковые масштабы с дополнительными прибамбасами. Еще потребуется цветовой повторяющийся масштаб. Но для начала нужно место, где мы могли бы разместить наши масштабы. Создадим непосредственно вызываемое функциональное выражение (Immediately

Invoked Function Expression – IIFE) `scalesDemo` в файле `chapter4/index.js`:

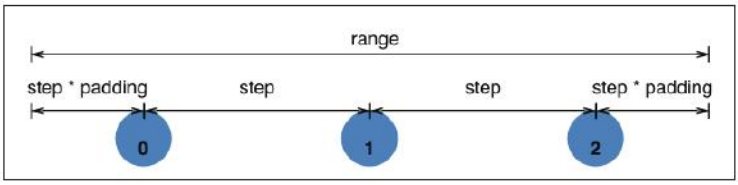
```
const scalesDemo = (enabled => {
  if (enabled) { // главный блок, здесь будет код }
})(true);
```

Затем определим все три масштаба и сгенерируем данные:

```
(function ordinalScales() {
  const data = d3.range(30);
  const colors = d3.scaleOrdinal(d3.schemeCategory10);
  const points = d3.scalePoint()
    .domain(data)
    .range([0, chart.height])
    .padding(1.0);
  const bands = d3.scaleBand()
    .domain(data)
    .range([0, chart.width])
    .padding(0.1);
}());
```

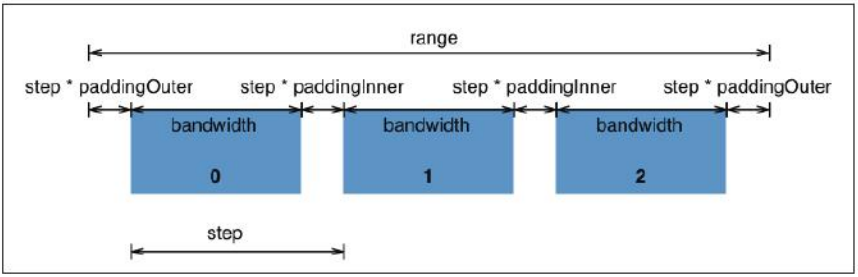
Данные – просто список чисел до 30, а цветовой масштаб взят из главы 2. В версии D3 v3 это был предопределенный порядковый масштаб с неопределенной областью определения и областью значений, содержащей 10 цветов, но теперь это просто массив, содержащий те же самые цвета. Чтобы получить из него порядковый масштаб, нужно выполнить присваивание, как мы и поступили в примере выше.

Затем мы определили два масштаба, разделяющих наш рисунок на две равные части. В масштабе `points` используется функция `.scalePoints()`, которая распределяет 30 точек по высоте рисунка через равные интервалы. С помощью метода `.padding()` мы задали коэффициент заполнения 1.0, т. е. расстояние от крайних точек до границы равно половине расстояния между точками. Расстояние от крайних точек до границы вычисляется по формуле `point_distance*padding/2`:



Эта картинка, взятая из документации по пакету `d3-scale`, объясняет, как вычисляются точечные масштабы.

Затем мы с помощью функции `.scaleBands()` делим всю ширину на 30 равных полос, промежуток между которыми определяется коэффициентом заполнения (0.1). Мы задали расстояние между полосами, `step*padding`, а если бы присутствовал третий аргумент, то можно было бы определить расстояние между крайними полосами и границей, `step*outerPadding`.



На этом рисунке, также взятом из документации по пакету `d3-scale`, показаны факторы, влияющие на вычисление полосного масштаба.

Со времен версии D3 v3 API претерпел довольно серьезные изменения. Методы `rangeBands()` и `rangePoints()` раньше были присоединены к порядковому масштабу, а теперь являются самостоятельными масштабами. Дополнительные сведения см. по адресу https://github.com/d3/d3/blob/master/CHANGES.md#scales-d3-scale.

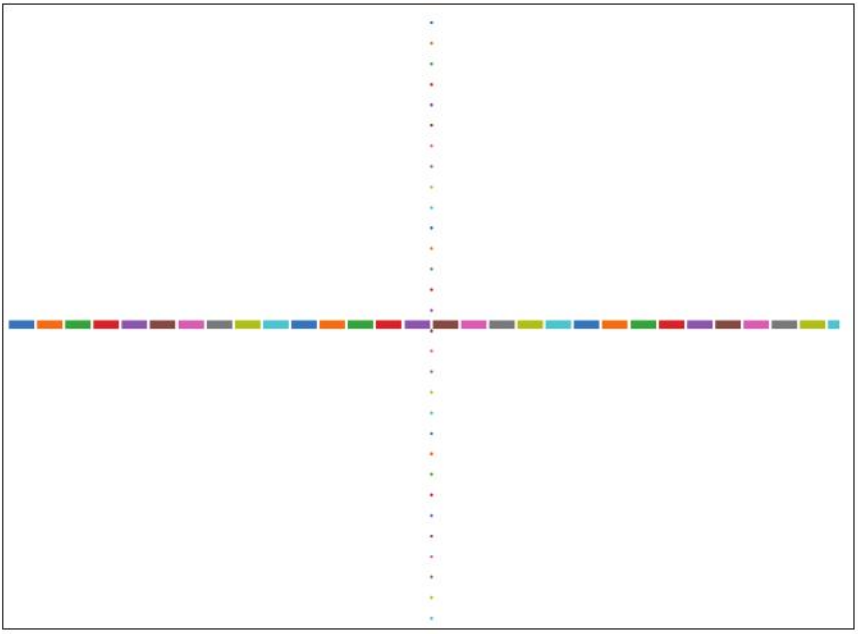
Мы воспользуемся кодом, знакомым по главе 2, чтобы нарисовать две прямые с такими масштабами. Добавьте следующий код в наше IIFE:

```
chart.container.selectAll('path')
  .data(data)
  .enter()
  .append('path')
  .attr('d', d3.symbol()
    .type(d3.symbolCircle)
    .size(10)
  )
  .attr('transform', d => &grave;translate(${(chart.width / 2)},
${points(d)})&grave;)
  .style('fill', d => colors(d));

chart.container.selectAll('rect')
  .data(data)
  .enter()
```

```
  .append('rect')
  .attr('x', d => bands(d))
  .attr('y', chart.height / 2)
  .attr('width', bands.bandwidth)
  .attr('height', 10)
  .style('fill', d => colors(d));
```

Чтобы получить позиции каждой точки или прямоугольника, мы вызываем масштабы как функции, а ширину прямоугольника получаем от функции `bands.rangeBand()`.

Получается такая картинка:



Применяя различные цветовые схемы, можно получить разные эффекты. Замените код, начиная со строки `chart.container.selectAll('rect')`, таким:

```
['10', '20', '20b', '20c'].forEach((scheme, i) => {
  const height = 10;
  const padding = 5;
  const categoryScheme = &grave;schemeCategory${scheme}&grave;;
  const selector = &grave;rect.scheme-${scheme}&grave;;
  const categoryColor = d3.scaleOrdinal(d3[categoryScheme]);
```

```
  chart.container.selectAll(selector)
    .data(data.slice())
    .enter()
    .append('rect')
    .classed(selector, true)
    .attr('x', d => bands(d))
    .attr('y', (chart.height / 2) - ((i * height) + (padding * i)))
    .attr('width', bands.bandwidth)
    .attr('height', height)
    .style('fill', d => categoryColor(d));
  });
```

Рисуются сразу все цветовые схемы. Ослепительно!

Дополнительные сведения о цветовых схемах в D3 см. на странице https://github.com/d3/d3-scale#category-scales.

Количественные масштабы

Существует несколько видов количественных масштабов, но у всех есть общая черта: непрерывная область определения. Непрерывный масштаб моделируется не множеством дискретных значений, а простой функцией. Перечислим семь типов количественных масштабов: линейный, тождественный, степенной, логарифмический, квантованный, квантильный и пороговый. Все они определяют преобразования входной области определения. У первых четырех область значений непрерывная, а у трех последних – дискретная.

Чтобы понять, как ведут себя эти масштабы, воспользуемся ими для манипулирования осью *Y* при рисовании функции Вейерштрасса, первой открытой функции, которая всюду непрерывна, но нигде не дифференцируема. Это означает, что график такой функции можно нарисовать, не отрывая пера от бумаги, но ни в какой точке невозможно вычислить угол наклона линии (производную).

Добавьте в выражение `scalesDemo` такой код:

```
(function quantitativeScales() {
  const weierstrass = (x) => {
    const a = 0.5;
    const b = (1 + 3 * Math.PI / 2) / a;
    return d3.sum(d3.range(100).map(
      n => Math.pow(a, n) * Math.cos(Math.pow(b, n) * Math.PI * x)));
  };
}());
```

Мы генерируем данные, получаем экстент функции `weierstrass` и используем линейный масштаб для `x`:

```
const data = d3.range(-100, 100).map(d => d / 200);
const extent = d3.extent(data.map(weierstrass));
const colors = d3.scaleOrdinal(d3.schemeCategory10);
const x = d3.scaleLinear()
  .domain(d3.extent(data))
  .range([0, chart.width]);
```

Функция рисования поможет избежать дублирования кода:

```
const drawSingle = line => chart.container.append('path')
  .datum(data)
  .attr('d', line)
  .style('stroke-width', 2)
  .style('fill', 'none');
```

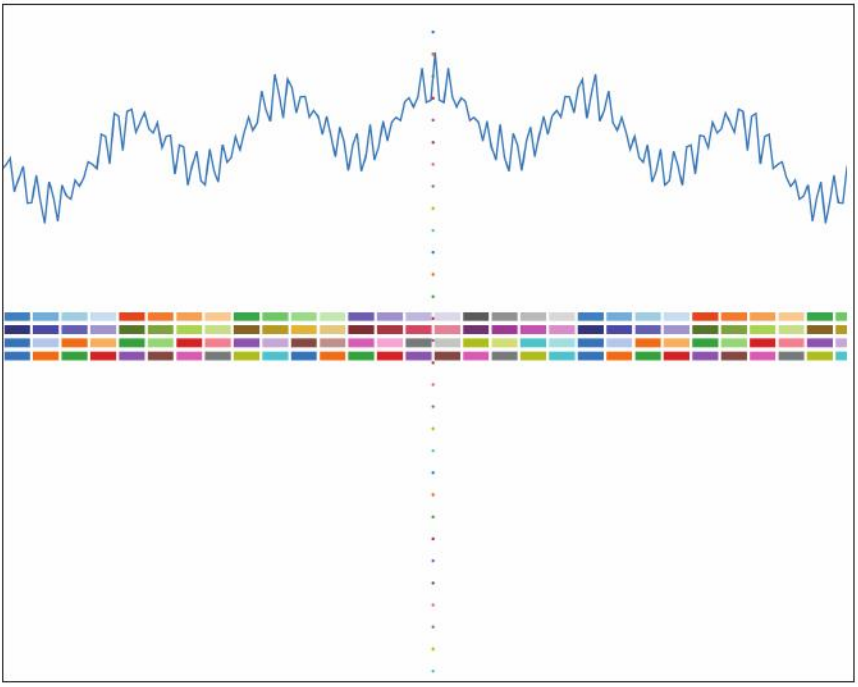
Масштабы с непрерывной областью значений

Для тестирования масштабов с непрерывной областью значений напишем такой код:

```
const linear = d3.scaleLinear()
  .domain(extent)
  .range([chart.height / 4, 0]);

const line1 = d3.line()
  .x(x)
  .y(d => linear(weierstrass(d)));

drawSingle(line1)
  .attr('transform', &grave;translate(0, ${chart.height / 16})&grave;)
  .style('stroke', colors(0));
```

Мы определили линейный масштаб, в котором область определения охватывает все значения, возвращаемые функцией `weierstrass`, а область значений простирается от 0 до ширины области рисования. В этом масштабе для отображения между входом и выходом применяется линейная интерполяция, он даже способен предсказывать значения в точках, лежащих вне области определения. Если нам это не нужно, можно воспользоваться функцией `.clamp()`. Задавая более двух чисел в областях определения и значений, мы можем создать полилинейный масштаб, в котором каждый участок ведет себя как отдельный линейный масштаб.

Линейный масштаб создает такую картинку:



Добавим сразу все остальные непрерывные масштабы:

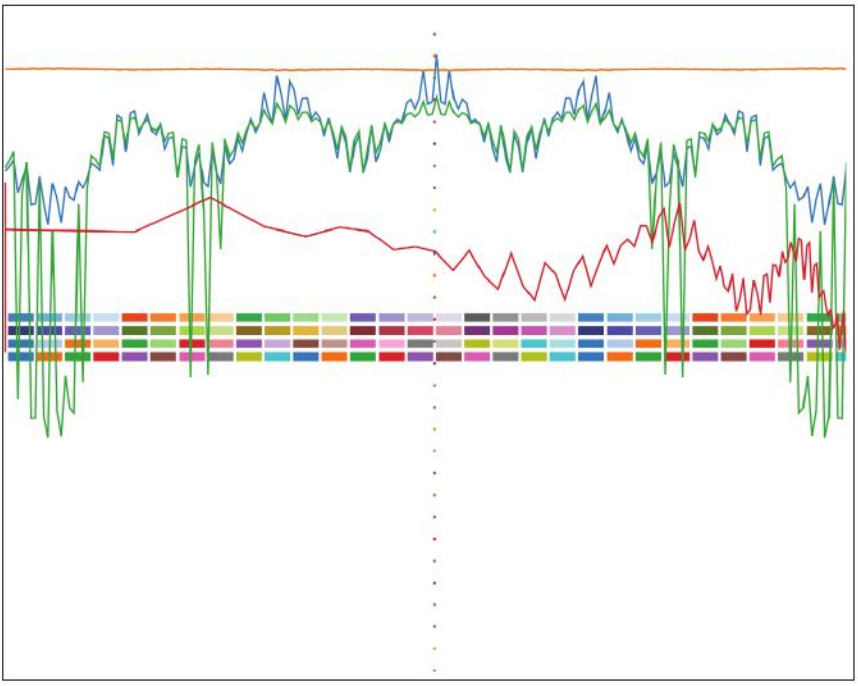
```
const identity = d3.scaleIdentity()
  .domain(extent);

const line2 = line1.y(d => identity(weierstrass(d)));

drawSingle(line2)
  .attr('transform', &grave;translate(0, ${chart.height / 12})&grave;)
  .style('stroke', colors(1));

const power = d3.scalePow()
  .exponent(0.2)
  .domain(extent)
  .range([chart.height / 2, 0]);

const line3 = line1.y(d => power(weierstrass(d)));

drawSingle(line3)
  .attr('transform', &grave;translate(0, ${chart.height / 8})&grave;)
```

```
  .style('stroke', colors(2));

const log = d3.scaleLog()
  .domain(d3.extent(data.filter(d => (d > 0 ? d : 0))))
  .range([0, chart.width]);

const line4 = line1.x(d => (d > 0 ? log(d) : 0))
  .y(d => linear(weierstrass(d)));

drawSingle(line4)
  .attr('transform', &grave;translate(0, ${chart.height / 4})&grave;)
  .style('stroke', colors(3));
```

Во всех масштабах, кроме степенного, мы повторно использовали одно и то же определение линии, изменяя лишь масштаб `y`, а в случае степенного масштаба изменение `x` улучшает пример.

Мы также учли, что логарифм определен только для положительных чисел, но обычно этот масштаб все равно не применяется к периодическим функциям. Он больше полезен, когда нужно на одном графике изобразить большие и малые числа.

Теперь картинка выглядит следующим образом:



Начинает походить на обложку альбома группы New Order или что-то в этом роде...

Тождественный масштаб оранжевый (с меткой 1), он почти постоянный, потому что данные, подаваемые на вход функции, изменяются всего от –0.5 до 0.5, степенной масштаб (с меткой 2) зеленый, а логарифмический (с меткой 3) красный.

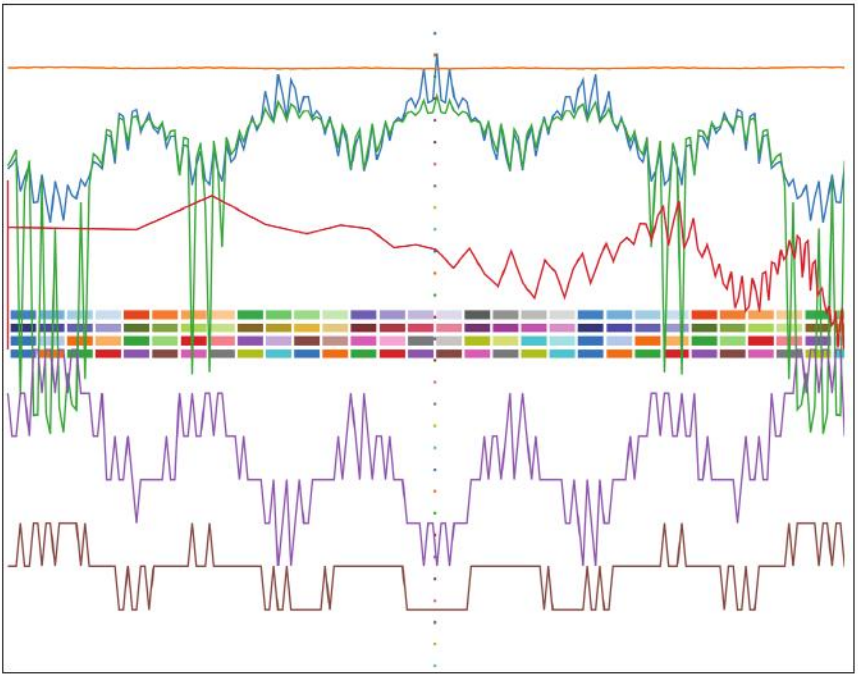
Масштабы с дискретной областью значений

Для сравнения нам интересны квантованные и пороговые масштабы. Квантованный масштаб разбивает область определения на равные части и отображает их на выходную область значений, а пороговый позволяет отображать произвольные участки области определения на дискретные значения:

```
const offset = 100;
const quantize = d3.scaleQuantize()
  .domain(extent)
  .range(d3.range(-1, 2, 0.5)
  .map(d => d * 100));

const line5 = line1.x(x)
  .y(d => quantize(weierstrass(d)));

drawSingle(line5)
  .attr('transform', &grave;translate(0, ${(chart.height / 2) + offset})&grave;)
  .style('stroke', colors(4));

const threshold = d3.scaleThreshold()
  .domain([-1, 0, 1])
  .range([-50, 0, 50, 100]);

const line6 = line1.x(x)
  .y(d => threshold(weierstrass(d)));

drawSingle(line6)
  .attr('transform', &grave;translate(0, ${(chart.height / 2) + (offset *
2)})&grave;)
  .style('stroke', colors(5));
```

Квантованный масштаб представляет функцию `weierstrass` в виде множества дискретных значений от 1 до 2 с шагом 0.5 (–1, –0.5, 0 и т. д.), а пороговый отображает значения, меньшие –1, в –50, –1 – в 0 и т. д.

Результат выглядит следующим образом:



Время

Время – штука сложная. Час может содержать 3600 секунд, а может и 3599 – если имеется високосная секунда. Завтра может отстоять от сегодня на 23 или на 25 часов, в месяце бывает от 28 до 31 дня, а в году – 365 или 366 дней. В одних декадах дней меньше, чем в других. Не забывайте об этом в следующий раз, когда захотите прибавить к временной метке 3600 секунд, чтобы сдвинуть ее вперед на час, или прибавить 24*3600 – чтобы получить то же самое время в следующих сутках. А если вспомнить, сколько наборов данных тесно связано со временем, то проблема встает во весь рост. Так как же все-таки работать с временными данными?

По счастью, в состав D3 включены средства для этой цели. Все они находятся в пакете `d3-time-format`.

Форматирование

Для создания форматера нужно передать форматную строку функции `d3.timeParse()`. Затем его можно использовать для преобразования строк в объекты `Date`.

Полностью язык описан в документации по D3, но несколько примеров мы здесь рассмотрим:

```
> format = d3.timeParse('%Y-%m-%d')
> format('2015-12-14')
Mon Dec 14 2015 00:00:00 GMT+0100 (CET)
```

Мы определили форматер с помощью `d3.timeParse()`, передав ей строку для разбора дат вида «год-месяц-день». Затем мы разобрали типичную для наборов данных дату и получили корректный объект даты, в котором часу, минуте и секунде присвоены значения по умолчанию.

Функция `d3.timeFormat()` работает иначе:

```
> format = d3.timeFormat('%Y-%m-%d')
> format(new Date());
"2017-03-12"
```

Форматер времени, полностью отвечающий стандарту ISO, дает функция `d3.isoFormat`. Она часто бывает полезна, поскольку во многих базах данных этот формат подразумевается по умолчанию, а кроме того, именно в таком формате представляет объект `Date` метод `toString()`.

Арифметика времени

Мы также получаем в свое распоряжение полный комплект арифметических функций для работы с объектами JavaScript `Date`.

- `d3.time`**`Interval`**, где **`Interval`** может принимать значения `second`, `minute`, `hour` и т. д. Эта функция возвращает новый временной интервал. Например, `d3.timeHour` возвращает часовой интервал.
- `d3.time`**`Interval`**`(Date)` – псевдоним функции `interval.floor()`, которая округляет `Date` с недостатком, так что все компоненты мельче **`Interval`** приравниваются нулю.
- **`interval`**`.offset(Date, step)` возвращает новую дату, отстоящую от `Date` на заданное число шагов.
- **`interval`**`.range(Date_start, Date_stop)` возвращает все интервалы между двумя датами.

Например, чтобы узнать, сколько времени будет через час, мы пишем:

```
> d3.timeHour.offset(new Date(), 1)
Sun Mar 19 2017 02:44:30 GMT+0100 (CET)
```

Выясняется, что уже очень поздно – пора заканчивать писать книги о JavaScript и укладываться спать.

Хотите узнать о времени побольше? `Moment.js` – потрясающая библиотека, которая точно вычисляет такие вещи, как часовые пояса и разность между двумя временными метками. Находится она на сайте http://momentjs.com.

Загрузка данных

Очень часто мы лишены счастья создавать данные с помощью генератора случайных чисел из пакета Math, а должны загружать их из внешнего источника. Хотя иногда проще сгенерировать набор данных программно, по большей части визуализировать с помощью D3 приходится реальные данные.

Есть много способов получить данные. Я рассмотрю только основные.

- **Создать HTTP-запрос вручную:** мы уже поступали так в предыдущих главах. Мы передаем URL функции, которая заставляет браузер отправить запрос. В эту категорию попадают функции `XMLHttpRequest` и `Fetch`. Для импорта JSON на сервере должен быть включен режим разделения ресурсов между разными источниками (Cross-Origin Resource Sharing –CORS), в котором сервер посылает заголовок вида `Access-Control-Allow-Origin: *`. Это связано с моделью безопасности браузера, но не относится к случаю, когда мы загружаем данные из того же домена, что и код, чтобы можно было обойтись без дополнительной работы.
- **Импортировать в виде модуля:** некоторые наборы данных доступны в виде модулей, загружаемых npm. Примером может служить набор `earth-topojson`, которым мы воспользуемся, когда начнем работать с картами. Это очень удобно, поскольку означает, что нужно всего лишь импортировать модуль, перед тем как присваивать значение переменной. К сожалению, требования статического анализа из ES2015+ диктуют, что конечный набор данных можно загружать только на этапе сборки.
- **Подключиться к веб-сокету:** веб-сокеты – это новая технология, которая позволяет браузеру подписаться на поток и полу-

чать от сервера данные в режиме реального времени. В случае динамических данных это весьма удобно, но потребляет много памяти и обычно означает, что серверную часть вы должны реализовывать самостоятельно.

- **Включить тег `script`, который загружает данные в виде JSONP и присоединяет их к глобальной переменной:** JSONP – это вариант JSON, обернутый обратным вызовом функции, заданной с помощью URL. Эта технология была популярна до стандартизации CORS. Ныне она встречается редко.

Ядро

D3 предоставляет собственную систему отправки запросов, которая находится в модуле `d3-request`. В ней используется механизм `XMLHttpRequest`, обернутый непритязательной функцией `d3.xhr()`, т. е. запросы формируются вручную.

Функция принимает URL и необязательный обратный вызов. Если обратный вызов задан, то запрос отправляется немедленно, и функция обратного вызова получает данные в виде аргумента по завершении запроса. Если же обратный вызов не задан, то мы должны создать запрос сами – полностью, включая все заголовки и метод отправки, – а затем отправить его. Ниже показано, как выглядит код создания запроса:

```
let xhr = d3.xhr('http://www.example.com/test.json');
xhr.mimeType('application/json');
xhr.header('User-Agent', 'SuperAwesomeBrowser');
xhr.on('load', function (request) { ... });
xhr.on('error', function (error) { ... });
xhr.on('progress', function () { ... });
xhr.send('GET');
```

Здесь мы отправляем `GET`-запрос, ожидая получить в ответ JSON, и говорим серверу, что клиентом является браузер `SuperAwesomeBrowser`. Можно сократить процедуру, задав функцию обратного вызова, но тогда мы не сможем ни определить собственные заголовки, ни получать другие события, помимо успешного завершения запроса.

Есть другой способ – вспомогательные функции D3. К ним относятся:

- `d3.text` – импорт текстового файла в виде строки:

```
d3.text('/data/transcript.txt', (error, text) => {
    console.log(text);
});
```

- `d3.csv` – загрузка таблицы в формате CSV с разделителями-запятыми. Для загрузки данных с разделителями-табуляторами служит функция `d3.tsv`:

```
d3.csv('/data/finance_data.csv', (error, data) => {
    console.log(data);
});
```

- `d3.json` JSON:

```
d3.json('http://www.aendrew.com/hello.json', (error, data) => {
    console.log(data);
});
```

- `d3.xml` – загрузка файла в формате XML:

```
d3.xml('/probably/made/by/java.xml', (error, data) => {
    console.log(data);
});
```

Отметим, что никто не заставляет использовать эти функции, если вы можете получить данные как-то иначе. Например, если CSV-файл загружен в строковую переменную, то для преобразования его в массив или в объект можно применить функцию `d3.csvParse` из модуля `d3-dsv`. Или воспользоваться встроенной в браузер функцией `JSON.parse()` для преобразования JSON-строки в объект. Или объектом `DOMParser` для разбора XML.

Для управления потоком во всех этих методах используются обратные вызовы, и это не слишком удобно, когда нужно выполнить несколько запросов. Поговорим об этом подробнее.

Управление потоком

JavaScript – однопоточный асинхронный язык, т. е. программа не порождает несколько потоков, как, например, в C++. Поэтому если некоторая функция блокирует главный поток, то останавливается все вообще. По счастью, благодаря асинхронности функциям нет нужды вести себя подобным образом, и пишутся они так, что следующая начинает выполнение еще до завершения предыдущей.

С одной стороны, это один из самых интересных аспектов языка JavaScript, лежащий в основе его эффективности, а с другой – такая организация затрудняет рассуждения о коде и увеличивает его сложность.

Хотя можно сделать довольно много, не залезая в дебри модели событий JavaScript, асинхронная природа JavaScript выступает на

передний план, когда нужно отправить запрос. Каким бы быстрым ни было ваше подключение к Интернету, все равно по законам физики неизбежна задержка при перемещении данных из одного места в другое. Если бы JavaScript блокировал приложение на все время с начала HTTP-запроса и до его завершения, то веб-страницы работали бы невыносимо медленно. Особенно это касается современных веб-приложений, которые отправляют десятки фоновых запросов, пока вы гуляете по сайту.

Традиционно программа начинает запрос и предоставляет функцию, которая получит управление, когда запрос завершится, обычно в виде обратного вызова:

```
function done() {
  console.log(this.responseText); // Закончился!
}

const req = new XMLHttpRequest();
req.addEventListener('load', done);
req.open('get', 'http://www.packt.com');
req.send();
```

Здесь функция обратного вызова `done` печатает содержимое запроса на консоли в виде строки. В этом тривиальном примере понять, что происходит, просто, но если запросы следуют один за другим, то ситуация резко усложняется:

```
const req1 = new XMLHttpRequest();

req1.addEventListener('load', function done1() {
  const response1 = JSON.parse(this.textContent);
  const req2 = new XMLHttpRequest();

  req2.addEventListener('load', function done2() {
    console.log(this.textContent); // закончился!
  });

  req2.get(response1.newEndpoint);
  req2.send();
});

req1.get('http://www.aendrew.com/api/1.json');
req1.send();
```

Здесь мы отправляем один запрос, дожидаемся его завершения, выделяем значение из полученного ответа в формате JSON и отправляем следующий запрос, зависящий от этого значения. Если бы нужно было отправить третий запрос, зависящий от результата второго, то

понадобился бы еще один уровень вложенности. Этот феномен разработчики на JavaScript называют «адом обратных вызовов», поскольку такой код трудно читать и высказывать о нем утверждения. К сожалению, хотя методы отправки запросов в D3 не такие пространные, как `XMLHttpRequest`, они все равно отмечены печатью старой школы:

```
d3.json('http://www.aendrew.com/api/1.json', (error1, data1) => {
  if (data1) {
    d3.json(data1.newEndpoint, (error2, data2) => {
      if (data2) {
        // ...и так далее...
      }
    });
  }
});
```

Всего два запроса, и уже четыре уровня отступов. Этот код можно было бы переписать, сделав его более удобочитаемым, но уже ясно, что такая организация программы крайне неудобна. По счастью, нас никто не заставляет применять исключительно методы D3 для запроса данных! На самом деле в современном JavaScript есть много других способов получения данных и структурирования кода, например:

- обещания;
- генераторы;
- наблюдаемые объекты.

Скоро мы покажем, как их использовать в сочетании с D3. Отметим, что они применимы отнюдь не только для ожидания данных, и это станет понятно, когда мы будем обсуждать генераторы.

Обещания

Из всех методов организации кода, рассматриваемых в этой главе, обещания – мой любимый. Их очень просто использовать, и они лежат в основе паттерна `await/async`, который постоянно встречается в этой книге. Коротко говоря, обещание – это объект, представляющий значение, которое может быть доступно сейчас, никогда или в любой момент между этими двумя крайностями. Вот небольшой пример:

```
const p = new Promise((resolve, reject) => {
  d3.json('http://www.aendrew.com/api/1.json', (err, data1) => {
    if (err) {
      reject(err);
    } else {
```

```
      resolve(data1);
    }
  });
});

p.then(data => console.log(data.newEndpoint))
.catch(err => console.error(err));
```

Мы создали новое обещание и присвоили его переменной `p`. Конструктор объекта Promise принимает две функции обратного вызова: `resolve` и `reject`. Ошибки передаются функции `reject`, а данные – функции `resolve`. Позже вне обратного вызова Promise мы можем вызвать метод `Promise.prototype.then()` объекта `p`, чтобы сделать что-то, когда обещание будет выполнено, или обработать ошибки с помощью метода `Promise.prototype.catch()`. Это, правда, довольно глупый пример, поскольку функции из пакета `d3-request` не возвращают обещаний и надо научиться как-то программировать в обход принятого в нем стиля. Начиная с этого момента, будем использовать `Fetch`.

Памятуя об этом, зададимся все же вопросом: что делать, если продолжить работу можно только после выполнения двух обещаний? Если используются обратные вызовы, то, наверное, придется обрабатывать вложенные запросы или придумать какой-то хитрый механизм обработки событий, даже если второму запросу не нужны данные, возвращенные в ответ на первый. С помощью обещаний мы можем дождаться выполнения сразу нескольких, один раз вызвав метод `Promise.all`:

```
Promise.all([fetch('./data1.json'), fetch('./data2.json')])
  .then(([data1, data2]) => [data1.json(), data2.json()])
  .then(([data1, data2]) => {
    console.log(data1); // Готов data1.json
    console.log(data2); // Готов data2.json
});
```

Напомним (см. главу 1), что `fetch` возвращает обещание, содержащее запрос, которое можно превратить в другое обещание, содержащее полученные данные, разобранные определенным способом (текст, двоичный блок или JSON). Здесь мы просто объединяем обещания с помощью `Promise.all` и дожидаемся их выполнения, как и раньше.

Но зачем вообще «заморачиваться» со старым синтаксисом обещаний, когда есть новомодная штучка – `async/await`?

```
async function getDataAsync() {
  const data1 = await (await fetch('./data1.json').json();
  const data2 = await (await fetch(&grave;./data${data1}.json&grave;).json();
```

```
  console.log(data1);
  console.log(data2);
}
```

Черт побери, да это же великолепно! Только понять бы, что тут происходит.

Любая функция, помеченная ключевым словом `async`, возвращает некоторое обещание. Даже если мы не указали возвращаемого значения (как в примере выше), такая функция все равно вернет обещание. Важно, однако, то, что впоследствии мы можем дождаться выполнения этого обещания с помощью ключевого слова `await` и не использовать метод `Promise.prototype.then()`. Таким образом, API обещаний оказывается совершенно прозрачным и позволяет писать асинхронный код в стиле, понятном всякому, кто привык к синхронным языкам.

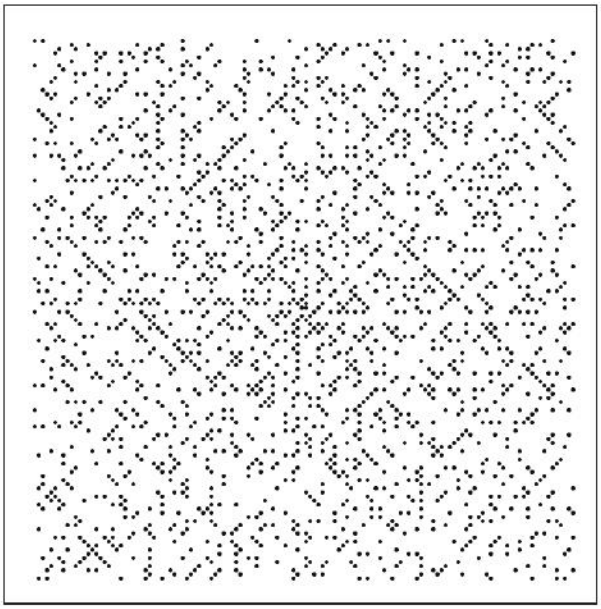
Не стану приводить здесь содержательного примера использования обещаний, потому что мы уже встречались с ними раньше и встретимся еще не раз. К концу книги вы прекрасно освоите все тонкости работы с ними.

Генераторы

Генераторы, впервые добавленные в версию ES7, – это функции, которые можно приостановить в ожидании данных. Созданный экземпляр функции-генератора возвращает объект, имеющий метод `next()` и свойство `finished`. При каждом вызове `next()` генератор *отдает* следующее значение. По достижении конца последовательности (если она конечна) свойство `finished` принимает значение `true`. По существу, генератор представляет собой фабрику итераторов, т. е. функций, способных перебирать коллекцию по одному объекту, сохраняя внутри себя состояние обхода. Дополнительные сведения о генераторах можно найти по адресу http://mdn.io/Iterators_and_Generators.

Поэкспериментируем с ними на примере нерешенной математической задачи о спирали Улама. Открытая в 1963-м, она вскрывает некоторые закономерности распределения простых чисел на двумерной плоскости. До сих пор никому не удалось найти объясняющую их формулу.

Мы будем строить спираль, имитируя метод пера и бумаги Улама: нанесем все натуральные числа по спирали, а затем вычеркнем составные числа. Только вместо чисел будем рисовать точки с помощью элемента SVG `circle`. На первом этапе эксперимента получится такая картина:



Не бог весть что, но ведь это только первые 2000 простых чисел. Заметили диагональные линии точек? Некоторые из них можно описать полиномами, что влечет за собой интересные выводы о предсказании простых чисел и, как следствие, о безопасности криптографии.

Создайте новую функцию в файле `lib/chapter3/index.js`. В конце файла отключите выполнение предыдущего выражения:

```
})(false);
```

Теперь создайте новое IIFE:

```
const ulamSpiral = ((enabled) => {
  if (!enabled) return;

  const chart = chartFactory();
})(true);
```

Пока что ничего нового; мы просто подготовились к рисованию новой диаграммы с помощью нашей фабрики `chartFactory` и создали экземпляр диаграммы.

Затем определим алгоритм, который генерирует список чисел и их координаты на спирали. Начнем с алгоритма построения спирали. Создайте новую функцию внутри только что написанного кода:

```
const generateSpiral = (n) => {
  const spiral = [];
  let x = 0;
  let y = 0;
  const min = [0, 0];
  const max = [0, 0];
  let add = [0, 0];
  let direction = 0;
  const directions = {
    up: [0, -1],
    left: [-1, 0],
    down: [0, 1],
    right: [1, 0],
  };
}
```

Функция построения спирали принимает единственный аргумент `n` – верхнюю границу. Здесь определены четыре направления перемещения и несколько переменных, используемых в алгоритме. Комбинация минимальной и максимальной известных координат скажет нам, когда поворачивать, в `x` и `y` хранится текущая позиция, а `direction` говорит, в какой части спирали мы сейчас находимся.

Теперь добавьте сам алгоритм в конец функции:

```
  d3.range(1, n).forEach((i) => {
    spiral.push({ x, y, n: i });
    add = directions[['up', 'left', 'down', 'right'][direction]];
    x += add[0];
    y += add[1];
    if (x < min[0]) {
      direction = (direction + 1) % 4;
      min[0] = x;
    }
    if (x > max[0]) {
      direction = (direction + 1) % 4;
      max[0] = x;
    }
    if (y < min[1]) {
      direction = (direction + 1) % 4;
      min[1] = y;
    }
    if (y > max[1]) {
      direction = (direction + 1) % 4;
      max[1] = y;
    }
  });
  return spiral;
```

`d3.range()` генерирует массив чисел между двумя аргументами, который мы затем обходим с помощью функции `.forEach`. На каждой итерации в массив спирали добавляется новая тройка `{x: x, y: y, n: i}`. Дальнейший код нужен просто для смены направления, когда мы упираемся в угол.

Теперь приступим к рисованию. Вернитесь в объемлющую функцию и добавьте такой код:

```
const dot = d3.symbol().type(d3.symbolCircle).size(3);
const center = 400;
const l = 2;
const x = (x, l) => center + (l * x);
const y = (y, l) => center + (l * y);
```

Мы определили генератор точек и две функции, которые помогут преобразовать координаты, полученные от функции построения спирали, в позиции пикселей. Здесь `l` длина стороны одной ячейки квадратной сетки.

Далее нужно вычислить простые числа. Конечно, можно получить огромный список таких чисел из сети, но это не так интересно, как использовать генератор.

Создайте метод `generatePrimes` в нашей функции `ulamSpiral` в файле `chapter3/index.js`:

```
generatePrimes(n) {}
```

И поместите в эту функцию следующие генераторы. Первым генератором будет функция, которая при каждом вызове возвращает следующее по порядку число:

```
function* numbers(start) {
  while (true) {
    yield start++;
  }
}
```

Ой, сколько нового и незнакомого! Ничего, разберемся. Звездочка после слова `function` означает, что это функция-генератор. Внутри цикла `while` (который никогда не завершается, так что мы можем запрашивать числа до морковкина заговенья) ключевое слово `yield` отдает следующее число. Как это используется, мы скоро увидим.

Далее создадим генератор простых чисел, который будет вызывать написанный ранее генератор чисел:

```
function* primes() {
  var seq = numbers(2); // начинаем с 2.
```

```
  var prime;
  while (true) {
    prime = seq.next().value;
    yield prime;
  }
}
```

Теперь понять этот код нетрудно. Генератор чисел, запускаемый с числа 2, присваивается переменной `seq`. Затем мы вызываем метод `.next()`, чтобы получить от него следующее значение.

Нам нужен еще один генератор для перебора простых чисел. Добавьте такой код:

```
function* getPrimes(count, seq) {
  while (count) {
    yield seq.next().value;
    count--;
  }
}
```

В конец функции `generatePrimes` поместите такие строки:

```
for (let prime of getPrimes(n, primes())) {
  console.log(prime);
}
```

А после `generatePrimes` – строку:

```
const primes = generatePrimes(2000);
```

Если вы запустите сервер разработки командой

```
$ npm start
```

то, перейдя по адресу `http://localhost:8080/`, увидите на консоли массив из 2000 последовательных целых чисел.

Нам еще нужно добавить фильтр простых чисел. Добавьте в `generatePrimes` новый генератор:

```
function* filter(seq, prime) {
  for (let num of seq) {
    if (num % prime !== 0) {
      yield num;
    }
  }
}
```

Он просто обходит все числа в сгенерированной к этому моменту последовательности и проверяет, делятся ли они нацело на потенци-

альное простое число. Если какое-то число в последовательности нацело делится на простое, значит, оно простым не является. Эта функция используется, чтобы отфильтровать все составные числа.

Далее в генераторе `primes` после строки `yield prime` добавьте такую строку:

```
seq = filter(seq, prime);
```

В результате ко всей уже имеющейся последовательности будет применен фильтр для проверки на простоту.

Найдите код

```
for (let prime of getPrimes(n, primes())) {
  console.log(prime);
}
```

и замените его таким:

```
let results = [];
for (let prime of getPrimes(n, primes())) {
  results.push(prime);
}
return results;
```

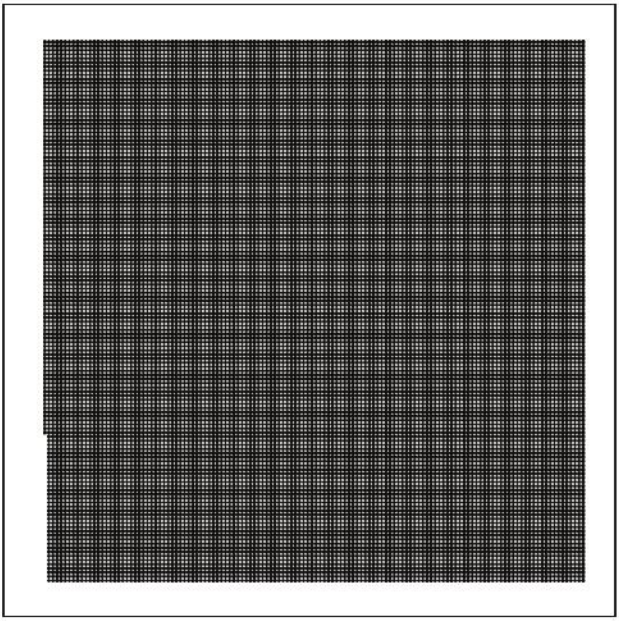
Теперь все простые числа находятся в массиве, и пора уже собрать все вместе. Добавьте в конструктор такой код:

```
const primes = generatePrimes(2000);
const sequence = generateSpiral(d3.max(primes));
```

В результате с помощью нашего генератора будет создан массив, содержащий 2000 простых чисел, после чего запускается генератор спирали, которому передается максимальное найденное простое число. Объединим это с генератором точек, чтобы получить картинку, которую мы видели раньше:

```
chart.container.selectAll('path')
  .data(sequence)
  .enter()
  .append('path')
  .attr('transform', d => &grave;translate(${x(d.x, 1)}, ${y(d.y, 1)})&grave;)
  .attr('d', dot);
```

Результат выглядит так:



Гм, не совсем то, что надо, но идея-то правильная!

Нам еще осталось отфильтровать составные числа из матрицы точек. Замените строку `const sequence` такой:

```
const sequence = generateSpiral(d3.max(primes)).filter(d =>
  primes.indexOf(d.n) > -1);
```

Если у вас медленный компьютер, то придется подождать, потому что мы изрядно нагрузили бедный браузер!

Очевидно, что с производительностью здесь не все в порядке. Конечно, наш проект – супер, и мы можем нагенерировать столько простых чисел, сколько захотим, но в действительности такой способ достижения результата чудовищно неэффективен (если попробовать сгенерировать примерно 2500 чисел, то у Chrome переполнится стек). Ранее отмечалось, что список из нескольких тысяч первых простых чисел всегда можно скачать из Сети, и «весит» он всего один-два килобайта; в большинстве случаев такой путь предпочтительнее, и далее в этой книге мы, как правило, будем выбирать его.

Сделаем пример более интересным – визуализируем плотность распределения простых чисел. Определим сетку с большей стороной ячейки и раскрасим ячейки в разные цвета в зависимости от того, сколько в них точек. Красными будут ячейки, в которых количество простых чисел меньше медианы, а зелеными – остальные. Оттенок цвета будет показывать, насколько количество простых чисел далеко от медианного значения.

Сначала воспользуемся вложенной структурой для определения новой сетки. Добавьте в конец конструктора такой код:

```
const scale = 8;
const regions = d3.nest()
  .key(d => Math.floor(d.x / scale))
  .key(d => Math.floor(d.y / scale))
  .rollup(d => d.length)
  .map(sequence);
```

Коэффициент 8 означает, что новая ячейка содержит 64 старых.

Функция `d3.nest()` преобразует данные во вложенные словари в соответствии с ключом. Первая функция `.key()` создает столбцы: каждое значение `x` отображается на соответствующее значение `x` новой сетки. Вторая функция `.key()` делает то же самое для `y`. Затем с помощью функции `.rollup()` мы преобразуем получившиеся списки в одно значение – количество точек.

Данные подаются на вход `.map()`, и мы получаем такую структуру:

```
{
  "0": {
    "0": 5,
    "-1": 2
  },
  "-1": {
    "0": 3,
    "-1": 4
  }
}
```

С первого взгляда не очень понятно, но на самом деле это коллекция столбцов, содержащих строки. В ячейке (0, 0) находятся 5 простых чисел, в ячейке (–1, 0) – 2 простых числа и т. д.

Чтобы найти медиану и количество оттенков, нужно поместить эти счетчики в массив:

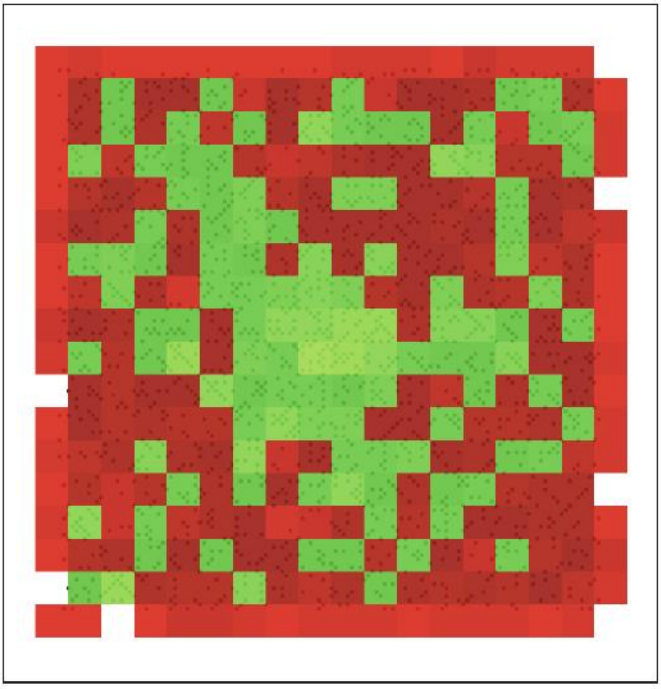
```
const values = d3.merge(d3.keys(regions)
  .map(_x => d3.values(regions[_x])));
const median = d3.median(values);
const extent = d3.extent(values);
const shades = (extent[1] - extent[0]) / 2;
```

С помощью функции `.map` мы сопоставляем ключам областей (координатам `x`) списки значений в каждом столбце, а затем с помощью `d3.merge()` линеаризуем получившийся массив массивов. Функция `d3.median()` дает медианное значение массива, а `d3.extent()` – наименьшее и наибольшее значение, на основании которых мы вычисляем количество необходимых оттенков.

Наконец, в еще одном цикле по координатам раскрасим новую сетку:

```
regions.each((_x, _xKey) => {
  _x.each((_y, _yKey) => {
    let color,
    red = '#e23c22',
    green = '#497c36';
    if (_y > median) {
      color = d3.rgb(green).brighter(_y / shades);
    } else {
      color = d3.rgb(red).darker(_y / shades);
    }
    chart.container.append('rect')
      .attr('x', x(_xKey, 1 * scale))
      .attr('y', y(_yKey, 1 * scale))
      .attr('width', 1 * scale)
      .attr('height', 1 * scale)
      .style('fill', color)
      .style('fill-opacity', 0.9);
  });
});
```

Получилось изображение, напоминающее случайные аватары, генерируемые WordPress:



Наблюдаемые объекты

Наблюдаемые объекты (Observable) – новый подход к управлению потоком, идея которого состоит в том, чтобы подписаться на источник данных, а затем выполнять различные функции по мере поступления событий. Наблюдаемые объекты можно использовать для чего угодно, но лучше всего они проявляют себя в ситуациях, когда данные часто обновляются, например при построении динамических графиков рыночных котировок. Также объекты Observable можно использовать для обновлений на основе *push*-уведомлений, например при подключении к веб-сокету; это позволяет отказаться от постоянного опроса единственной оконечной точки.

В настоящее время для работы с наблюдаемыми объектами чаще всего применяется библиотека RxJS. Если в случае обещаний мы пишем код вида:

```
const data = await (await fetch(url)).json()
```

то при использовании RxJS мы написали бы примерно следующее:

```
Rx.Observable
  .fromPromise(fetch('/data/cultural.json'))
  .flatMap(v => v.json())
  .subscribe(
    (data) => {
      console.dir(data);
    },
    (error) => {
      console.dir(error);
    },
    () => {
      console.log('complete!');
    }
  );
```

В этой книге нет никакой возможности сколько-нибудь подробно рассказать о наблюдаемых объектах, потому что это действительно сложная тема, интересная в основном тем, кто имеет дело с динамическими данными. Но если у вас разыгрался аппетит, поинтересуйтесь книгой по RxJS, бесплатно доступной по адресу https://xgrommx.github.io/rx-book/.

География

Геопространственные типы данных используются, например, при нанесении на карту данных о погоде или плотности населения. Преобразование физических координат в нечто, что можно представить на двумерной карте, – трудная математическая задача, которая занимала воображение ученых на протяжении многих столетий (и до сих пор *окончательно* не решена, если судить по огромному числу проекций, поставляемых в составе пакета `d3-geoprojection`).

D3 включает три инструмента для работы с географическими данными:

- **пути** порождают конечные пиксели;
- **проекции** преобразуют сферические координаты в декартовы;
- **потоки** ускоряют работу.

Основной формат данных, который мы будем использовать, – TopoJSON, более компактная версия формата GeoJSON, предложенного Майком Бостоком. В некотором смысле TopoJSON относится к GeoJSON, как DivX к видео. Если в GeoJSON формат JSON при-

меняется для кодирования географических данных – точек, прямых линий и многоугольников, то в TopoJSON основные объекты кодируются с помощью дуг и повторно используются для построения все более сложных объектов. Конечный файл может оказаться процентов на 80 меньше, чем при использовании GeoJSON.

Получение геоданных

По сравнению с другими типами данных, форматы геоданных малопригодны для онлайновой презентации, поэтому их необходимо преобразовывать в TopoJSON. Мы находим какие-то данные в формате Shapefile или GeoJSON, а затем переводим их в формат TopoJSON с помощью командной утилиты `topojson`. Найти детальные данные не всегда легко, но все же возможно; если вам нужна карта своей страны, поищите данные национальной службы проведения переписей. Например, Бюро переписи населения США выкладывает много полезных наборов данных по адресу https://www.census.gov/geo/maps-data/, в Великобритании аналогом является сайт https://geoportal.statistics.gov.uk/geoportal/, а в Канаде – https://www12.statcan.gc.ca/census-recensement/2011/geo/bound-limit/bound-limit-2011-eng.cfm.

Еще один великолепный источник геоданных с различным уровнем детализации – сайт *Natural Earth*. Его главное достоинство – в том, что различные слои (океаны, страны, дороги и т. д.) тщательно подогнаны друг к другу и часто обновляются. Найти эти наборы данных можно по адресу http://www.naturalearthdata.com. Можно также получить политическую карту мира, уже преобразованную в формат TopoJSON, установив с помощью npm пакет `world-atlas`.

Мы преобразуем данные Natural Earth в TopoJSON самостоятельно, потому что в пакете `world-atlas` не показаны такие вещи, как ландшафт и реки. Сначала скачайте файл http://naciscdn.org/naturalearth/packages/natural_earth_vector.zip, содержащий все данные. Распакуйте его. Затем установите глобальные зависимости:

```
npm install --global topojson shapefile
```

Мы устанавливаем пакет `topojson` глобально, потому что нам понадобится входящая в него командная утилита `geo2topo` для преобразования GeoJSON в TopoJSON.

Распаковав zip-файл и перейдя в каталог, где находится результат, выполните такие команды:

```
$ mkdir output
$ geo2topo water=<(shp2json
```

```
50m_physical/ne_50m_rivers_lake_centerlines.shp) >water.json
$ geo2topo land=<(shp2json 50m_physical/ne_50m_land.shp) > land.json
$ geo2topo -n ne_50m_admin_0_boundary_lines_land=<(shp2json -n
50m_cultural/ne_50m_admin_0_boundary_lines_land.shp)
ne_50m_urban_areas=<(shp2json -n 50m_cultural/ne_50m_urban_areas.shp) >
cultural.json
```

Скопируйте получившиеся файлы данных в каталог `data/` проекта.

Рисование на картах

Рабочей лошадкой для визуализации географических данных является функция `d3.geoPath()`. Она аналогична генераторам путей SVG, о которых мы говорили выше, с тем отличием, что рисует географические данные и понимает, когда нужно рисовать линию, а когда область.

Чтобы изобразить сферические объекты, например планеты, на двумерной плоскости, `d3.geoPath()` пользуется проекциями. Существуют разные проекции для отображения различных аспектов данных, и внешний вид карты можно кардинально изменить, взяв другую проекцию или сместив центр проекции.

Нарисуем карту мира с центром в Европе. Возможность навигации на карте мы реализуем в главе 5.

Прежде всего установим файлы TopoJSON в свой проект.

```
$ npm install topojson --save
```

Теперь импортируем topojson в начале файла `lib/chapter4/index.js`.

```
import * as topojson from 'topojson';
```

Создадим новое исполняемое выражение для всего этого в файле `chapter4/index.js`:

```
const geoDemo = ((enabled) => {
  if (!enabled) return;
  const chart = chartFactory();
})(true);
```

После строки, начинающейся в `const chart`, определим в конструкторе географическую проекцию:

```
const projection = d3.geoEquirectangular()
  .center([8, 56])
  .scale(500);
```

Равнопромежуточная проекция – одна из ста с лишним проекций в пакете `d3-geoprojection`. Это, пожалуй, самая распространенная проекция, к которой мы привыкли еще в школе.

Равнопромежуточная проекция не сохраняет площадь и не очень хорошо представляет поверхность Земли. Споры о том, как лучше всего спроецировать сферу на двумерную поверхность, имеют давнюю историю, а огромный набор географических проекций в D3 – яркое свидетельство того, сколь разнообразные подходы применялись на протяжении веков: https://github.com/d3/d3-geo-projection.

В следующих двух строках определяются центр и масштаб карты. Методом проб и ошибок я подобрал все три значения: широта 8, долгота 56 и масштабный коэффициент 500. Повозитесь сами, чтобы получить другой вид.

Теперь загрузим данные:

```
const world = await Promise.all([
  (await fetch('data/water.json')).json(),
  (await fetch('data/land.json')).json(),
  (await fetch('data/cultural.json')).json(),
]);
```

Мы воспользовались обещаниями из ES2015, чтобы запустить все три операции загрузки одновременно. В каждом случае мы вызываем функцию `d3.json()` для загрузки и разбора данных; она либо отвергает (если имела место ошибка), либо выполняет обещание (если аргумент функции обработки ошибки не определен или равен `null`). Функция `Promise.all()` собирает все обещания и, когда определится исход каждого, возвращает результат `await`.

Перед тем как начать рисование, нам нужна еще функция, добавляющая объект на карту, чтобы уменьшить дублирование кода:

```
const addToMap = (collection, key) => chart.container.append('g')
  .selectAll('path')
  .data(topojson.feature(collection, collection.objects[key]).features)
  .enter()
  .append('path')
  .attr('d', d3.geoPath().projection(projection));
```

Эта функция принимает коллекцию объектов и ключ отображаемого объекта. Функция `topojson.object()` преобразует объект из формата TopoJSON в GeoJSON, ожидаемый функцией `d3.geoPath()`.

Что эффективнее – преобразовывать в формат GeoJSON или передавать данные сразу в конечном формате, зависит от конкретной ситуации. Для преобразования данных нужны вычислительные ресурсы, но передача нескольких мегабайтов вместо нескольких килобайтов может негативно отразиться на времени реакции приложения.

Наконец, мы создаем новый экземпляр `d3.geoPath()` и просим его использовать нашу проекцию. Помимо генерации строки пути SVG, `d3.geoPath()` умеет еще вычислять различные свойства географического объекта, например площадь (`.area()`) и ограничивающий прямоугольник (`.bounds()`).

Вот теперь можно приступить к рисованию:

```
const draw = (worldData) => {
  const [sea, land, cultural] = worldData;
  addToMap(sea, 'water').classed('water', true);
  addToMap(land, 'land').classed('land', true);
  addToMap(cultural, 'ne_50m_admin_0_boundary_lines_land').classed('boundary',
true);
  addToMap(cultural, 'ne_50m_urban_areas').classed('urban', true);
  chart.svg.node().classList.add('map');
};
```

Наша функция рисования принимает все три набора данных и перепоручает настоящую работу функции `addtoMap`. Здесь используется новая возможность ES2015 – деструктуризация аргумента, чтобы присвоить различным переменным значения элементов массива.

Что такое деструктуризация? На сайте Mozilla Developer's Network сказано, что деструктурирующее присваивание позволяет «выделить данные из массивов или объектов, применяя синтаксис, зеркально отражающий конструирование литеральных массивов и объектов». Эквивалентный код в ES5 выглядит так: `var land = values[0]; var sea = values[1]; var cultural = values[2]`. Дополнительные сведения о деструктуризации и ее пользе см. по адресу https://mdn.io/Destructuring_assignment.

Добавьте новые стили в файл `styles/index.css`:

```
.river {
  fill: none;
  stroke: #759dd1;
  stroke-width: 1;
}

.land {
  fill: #ede9c9;
  stroke: #7b5228;
  stroke-width: 1;
}

.boundary {
  stroke: #7b5228;
  stroke-width: 1;
```

```
  fill: none;
}

.urban {
  fill: #e1c0a3;
}

.map {
  background: #79bcd3;
}
```

И проверьте, что он импортирован в начале файла `lib/main.js`:

```
import '../styles/index.css';
```

Убедитесь, что сервер `webpack-dev-server` работает, и взгляните на свою страницу. Она должна выглядеть так:

Мы имеем медленно отрисовываемую карту мира с центром в Европе, на которой городские территории показаны темными пятнами.

Причин медленной работы много. Мы производим преобразование из TopoJSON в GeoJSON при каждом вызове `addToMap`. Мы используем слишком детальные данные для карты такого крупного масштаба и рисуем карту всего мира, только чтобы взглянуть на малую его часть. В данном случае мы пожертвовали скоростью ради гибкости. А можно было бы значительно повысить скорость отрисовки, уменьшив размер TopoJSON-файлов, для чего нужно было бы обработать их программой `toposimplify` из пакета `topojson-simplify`.

Я привел более подробное руководство по командным утилитам TopoJSON в статье https://medium.com/@aendrew/creating-topojson-using-d3-v4-10838d1a9538. А если вам нужно совсем уж детальное

описание всех инструментов, прямо от Майка Бостока, прочитайте серию его статей, начинающуюся на странице https://medium.com/@mbostock/command-line-cartography-part-1-897aa8f8ca2c.

Географические данные как основа

Говоря о географии, мы имеем в виду прежде всего рисование карт. Обычно карта – это основа, используемая для представления каких-то данных.

Обратимся к карте мировых аэропортов. А точнее, займемся еще более интересной задачей. Составим карту авиаперевозок лиц, задержанных в рамках процедуры экстрадиции, рейсами из США по инициативе ЦРУ. Для этого нам все-таки понадобятся сведения об аэропортах, поскольку в наборе данных правозащитной организации Rendition Project аэропорты представлены короткими кодами, а не широтой и долготой.

Первый шаг – скачать набор данных `airports.dat` по адресу http://openflights.org/data.html и данные о рейсах из США с сайта Rendition Project по адресу http://www.therenditionproject.org.uk/pdf/XLS%201%20-%20Flight%20data.%20US%20FOI%20resp.xls. Эти файлы можно найти в примерах на GitHub по адресу https://github.com/aendrew/learning-d3/blob/master/chapter4/data/airports.dat и https://github.com/aendrew/learning-d3/blob/master/chapter4/data/renditions.csv соответственно.

Откройте набор данных об экстрадициях в Excel и сохраните его как CSV-файл. Я уже это сделал и положил результат на GitHub. Понадобится также установить пакет `d3-dsv` для разбора CSV-файлов:

```
npm install d3-dsv --save
```

Теперь импортируйте из него функции `parseCSV` и `csvParseRows`, поместив следующую строку в начало файла `chapter4/index.js`:

```
import { csvParseRows, parseCsv } from 'd3-dsv';
```

Убедитесь, что файлы airports.dat и renditions.csv находятся в каталоге `data`, и добавьте еще два вызова в `Promise.all()`, чтобы загрузить их. После `draw()` добавьте обращение к `addRenditions()`:

```
addRenditions(
  await (await fetch('data/airports.dat')).text(),
  await (await fetch('data/renditions.csv')).text()
);
```

Здесь мы загружаем оба набора данных, а затем вызываем еще не написанную функцию `addRenditions` для их визуализации.

В функции addRenditions мы сначала преобразуем данные в объекты JavaScript: словарь аэропортов, индексированных идентификатором, а затем используем этот словарь для получения широты и долготы аэропортов вылета и назначения:

```
function addRenditions(airportData, renditions) {
  const airports = csvParseRows(airportData)
  .reduce((obj, airport) => {
    obj[airport[4]] = {
     lat: airport[6],
     lon: airport[7],
  };
  return obj;
}, {});
const routes = csvParse(renditions).map((v) => {
  const dep = v['Departure Airport'];
  const arr = v['Arrival Airport'];
  return {
    from: airports[dep],
    to: airports[arr],
  };
})
.filter(v => v.to && v.from)
.slice(0, 100);
}
```

Мы воспользовались функцией d3.csvParseRows для преобразования CSV-файлов в массивы, а затем вручную построили из них словари. Числовые индексы, к сожалению, не несут информации, но понять, что они означают, можно, взглянув на исходные данные:

```
1,"Goroka","Goroka","Papua New Guinea","GKA","AYGA",
-6.081689,145.391881,5282,10,"U"2,"Madang","Madang","Papua New
Guinea","MAG","AYMD",-5.207083,145.7887,20,10,"U"
```

Затем мы производим отображение всех рейсов с целью экстрадиции, так что получается просто словарь координат вылета и прибытия. Отфильтровываются элементы, в которых отсутствует поле to или from, что может случиться, если наша функция не смогла найти короткий код аэропорта. Кроме того, поскольку набор данных очень велик, то его визуальное изображение окажется неразборчивой кашей линий, поэтому мы оставили только первые 100 объектов, воспользовавшись методом Array.prototype.slice.

Теперь нарисуем сами прямые, пользуясь проекцией для преобразования широты и долготы в нечто, помещающееся на экране. Все в ту же функцию drawRenditions добавьте такой код:

```
chart.container.selectAll('.route')
  .data(routes)
  .enter()
  .append('line')
  .attr('x1', d => projection([d.from.lon, d.from.lat])[0])
  .attr('y1', d => projection([d.from.lon, d.from.lat])[1])
  .attr('x2', d => projection([d.to.lon, d.to.lat])[0])
  .attr('y2', d => projection([d.to.lon, d.to.lat])[1])
  .classed('route', true);
```

Маршруты не будут видны, пока мы не стилизуем их. Добавьте такие строки в файл `index.css`:

```
.route {
  stroke-width: 2px;
  stroke: goldenrod;
}
```

Результат показан на рисунке ниже:

Негусто. Всего-то один маршрут, изображенный желтой линией в середине рисунка. Наверное, масштаб слишком крупный. Исправим это. Вернитесь туда, где мы определяли проекцию, и задайте масштаб 200, а центр в точке –50, 56:

```
const projection = d3.geoEquirectangular()
  .center([-50, 56])
  .scale(200);
```

Ага, вот оно! Ну прямо как в интерактивных новостях!

Мы решили проблему *слишком большого числа маркеров* – при уменьшении масштаба карта оказывается забитой данными, – просто ограничив объем демонстрируемых данных. Пожалуй, это не самое удачное решение; было бы лучше либо кластеризовать данные (скажем, кодировать аэропорты вылета одним цветом, а аэропорты назначения – другим), либо включить интерфейсные элементы, позволяющие переключать аспекты набора данных. Интерактивностью мы займемся в последующих главах – готовьтесь!

Резюме

Вот и закончилась глава о данных!

Мы добрались до самой сути D3 – манипулирования данными. Мы рассмотрели различные виды управления потоком в JavaScript, выполнили столько обещаний, что всего даже не упомнишь, и преобразовали файлы из картографического формата в TopoJSON. Сейчас оценка вашего владения ES2015 приближается к 9000 (а то и выше).

Наконец, мы нанесли на карту некоторые данные, для чего потребовалось скомбинировать несколько наборов данных. Скоро вы поймете, что самые интересные визуализации получаются в результате объединения разных наборов данных, а вновь приобретенные навыки работы с обещаниями – путь к комбинированию любых данных на свете.

В следующей главе мы покажем, как с помощью анимации заставить наши данные жить своей жизнью. Будет по-настоящему интересно!

Глава 5

Все для удобства пользователя

Анимация – как соль. В небольшом количестве очень полезна и помогает лучше понять графику, привлекая внимание зрителя к важным аспектам содержания, но стоит переборщить – и все внимание будет обращено только на нее. Правильное взаимодействие с пользователем (User Experience – UX), т. е. идиомы работы с компьютером, которые вы используете в своих проектах, больше напоминают соус гуакамоле. Если он приготовлен правильно, то воспринимается как приятный легкий штрих, улучшающий общее качество продукта, и все довольны. Но плохой соус забивает собой все и безнадежно портит вкус буррито.

В этой главе мы обсудим анимацию и взаимодействие с пользователем, обращая особое внимание на улучшение качества визуализации данных. Мы также будем использовать поведения D3, чтобы сделать карту из предыдущей главы суперкрутой. Мы будем отмечать, когда анимация или интерактивность уместны, а когда – нет.

Способность библиотеки D3 нестандартно отображать данные – одна из главных причин ее использования; интерактивность и анимация позволяют не только показать, но и **объяснить** данные. От того, как устроено взаимодействие с пользователем в вашем интерфейсе, зависит, что вы создаете: *исследовательскую* графику, когда пользователю предоставляется доступ ко всем данным и возможность изменять способ отображения с помощью сортировки, фильтрации и т. п., или *объяснительную* графику, когда имеется минимальная интерактивность, направляющая пользователя в его изучении релевантных данных. На практике оба подхода обычно комбинируются, но четко

понимать, какой тип взаимодействия требуется обеспечить в данном месте, полезно при планировании проектов.

В этой главе мы рассмотрим различие между обоими подходами.

Анимация

Первый вопрос, на который следует ответить: «Почему анимация может улучшить данный проект?»

Если вы делаете что-то, предназначенное не для представления данных, а просто чтобы произвести впечатление на участников тусовки в помещении местного склада, то ответ «потому что это реально круто» вполне годится. И пожалуйста, сколько угодно ваяйте переливающиеся цвета радуги, расцвечивающие ваши диаграммы, если считаете это забавным (по собственному опыту знаю, что создание безумных анимированных картинок средствами D3 – неплохой способ провести субботний вечерок).

Но если вы все-таки отображаете данные, то стоить подойти к этому более ответственно. Каков характер данных? Если это значение, возрастающее со временем, то анимировать линию, поднимающуюся из левого нижнего угла в правый верхний, разумнее, чем медленно проявлять всю линию целиком.

Раньше мы задавали атрибуты различных объектов SVG, когда хотели, чтобы они появились, как только изображение будет полностью построено. Теперь же будем использовать анимацию, чтобы с помощью графики направлять внимание зрителя и помогать ему в интерпретации отображаемых данных. Для этого мы будем анимировать подходящие свойства объектов SVG.

Анимация с помощью переходов

В D3 переходы – один из способов анимации. Они основаны на хорошо известном принципе изменения атрибутов выборки, но только изменения растянуты во времени. Переходы, варианты динамики анимации и другие подобные вещи находятся в пакете `d3-transition`.

Чтобы медленно перекрасить все прямоугольники на странице в красный цвет, можно использовать такой код:

```
d3.selectAll('rect').transition().style('fill', 'red');
```

Мы запускаем новый переход методом `.transition()`, а затем определяем конечное состояние каждого анимируемого атрибута. По умолчанию каждый переход длится 250 миллисекунд, но этот промежуток

можно изменить методом `.duration()`. Переходы применяются одновременно ко всем свойствам, если не задать задержку методом `.delay()`.

Задержки удобны, когда требуется организовать последовательность переходов. Не будь задержки, все переходы выполнялись бы в одно и то же время, зависящее от внутреннего таймера. Простейший способ создать последовательность переходов – вложенные вызовы, тогда каждая задержка отсчитывается от непосредственно предшествующего ей перехода:

```
d3.select('rect')
  .style('fill', 'green')
  .transition()
  .delay(2000)
    .style('fill', 'red')
    .transition()
    .delay(1000)
      .style('fill', 'blue')
```

Здесь закрашивается выбранный зеленый прямоугольник: через две секунды он перекрашивается в красный цвет, а еще через секунду в синий.

Отсчет задержек от последнего перехода – важное изменение, по сравнению с версией D3 v3! Раньше все задержки отсчитывались от самого первого перехода в цепочке, поэтому координация несколько усложнялась.

Если вы хотите сделать что-то до начала перехода или желаете получить уведомление о его завершении, воспользуйтесь методом `.on()` соответствующего события. Добавьте показанный ниже код к написанному ранее:

```
.style('fill', 'red')
  .on('start', () => { console.log("Я краснею!"); })
  .on('end', () => { console.log("Я весь покраснел!"); })
```

Это довольно важное изменение в версии D3 v4; раньше для прослушивания событий переходов приходилось использовать метод `transition.each()`. В версии D3 v4 метод `transition.each()` работает точно так же, как `selection.each()`, т. е. принимает функцию обратного вызова, которой передается текущий элемент данных, его индекс и контекст; тем самым мы получаем возможность выполнить произвольный код для каждого элемента.

Это удобно, когда нужно произвести мгновенные изменения до или после перехода. Только не забывайте, что переходы выполняются

независимо и что нельзя рассчитывать на то, что вне текущей функции обратного вызова переходы находятся в каком-то определенном состоянии.

Интерполяторы

Для вычисления промежуточных значений в процессе перехода в D3 применяются функции-интерполяторы, отображающие отрезок `[0,1]` в целевой диапазон: цветов, чисел или строк. Это позволяет плавно перейти от начального значения к конечному, поскольку интерполятор возвращает промежуточные значения. Под капотом масштабы тоже реализованы с помощью интерполяторов.

Встроенные в D3 интерполяторы применимы чуть ли не к чему угодно, но чаще всего к числам и цветам, иногда к строкам. На первый взгляд, странно, но в действительности бывает весьма полезно. Чтобы D3 выбрала нужный интерполятор, мы просто пишем `d3.interpolate(a, b)`, а библиотека выбирает функцию, соответствующую типу аргумента `b`. Здесь `a` – начальное значение, `b` – конечное.

Если `b` – число, то `a` приводится к числовому типу и используется функция `.interpolateNumber()`. Следует избегать интерполяции с начальным или конечным нулевым значением, потому что числовые значения в конечном итоге преобразуются в строковые значения соответствующего атрибута, а для очень маленьких чисел может получиться представление в научной нотации. CSS и HTML не понимают, что такое 1e–7 (10 в степени –7), наименьшее допустимое число равно 1e–6.

Если `b` – строка, то D3 проверяет, совпадает ли она с одним из цветов CSS, и если да, то преобразует в соответствующий цвет. В этом случае `a` тоже преобразуется в цвет, и D3 вызывает `.interpolateRgb()` или другой интерполятор, соответствующий вашему цветовому пространству.

Еще более удивительные вещи происходят, когда строка не является названием цвета. D3 и здесь справляется! Встретив строку, D3 пытается разобрать ее как число, а затем вызывает `.interpolateNumber()` для каждого числового фрагмента. Это полезно для интерполяции смешанных определений.

Например, для перехода шрифтов можно написать такой код:

```
d3.select('svg')
  .append('text')
  .attr('x', 100)
  .attr('y', 100)
```

```
  .text("Я расту!")
  .transition()
    .styleTween('font', () =>
      d3.interpolate('12px Helvetica', '36px Comic Sans MS'));
```

Для определения перехода вручную мы воспользовались функцией `.styleTween()`. Это особенно полезно, когда нужно задать начальное значение перехода, не полагаясь на текущее состояние. Первый аргумент определяет, для какого атрибута стиля производится переход, а второй задает интерполятор.

Функцию `.tween()`, можно использовать и для других атрибутов, а не только для стиля.

Числовые части строки интерполируются от начального до конечного значения, а строковые переходят в конечное состояние сразу же. Интересное применение этой возможности – интерполяция определений пути: можно анимировать изменение формы фигуры. Ну не круто ли?

Помните, что интерполировать можно только строки с одинаковым числом и местоположениями контрольных точек (чисел внутри строки). Использовать интерполяторы для всего на свете не получится.

Можно создать и пользовательский интерполятор, для этого достаточно определить функцию, которая принимает один параметр `t` и возвращает начальное значение для `t = 0`, конечное – для `t = 1`, промежуточные – для всех остальных `t`.

Вот, например, как выглядит функция `interpolateNumber`, входящая в состав D3:

```
function interpolateNumber(a, b) {
  return function(t) {
    return a + t * (b - a);
  };
}
```

Как видите, ничего сложного.

Можно даже интерполировать целые массивы и объекты, получая тем самым составные интерполяторы нескольких значений. Скоро мы этим воспользуемся.

Динамика анимации

Динамика анимации (easing) модифицирует поведение интерполятора, управляя аргументом времени (`t`). Это применяется, чтобы

сделать анимацию более естественной, добавить эффект упругого отскока и т. д. В большинстве случаев динамика используется, чтобы избежать искусственности, возникающей при линейной интерполяции.

Давайте сравним функции динамики, включенные в D3, и посмотрим, что они делают. Мы будем помещать код в разные файлы, поскольку это поможет управлять им впоследствии, когда мы перейдем к расширению функциональности диаграмм. Сначала создайте файл `lib/chapter5/index.js` и поместите в него такой код:

```
import easingChart from './easingChart';
easingChart(true);
```

Затем создайте файл `lib/chapter5/easingChart.js` и добавьте в него новую функциональную обертку:

```
import * as d3 from 'd3';
import chartFactory from '../common';
function easingChart(enabled) {
  if (!enabled) return;
  const chart = chartFactory();
}
export default easingChart;
```

Далее нам понадобится массив функций, описывающих динамику, и масштаб для размещения их вдоль вертикальной оси. Добавьте следующий код после `const chart`:

```
const easings = ['easeLinear', 'easePolyIn(4)', 'easeQuadIn',
  'easeCubicIn', 'easeSinIn', 'easeExpIn', 'easeCircleIn', 'easeElasticIn(10,-5)',
  'easeBackIn(0.5)', 'easeBounceIn', 'easeCubicIn', 'easeCubicOut',
  'easeCubicInOut'];
const y = d3.scaleBand()
  .domain(easings)
  .range([50, 500]);
const svg = chart.container;
```

Обратите внимание, что функции `easePolyIn`, `easeElasticIn` и `easeBackIn` принимают аргументы; поскольку это просто строки, нам придется вручную превратить их в настоящие аргументы позже. Функция `easePolyIn` – полином, так что `easePolyIn(2)` совпадает с `easeQuadIn`, а `easePolyIn(3)` – с `easeCubicIn`. Для тех, кто перестал заниматься математикой уже в средних классах, напомню, что чем выше степень по-

линома, тем глубже кривая; например, `easePolyIn(4)` (совпадающая с `easeQuadIn`) характеризуется задержкой в начале, в конце или в обеих граничных точках в зависимости от того, как настроить динамику (см. фрагмент кода ниже). Чем выше степень, тем больше задержка. Экспериментируйте, делайте все, что вам нравится.

Функция `easeElasticIn` имитирует резинку, а два ее аргумента задают натяжение. Рекомендую поиграть с их значениями, пока не получите желаемого эффекта. Функция `easeBackIn` призвана имитировать парковку. Аргумент управляет величиной перехода за пределы требуемого положения.

Имена последних двух функций содержат суффиксы `out` и `inout`. Это означает следующее:

- `-In`: нормальное поведение;
- `-Out`: изменение направления динамики на противоположное;
- `-InOut`: на отрезке `[0, 0.5]` нормальное поведение, а на отрезке `[0.5, 1]` – зеркально отраженное;
- `-OutIn`: на отрезке `[0.5, 1]` нормальное поведение, а на отрезке `[0, 0.5]` – зеркально отраженное.

Такие суффиксы могут быть у любой функции динамики, пробуйте. Теперь нарисуем несколько кругов, анимированных с помощью каждой функции динамики:

```
easings.forEach((easing) => {
  const transition = svg.append('circle')
    .attr('cx', 130)
    .attr('cy', y(easing))
    .attr('r', (y.bandwidth() / 2) - 5)
    .transition()
    .delay(400)
    .duration(1500)
    .attr('cx', 400);
});
```

Мы в цикле обходим список динамик и на каждой итерации создаем круг; его вертикальную позицию задает масштаб `y()`, а радиус равен `y.bandwidth()`. Таким образом, мы можем легко добавлять и удалять примеры. Каждый переход начинается задержкой чуть меньше полсекунды, чтобы у нас было время понять, что происходит. Продолжительности перехода 1500 мс при конечной позиции 400 должно хватить для наблюдения за динамикой анимации.

Саму динамику мы определяем в конце функции, перед строкой });:

```
if (easing.indexOf('(') > -1) {
  const args = easing.match(/[0-9]+/g);
  const funcName = easing.match(/^[a-z]+/i).shift();
  const type = d3[funcName];
  transition.ease(type, args[0], args[1]);
} else {
  const type = d3[easing];
  transition.ease(type);
}
```

Этот код проверяет наличие скобок в строке, описывающей динамику, выделяет имя и аргументы функции и передает их функции `transition.ease()`. Без скобок `ease` означает просто тип динамики.

Отметим, что в версии D3 v4 используются символы, присоединенные к объекту D3, например `d3.easeCubicIn`. Раньше нужно было передавать строку функции `transition.ease()`, например `'cubicIn'`.

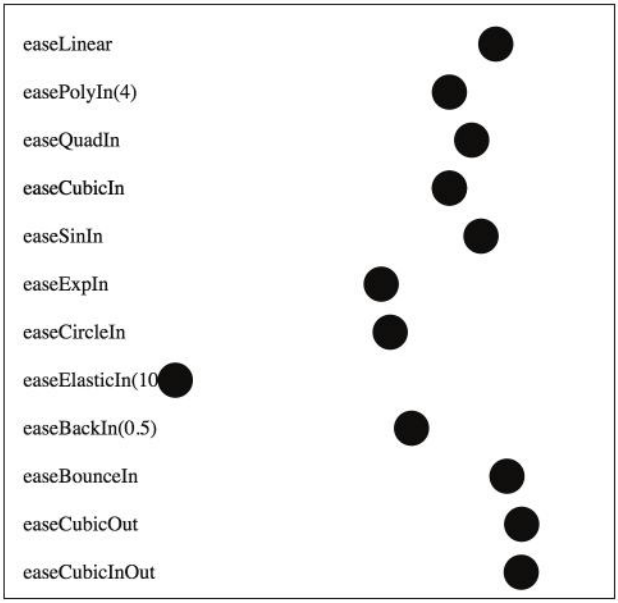
Добавим еще текст, чтобы можно было различать примеры:

```
svg.append('text')
  .text(easing)
  .attr('x', 10)
  .attr('y', y(easing) + 5);
```

Убедитесь, что сервер работает (если нет, выполните команду `$ npm start`), и зайдите на страницу `http://127.0.0.1:8080`.

Чтобы не повторять многократно одно и то же, я буду предполагать, что вы уже поняли, как импортировать и выполнять создаваемые нами модули, а также как запускать сервер разработки. Поэтому в дальнейшем эту часть я буду опускать.

Вот как выглядит сумятица точек:



На статическом рисунке невозможно показать анимацию, так что попробуйте сами запустить пример в браузере. Или посмотрите на кривые динамики на сайте http://easings.net/.

Динамика – изящный завершающий штрих, уместный в большинстве анимаций. В реальном мире постоянная скорость встречается редко, эвристическое правило состоит в том, чтобы подобрать для каждого элемента динамику, соответствующую его размеру и положению на странице. Иными словами, маленькие элементы должны двигаться быстрее больших, а интервалы замедления и ускорения в функциях с суффиксами `-in` и `-out` должны быть короче. Всегда важно думать о логичном перемещении предметов, а не просто вставлять куда ни попадя постепенное проявление с постоянной скоростью.

Таймеры

Для планирования переходов в D3 используются таймеры. Даже немедленный переход начинается с задержкой 17 мс.

D3 не приберегает таймеры только для себя, а предоставляет их и нам, чтобы мы могли не ограничиваться моделью перехода с двумя опорными кадрами. Для тех, кто незнаком с терминологией, применяемой в анимации, скажу, что опорные кадры определяют начало и конец плавного перехода.

Для создания таймера служит функция `d3.timer()`; она принимает функцию, задержку (в миллисекундах) и начальную временную метку. Начиная с момента, равного временной метке плюс задержка, функция повторно выполняется, пока не вернет `true`. Временная метка должна быть представлена в виде количества миллисекунд от «начала времен» в смысле Unix (эту величину возвращает `Date.getTime()`); если она не задана, то по умолчанию D3 берет `Date.now()`.

Давайте анимируем рисование параметрической функции по аналогии с детской игрушкой спирограф. Мы создадим таймер, дадим ему поработать несколько секунд, а миллисекундную метку будем использовать в качестве параметра функции.

Передайте значение `false` функции `easingChart()` в файле `chapter5/index.js` и добавьте в начало такой код:

```
import spirograph from './spirograph';
spirograph(true);
```

Затем создайте новый файл `lib/chapter5/spirograph.js` и поместите в него следующий код:

```
import * as d3 from 'd3';
import chartFactory from '../common';

function spirograph(enabled) {
  if (!enabled) return;

  const chart = chartFactory();
}
export default spirograph;
```

Мы построим параметрическое уравнение функции. Следующая функция основана на примере из статьи в Википедии по адресу http://en.wikipedia.org/wiki/Parametric_equations:

```
const position = (t) => {
  const a = 80;
  const b = 1;
  const c = 1;
```

```
  const d = 80;

  return {
    x: Math.cos(a * t) - Math.pow(Math.cos(b * t), 3),
    y: Math.sin(c * t) - Math.pow(Math.sin(d * t), 3),
  };
};
```

Эта функция возвращает координаты точки, вычисляемые по положительному возрастающему параметру. Поведение спирографа можно настроить, изменяя переменные `a`, `b`, `c`, `d`; примеры имеются в указанной выше статье из Википедии.

Эта функция возвращает координаты от –2 до 2; чтобы сделать ее график видимым на экране, нам понадобятся масштабы:

```
const tScale = d3.scaleLinear()
  .domain([500, 25000])
  .range([0, 2 * Math.PI]);

const x = d3.scaleLinear()
  .domain([-2, 2])
  .range([100, chart.width - 100]);

const y = d3.scaleLinear()
  .domain([-2, 2])
  .range([chart.height - 100, 100]);
```

Масштаб `tScale` преобразует время в параметры функции, а `x` и `y` вычисляют конечное положение изображения.

Далее нужно определить кисть `brush`, которая летает по экрану и рисует линии. Понадобится также переменная `previous` для хранения предыдущей позиции:

```
const brush = chart.container.append('circle').attr('r', 4);
let previous = position(0);
```

Теперь нужно определить функцию анимации `step`, которая перемещает кисть и рисует отрезок прямой между предыдущей и текущей точками:

```
const step = (time) => {
  if (time > tScale.domain()[1]) {
    return true;
  }

  const t = tScale(time);
  const pos = position(t);

  brush
    .attr('cx', x(pos.x))
```

```
    .attr('cy', y(pos.y));

  chart.container.append('line')
    .attr('x1', x(previous.x))
    .attr('y1', y(previous.y))
    .attr('x2', x(pos.x))
    .attr('y2', y(pos.y))
    .attr('stroke', 'steelblue')
    .attr('stroke-width', 1.3);
    previous = pos;
}
```

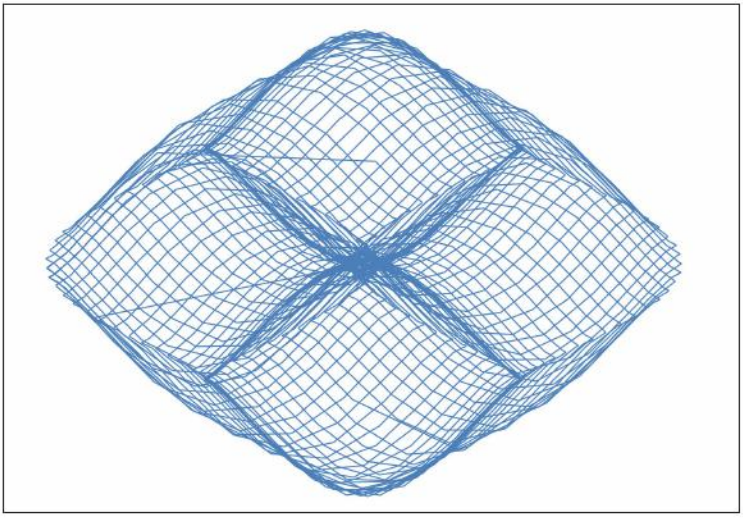
Первое условие останавливает таймер, когда текущее значение времени выходит за пределы области определения `tScale`. Затем мы с помощью `tScale()` преобразуем время в значение параметра и получаем новую позицию кисти.

Далее мы перемещаем кисть – никакого перехода нет, потому что мы делаем это самостоятельно, – рисуем синим цветом отрезок прямой между предыдущей и текущей позициями (`pos`).

В заключение переустанавливается предыдущая позиция. Осталось только создать таймер:

```
d3.timer(step, 500);
```

Вот и все. Спустя секунду после обновления страницы программа начнет рисовать кривую и закончит через 25 секунд. В итоге получится такая вот картина:



Соберем все вместе – последовательность анимаций

Пришло время с пользой применить полученные знания. Оставим ненадолго игрушки и создадим настоящую диаграмму.

Мы будем использовать набор данных о количестве заключенных в тюрьмах Великобритании за период с 1900 по 2015 год, доступный в виде файла `data/uk_prison_population_1900-2015.csv` в репозитории книги. На основе этого набора я создал серию аналогичных диаграмм для газеты *Times* в 2016 году, причем моя группа испробовала несколько вариантов, прежде чем остановилась на варианте взаимодействия, с которым мы познакомимся в следующем разделе.

Закомментируйте весь код в файле `chapter5/index.js` и добавьте в начало вот что:

```
import prisonChart from './prisonChart';
(async (enabled) => {
  if (!enabled) return;
  await prisonChart.init();
  const data = prisonChart.data;
  function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
  }
  const randomChart = () => {
    try {
      const from = getRandomInt(0, data.length - 1);
      const to = getRandomInt(from, data.length - 1);
      prisonChart.update(data.slice(from, to));
    } catch (e) {
      console.error(e);
    }
  };
  prisonChart.update(data);
  setInterval(randomChart, 5000);
})(true);
```

Мы создадим диаграмму, на которой будет случайным образом отображаться подмножество данных с анимацией переходов между состояниями. На реальном сайте этого делать не стоит, но подобные тесты полезны, когда нужно удостовериться, что анимация работает правильно, какие бы данные ей ни подсунуть. Мы еще не написали функцию `prisonChart.init()`, но ждать осталось недолго.

Создайте файл `lib/chapter5/prisonChart.js` и поместите в него следующий код:

```
import * as d3 from 'd3';
import { csvParse } from 'd3-dsv';
import chartFactory from '../common';

const prisonChart = chartFactory({
  margin: { left: 50, right: 0, top: 50, bottom: 50 },
  padding: 20,
  transitionSpeed: 500,
});
```

Мы создаем диаграмму с помощью нашей доброй старой функции `chartFactory`, задав для нее поля и некоторые значения по умолчанию.

Далее нужно определить методы нового объекта:

```
prisonChart.resolveData = async function resolveData() {
  return csvParse(await (await
    fetch('data/uk_prison_data_1900-2015.csv')).text());
};
```

Здесь мы создали асинхронный метод получения данных. Он просто скачивает файл в виде текстовой строки и разбирает его средствами пакета `d3-dsv`.

Далее создадим еще одну асинхронную функцию для инициализации масштабов. Она сделана асинхронной, чтобы имела возможность дожидаться выполнения обещания, созданного `resolveData()`:

```
prisonChart.init = async function init() {
  this.data = this.data || await this.resolveData();
  this.innerHeight = () =>
    this.height - (this.margin.bottom + this.margin.top + this.padding);
  this.innerWidth = () =>
    this.width - (this.margin.right + this.margin.left + this.padding);

  this.x = d3.scaleBand()
    .range([0, this.innerWidth()])
    .padding(0.2);

  this.y = d3.scaleLinear()
    .range([this.innerHeight(), 0]);

  this.x.domain(this.data.map(d => d.year));
  this.y.domain([0, d3.max(this.data, d => Number(d.total))]);

  this.xAxis = d3.axisBottom().scale(this.x)
    .tickValues(this.x.domain().filter((d, i) => !(i % 5)));
  this.yAxis = d3.axisLeft().scale(this.y);

  this.xAxisElement = this.container.append('g')
    .classed('axis x', true)
```

```
    .attr('transform', &grave;translate(0, ${this.innerHeight()})&grave;)
    .call(this.xAxis);

  this.yAxisElement = this.container.append('g')
    .classed('axis y', true)
    .call(this.yAxis);

  this.barsContainer = this.container.append('g')
    .classed('bars', true);
};
```

Вы, наверное, обратили внимание, что мы здесь используем не анонимную функцию с двойной стрелкой, а развернутый синтаксис с ключевыми словами `function` и `this`. Когда применяется нотация с двойной стрелкой, контекст `this` ссылается на родительский блок, а нам это не нужно, потому что мы хотим задать внутренние свойства объекта `prisonChart`. Это окажется полезно, когда впоследствии мы захотим расширить диаграмму.

По существу, в показанном выше коде просто задаются масштабы и оси, а затем запоминаются во внутреннем состоянии объекта с помощью `this`. Кроме того, мы создаем контейнер для столбиков и запоминаем ссылку на него в переменной `this.barsContainer`. Но самое главное – мы вызываем функцию `resolveData` для получения данных и присваиваем их свойству `data` объекта `prisonChart`.

Далее мы сделаем то, чего не делали раньше: создадим отдельный метод для обновления данных в диаграмме. Когда ниже в этой главе мы займемся реализацией разных интересных интерактивных операций, этот метод будет использоваться для изменения состояния диаграммы.

Я представлю этот метод по частям, он опирается на полученные ранее знания о соединении с данными. Сначала создадим функцию и присвоим ее свойству объекта `prisonChart`:

```
prisonChart.update = function update(_data) {
  const data = _data || this.data;
  const TRANSITION_SPEED = this.transitionSpeed;
};
```

Здесь просто задается константа, которая понадобится ниже при создании переходов. Мы также завели локальную переменную `data`, в которую поместим либо подмножество данных, переданное в качестве аргумента, либо – если этот аргумент отсутствует – данные, сохраненные во внутреннем состоянии объекта функцией `init()`.

Далее добавьте в функцию `update()` следующие строки:

```
// Обновление
const bars = d3.select('.bars').selectAll('.bar');
const barsJoin = bars.data(data, d => d.year);

// Обновление масштабов
this.x.domain(data.map(d => +d.year));
this.xAxis.tickValues(this.x.domain()
  .filter((d, i, a) => (a.length > 10 ? !(i % 5) : true)));

// Вызвать this.xAxis после удвоения тайм-аута TRANSITION_SPEED
d3.timeout(this.xAxis.bind(this, this.xAxisElement), TRANSITION_SPEED * 2);
```

Здесь мы выбираем все элементы класса `.bar` в контейнере `.bars` и сохраняем их в локальной переменной `bars`. Затем мы соединяем эту переменную с набором данных и сохраняем результат в переменной `barsJoin`. Обратите внимание на второй аргумент метода `selection.data()`, который говорит D3, какие части данных соответствуют каждому элементу массива; обычно D3 считает ключами объектов индексы массива, но когда в массив добавляются новые элементы и удаляются старые, полагаться на индексы нельзя. Мы обновляем область определения масштаба `x`, так чтобы она включала все годы, представленные в массиве, и настраиваем ось x так, что в случае, когда число элементов в массиве больше 10, отображается каждое пятое деление. Затем конфигурируется таймер – он должен сработать по прошествии времени, равного удвоенной скорости перехода, в результате чего произойдет обновление оси x (поясним, что вызов `this.axisElement.call(this.xAxis)` обновляет ось немедленно, а вызов `this.xAxis.bind(this, this.xAxisElement)` возвращает функцию, которая обновит ось, когда будет вызвана, – таким образом, мы выходим из `d3.timeout`, а функция запустится, когда сработает таймер).

Продолжаем – добавьте в `prisonChart.update()` следующий код:

```
// Удаление
barsJoin.exit()
  .transition()
  .duration(TRANSITION_SPEED)
  .attr('height', 0)
  .attr('y', this.y(0))
  .remove();
```

Здесь мы настраиваем состояние `exit()` нашего соединения с данными. Мы задаем новый переход длительностью `TRANSITION_SPEED` и говорим, что в конце перехода высота каждого столбика должна быть равна 0, а значение `y` – лежать на оси x. В результате все столбики сожмутся и покинут сцену.

Далее настроим состояние `enter()`:

```
const newBars = barsJoin.enter()
  .append('rect')
  .attr('x', d => this.x(+d.year))
  .classed('bar', true);
```

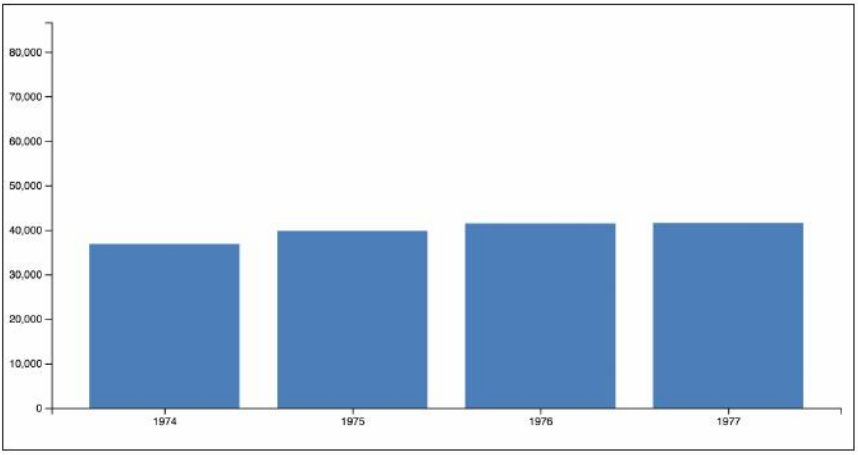
Здесь в локальную переменную записываются новые столбики, созданные в процессе соединения с данными. Мы задаем для них значение `x` и класс `bar`.

И последний фрагмент `prisonChart.update()`, в котором обрабатываются как новые, так и оставшиеся старые столбики:

```
barsJoin.merge(newBars) // Обновление
  .transition()
  .duration(TRANSITION_SPEED)
  .attr('height', 0)
  .attr('y', this.y(0))
  .transition()
  .attr('x', d => this.x(+d.year))
  .attr('width', this.x.bandwidth())
  .transition()
  .attr('x', d => this.x(+d.year))
  .attr('y', d => this.y(+d.total))
  .attr('height', d => this.innerHeight() - this.y(+d.total));
```

Здесь мы пользуемся появившейся в версии D3 v4 функцией `merge()`, чтобы обновить существующие столбики. Мы объединяем выборки старых и новых столбиков и выполняем операции над теми и другими сразу. Сначала инициализируется переход, длящийся один интервал, в течение которого все столбики сжимаются до нулевой высоты и стягиваются к оси x – то же самое мы проделали при удалении столбиков. Затем начинается новый переход, длящийся 250 мс (значение по умолчанию), который перемещает все столбики в новые положения и устанавливает их ширину. И последний переход устанавливает высоту столбиков.

Обновите страницу и полюбуйтесь на гиперактивную, быстро обновляющуюся диаграмму:



Столбики растут, данные изменяются, столбики сжимаются, оси обновляются, и столбики снова растут. Восхитительно!

Это все, что мы хотели сказать об анимации с помощью переходов D3. Далее мы обсудим, как добавить взаимодействие, применяя поставляемые в составе D3 поведения.

Есть ли другие способы анимации в SVG?
Хороший вопрос!
В самом начале был язык SMIL (Synchronized Multimedia Integration Language). Это подразумевало использование тегов `<animate>`, вставляемых в SVG-разметку. Если вам уже кажется, что это чересчур, то вы не одиноки. По счастью, в версии Chrome 45 эта технология объявлена нерекомендуемой, а стало быть, тратить время на ее изучение нет смысла. Неудержимо любознательные могут зайти на страницу https://mdn.io/SVG_animation_with_SMIL.
Еще один способ анимации – переходы CSS. Для простых анимаций с ними ничто не сравнится, поскольку они могут использовать графический процессор для отрисовки, а это означает отсутствие блокировок и повышенное быстродействие. В предыдущем издании этой книги я ими и пользовался. Но переходы CSS не работают с холстом, и их трудно выстроить в последовательность, если в проекте используется также пакет `d3-transition`. Поэтому лучше оставить их для таких вещей, как анимация кнопок при наведении мыши и других анимаций пользовательского интерфейса.

Дополнительные сведения см. на странице https://developer.mozilla.org/Web/CSS/CSS_Transitions/Using_CSS_transitions.

В будущем должен появиться еще один способ анимации с помощью JavaScript – Web Animation API. Пока что эта технология не поддерживается продуктами Microsoft и Apple, но в Chrome она включена, начиная с версии 36, а в Firefox – начиная с версии 41. У нее очень удобный синтаксис, основанный на JavaScript, и она прекрасно работает совместно с D3. Вот пример:

```
d3.selectAll('.bar').each(function(d, i){
 this.animate([
   {transform: &grave;translate(${x(i)}, ${y(d)})&grave;}
 ],{
     duration: 1000,
     iterations: 5,
     delay: 100
 });
});
```

Увы! Как всегда, веб-разработка – сундук, полный сокровищ, вот только надежного способа воспользоваться ими прямо сейчас не существует. Если хотите поэкспериментировать с технологией Web Animation до включения ее в стандарт, скачайте фантастический полифил по адресу https://github.com/web-animations/web-animations-js – он позволяет использовать веб-анимацию в большинстве современных браузеров. Но, повторюсь, пока она находится в младенчестве; если вам удастся использовать Web Animation API и D3 в одном проекте, черкните мне и расскажите о своих успехах!

Взаимодействие с пользователем

Добрались. Именно здесь сходятся все намеки на пользовательский интерфейс, разбросанные в этой главе, и все описанные выше идеи функционального программирования. Сейчас мы создадим простую объяснительную графику, которая интерактивно направляет пользователя в осмыслении данных.

Первый шаг разработки любой интерактивной визуализации – четко спланировать, для чего предназначена визуализация, как пользователи будут с ней взаимодействовать и что именно вы хотите сказать о данных. В чем смысл истории и как ее лучше подать?

В наборе данных о тюрьмах мы имеем числовое представление сведений о динамике количества заключенных за сто с лишним лет в одной западной стране. Взглянуть на эти данные можно разными

способами. Можно сопоставить рост числа заключенных с общим ростом населения или посмотреть, как оно коррелирует с известными историческими событиями. Зачастую одной диаграммы недостаточно; когда я работал с этими данными для газеты *Times*, в материале было целых пять диаграмм и одна карта, и читатель переходил от одной графической вставки к другой. Обдумывая подобные сложные последовательные взаимодействия, полезно записать план в виде перечня пунктов или, быть может, нарисовать блок-схему и только потом приступать к кодированию. Настоящая работа нередко начинается задолго до написания первой строки кода на JavaScript.

В данном случае, имея данные за сто лет, мы можем рассмотреть несколько важных исторических событий. Графика будет состоять из пяти частей, а переход от одной к другой будет осуществляться нажатием кнопки.

- **Начало.** 1900–2015. Дает общее представление о росте числа заключенных с течением времени.
- **Период 1900–1930.** Выделены годы с 1914 по 1918. В тексте объясняется, что число заключенных выросло в связи с окончанием Первой мировой войны.
- **Период 1930–1960.** Выделены годы с 1939 по 1945. В тексте объясняется, почему число заключенных выросло после Второй мировой войны.
- **Период 1960–1990.** Обсуждаются возникновение общества потребления и его влияние на преступность.
- **Период 1990–2015.** Выделен 1993 год, и объясняется резкий рост числа заключенных после убийства Джеймса Балджера, давшего политикам повод увеличить тяжесть наказания.

Мы сознательно стремились к простоте пользовательского интерфейса, но помните, что и в общем случае чем проще, тем обычно и лучше, особенно когда создается интерфейс для мобильной аудитории (например, использовать ползунки на сенсорном устройстве гораздо труднее, чем кнопки).

Основы взаимодействия

Как и всё в стране под названием JavaScript, принцип взаимодействия прост – присоединить прослушиватель события к элементу и сделать что-то, когда он сработает. Для добавления и удаления прослушивателей служит метод выборки `.on()`, которому передаются тип события (например, щелчок мышью) и функция-прослушиватель, вызываемая в момент возникновения события.

Можно установить флаг захвата, тогда наш прослушиватель будет вызван первым, а все остальные будут ждать, пока он завершится. События, всплывающие от дочерних элементов, не будут приводить к вызову нашего прослушивателя.

Мы можем быть уверены, что для каждого события данного элемента существует только один прослушиватель, потому что старые удаляются при добавлении новых. Это очень помогает избегать неожиданного поведения.

Как и другие функции, применяемые к выборкам элементов, прослушиватели событий получают текущий элемент и индекс и устанавливают контекст `this` на элемент DOM. Глобальная переменная `d3.event` дает доступ к самому объекту события.

Создадим функцию, которая будет добавлять кнопки, и свяжем все воедино. Сначала закомментируйте весь код в файле `lib/chapter5/index.js` и добавьте такой:

```
import buttonPrisonChart from './buttonChart';
  (async (enabled) => {
    if (!enabled) return;
    await buttonPrisonChart.resolveData();
    await buttonPrisonChart.init();
    buttonPrisonChart.addUIElements();
})(true);
```

Это похоже на упрощенную версию исполняемого выражения в файле `index.js`. Мы дожидаемся получения данных, а затем выполняем инициализацию.

Создайте новый файл `lib/chapter5/buttonChart.js` и поместите в него такой код:

```
import * as d3 from 'd3';
import scenes from '../../data/prison_scenes.json';
import PrisonPopulationChart from './prisonChart';

const buttonPrisonPopulationChart = Object.create(PrisonPopulationChart);
buttonPrisonPopulationChart.scenes = scenes;
buttonPrisonPopulationChart.addUIElements = function addUI() {}
```

Мы импортируем D3, JSON-файл, содержащий описания каждой части диаграммы, и последнюю диаграмму. Вместо того чтобы создавать новую диаграмму с помощью фабрики `chartFactory`, мы создали объект методом `Object.create()`, указав в качестве прототипа `prisonChart`. Это означает, что нам доступны все методы и свойства `prisonChart`, и мы можем взаимодействовать с внутренним API этого объекта, добавляя новые методы или переопределяя существующие.

Мы также создали функцию `addUIElements()`, на которую раньше ссылались в `index.js`.

Добавьте в функцию `addUIElements()` такой код:

```
// Оставить место для кнопок
this.height -= 100;

// Необходимо обновить масштаб и ось y
this.y.range([this.innerHeight(), 0]);
this.yAxisElement.call(this.yAxis);

// ... и то  же для масштаба и оси x
this.xAxisElement.attr('transform', &grave;translate(0,
    ${this.innerHeight()})&grave;);
```

Мы задали высоту на 100 пикселей меньше, чем в последней диаграмме (где использовалась вся высота экрана), и соответственно изменили масштабы и оси. Далее добавьте такой код:

```
this.buttons = d3.select('body')
  .append('div')
  .classed('buttons', true)
    .selectAll('.button');

this.buttons.data(this.scenes)
  .enter()
  .append('button')
  .classed('scene', true)
  .text(d => d.label)
  .on('click', d => this.loadScene(d))
  .on('touchstart', d => this.loadScene(d));

this.words = d3.select('body').append('div');
this.words.classed('words', true);
this.loadScene(this.scenes[0]);
}; // Конец метод addUI()
```

Здесь на страницу добавлены кнопки, и обработчики событий `click` и `touchstart` установлены так, чтобы вызывался метод загрузки сцены.

Мы также создали элемент `div`, в котором будут размещены описания, и сохранили ссылку на него в свойстве `words`. И напоследок мы загрузили первую часть диаграммы, в которой ничего не выделено.

С функцией `addUIElements()` покончено. Пора добавить еще несколько функций – для выборки столбиков и для сброса выборки.

```
buttonPrisonPopulationChart.clearSelected = function clearSelected() {
  d3.timeout(() => {
    d3.selectAll('.selected').classed('selected', false);
```

```
  }, this.transitionSpeed);
};

buttonPrisonPopulationChart.selectBars = function selectBars(years) {
  this.clearSelected();
  d3.timeout(() => {
    d3.select('.bars').selectAll('.bar')
      .filter(d => years.indexOf(Number(d.year)) > -1)
      .classed('selected', true);
  }, this.transitionSpeed);
};
```

В функции `clearSelected()` мы просто удаляем класс `selected` из всех столбиков, а в функции `selectBars()` назначаем этот класс столбикам, соответствующим переданным в качестве аргумента годам. Столбики выбираются внутри вызова `d3.timeout()`, так что мы синхронизировались с переходами столбиков в функции `prisonChart.update()`.

И наконец, напишем метод `loadScene()`:

```
buttonPrisonPopulationChart.loadScene = function loadScene(scene) {
  const range = d3.range(scene.domain[0], scene.domain[1]);
  this.update(this.data.filter(d => range.indexOf(Number(d.year)) > -1));
  this.clearSelected();

  if (scene.selected) {
    const selected = scene.selected.range
      ? d3.range (...scene.selected.range) : scene.selected;
    this.selectBars(selected);
}

  this.words.html(scene.copy);

  d3.selectAll('button.active').classed('active', false);
  d3.select((d3.event && d3.event.target) ||
    this.buttons.node()).classed('active', true);
};
```

Здесь мы сначала вычисляем диапазон столбиков, которые затем передадим функции `update` для установки новых столбиков и осей. Затем, если у выбранной части есть свойство `range`, мы получаем массив значений методом `d3.range`, в противном случае предполагаем, что выбранная часть и есть массив значений, подлежащих выборке, и в любом случае передаем этот массив методу `selectBars()`. Если выбранных столбиков нет, то этот метод очистит все ранее выбранные столбики. В качестве описания мы устанавливаем копию текста из JSON-файла, делаем все кнопки неактивными, а в самом конце делаем активной выбранную кнопку.

И наконец, экспортируем нашу диаграмму:

```
export default buttonPrisonPopulationChart;
```

Оформим HTML-макет с помощью CSS. Создайте файл `lib/chapter5/prisonChart.css` и поместите в него такой код:

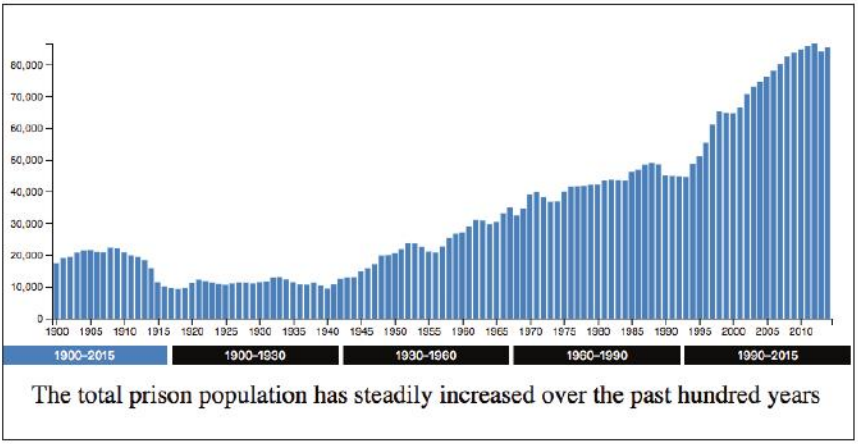
```
.selected {
  fill: red;
}

.buttons {
  display: flex;
}

.buttons button {
  flex-grow: 1;
  color: white;
  background: black;
  border: 2px solid white;
  transition: .5s background;
}

.buttons button.active {
  background: steelblue;
}

.words {
  padding: .5em;
  text-align: center;
  font-size: 1.5em;
}
```

В результате макет растягивается на всю страницу по ширине и добавляется CSS-переход при выборе кнопки. Импортируйте этот файл в файл `prisonChart.js`, добавив в начало последнего строку:

```
import './prisonChart.css';
```

Как и раньше, мы используем загрузчики CSS-таблиц из Webpack, чтобы импортировать стили в JavaScript-модуль.

Диаграмма должна выглядеть следующим образом:



Вот и готова ваша первая интерактивная визуализация данных!

Поведения

В предыдущем разделе мы продемонстрировали *объяснительную* графику, в которой взаимодействие использовалось, чтобы направить пользователя в его ознакомлении с данными. Но часто наша цель состоит в том, чтобы сделать интерактивным сам набор данных и предоставить пользователю возможность манипулировать им. Такой подход называется *исследовательской* графикой.

Поведение D3 позволяет сэкономить уйму времени при настройке сложных взаимодействий с диаграммой. Кроме того, они учитывают различия между устройствами ввода, так что один раз реализованное поведение будет работать и с мышью, и с сенсорным устройством. В настоящее время поддерживаются два поведения: буксировка и масштабирование.

Буксировка

Что, если вместо нажатия кнопок в предыдущем примере мы дадим пользователю возможность просто буксировать область диаграммы, чтобы увидеть, как менялось количество заключенных в тюрьмах Ве-

ликобритании? Это потребует от пользователя чуть больше усилий, зато позволит свободно перемещаться по диаграмме, что в некоторых случаях желательно.

Еще раз расширим объект `prisonChart`. Закомментируйте весь код в файле `chapter5/index.js` и добавьте такой:

```
import draggablePrisonChart from './draggableChart';
(async (enabled) => {
  if (!enabled) return;
  await draggablePrisonChart.init();
  draggablePrisonChart.addDragBehavior();
})(true);
```

Теперь создайте в каталоге `lib/chapter5` новый файл `draggableChart.js`:

```
import * as d3 from 'd3';
import PrisonPopulationChart from './prisonChart';

const draggablePrisonPopulationChart = Object.create(PrisonPopulationChart);
draggablePrisonPopulationChart.addDragBehavior = function addDrag() {};
```

Мы уже пользовались подобным способом запуска диаграммы. В функцию `addDrag` поместите такой код:

```
  this.x.range([0, this.width * 4]);
  this.update();
  const bars = d3.select('.bars');
  bars.attr('transform', 'translate(0,0)');
```

В качестве области значений масштаба `x` мы задаем учетверенную ширину экрана, обновляем диаграмму, чтобы в ней появился начальный рисунок и настроенные оси. Мы также задаем начальное значение сдвига для столбиков, впоследствии оно понадобится нам при буксировке диаграммы.

Вы, вероятно, заметили неприятное мигание при обновлении осей. Избавиться от него можно несколькими способами – изменить функцию `prisonChart`, так чтобы она вызывала функции настройки осей вне метода `init()`, или с помощью CSS скрыть диаграмму на начальном этапе и сделать ее видимой только после обновления осей. Оставляю это в качестве упражнения для читателя.

Далее добавьте в функцию `addDrag` такой код:

```
  const dragContainer = this.container.append('rect')
    .classed('bar-container', true)
    .attr('width', this.svg.node().getBBox().width)
```

```
    .attr('height', this.svg.node().getBBox().height)
    .attr('transform', &grave;translate(${this.margin.left}, ${this.margin.
top})&grave;)
    .attr('x', 0)
    .attr('y', 0)
    .attr('fill-opacity', 0);

  const xAxisTranslateY =
    d3.select('.axis.x').node().transform.baseVal[0].matrix.f;
```

Мы добавляем новый невидимый элемент поверх всего и получаем величину сдвига по оси x, чтобы не изменять координату Y при буксировке. Для этого мы делаем ось x групповым элементом, а затем получаем свойства элемента `transform.baseVal[0].matrix`. Свойство `e` соответствует сдвигу по оси y, а свойство `f` – сдвигу по оси x.

Раньше существовала удобная функция `d3.transform`, которая существенно упрощала эту процедуру. Увы, в версии D3 v4 она пропала.

Далее мы настраиваем само поведение буксировки и вызываем его:

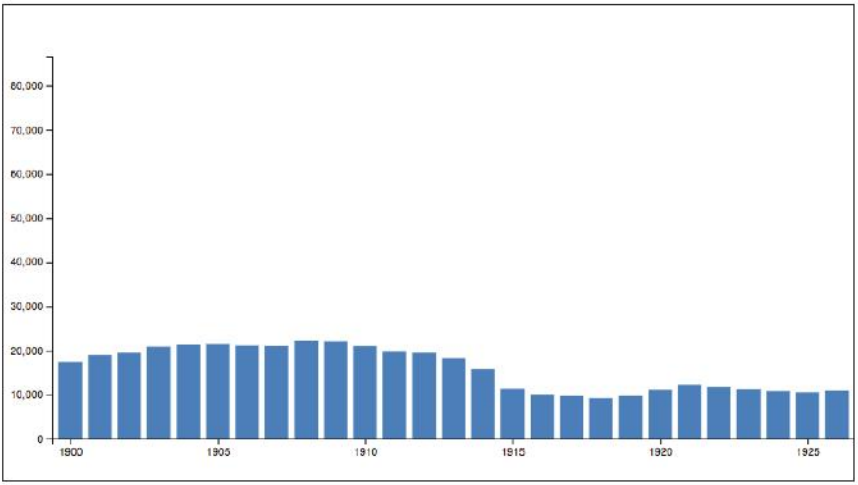
```
  const drag = d3.drag().on('drag', () => {
    const barsTranslateX = bars.node().transform.baseVal[0].matrix.e;
    const barsWidth = bars.node().getBBox().width;
    const xAxisTranslateX = d3.select('.axis.x').node()
      .transform.baseVal[0].matrix.e;
    const dx = d3.event.dx;

    if (barsTranslateX + dx < 0 && barsTranslateX + dx > -barsWidth + this.
innerWidth()) {
      bars.attr('transform', &grave;translate(${barsTranslateX + dx},
0)&grave;);
      d3.select('.axis.x').attr('transform', &grave;
translate(${xAxisTranslateX + d3.event.dx},
${xAxisTranslateY})&grave;);
    }
  });
  dragContainer.call(drag);
```

Мы присоединяем поведение буксировки к элементу `dragContainer`. Когда пользователь буксирует этот элемент, возникает событие перетаскивания, в котором мы сдвигаем столбики и ось x по горизонтали. У объекта `d3.event` имеются свойства `dx` и `dy` – соответственно расстояния буксировки по горизонтали и по вертикали. Предложение `if` служит для того, чтобы не проскочить конец или начало области, занятой столбиками.

Вне функции `addDrag()` добавьте строку для экспорта диаграммы:

```
export default draggablePrisonPopulationChart;
```

Сохранитесь, обновите страницу – и перед вами появится новая диаграмма:



Кисти

Поведение кисти аналогично буксировке, но используется оно для запоминания выборок. Вместо перемещения объектов кисть рисует прямоугольник и запоминает все попавшие в него объекты. Документация имеется в составе пакета `d3-brush`.

Для создания новой кисти мы вызываем функцию `d3.brush()`, а затем применяем кисть к элементу. Она порождает события `start`, `brush` и `end`, обработчику каждого из которых передаются объекты `selection`, `sourceEvent` и `target` в составе `d3.event`.

Пора перейти к примеру!

Вернемся к нашей навороченной диаграмме количества заключенных. Обещаю, это в последний раз. Мы дадим пользователю возможность увеличить масштаб группы столбиков в два приема: сначала выбрать их кистью, а затем масштабировать правой кнопкой мыши. Закомментируйте весь код в файле `chapter5/index.js` и добавьте показанный ниже – как и в предыдущих примерах:

```
import brushableChart from './brushableChart'; (async (enabled) => {
  if (!enabled) return;
```

```
  await brushableChart.init();
  brushableChart.addBrushBehavior();
})(true);
```

Создайте файл `lib/chapter5/brushableChart.js` и добавьте в него такой код:

```
import * as d3 from 'd3';
import interactivePrisonChart from './buttonChart';

const brushablePrisonPopulationChart =
Object.create(interactivePrisonChart);
brushablePrisonPopulationChart.addBrushBehavior = function () {
  this.brush = d3.brush();
  this.container.append('g')
    .classed('brush', true)
    .call(this.brush
      .on('brush', this.brushmove.bind(this))
      .on('end', this.brushend.bind(this)));

  this.update();
};
```

Первая часть такая же, как и раньше. Затем метод `addBrushBehavior()` настраивает поведение кисти. Мы создаем новый групповой элемент для кисти, добавляем его в диаграмму и присоединяем к нему прослушивателей событий. Затем вызывается метод `update()` для построения диаграммы.

Создайте новую функцию после `addBrushBehavior()`:

```
brushablePrisonPopulationChart.brushmove = function () {
  const e = d3.event.selection;
  if (e) {
    d3.selectAll('.bar').classed('selected',
       d => e[0][0] <= this.x(d.year) && this.x(d.year) <= e[1][0]
    );
  }
};
```

Здесь мы получаем размеры и координаты области, обведенной кистью. Объект `d3.event.selection` содержит два массива: в одном находятся координаты точки, где кисть начала движение, в другом – точки, где движение закончилось. Мы анализируем значения x для каждой точки, чтобы получить диапазон выбранных кистью столбиков, а затем добавляем к каждому из них класс `.selected`.

По завершении рисования кистью возникает событие `end`, его обрабатывает показанная ниже функция:

```
brushablePrisonPopulationChart.brushend = function () {
  if (!d3.event.sourceEvent) return;// переход только после ввода
  if (!d3.event.selection) return;// игнорируем пустые выборки
  const selected = d3.selectAll('.selected');
  const data = selected.data();

  // Очистить объект кисти
  d3.select('g.brush').call(d3.event.target.move, null);

  // Увеличить масштаб выбранных столбиков
  if (data.length >= 2) {
    const start = data.shift();
    const end = data.pop();

    this.update(this.data.filter(d =>
      d3.range(start.year, end.year + 1).indexOf(Number(d.year)) > -1));

    const hitbox = this.svg
      .append('rect')
      .classed('hitbox', true)
      .attr('width', this.svg.attr('width'))
      .attr('height', this.svg.attr('height'))
      .attr('fill-opacity', 0);

    hitbox.on('contextmenu', this.rightclick.bind(this));
  }
};
```

Тут много интересного. Сначала мы очищаем очерченную кистью область, вызвав метод `brush.move()` со вторым аргументом `null`, затем находим первый и последний элементы, выбранные кистью. Из них мы извлекаем начальный и конечный годы, которые передаем методу `d3.range`, и перерисовываем диаграмму, как в функции `buttonChart`. И наконец, мы добавляем элемент `hitbox`, который используется для прослушивания события `contextmenu` (это аналог события `click` для щелчка правой кнопкой мыши).

И еще нам необходим обработчик события щелчка правой кнопкой мыши по диаграмме. Добавьте в конец файла `brushableChart.js` следующий код:

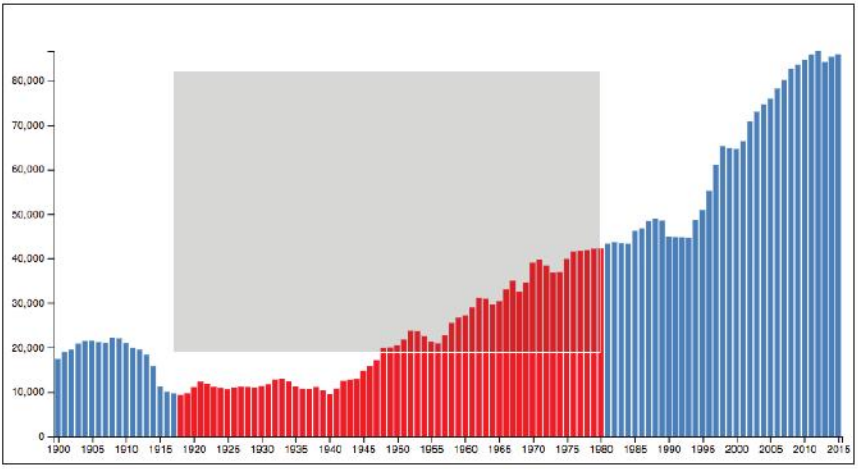
```
brushablePrisonPopulationChart.rightclick = function () {
  d3.event.preventDefault();
  this.clearSelected();
  this.update();
  this.svg.select('.hitbox').remove();
};
```

Он просто предотвращает появление стандартного контекстного меню, очищает выбранные столбики и выполняет обновление для всего набора данных. Заодно удаляется элемент `hitbox`.

Осталось только экспортировать и сохранить:

```
export default brushablePrisonPopulationChart;
```

После выборки некоторых столбиков диаграмма будет выглядеть так:



Затем эти столбики будут *приближены* благодаря вызову `update()` с данными, относящимися к выбранным столбикам.

Масштабирование

Несмотря на название, поведение масштабирования не ограничивается одним масштабированием – оно позволяет также панорамировать изображение! Как и буксировка, масштабирование автоматически обрабатывает события мыши и касания и генерирует событие масштабирования более высокого уровня. Да-да, сюда входит и изменение масштаба щипком!

Помните карту из главы 4 – ту, на которой были показаны рейсы для экстрадиции? Давайте-ка добавим к ней возможности масштабирования и панорамирования.

Мы должны присоединить поведение масштабирования к элементу, а оно будет генерировать события, когда пользователь станет взаи-

модействовать с этим элементом. Результатом является некоторое значение, которое можно либо применить для преобразования контейнерного элемента, либо использовать для изменения геометрии проекции. Начнем со второго.

Закомментируйте весь код в файле `chapter5/index.js` и добавьте такой:

```
import { addZoomBehavior } from './zoomMap';

(async (enabled) => {
  if (!enabled) return;
  addZoomBehavior();
})(true);
```

Затем создайте файл `lib/chapter5/zoomMap.js` и поместите в него такой код:

```
import * as d3 from 'd3';
import '../chapter4';

const zoomMap = {};

const NE_SCALE = 200;
const projection = d3.geoEquirectangular()
  .center([-50, 56])
  .scale(NE_SCALE);
```

Здесь мы импортируем файл `chapter4/index.js` целиком, а осталось в нем только исполняемое выражение `geoDemo`. В данном случае проектное решение не слишком удачное, поскольку означает, что у нас больше нет доступа к внутреннему состоянию объекта диаграммы. Мы можем либо вернуться и экспортировать `geoDemo` из `chapter4/index.js`, либо отбросить сомнения и работать с тем, что есть, применив поведение масштабирования к существующей диаграмме после ее отрисовки. Хотя с точки зрения сопровождения первый вариант лучше, мы все же выберем второй – он прост и демонстрирует, что D3 можно использовать и для манипуляции сторонними элементами SVG.

Однако нам необходимо скопировать прежнюю проекцию, чтобы можно было правильно изобразить обновленные пути. Проекция такая же, как в файле `chapter4/index`. Теперь добавим поведение масштабирования. Вставьте такой код в файл `zoomMap.js`:

```
zoomMap.addZoomBehavior = () => {
  const chart = d3.select('#chart');
  const zoom = d3.zoom()
    .scaleExtent([0.5, 2])
```

```
    .on('zoom', zoomMap.onZoom);

  const center = projection(projection.center());

  chart.call(zoom)
    .call(zoom.transform, d3.zoomIdentity.translate(center[0], center[1]));
};
```

Мы создаем поведение масштабирования и с помощью функции `scaleExtent()` определяем минимальное и максимальное значения масштаба, прежде чем присоединять обработчик события масштабирования. Затем мы вычисляем центр проекции, передав исходное описание центра функции `projection`. Новый центр используется в следующей строке, где мы вызываем поведение масштабирования для нашей диаграммы методом `call()`. Обратите внимание, что `.call()` встречается дважды – во второй раз мы вызываем метод поведения `transform`, чтобы установить начальное преобразование, которому передается центр проекции. Если этого не сделать, то в самом начале взаимодействие будет «дёрганым», потому что пересчитанная проекция диаграммы не согласована с исходной.

Отметим, что в версии D3 v3 начальный вектор масштабирования следовало устанавливать с помощью методов `zoom.scale()` и `zoom.translate()`; теперь они заменены показанной выше формой с применением `transform`.

Теперь сам обработчик:

```
zoomMap.onZoom = () => {
  const { x, y, k } = d3.event.transform;
  projection
    .scale(k * NE_SCALE)
    .translate([x, y]);

  d3.selectAll('path')
    .attr('d', d3.geoPath().projection(projection));

  d3.selectAll('line.route')
    .attr('x1', d => projection([d.from.lon, d.from.lat])[0])
    .attr('y1', d => projection([d.from.lon, d.from.lat])[1])
    .attr('x2', d => projection([d.to.lon, d.to.lat])[0])
    .attr('y2', d => projection([d.to.lon, d.to.lat])[1]);
};
```

Свойства преобразования масштабирования передаются в объекте `d3.event.transform`; мы деструктурируем его, присваивая отдельные свойства константам `x` (абсцисса), `y` (ордината) и `k` (масштаб). Затем мы изменяем проекцию в соответствии с полученными от поведения

значениями и используем новую проекцию, чтобы обновить значение d для всех путей и конечных точек рейсов.

Сохранитесь и получите невыносимо медленную карту, поддерживающую масштабирование и панорамирование! Да уж, с производительностью дело обстоит ужасно!

Пересчитывать проекцию Земли при каждом движении мыши с точки зрения производительности – безумие. Вряд ли вам часто придется совершать такие действия с картой на практике, но полезно знать о том, что пакет d3-zoom позволяет осуществлять программное масштабирование с помощью преобразований. Подойдем к задаче по-другому – будем просто преобразовывать родительский контейнер.

В файле chapter5/index закомментируйте строку

```
zoomMap.addZoomBehavior();
```

и добавьте вместо нее такую:

```
zoomMap.addZoomBehaviorTransform();
```

Сохранитесь и вернитесь в файл zoomMap.js:

```
zoomMap.addZoomBehaviorTransform = () => {
  const chart = d3.select('#chart');

  const zoom = d3.zoom()
    .scaleExtent([0.5, 2])
    .on('zoom', zoomMap.onZoomTransform);

  chart.call(zoom);
};
```

Гораздо проще, не правда ли? Все то же, что и раньше, только не надо настраивать проекции и начальные преобразования. Перейдем к обработчику события:

```
zoomMap.onZoomTransform = () => {
  const container = d3.select('#container');
  container.attr('transform', d3.event.transform);
  container.selectAll('path')
    .style('stroke-width', 1 / d3.event.transform.k);
};
```

А это еще проще! Мы выбираем контейнер и записываем в его свойство transform содержимое d3.event.transform (в строковом виде оно как раз содержит значение, которое можно присвоить свойству transform). Мы даже позволили себе экстравагантность – установили ширину линии в соответствии с величиной масштаба.

Сохранитесь и перезагрузите карту. Не в пример лучше, да? D3 предоставляет свободу реализации; чтобы понять, какой способ лучше в конкретном проекте, нужны практика и знание потенциальной аудитории. Но, вообще говоря, лучше не пересчитывать проекцию, если есть возможность этого избежать, поскольку работает это очень медленно.

А нужна ли вам вообще интерактивность?

Закончим на поучительной ноте – прежде чем наделять проект интерактивностью, задайтесь вопросом, так ли это нужно. Если вы принадлежите к когорте людей, использующих D3 как средство визуализации данных для СМИ, имейте в виду, что число тех, кто пользуется функциональностью интерактивной графики, удручающе мало; большинство читателей просто бросает мимолетный взгляд на графику, которую вы так тщательно и так долго шлифовали. Заместитель начальника отдела графики в газете *New York Times* Арчи Це сформулировал три правила визуального повествования:

- если вы заставляете читателя щелкнуть мышью или произвести еще какое-то действие, кроме прокрутки, то вознаградите его чем-то, заслуживающим внимания;
- если вы включаете всплывающее пояснение или иное поведение при наведении мыши, предполагайте, что его никто не увидит. Не скрывайте то, что читатель должен увидеть;
- размышляя о том, стоит ли сделать нечто интерактивным, помните, что обеспечить работу на всех платформах станет дорого.

Источник можно найти по адресу https://github.com/archietse/malofiej-2016/blob/master/tse-malofiej-2016-slides.pdf.

На самом деле моя группа в *Financial Times* очень редко делает интерактивную графику, поскольку это отвлекает от того, что мы в действительности хотим передать. Начальник отдела интерактивных новостей в *Financial Times* Мартин Штабе говорит, что визуализация призвана прояснить данные, но когда интерактивности слишком много, то понимание страдает, т. к. ответственность за выделение важных моментов переходит от дизайнера к читателю. Источник (платный) – https://www.ft.com/content/c62b21c6-7feb-11e6-8e50-8ec15fb462f4.

Одно из решений – навесить интерактивность на события прокрутки браузера, показывая пользователю данные по мере того, как он прокручивает страницу, но и тут таится опасность, потому что не всегда легко сделать это правильно, да к тому же некоторые злятся, когда кто-то посторонний вмешивается в прокрутку. Тем, кто ре-

шится реализовать такой подход к интерактивному повествованию, может пригодиться библиотека слежения за прокруткой от журнала Wall Street Journal по адресу https://github.com/WSJ/scroll-watcher.

Наконец, даже если ваш проект не рассчитан на потребителей новостей, помните, что к построению интерфейса надо подходить очень осторожно. Вы как разработчик с компьютером на ты, но о большинстве людей этого не скажешь – на самом деле недавнее исследование навыков работы с компьютером в государствах, входящих в Организацию экономического сотрудничества и развития, показало, что целых 24% вообще не могут пользоваться компьютером сколько-нибудь эффективно, а еще 16% обладают лишь самыми базовыми навыками (https://www.nngroup.com/articles/computer-skill-levels/). Особенно хочу предупредить тех, кто работает в правительстве и использует D3 для донесения важной информации до избирателей: сведение интерактивности к минимуму – зачастую очень правильная мысль.

Резюме

Ах, что за глава!

Вы заставили объекты скакать по странице, чуть не довели до истощения компьютер и собственное терпение, ожидая отрисовки масштабируемой карты, и создали потрясающую столбчатую диаграмму. Отличная работа!

В этой главе мы выполняли анимацию с помощью переходов, интерполяторов и таймеров. Мы узнали о различии между объяснительной и исследовательской графикой и применили интерактивности для реализации первой. Затем мы занялись исследовательской визуализацией, добавив поведения в предыдущие проекты. Получается довольно эффектно, правда?

В следующей главе мы построим целую кучу роскошных диаграмм с помощью иерархических макетов D3. В сочетании знания, полученные в этой и в следующей главах, позволят вам создавать просто фантастические диаграммы. Надеюсь, вы готовы – это вам придется по нраву!

Глава 6

Иерархические макеты в D3

Один из способов научиться делать интересные вещи с помощью D3 – познакомиться с примерами на сайте `bl.ocks.org`. Там вы найдете динамичную, допускающую разветвление кодовую базу, которую можно модифицировать и приспособить к своему случаю. Но изучение D3 осложняется, в частности, тем, что зачастую эти примеры основаны на макетах – алгоритмах, которые некоторым образом изменяют структуру данных. И если не знать, как они работают, то разобраться в примере будет нелегко.

Новое в версии D3 v4! Все, о чем говорится в этой и последующих главах, подверглось значительной переработке в D3 v4. Теперь особенно внимательно проверяйте, для какой версии D3 написан пример, найденный вами в Сети.

В этой и следующей главах мы будем заниматься преимущественно макетами и построим кучу диаграмм. Начнем мы с иерархических макетов, в которых предполагается, что данные обладают древовидной структурой, состоящей из узлов-родителей и узлов-потомков. А в следующей главе рассмотрим другие макеты, в т. ч. секторные диаграммы и диаграммы на основе действующих сил. Приступим.

Что такое макеты и зачем вам о них знать?

Макеты D3 – это модули, преобразующие данные в правила рисования. Простейший макет всего лишь преобразует массив объектов в координаты, как масштаб.

Но обычно макеты используются для более сложной визуализации, например рисования графа действующих сил (force-directed graph) или дерева. В таких случаях макет помогает отделить вычисление координат от размещения пикселей на экране. Мало того что при этом код становится чище, так еще один и тот же макет можно повторно использовать в разных визуализациях.

Встроенные макеты

В состав D3 входит с десяток встроенных макетов, охватывающих наиболее распространенные визуализации. Их можно условно разделить на **обычные** и **иерархические**. Обычный макет представляет данные в виде плоской иерархии, а иерархический – в виде древовидной структуры.

К обычным макетам относятся следующие:

- гистограммы;
- секторные диаграммы;
- стопка;
- хорда;
- действующие силы.

К иерархическим макетам относятся следующие:

- дерево;
- кластер;
- древовидная карта;
- разбиение;
- упаковка.

В этой и следующей главах будет много примеров. Мы начнем с иерархических макетов (модуль `d3-hierarchy`), потому что они более единообразны с точки зрения структуры данных, которая выглядит примерно так:

```
{ "name": "Tywin Lannister",
    "children": [{
      "name": "Jamie Lannister",
      "children": [
        { "name": "Joffery Baratheon" },
        { "name": "Myrcella Baratheon" },
        { "name": "Tommen Baratheon" }
      ]}, {
      "name": "Tyrion Lannister",
      "children": []
    }]
}
```

Иерархия может включать и много других данных, но главное – она должна быть вложенной совокупностью объектов, в которой иерархичность определяется неявно посредством некоторого атрибута (например, с именем `children`). Можно использовать также вспомогательную функцию из модуля `d3-hierarchy`, `d3.stratify()`, которая преобразует плоскую последовательность типа показанной ниже в структуру, необходимую иерархическим макетам:

```
[
  {"name": "Westeros", "parent": ""},
  {"name": "Tywin Lannister", "parent": "Westeros"},
  {"name": "Jamie Lannister", "parent": "Tywin Lannister"},
  {"name": "Tyrion Lannister", "parent": "Tywin Lannister"},
  {"name": "Joffery Baratheon", "parent": "Jamie Lannister"},
]
```

Отметим, что в любой такой последовательности должен быть один и только один корневой узел, т. е. узел, не имеющий родителей и являющийся предком всех остальных узлов. Если корневого узла нет или их несколько, то D3 возбудит исключение. В данном случае корневой узел назван `Westeros`, поскольку там проживают все перечисленные далее люди.

Предупреждаю, что мы будем использовать набор данных, относящийся к фантастическому телевизионному сериалу вплоть до шестого сезона. Возможно, глубоко проанализировав эти данные, вы раскроете несколько сюрпризов и испортите себе удовольствие, так что предупреждаю заранее, если вас это тревожит.
Если «Игра престолов» вам не нравится, прошу меня простить.

Для генерации надписей мы будем пользоваться сторонней библиотекой. Установите ее следующей командой:

```
npm install d3-svg-legend --save
```

Можно было бы создать надписи самостоятельно, и это могло бы стать интересным упражнением, но вообще-то вы уже и так понимаете, что происходит, – библиотека пользуется масштабом для создания квадратиков и текстовых меток SVG. Скучно...

Давненько мы ничего не добавляли в файл `lib/common/index.js`, но теперь все изменится, потому что нам понадобится много общей для всех диаграмм функциональности. Добавьте такой код:

```
export const colorScale = d3.scaleOrdinal()
    .range(d3.schemeCategory20);
export const heightOrValueComparator =
```

```
    (a, b) => b.height - a.height || b.value - a.value;
export const valueComparator = (a, b) => b.value - a.value;

export const fixateColors = () => {};
export const addRoot = () => {};
export const tooltip = () => {};
export const descendantsDarker = () => {};
```

Вначале создается цветовой масштаб со схемой `Category20`, содержащей 20 цветов, а также несколько пользовательских функций сравнения. Заодно мы создали несколько пустых функций, которые скоро наполним кодом.

Пора уже построить какую-нибудь диаграмму! Мы воспользуемся функцией `chartFactory` для создания базовой диаграммы, которая пригодится неоднократно. Создайте в папке `lib/` новую подпапку `chapter6/`, а в ней файл `index.js`. Поместите в него такой код:

```
import * as d3 from 'd3';
import * as legend from 'd3-svg-legend';
import chartFactory, {
  fixateColors,
  addRoot,
  colorScale as color,
  tooltip,
  heightOrValueComparator,
  valueComparator,
  descendantsDarker,
} from '../common';
```

Здесь мы импортируем D3, стороннюю библиотеку создания надписей и много всякого из нашего модуля общей функциональности.

Затем добавьте такой код:

```
const westerosChart = chartFactory({
  margin: { left: 50, right: 50, top: 50, bottom: 50 },
  padding: { left: 10, right: 10, top: 10, bottom: 10 },
});
```

Он добавляет объект `padding` в прототип нашей диаграммы, это немного упростит ряд операций. Следующей добавим функцию загрузки данных, различающую CSV и JSON-файлы:

```
westerosChart.loadData = async function loadData(uri) {
  if (uri.match(/.csv$/)) {
    this.data = d3.csvParse(await (await fetch(uri)).text());
  } else if (uri.match(/.json$/)) {
    this.data = await (await fetch(uri)).json();
  }
```

```
  return this.data;
};
```

Да, снова `async` и `await` – просто и красиво. Разобранные данные присваиваются переменной `data` в локальном контексте, после чего эта переменная возвращается. Добавим функцию, которая будет инициализировать диаграммы и передавать им аргументы:

```
westerosChart.init = function initChart(chartType, dataUri, ...args) {
  this.loadData(dataUri).then(data => this[chartType].call(this, data, ...args));
  this.innerHeight = this.height - this.margin.top - this.margin.bottom -
                     this.padding.top - this.padding.bottom;
  this.innerWidth = this.width - this.margin.left - this.margin.right -
                    this.padding.left - this.padding.right;
};
```

Поскольку функция `loadData()` помечена как `async`, она возвращает обещание. Можно было бы сделать `init()` `async`-функцией и просто дождаться результатов от `loadData()` с помощью `await`, но не будем усложнять и воспользуемся методом `.then()`, чтобы подождать, когда исполнится обещание, возвращенное `loadData()`. Тогда оставшаяся часть `init()` не должна дожидаться получения данных.

Конструкция `...args` в аргументах функции – новая возможность, появившаяся в ES2015, она называется «**оставшиеся параметры**». Она позволяет представить все аргументы, кроме явно заданных, в виде массива. Мы используем еще одну новую конструкцию из ES2015 – **оператор расширения** – в выражении `this[chartType].call(this, ...args)`, чтобы деструктурировать массив, расчленив его на отдельные значения. Это позволяет задавать в методах диаграмм столько аргументов, сколько необходимо.

Далее в `.init()` просто задаются два свойства, `innerHeight` и `innerWidth`, которые понадобятся, чтобы упростить и сократить код добавления надписей и осей.

Иерархические макеты

Все иерархические макеты основаны на абстрактном макете, предназначенном для представления иерархических данных. Можете держать в уме дерево или организационную схему.

Весь код макетов `partition`, `tree`, `cluster`, `pack` и `treemap` находится в модуле `d3-hierarchy`, и все они устроены одинаково. Макеты очень похожи и имеют много общих черт, поэтому во избежание повторений мы сначала рассмотрим общее, а затем займемся различиями.

Прежде всего нам понадобятся какие-нибудь иерархические данные. В репозитории кода книги имеется файл `GoT-lineages-screentimes.json`, в котором хранятся родословные всех персонажей (по отцовской линии), время их присутствия на экране и количество эпизодов с их участием. Родословные используются для создания иерархии, а экранное время – чтобы задать размер различных элементов страницы в тех макетах, где это нужно (в примерах деревьев и кластеров экранное время игнорируется, чтобы диаграмма была короче и проще для восприятия).

Каждый объект в файле `GoT-lineages-screentimes.json` имеет вид:

```
{
  "itemLabel": "Lysa Arryn",
  "fatherLabel": "Hoster Tully",
  "screentime": 16.3,
  "episodes": 5
},
```

Поскольку данные не представлены в формате дерева, мы воспользуемся функцией `d3.stratify()` для создания иерархии, считая `itemLabel` уникальным идентификатором узла, а `fatherLabel` – идентификатором родительского узла.

Генеалогическое древо

Начнем с простейшей иерархической диаграммы – дерева. Создайте новую функцию и поместите в нее такой код:

```
westerosChart.tree = function Tree(_data) {
  const data = getMajorHouses(_data);
  const chart = this.container;
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);
  const root = stratify(data);
  const layout = d3.tree()
    .size([
      this.innerWidth,
      this.innerHeight,
    ]);
}
```

Мы воспользовались еще не написанной функцией `getMajorHouses()`, чтобы отфильтровать персонажей, у которых нет свойства `fatherLabel` или `itemLabel` не совпадает со свойством `fatherLabel` какого-нибудь

другого персонажа. Затем мы создаем объект `stratify` и определяем для него функции-акцессоры: `parentId()`, возвращающую `fatherLabel`, и `id()`, возвращающую `itemLabel`. Для этого набора данных такое допустимо, потому что все значения `itemLabel` заведомо различны; если бы это было не так (например, если бы в наборе данных было несколько Джонов Смитов), то пришлось бы использовать в качестве уникального идентификатора не имя персонажа, а какое-то другое свойство. Затем мы передаем данные только что определенной функции `stratify` для создания корня. Мы также создаем экземпляр макета `tree` и задаем его внутренние размеры.

Теперь можно заняться функциями манипулирования структурой данных. Первой мы добавим функцию `addRoot`, которая экспортируется из `lib/common/index.js`. Откройте этот файл и добавьте такой код в функцию в `addRoot`:

```
export function addRoot(data, itemKey, parentKey, joinValue) {
  data.forEach((d) => { d[parentKey] = d[parentKey] || joinValue; });
  data.push({
    [parentKey]: '',
    [itemKey]: joinValue,
  });

  return data;
}
```

Функция `addRoot()` принимает четыре аргумента: данные, имя свойства, содержащего идентификатор объекта, имя свойства, содержащего идентификатор родительского объекта, и значение, присваиваемое этому свойству в случае его отсутствия. Функция восполняет отсутствующие значения родителей, затем добавляет корневой элемент в массив `data` и возвращает этот массив.

Заодно напишем функции `fixateColors` и `uniques()`:

```
export const uniques = (data, name) => data.reduce(
  (uniqueValues, d) => {
    uniqueValues.push(
      (uniqueValues.indexOf(name(d)) < 0 ? name(d) : undefined));
    return uniqueValues;
  }, [])
  .filter(i => i); // фильтр по идентификатору

export function fixateColors(data, key) {
  colorScale.domain(uniques(data, d => d[key]));
}
```

Здесь для экспортируемого цветового масштаба задается область определения, содержащая уникальные значения переданных функции данных. Функция `uniques()` работает следующим образом: создает массив и добавляет в него элементы, определяемые функцией, переданной в качестве второго аргумента, или пропускает их, если такой элемент уже есть в массиве.

Добавим функцию, которая получает название дома, поднимаясь вверх по дереву и выделяя фамилию старейшего предка:

```
export const getHouseName = (d) => {
  const ancestors = d.ancestors();
  let house;
  if (ancestors.length > 1) {
    ancestors.pop();
    house = ancestors.pop().id.split(' ').pop();
  } else {
    house = 'Westeros';
  }

  return house;
};
```

Еще добавим функцию, которая возвращает список названий всех главных домов:

```
export const houseNames = root =>
  root.ancestors().shift().children.map(getHouseName);
```

Затем удалим объекты, для которых нет данных о потомстве. После секции импорта добавьте в файл `chapter6/index.js` такой код:

```
const getMajorHouses = data =>
  addRoot(data, 'itemLabel', 'fatherLabel', 'Westeros')
  .map((d, i, a) => {
    if (d.fatherLabel === 'Westeros') {
      const childrenLen = a.filter(
        e => e.fatherLabel === d.itemLabel).length;
        return childrenLen > 0 ? d : undefined;
    } else {
      return d;
    }
  })
  .filter(i => i);
```

Мы передаем наши данные функции `addRoot`, чтобы добавить `Westeros` в качестве корневого узла, и отфильтровываем из результата узлы, в которых отцом является `Westeros`, а сами они ничьим отцом не являются.

Далее изобразим данные, применив древовидный макет. Добавьте такой код в объект `westerosChart.tree`:

```
const links = layout(root)
  .descendants()
  .slice(1);

fixateColors(getHouseNames(root), 'id');
const line = d3.line().curve(d3.curveBasis);

chart.selectAll('.link')
  .data(links)
    .enter()
    .append('path')
    .attr('fill', 'none')
    .attr('stroke', 'lightblue')
    .attr('d', d => line([
      [d.x, d.y],
      [d.x, (d.y + d.parent.y) / 2],
      [d.parent.x, (d.y + d.parent.y) / 2],
      [d.parent.x, d.parent.y]],
));
```

Вот тут начинается самое интересное. Мы передаем созданный ранее корневой объект генератору макета и получаем в ответ объект макета. У этого объекта есть метод `descendants()`, который возвращает всех потомков корневого узла, начиная с него самого в топологическом порядке (сам корневой узел мы отрезаем, потому что в него ничего не ведет). Затем мы фиксируем цвета, вызвав написанную выше функцию. Мы также создаем генератор линий с параметрами по умолчанию, задав для красоты интерполятор `curveBasis`.

После этого мы создаем новую выборку, соединяем ее с данными и добавляем элементы пути для каждого ребра дерева, задав для кривой две управляющие точки посередине.

Способ генерации ребер не так интуитивно понятен, как хотелось бы, и разработчики, сопровождающие D3, понимают, что это необходимо улучшить. Следите за проблемой #27 для модуля `d3-shape` по адресу `github.com/d3/d3-shape/issues/27`, поскольку здесь могут произойти изменения.

Далее нарисуем круги, представляющие каждый узел:

```
const nodes = chart.selectAll('.node')
  .data(root.descendants())
  .enter()
    .append('circle')
```

```
    .attr('r', 4.5)
    .attr('fill', getHouseColor)
    .attr('class', 'node')
    .attr('cx', d => d.x)
    .attr('cy', d => d.y);
```

Мы снова получаем все узлы от функции `root.descendants()` и передаем их новой выборке, добавляя круги к каждому узлу. Затем мы закрашиваем круги цветом, полученным от функции `getHouseColor` (которую скоро напишем), и задаем радиус `4.5`. Теперь определим функцию `getHouseColor`; поместите следующий код в начало файла `chapter6/index.js`, сразу после `getMajorHouses()`:

```
const getHouseColor = (d) => {
  const ancestors = d.ancestors();
  let house;
  if (ancestors.length > 1) {
    ancestors.pop();
    house = ancestors.pop().id.split(' ').pop();
  } else {
    house = 'Westeros';
  }

  return color(house);
};
```

Мы получаем строковый идентификатор, разбиваем его на части по пробелу и берем последний элемент получившегося массива. Предполагается, что перед фамилией стоит пробел и что фамилия – последняя часть имени персонажа (например, при разборе имени *Sammy Davis Jr.* возникли бы проблемы). Если у объекта только один предок (он сам), то мы берем строку `Westeros`, потому что это наш корневой узел. Затем найденное значение передается цветовому масштабу.

Далее добавим надпись. Включите в объект `westerosChart.tree` такой код:

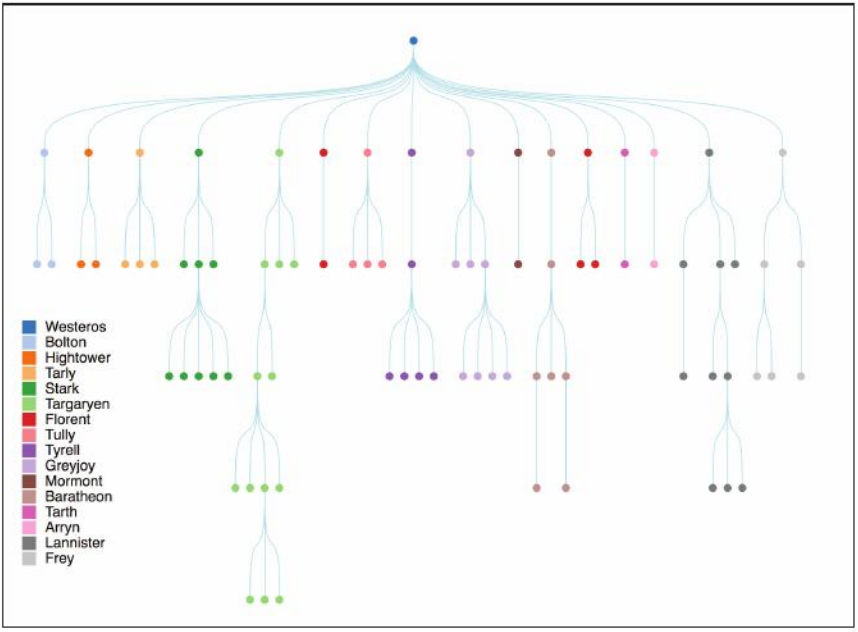
```
const legendGenerator = legend
  .legendColor()
  .scale(color);

this.container
  .append('g')
  .attr('id', 'legend')
  .attr('transform', &grave;translate(0, ${this.innerHeight / 2})&grave;)
  .call(legendGenerator);
```

Здесь создается генератор надписей, который помещается в контейнер – слева посередине. Мы передаем ему наш цветовой масштаб, чтобы генератор знал, какие надписи формировать.

Перейдите в файл `lib/main.js`, удалите все, что в нем есть, и добавьте такой код:

```
import '../styles/index.css';
import westerosChart from './chapter6/index';
westerosChart.init('tree', 'data/GoT-lineages-screentimes.json');
```

Зайдя в браузер (не забудьте предварительно выполнить `npm start`, если локальный сервер разработки не запущен), вы увидите такую картину:



Но это еще не всё, мы пока не добавили меток.

Можно было бы дописать текстовые метки к каждому узлу (это несложно, поскольку в каждом элементе данных есть свойство `itemLabel`), но поскольку узлов много, то на типичном экране метки, скорее всего, будут наползать друг на друга. Пожалуй, самое время познакомиться с всплывающими пояснениями! Они будут появляться на экране, когда пользователь наводит мышь на элемент диаграммы.

Перейдите в файл common/index.js и измените функцию tooltip():

```
export function tooltip(text, chart) {
  return (selection) => {
    function mouseover(d) {
      const path = d3.select(this);
      path.classed('highlighted', true);

      const mouse = d3.mouse(chart.node());
      const tool = chart.append('g')
        .attr('id', 'tooltip')
        .attr('transform',
          &grave;translate(${mouse[0] + 5},${mouse[1] + 10})&grave;);

      const textNode = tool.append('text')
        .text(text(d))
        .attr('fill', 'black')
        .node();

      tool.append('rect')
        .attr('height', textNode.getBBox().height)
        .attr('width', textNode.getBBox().width)
        .style('fill', 'rgba(255, 255, 255, 0.6)')
        .attr('transform', 'translate(0, -16)');

      tool.select('text')
        .remove();

      tool.append('text').text(text(d));
    }

    function mousemove() {
      const mouse = d3.mouse(chart.node());
      d3.select('#tooltip')
        .attr('transform',
          &grave;translate(${mouse[0] + 15},${mouse[1] + 20})&grave;);
    }

    function mouseout() {
      const path = d3.select(this);
      path.classed('highlighted', false);
      d3.select('#tooltip').remove();
    }

    selection.on('mouseover.tooltip', mouseover)
      .on('mousemove.tooltip', mousemove)
      .on('mouseout.tooltip', mouseout);
  };
}
```

Здесь мы написали фабрику, которая возвращает новую функцию, принимающую выборку в качестве аргумента. Затем мы присоединили обработчики событий входа и выхода мыши: когда мышь находится в области элемента, мы изменяем текст и координаты элемента, содержащего пояснение. Когда мышь покидает эту область, пояснение становится невидимым. Когда мышь перемещается внутри области элемента, мы изменяем координаты пояснения.

Добавьте в конец объекта `westerosChart.tree` строку:

```
nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

Напомним, что генератор всплывающих пояснений принимает два аргумента: функцию-акцессор и целевой контейнер. Мы передаем эти аргументы генератору, а затем вызываем его для каждого узла.

И вот они – пояснения!

Кластер-блокбастер!

На дерево похожа дендрограмма: для ее построения используется макет `cluster`, а все листовые узлы располагаются на одной глубине. Построим ее. Закомментируйте в файле `main.js` строку `westerosChart.init()` и поместите под ней такую:

```
westerosChart.init('cluster', 'data/GoT-lineages-screentimes.json');
```

В файл `chapter6/index.js` добавьте следующий код:

```
westerosChart.cluster = function Cluster(_data) {
  const data = getMajorHouses(_data);
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data);

  fixateColors(houseNames(root), 'id');

  const layout = d3.cluster()
    .size([
      this.innerWidth - 150,
      this.innerHeight,
    ]);

  const links = layout(root)
    .descendants()
    .slice(1);
}
```

Все это уже знакомо – получаем данные, создаем генератор `stratify` и применяем его к данным. Далее создается макет `cluster`, задается его размер (при этом мы оставляем 150 пикселей для надписей) и с помощью функции `layout.descendants()` генерируются ребра.

В тот же объект `westerosChart.cluster` добавьте такой код:

```
const line = d3.line().curve(d3.curveBasis);

this.container.selectAll('.link')
  .data(links)
  .enter()
  .append('path')
  .attr('fill', 'none')
  .attr('stroke', 'lightblue')
  .attr('d', d => line([
    [d.y, d.x],
    [(d.y + d.parent.y) / 2, d.x],
    [(d.y + d.parent.y) / 2, d.parent.x],
    [d.parent.y, d.parent.x]],
  ));
```

Это тот же код рисования путей, что и раньше, только мы поменяли местами `x` и `y`, чтобы диаграмма располагалась горизонтально, а не вертикально.

Добавляем круги, представляющие узлы:

```
const nodes = this.container.selectAll('.node')
  .data(root.descendants())
  .enter()
  .append('circle')
  .classed('node', true)
    .attr('r', 5)
    .attr('fill', getHouseColor)
    .attr('cx', d => d.y)
    .attr('cy', d => d.x);
```

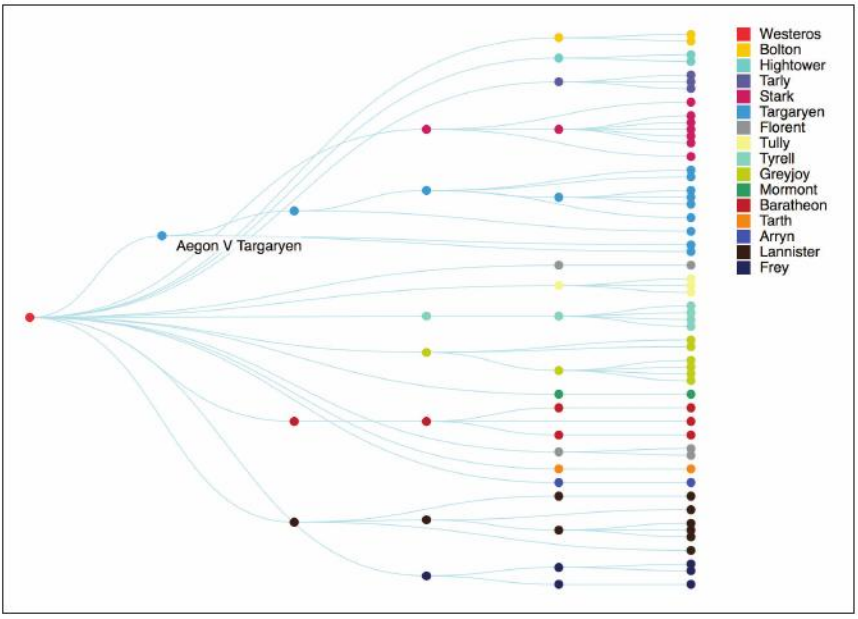
И добавляем надписи:

```
const l = legend
  .legendColor()
  .scale(color);

this.container
  .append('g')
  .attr('id', 'legend')
  .attr('transform', &grave;translate(${this.innerWidth - 100}, 0)&grave;)
  .call(l);
```

Наконец, вызываем фабрику всплывающих пояснений – и все!

```
nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

После сохранения вы увидите такую диаграмму:



Карты древовидные – справные да видные

Несмотря на похожее название, древовидные карты (treemap) визуально мало напоминают рассмотренный выше макет tree; они разбивают область экрана на смежные прямоугольные участки. Для этого необходим какой-то способ определения количественной величины данных. В данном случае размер участка древовидной карты будет определяться экранным временем, и каждый участок будет вложен в своего родителя. Таким образом, размер родителя равен сумме размеров его детей плюс собственный размер.

Все это очень похоже на то, что мы уже встречали, поэтому начнем с написания общих функций. В файл `common/index.js` добавьте функцию, которая будет давать все более мелкие разбиения, исходя из иерархии:

```
export const descendantsDarker = (d, color, invert = false, dk = 5) =>
  d3.color(
    color(
      d.ancestors()[d.ancestors().length - 2].id.split(' ').pop()
    ) )[invert ? 'brighter' : 'darker'](d.depth / dk);
```

Она принимает элемент данных, цветовой масштаб и три опции: булев флаг, показывающий, нужно ли делать масштаб ярче, а не темнее, числовой множитель, определяющий силу эффекта, и строку, содержащую имя свойства, в котором хранится идентификатор (по умолчанию `itemLabel`).

Блестяще! Теперь – стандартная настройка, которую мы уже видели в предыдущих диаграммах. Перейдите в файл `lib/main.js`, закомментируйте последний вызов `westerosChart.init()` и добавьте вместо него такой:

```
westerosChart.init('treemap', 'data/GoT-lineages-screentimes.json');
```

Вернитесь в файл `chapter6/index.js` и добавьте такой код:

```
westerosChart.treemap = function Treemap(_data) {
  const data = getMajorHouses(_data);
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(heightOrValueComparator);

  const cellPadding = 10;
  const houseColors = color.copy().domain(houseNames(root));
}
```

Здесь делается все то же, что и раньше, только теперь мы для сортировки используем функцию сравнения `heightOrValue`, находящуюся в файле `common/index.js`. Если вы забыли, напомню, как она выглядит:

```
(a, b) => b.height - a.height || b.value - a.value
```

В таком виде функция работает, что бы ни было задано: `height` или `value`. Мы также получаем для каждого персонажа сумму его экранного времени и экранного времени всех его детей. Мы должны отсортировать и просуммировать экранное время от корня, прежде чем передать его генератору древовидной карты, иначе он не будет знать, как вычислять размеры участков.

Кроме того, мы определили константу, задающую промежуток между ячейками (я выбрал значение 10, потому что считаю, что при

таком наборе данных лучше, чтобы у участков были широкие поля), и создали новый цветовой масштаб специально для названий домов. Нам нужно два цветовых масштаба, потому что иначе надпись стала бы очень большой и громоздкой, а мы хотим всего лишь отобразить категориальные цвета, ассоциированные с каждым домом.

Наконец, добавим генератор макета древовидной карты:

```
const layout = d3.treemap()
  .size([
    this.innerWidth - 100,
    this.innerHeight,
  ])
  .padding(cellPadding);
```

Он мало чем отличается от всех остальных. Мы только задали промежуток между ячейками и уменьшили ширину на 100 пикселей, чтобы оставить место для надписи.

Теперь применим наш макет к данным, в результате чего к ним будут добавлены релевантные свойства; эта операция вносит изменения прямо в нашу иерархию, поэтому присваивать результат ничему не будем.

```
layout(root);
```

Да, я знаю – функциональный программист во мне тоже громко протестует.

Теперь рисуем все узлы, они будут представлены просто прямоугольниками:

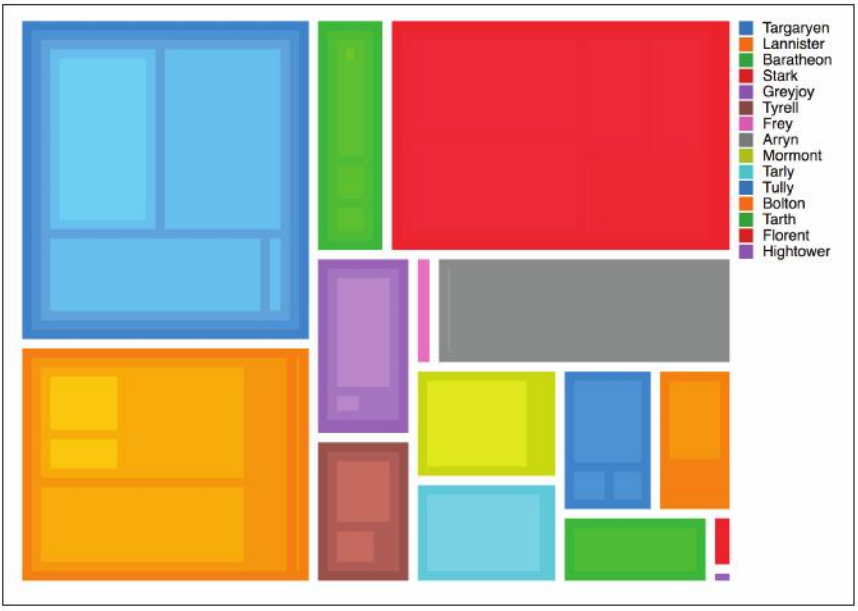
```
const nodes = this.container.selectAll('.node')
  .data(root.descendants().slice(1))
  .enter()
  .append('g')
    .attr('class', 'node');

nodes.append('rect')
  .attr('x', d => d.x0)
  .attr('y', d => d.y0)
  .attr('width', d => d.x1 - d.x0)
  .attr('height', d => d.y1 - d.y0)
  .attr('fill', d => descendantsDarker(d, color, true, 3));
```

Поскольку корневой узел нам здесь не нужен, мы отрезаем после вызова `root.descendants()`. Затем добавляется новая группа для каждого узла. После этого в каждую группу добавляется прямоугольник, и задаются его положение и размер. Далее недавно написанная функция `descendantsDarker` дает нам последовательность похожих цветов.

В следующем фрагменте устанавливаются надпись и всплывающие пояснения:

```
this.container
  .append('g')
  .attr('id', 'legend')
  .attr('transform',
    &grave;translate(${this.innerWidth - 100}, ${cellPadding})&grave;)
  .call(legend.legendColor().scale(houseColors));

nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

После сохранения мы получим третью диаграмму. Уже полпути пройдено!



Очарованные разбиением

На нескольких следующих диаграммах мы не будем останавливаться столь же подробно, все они похожи, и никаких новых общих функций не появляется.

Диаграммы смежности похожи на древовидные карты, но заполняют предоставленное пространство. Они полезны, например, для

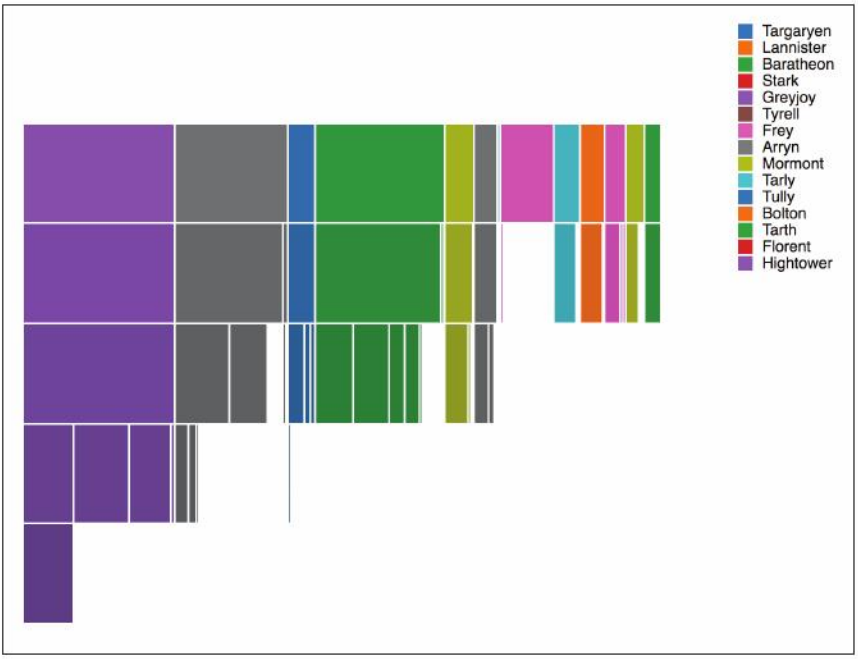
визуализации использования диска. Для их создания используется макет разбиения `partition`. Ниже приведен пример от начала до конца:

```
westerosChart.partition = function Partition(_data) {
  const data = getMajorHouses(_data);
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(heightOrValueComparator);

  const layout = d3.partition()
    .size([
      this.innerWidth - 100,
      this.innerHeight,
    ])
    .padding(2)
    .round(true);

  layout(root);

  const nodes = this.container.selectAll('.node')
    .data(root.descendants().slice(1))
    .enter()
    .append('g')
      .attr('class', 'node');

  nodes.append('rect')
    .attr('x', d => d.x0)
    .attr('y', d => d.y0)
    .attr('width', d => d.x1 - d.x0)
    .attr('height', d => d.y1 - d.y0)
    .attr('fill', d => descendantsDarker(d, color, false));

  this.container
    .append('g')
    .attr('id', 'legend')
    .attr('transform', &grave;translate(${this.innerWidth - 100}, 0)&grave;)
    .call(legend.legendColor().scale(houseColors));

  nodes.call(tooltip(d => d.data.itemLabel, this.container));
};
```

Знакомо? Держу пари, что да. Как и в случае древовидных карт, мы отрезаем корневой узел и вычисляем длины сторон прямоугольников, вычитая левый верхний угол из правого нижнего. Поместите следующую строку в файл `main.js`, закомментировав предыдущую инициализацию `westerosChart`:

```
westerosChart.init('partition', 'data/GoT-lineages-screentimes.json');
```

Диаграмма будет выглядеть так:



Нравится? Лично я предпочитаю древовидные карты.

Раз, два, три, четыре, пять – начинаем паковать

Макет упаковка (pack) порождает такие же диаграммы, как древовидная карта, только с круглыми узлами. Это, пожалуй, лучшая из трех последних диаграмм для такого представления иерархии – вспомните, что в древовидной карте нам пришлось задать несуразно большие промежутки, чтобы родительские узлы были хорошо различимы. Поскольку для упаковки кругов нужно больше места, родительские узлы видны отчетливо, и связь между родителями и детьми более ярко выражена.

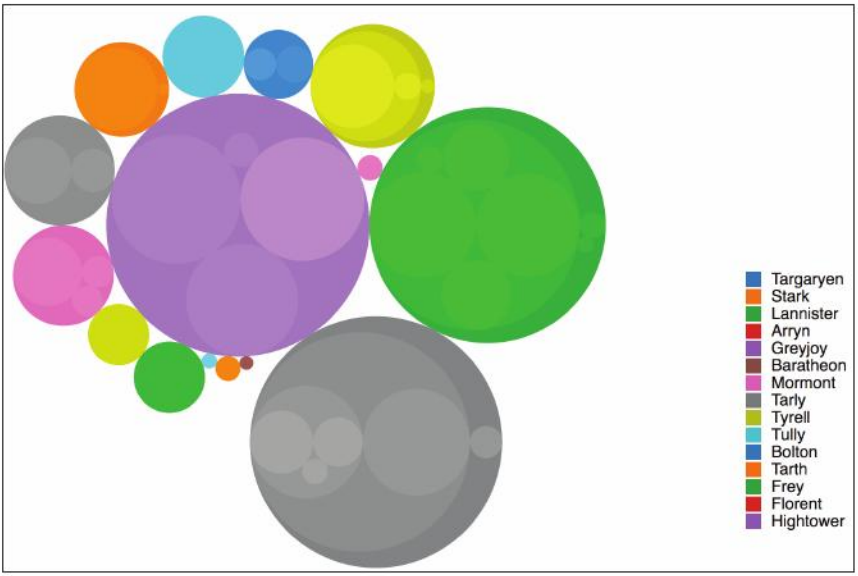
Как и раньше, закомментируйте все предыдущие вызовы `westerosChart.init()` в файле `main.js` и добавьте такой:

```
westerosChart.init('partition', 'data/GoT-lineages-screentimes.json');
```

Затем поместите такой код в файл `chapter6/index.js`:

```
westerosChart.pack = function Pack(_data) {
  const data = getMajorHouses(_data);

  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(valueComparator);

  const houseColors = color.copy().domain(houseNames(root));
  fixateColors(data, 'itemLabel');

  const layout = d3.pack()
    .size([
      this.innerWidth - 100,
      this.innerHeight,
    ]);

  layout(root);

  const nodes = this.container.selectAll('.node')
    .data(root.descendants().slice(1))
    .enter()
    .append('circle')
      .attr('class', 'node')
      .attr('cx', d => d.x)
      .attr('cy', d => d.y)
      .attr('r', d => d.r)
      .attr('fill', d => descendantsDarker(d, color, true, 5));

  this.container
    .append('g')
    .attr('id', 'legend')
    .attr('transform',
      &grave;translate(${this.innerWidth - 100}, ${this.innerHeight / 2})&grave;)
    .call(legend.legendColor().scale(houseColors));

  nodes.call(tooltip(d => d.data.itemLabel, this.container));
};
```

Практически то же самое, что для предыдущей диаграммы, только надпись расположена ближе к нижнему краю и вместо прямоугольников добавляются круги. Должно получиться вот что:



И на закуску – солнце светит из-за туч!

Думали, всё? А вот и нет – давайте-ка выжмем из `d3-hierarchy` еще одну диаграмму, а уж потом двинемся дальше.

Если помните, диаграмма разбиения слегка неопрятна, потому что применяется в основном для наборов данных, в которых значения ассоциированы только с листовыми узлами (самыми внешними, не имеющими потомков). Поскольку для некоторых родителей в нашем наборе данных свойство `screentime` определено, в диаграмму вносятся искажения, что и придает ей странность. Мы перерисуем ее, но на этот раз сделаем аккуратной и круговой.

Порядок действий уже известен. Сначала `main.js`:

```
westerosChart.init('radialPartition', 'data/GoT-lineagesscreentimes.json');
```

Затем `chapter6/index.js`:

```
westerosChart.radialPartition = function RadialPartition(_data) {
  const data = getMajorHouses(_data)
```

```
    .map((d, i, a) => Object.assign(d, {
      screentime: a.filter(v =>
        v.fatherLabel === d.itemLabel).length ? 0 : d.screentime,
    }) );
  const radius = Math.min(this.innerWidth, this.innerHeight) / 2;
};
```

Начинаем с радиуса, равного половине наименьшего размера, и отображаем данные, так что для всех узлов, являющихся чьими-то предками, свойство `screentime` сбрасывается в нуль.

Далее создаем корень и цветовой масштаб домов, как и раньше:

```
  const stratify = d3.stratify()
    .parentId(d => d.fatherLabel)
    .id(d => d.itemLabel);

  const root = stratify(data)
    .sum(d => d.screentime)
    .sort(null);

  const houseColors = color.copy().domain(root.ancestors().shift()
    .children.map(d => d.id.split(' ')[d.id.split(' ').length - 1])
  );
```

Создаем макет разбиения, но ширину и высоту устанавливаем равными половине родительской диаграммы. Это связано с тем, что мы вычисляем радиусы, а значит, в итоге все установленные сейчас значения удвоятся.

```
  const layout = d3.partition()
    .size([
      this.innerWidth / 2,
      this.innerHeight / 2,
    ])
    .padding(1)
    .round(true);
```

Создаем масштаб `x` и генератор дуг:

```
  const x = d3.scaleLinear()
    .domain([0, radius])
    .range([0, Math.PI * 2]);

  const arc = d3.arc()
    .startAngle(d => x(d.x0))
    .endAngle(d => x(d.x1))
    .innerRadius(d => d.y0)
    .outerRadius(d => d.y1);
```

В качестве области определения указываем радиус, а в качестве области значений – удвоенное число PI.

Применяем макет к данным и создаем соединение:

```
layout(root);

const nodes = this.container
  .append('g')
  .attr('class', 'nodes')
  .attr('transform',
    &grave;translate(${this.innerWidth / 2}, ${this.innerHeight / 2}))
  .selectAll('.node')
    .data(root.descendants().slice(1))
    .enter()
    .append('g')
    .attr('class', 'node');
```

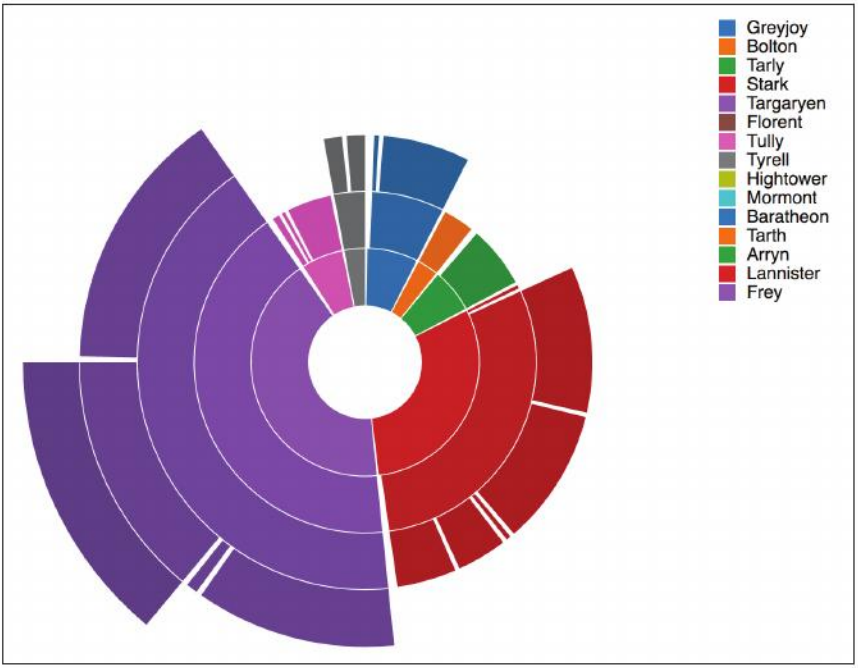
Добавляем пути, указывая в качестве значения d генератор пути:

```
nodes.append('path')
  .attr('d', arc)
  .attr('fill', d => descendantsDarker(d, color, false));
```

Наконец, настраиваем надпись и всплывающие пояснения:

```
this.container
  .append('g')
  .attr('id', 'legend')
  .attr('transform', &grave;translate(${this.innerWidth - 100}, 0)&grave;)
  .call(legend.legendColor().scale(houseColors));

nodes.call(tooltip(d => d.data.itemLabel, this.container));
```

Всё! Полюбуйтесь:



Должен сказать, это определенно лучше, чем прямоугольная версия.

Резюме

Хотя, на первый взгляд, макеты D3 обладают чуть ли не волшебным могуществом, на деле это не что иное, как вспомогательные объекты для преобразования данных в коллекцию координат. С помощью иерархических макетов мы создали целый ряд различных диаграмм, причем различия были сосредоточены всего в нескольких строчках.

Эти диаграммы настолько просты, что мы могли бы существенно расширить наш базовый объект; на такие вещи, как фиксация цветов и добавление надписи в методе `init()` (в файлах `chapter6/index.js` и `common/index.js`), ушло порядка 600 строк, а можно было бы пару сотен строк сэкономить. Однако и создавать диаграммы по отдельности тоже полезно – это способствует лучшему усвоению порядка использования иерархических макетов, потому я так и поступил.

В следующей главе мы рассмотрим еще несколько макетов. С точки зрения написания они напоминают иерархические, но придется немного по-другому подготовить данные, чтобы они соответствовали различным стилям, которые предлагают неиерархические макеты. К концу книги вы освоите все базовые вещи. Надеюсь, вас впечатляет то, что мы делаем.

Глава 7

Другие макеты

В предыдущей главе мы использовали иерархические макеты из модуля `d3-hierarchy` для создания изящных диаграмм. В этой главе мы рассмотрим другие макеты, включенные в версию D3 v4 и разбросанные по нескольким модулям. Мы будем называть их не *обычными* и не *неиерархическими*, а просто *другими*, потому что между ними мало общего.

Да здравствует модульный код

Приступим. Откройте файл `lib/main.js` и добавьте такую строку в секцию импорта:

```
import westerosChart from './chapter7/index';
```

Создайте каталог `chapter7` и в нем файл `index.js`. Поместите в него следующий код:

```
import * as d3 from 'd3';
import * as legend from 'd3-svg-legend';
import baseChart from '../chapter6/';
import './chapter7.css';
import {
  colorScale as color,
  tooltip,
  connectionMatrix,
  uniques,
} from '../common';
```

Здесь мы импортируем объект `westerosChart` из предыдущей главы под именем `baseChart`, намереваясь его расширить. Вот, собственно, всё и готово для создания первой диаграммы.

Летит пирог – румяный бок

Секторные диаграммы – очень распространенный способ представления количественных данных, но и у них есть ограничения – человеку труднее воспринять размер круговой области, и, чтобы можно было уверенно сравнивать секторы, их следует упорядочить по убыванию угла по часовой стрелке. При всем при том такие диаграммы настолько популярны, что без умения создавать их не обойтись никому, кто занимается визуализацией данных, потому что рано или поздно его попросят это сделать.

Макет секторной диаграммы находится в пакете `d3-shape` и занимает место где-то между макетами из предыдущей главы и генераторами линий из главы 3. Мы создаем макет секторной диаграммы, передавая ему массив чисел, а затем передаем макет генератору дуг для создания диаграммы. Перейдем к делу.

Для начала отфильтруем персонажей, проведших на экране меньше 60 минут, и создадим генератор секторов, при этом метод `value()` будет сообщать генератору, какое поле использовать для определения размера сектора. Далее создадим генератор, задав внешний радиус равным четверти размера экрана. Наконец, создадим группу и сдвинем ее в центр экрана:

```
westerosChart.pie = function Pie(_data) {
  const data = _data.filter(d => d.screentime > 60);
  const pie = d3.pie().value(d => +d.screentime);
  const arc = d3.arc()
    .outerRadius(this.innerWidth / 4)
    .innerRadius(null);

  const chart = this.container.append('g')
    .classed('pie', true)
    .attr('transform', &grave;translate(${this.innerWidth / 2},
        ${this.innerHeight / 2})&grave;);
  };
  export default westerosChart;
```

Следующий шаг – рисование секторов. Добавьте такой код в функцию `Pie()`:

```
  const slices = chart.append('g')
    .attr('class', 'pie')
    .selectAll('.arc')
    .data(pie(data).sort((a, b) => b.data.screentime - a.data.screentime))
    .enter()
    .append('path')
```

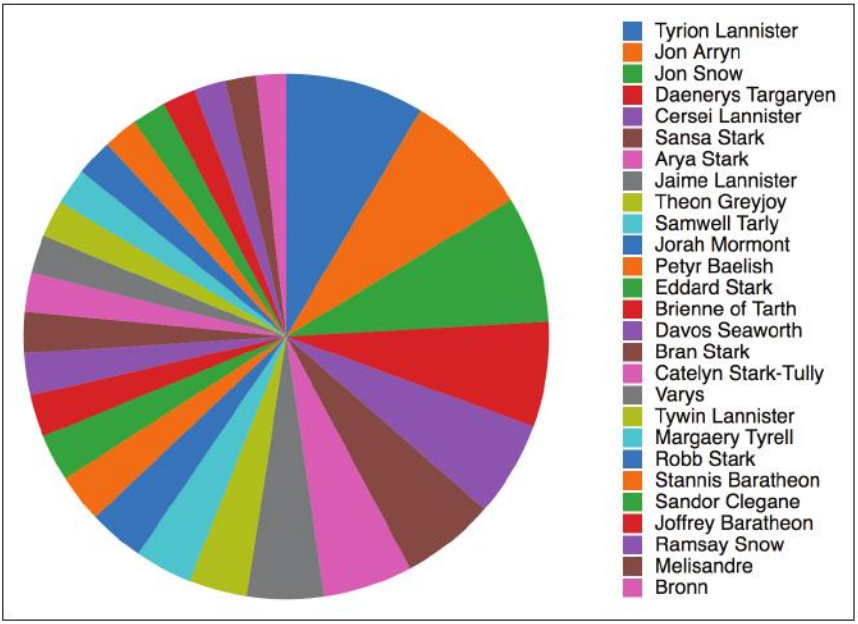
```
    .attr('d', arc)
    .classed('arc', true)
    .attr('fill', d => color(d.data.itemLabel));
```

Здесь мы передаем данные генератору секторов, а затем сортируем возвращенный им объект по свойству `screentime` (которое теперь находится в свойстве `data` элемента данных, поскольку мы смотрим на результат работы макета). Результат передается генератору дуг и используется для генерации путей. Затем мы передаем наш идентификатор `itemLabel` (он тоже находится в свойстве `data`) в цветовой масштаб.

Добавим надпись и всплывающие пояснения:

```
slices.call(tooltip(d => d.data.itemLabel, this.container));
this.container
  .append('g')
  .attr('id', 'legend')
  .attr('transform',
    &grave;translate(${this.innerWidth - 150}, ${(this.innerHeight / 2) -
250})&grave;)
  .call(legend.legendColor().scale(color));
```

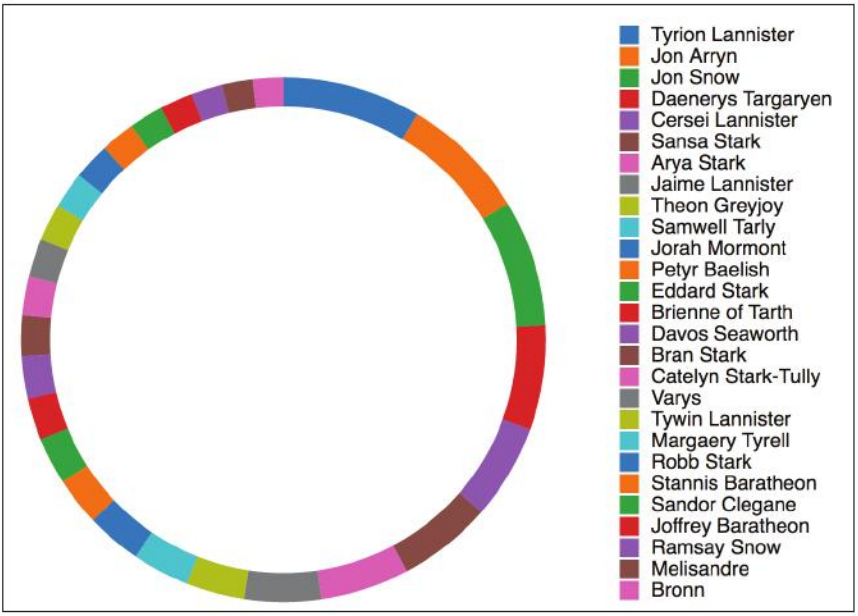
Вот как это выглядит:



Чтобы создать кольцевую диаграмму (секторную с дыркой внутри), просто задайте значение внутреннего радиуса `arc.innerRadius`:

```
.innerRadius(this.innerWidth / 4.5);
```

Получится такая картина:



Вот, пожалуй, и все, что можно сказать о макете секторной диаграммы. Несложно, правда?

Вы, вероятно, обратили внимание, что в этой диаграмме цвета повторяются, что является следствием порядкового масштаба. Дело в том, что данные содержат больше 10 элементов, поэтому наша цветовая схема повторяется. В предыдущей главе это не составляло проблемы, поскольку для интерпретации цветов мы полагались на естественную иерархию данных.

Для решения проблемы есть несколько вариантов. Можно было бы взять категориальную цветовую схему с 20 цветами, например `d3.schemeCategory20`, но это не выход, т. к. у нас больше 20 объектов. Можно было бы подойти к задаче так же, как в предыдущей главе, и кодировать отдельных персонажей цветами их дома, но тогда было бы трудно понять, к кому относится данный сектор. На практике лучший подход одновременно самый простой – ограни-

чить число секторов десятью, отобрав персонажей с наибольшим экранным временем, и создать отдельный сектор «прочие», в котором агрегировать всех остальных.
На каком подходе остановиться, зависит от аудитории и данных – просто помните, что чем больше секторов в диаграмме, тем труднее читателю интерпретировать размер конкретного сектора.
Оставляю решение проблемы в качестве упражнений для читателей, поскольку нам в этой главе предстоит рассмотреть еще много диаграмм, а секторная – наименее интересная из них.

Гистограммы-тристаграммы

Еще один пример макета – гистограмма, он упрощает создание столбчатых диаграмм в случае непрерывных данных. Этот макет можно также использовать для распределения множества значений по интервалам, чтобы не мучиться самостоятельно с функциями `Array.prototype.reduce` и `Array.prototype.map`.

В данном случае мы создадим порядковый масштаб эпизодов и сезонов и с его помощью построим гистограмму. Для этой цели мы воспользуемся другим набором данных, `GoT-deaths-by-season.json`, который тоже находится в каталоге `data/`. В него включены данные обо всех смертях, приключившихся в сериале, в следующем формате:

```
{
  "name": "Will",
  "role": "Ranger of the Night's Watch",
  "death": {
    "season": 1,
    "episode": 1
  },
  "execution": "Beheaded for desertion by Ned Stark",
  "likelihoodOfReturn": "0%"
},
```

Единственное, что нас здесь интересует, – объект `death`, который понадобится для создания порядкового масштаба.

Для начала закомментируйте в файле `main.js` все строки, содержащие `westerosChart`, и добавьте вместо них такую:

```
westerosChart.init('histogram', 'data/GoT-deaths-by-season.json');
```

Откройте файл `chapter7/index.js` и добавьте в него такой код:

```
westerosChart.histogram = function histogram(_data) {
  const data = _data.data.map(d =>
```

```
    Object.assign(d, { death: (d.death.season * 100) + d.death.episode }))
      .sort((a, b) => a.death - b.death);
  const episodesPerSeason = 10;
  const totalSeasons = 6;
  const allEpisodes = d3.range(1, totalSeasons + 1).reduce((episodes, s) =>
    episodes.concat(d3.range(1, episodesPerSeason + 1).map(e => (s * 100) +
e)), []);
```

Здесь мы заменяем объект `death` строкой, в которой номер сезона умножается на 100 и складывается с номером эпизода, после чего сортируем данные сначала по сезону, а затем по эпизоду. Кроме того, мы создаем массив из 60 элементов описанного выше формата.

Последний шаг необязателен, но мы хотим показать все эпизоды (даже если в них не было смертей), поэтому инициализируем масштаб по оси x, как описано ниже. Это несколько отличается от типичного использования гистограмм для представления непрерывных данных. Увы, в этом наборе непрерывных данных нет, поэтому придется удовольствоваться этим.

Далее создаем масштаб по оси x:

```
const x = d3.scaleBand()
  .range([0, this.innerWidth])
  .domain(allEpisodes)
  .paddingOuter(0)
  .paddingInner(0.25);
```

Здесь мы используем порядковый полосный масштаб и в качестве области определения указываем только что созданный массив `allEpisodes`.

Следующий шаг – создать генератор макета гистограммы и передать ему данные, как показано ниже:

```
const histogram = d3.histogram()
  .value(d => d.death)
  .thresholds(x.domain());
const bins = histogram(data);
```

Мы конфигурируем акцессор `value`, так чтобы он возвращал значение `death`, и с помощью метода `histogram.thresholds()` настраиваем границы каждого интервала.

Метод `histogram.thresholds()` принимает массив, содержащий последовательность значений, определяющих границы интервалов: первый интервал расположен между первым и вторым элементами массива, второй – между вторым и третьим и т. д.

В ответ метод возвращает массив интервалов. Каждый *интервал* (bin) – это массив, содержащий все отнесенные к интервалу данные, его свойство `length` равно числу элементов в массиве, а свойства `x0` и `x1` определяют границы интервала, как было объяснено выше. Нижняя граница (`x0`) включается, верхняя (`x1`) не включается (во все интервалы, кроме последнего).

Далее создаем масштаб по оси `y`:

```
const y = d3.scaleLinear()
  .domain([0, d3.max(bins, d => d.length)])
  .range([this.innerHeight - 10, 0]);
```

Тут ничего сложного: мы получаем максимальное количество элементов в интервале методом `d3.max()` и задаем область значений на 10 пикселей меньше, чем `innerHeight` (под осью `x` еще нужно разместить метки, а высота каждой строки составляет 10 пикселей).

Пора уже добавить столбики:

```
const bar = this.container.selectAll('.bar')
  .data(bins)
  .enter()
    .append('rect')
    .attr('x', d => x(d.x0))
    .attr('y', d => y(d.length))
    .attr('fill', tomato')
    .attr('width', () => x.bandwidth())
    .attr('height', d => (this.innerHeight - 10) - y(d.length));
```

Каждый столбик укорачивается на 10 пикселей, чтобы оставить место для надписей, но во всех остальных отношениях гистограмма выглядит так же, как столбчатая диаграмма. Для задания ширины мы воспользовались методом `bandwidth()`, а если бы не использовали здесь порядковый масштаб `bandScale`, то могли бы вычесть `x0` из `x1` и передать результат x-масштабу для получения ширины каждого столбика.

Почти готово! Надо еще добавить ось x:

```
const xAxis = this.container.append('g')
  .attr('class', 'axis x')
  .attr('transform', &grave;translate(0, ${this.innerHeight - 10})&grave;)
  .call(d3.axisBottom(x).tickFormat(
    d => &grave;S${(d - (d % 100)) / 100}E${d % 100}&grave;));
xAxis.selectAll('text')
  .each(function (d, i) {
    const yVal = d3.select(this).attr('y');
```

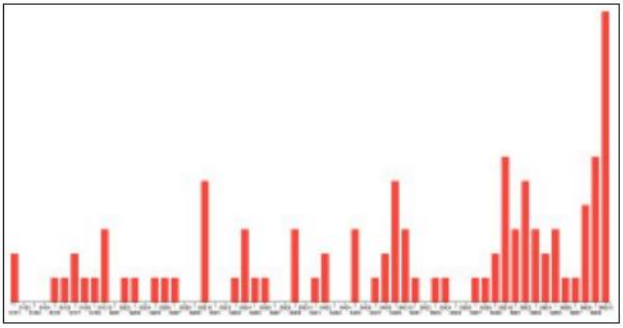
```
    d3.select(this).attr('y', i % 2 ? yVal : (yVal * 2) + 2)
  });
xAxis.selectAll('line')
  .each(function (d, i) {
  const y2 = d3.select(this).attr('y2');
  d3.select(this).attr('y2', i % 2 ? y2 : y2 * 2)
});
```

Все почти так же, как для любой оси x, только мы решили украсить гистограмму, сделав надпись двухэтажной, для чего сдвинули каждую вторую метку вниз на величину, чуть большую ее удвоенного значения `y` (заметим, что метка находится внутри группы, поэтому значение `y` измеряется относительно нее) и соответственно продлив отвечающее ей деление. Чтобы выполнить эту операцию для каждого элемента в выборке, мы воспользовались методом `d3.each`, который записывает в контекст `this` ссылку на текущий элемент. А чтобы сохранить контекст, установленный `d3.each`, мы применяем обычные функции, а не двойную стрелку.

Наконец, настраиваем наш любимый генератор всплывающих пояснений:

```
bar.call(tooltip((d) =>
  &grave;${d.x0}: ${d.length} deaths&grave;, this.container));
```

После сохранения вы увидите вот такую симпатичную столбчатую диаграмму:



Ого, шестой-то сезон оказался каким урожайным!

Секундочку, а разве так можно использовать гистограмму? Вообще-то, нет; обычно гистограмма получает последовательность отсчетов и разбивает ее на дискретные порции. Можно было бы возразить, что простая столбчатая диаграмма с порядковым

масштабом по оси x была бы уместнее, но, по крайней мере в данном случае, гистограмма оказывается немного проще, потому что автоматически производит все необходимое агрегирование, пусть даже нам пришлось проделать не вполне естественные манипуляции, чтобы ось x имела какой-то смысл. Вообще, если вы не уверены, какие деления должны быть на оси x, и имеется большой объем данных, которые желательно представить в виде столбчатой диаграммы, подумайте об использовании гистограммы. Если данные уже классифицированы, так что можно обойтись порядковым масштабом по оси x, то так и поступите.

Хордовый аккорд

Макет `chord` создает круговую диаграмму, показывающую связи в наборе данных. В этом разделе мы будем использовать еще один набор данных из каталога `data/` – `stormofswords.csv`, взятый со страницы https://www.macalester.edu/~abeverid/thrones.html.

Этот набор отражает близость имен персонажей в тексте книг, на основе которых написан сценарий, а его цель – определить веса связей между персонажами. Это идеальный набор данных для двух следующих примеров, в которых представлены произвольные неиерархические связи между данными.

Прежде всего закомментируйте в файле `main.js` последний пример и добавьте такую строку:

```
westerosChart.init('chord', 'data/stormofswords.csv');
```

В файл `chapter7/index.js` добавьте заготовку для конструктора новой диаграммы:

```
westerosChart.chord = function Chord(_data) {};
```

Все до боли знакомо. Далее создадим массив источников и связей между ними, оставляя только связи с весом больше 20. Добавьте следующий код в функцию `Chord`:

```
const minimumWeight = 20;
const majorLinks = _data.filter(d => +d.Weight > minimumWeight);
const majorSources = uniques(majorLinks, d => d.Source);
const data = majorLinks.filter(d => majorSources.indexOf(d.Target) > -1);
```

Для получения массива данных мы отфильтровываем связи, конец которых не совпадает ни с одним источником.

Передадим этот массив функции для построения матрицы связей, представляющей собой массив массивов. Каждый элемент этой мат-

рицы ссылается на другие значения циклически. Поместите функцию connectionMatrix в файл common.js:

```
export function connectionMatrix(data, sourceKey = 'source', targetKey =
'target', valueKey = 'value') {
  const nameIds = nameId(allUniqueNames(data, 'Source', 'Target'), d => d);
  const uniqueIds = nameIds.domain();
  const matrix = d3.range(uniqueIds.length).map(() =>
    d3.range(uniqueIds.length).map(() => 1));
  data.forEach((d) => {
    matrix[nameIds(d[sourceKey])][nameIds(d[targetKey])] += Number(d[valueKey]);
  });

  return matrix;
}
```

Мы создаем масштаб, содержащий все уникальные имена в массиве data. Затем получаем массив уникальных имен, запрашивая область определения этого масштаба. После этого дважды используем d3.range(), чтобы получить матрицу массивов, заполненных единицами. Потом заменяем каждую единицу весом связи, взятым из набора данных, и, наконец, возвращаем матрицу.

Откройте файл chapter7/index.js и добавьте следующий код в функцию Chord:

```
const matrix = connectionMatrix(data, 'Source', 'Target', 'Weight');
const outerRadius = (Math.min(this.width, this.height) * 0.5) - 40;
const innerRadius = outerRadius - 30;
```

Мы создаем матрицу связей, а затем определяем внешний и внутренний радиусы. Далее мы создаем генератор хордового макета, генератор дуг и генератор лент:

```
const chord = d3.chord()
  .padAngle(0.05)
  .sortSubgroups(d3.descending);

const arc = d3.arc()
  .innerRadius(innerRadius)
  .outerRadius(outerRadius);

const ribbon = d3.ribbon()
  .radius(innerRadius);
```

А теперь за дело. Сначала добавим в контейнер группу, затем передадим нашу матрицу хордовому макету:

```
const chart = this.container
  .append('g')
  .attr('class', 'chord')
  .attr('transform', &grave;translate(${this.innerWidth / 2},
      ${this.innerHeight / 2})&grave;)
  .datum(chord(matrix));
```

Сдвинем ее в середину, потому что иначе она оказалась бы в левом верхнем углу.

Создадим группы для каждого элемента набора данных:

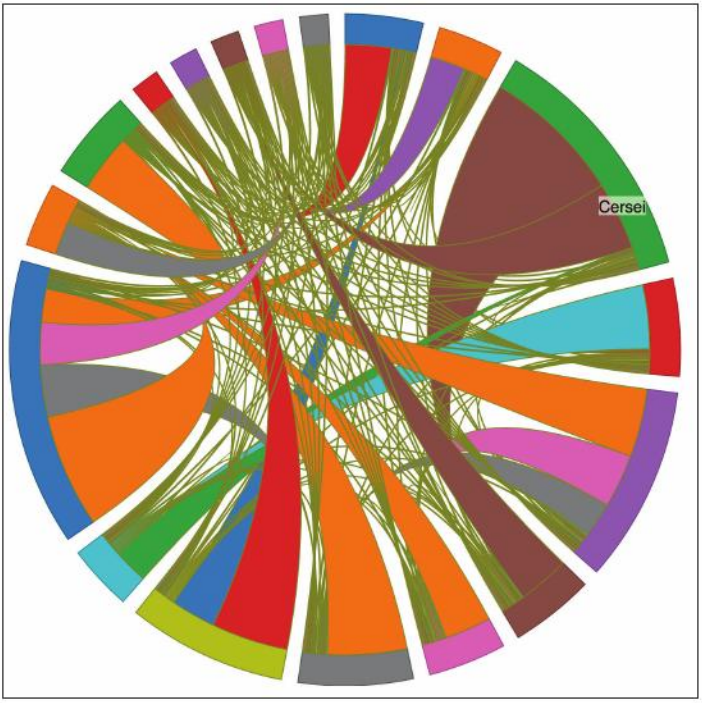
```
const group = chart.append('g').attr('class', 'groups')
  .selectAll('g')
  .data(chords => chords.groups)
    .enter()
    .append('g');

group.append('path')
  .style('fill', d => color(d.index))
  .style('stroke', d => d3.color(color(d.index)).darker())
  .attr('d', arc);
```

И напоследок добавим ленты, соединяющие группы:

```
const ribbons = chart.append('g').attr('class', 'ribbons')
  .selectAll('path')
  .data(chords => chords)
  .enter()
    .append('path')
    .attr('d', ribbon)
    .style('fill', d => color(d.target.index))
    .style('stroke', d => d3.color(color(d.index)).darker());
```

Осталось только сформировать всплывающие пояснения для лент и групп:

```
ribbons.call(tooltip(d => majorSources[d.target.index], this.container));
group.call(tooltip(d => majorSources[d.index], this.container));
```



Красиво-то как!

Да пребудет с вами сила

Один из наиболее интересных макетов, имеющихся в D3 (в пакете `d3-force`), –моделирование действующих сил. Он действительно полезен, т. к. позволяет расположить элементы на экране в соответствии с моделируемыми физическими свойствами. Это важно, когда позиционирование элемента на экране никак не связано с данными, например в диаграмме сети, где связи между объектами важнее их положения на экране.

Мы воспользуемся набором данных о связях из предыдущего примера и создадим сеть связей в сериале. В файле `main.js` закомментируйте последний пример и добавьте такую строку:

```
westerosChart.init('force', 'data/stormofswords.csv');
```

Добавьте в файл `chapter7/index.js` новую функцию:

```
westerosChart.force = function Force(_data) {
  const nodes = uniques(
    _data.map(d => d.Target)
      .concat(_data.map(d => d.Source)), d => d)
      .map(d => ({ id: d, total: _data.filter(e => e.Source === d).length }));

  fixateColors(nodes, 'id');
}
```

Мы создаем массив уникальных узлов формата

```
{id: <имя-персонажа>, total: <всего связей>}
```

Больше в этом наборе практически никаких данных нет, а связать его с другим набором данных затруднительно, поскольку отсутствуют фамилии персонажей. Будь у нас больше данных, мы могли бы сделать это на данном этапе (например, создать справочную таблицу с идентификаторами персонажей в наборе данных о связях в качестве ключа, а затем соединить ее по этому ключу с другим набором данных) и присоединить к каждому узлу дополнительные данные. В данном же случае мы получаем число связей для каждого узла, впоследствии мы используем эту величину для задания размеров узлов.

Мы также фиксируем цветовой масштаб и создаем массив связей между узлами, причем вес связи характеризует силу данного отношения. Веса используются для задания толщины линий, соединяющих узлы. Добавьте следующий код в функцию `westerosChart.force`:

```
  const links = _data.map(d =>
    ({ source: d.Source, target: d.Target, value: d.Weight }));
```

Сначала мы добавляем все узлы и связи, а затем настраиваем моделирование, чтобы вся система могла перемещаться по экрану. Добавьте в конец `westerosChart.force` следующий код:

```
const link = this.container.append('g').attr('class', 'links')
  .selectAll('line')
  .data(links)
    .enter()
    .append('line')
    .attr('stroke', d => color(d.source))
    .attr('stroke-width', d => Math.sqrt(d.value));
```

В результате на экране проводятся линии, соединяющие узлы. В качестве толщины линии мы берем квадратный корень из значения и раскрашиваем каждую связь в цвет персонажа в ее начале. Поместите вслед за этим такой код:

```
const radius = d3.scaleLinear().domain(
  d3.extent(nodes, d => d.total)).range([4, 20]);
const node = this.container.append('g').attr('class', 'nodes')
  .selectAll('circle')
  .data(nodes)
    .enter()
    .append('circle')
    .attr('r', radius)
    .attr('fill', d => color(d.id))
    .call(d3.drag()
      .on('start', dragstart)
      .on('drag', dragging)
      .on('end', dragend));
```

Мы определяем масштаб для задания размеров радиусов в диапазоне от 4 до 20 с линейной интерполяцией. Затем помещаем на экран ряд кругов и к каждому присоединяем обработчики событий, которые напишем чуть ниже.

Сопоставим узлам всплывающие пояснения:

```
node.call(tooltip(d => d.id, this.container));
```

Наконец, настроим моделирование. Мы ограничиваемся самыми простыми параметрами, хотя в пакете `d3-force` есть много других настраиваемых свойств:

```
const sim = d3.forceSimulation()
  .force('link', d3.forceLink().id(d => d.id).distance(200))
  .force('charge', d3.forceManyBody())
  .force('center', d3.forceCenter(this.innerWidth / 2, this.innerHeight / 2));
```

Здесь задается макет моделирования сил с тремя силами: `link`, `charge` и `center`.

- Сила `link` описывает натяжение, возникающее из-за наличия материального соединения между двумя телами. Ее можно использовать для придания эластичности связям или для того, чтобы притянуть два узла ближе друг к другу. Мы задали расстояние 200, чтобы узлы находились достаточно далеко друг от друга. Если бы мы захотели больше полагаться на другие силы, то не стали бы задавать расстояние, потому что это слишком жесткое условие.
- Сила `charge` означает, что объекты на экране моделируются как электрические заряды, а для вычисления их положения решается *задача многих тел*. Это полезно, потому что мы можем моделировать притяжение и отталкивание объектов, причем

последнее – эффективный способ гарантировать, что никакие объекты не перекрываются. Мы настроили модель со стандартными параметрами, при которых всем объектам назначается небольшой отрицательный заряд, чтобы они немного отталкивались.

- Наконец, сила `center` вызывает притяжение к центральной точке, координаты которой заданы в конструкторе.

Теперь добавьте такие строки:

```
sim.nodes(nodes).on('tick', ticked);
sim.force('link').links(links);
```

Тем самым мы добавляем в модель узлы и связи.

Метод `.nodes()` устанавливает следующие свойства для каждого переданного ему объекта:

- `index`: индекс узла, начинающийся с нуля;
- `x`, `y`: текущие координаты узла;
- `vx`, `vy`: текущие компоненты скорости узла в направлениях x и y.

Можно также задать свойства `fx` и `fy`, чтобы зафиксировать узел в определенной позиции. Мы используем их в обработчиках события буксировки, чтобы была возможность перемещать узел.

Это еще не всё. Необходимо написать все анонсированные ранее обработчики. Начнем с того, что происходит в момент *такта* (tick) моделирования. Добавьте в конец функции `Force()` такой код:

```
function ticked() {
  link.attr('x1', d => d.source.x)
      .attr('y1', d => d.source.y)
      .attr('x2', d => d.target.x)
      .attr('y2', d => d.target.y);

  node.attr('cx', d => d.x)
      .attr('cy', d => d.y);
}
```

Моделирование сил просто обновляет значения свойств данных, но не приводит к перерисовке. Это мы должны сделать сами, задав определенные свойства элементов, представляющих узлы и связи.

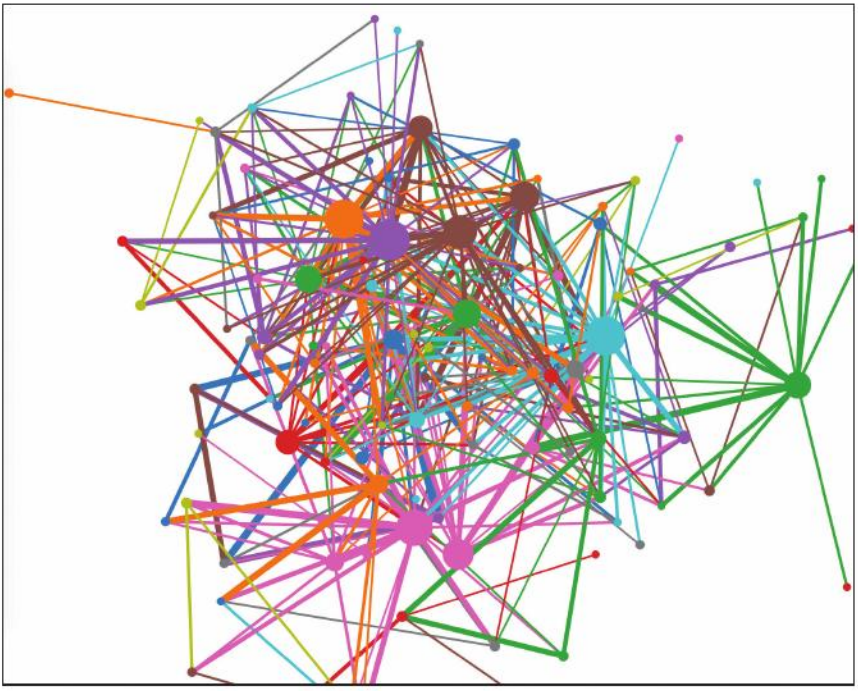
Сразу вслед за этим добавьте еще три функции, определяющие, как узлы должны взаимодействовать с событиями мыши:

```
function dragstart(d) {
  if (!d3.event.active) sim.alphaTarget(0.3).restart();
  d.fx = d.x;
  d.fy = d.y;
```

```
}

function dragging(d) {
  d.fx = d3.event.x;
  d.fy = d3.event.y;
}

function dragend(d) {
  if (!d3.event.active) sim.alphaTarget(0);
  d.fx = null;
  d.fy = null;
}
```

Мы задаем свойства, описывающие фиксированную позицию, равными координатам события мыши. После прекращения буксировки они сбрасываются в `null`, чтобы узел вернулся туда, где был.

После сохранения вы увидите вот такую красоту:



В модуле `d3-force` есть гораздо больше, чем мы смогли продемонстрировать здесь. Посмотрите в документации, какие еще есть стили сил и какие значения можно задать.

По стопочке по маленькой налей, налей, налей...

Стопочный макет позволяет составлять из участков диаграммы стопки, что бывает полезно, например, для построения накопительных диаграмм (stacked area chart). Макет находится в пакете `d3-shape`.

Закомментируйте последний пример в файле `main.js` и добавьте вместо него такой код:

```
westerosChart.init('stack', 'data/GoT-deaths-by-season.json', false);
```

Мы добавили дополнительный параметр, потому что собираемся построить в этом примере две диаграммы. На этот раз мы пользуемся набором данных `deaths-by-season`.

Добавьте следующий код в конец файла `chapter7/index.js`:

```
westerosChart.stack = function Stack({ data }, isStream = false) {
  const episodesPerSeason = 10;
  const totalSeasons = 6;
  // Создать вложенную структуру, содержащую сведения о смертях в каждом эпи-
зоде
  const seasons = d3.nest()
    .key(d => d.death.episode)
    .key(d => d.death.season)
    .entries(data.filter(d => !d.death.isFlashback))
    .map(v => {
      return d3.range(1, totalSeasons + 1)
        .reduce((item, episodeNumber) => {
          const deaths = v.values.filter(d =>
            +d.key === episodeNumber)
            .shift() || 0;
        item[&grave;season-${episodeNumber}&grave;] = deaths ? deaths.values.
length : 0;
        return item;
      }, { episode: v.key });
    })
    .sort((a, b) => +a.episode - +b.episode);
}
```

Мы написали функцию с двумя аргументами, один из которых должен быть объектом с атрибутом `data`, а другой – булевым флагом, определяющим, хотим мы построить потоковую диаграмму или нет. Затем создается вложенная структура объектов и отфильтровываются персонажи, умершие в ретроспективных кадрах. В результате полу-

чается массив эпизодов, в каждом из которых хранится число смертей в сезоне. Доступ к этому массиву производится по ключу `season-n`.

Затем мы настраиваем генератор стопочного макета и отображаем его на ключ `season-n`:

```
const stack = d3.stack()
  .keys(d3.range(1, totalSeasons + 1)
  .map(key => &grave;season-${key}&grave;));
```

Следующая строка позволяет сделать диаграмму потоковой, если второй аргумент функции-конструктора принимает значение «истина».

```
if (isStream) stack.offset(d3.stackOffsetWiggle);
```

Далее мы настраиваем масштабы по осям x и y:

```
const x = d3.scaleLinear()
  .domain([1, episodesPerSeason])
  .range([this.margin.left, this.innerWidth - 20]);

const y = d3.scaleLinear()
  .domain([
    d3.min(stack(seasons), d => d3.min(d, e => e[0])),
    d3.max(stack(seasons), d => d3.max(d, e => e[1]))
  ])
  .range([this.height - (this.margin.bottom + this.margin.top + 30), 0]);
```

Стопочный макет создает массив массивов, каждый из которых содержит максимальное и минимальное значения `y`. Мы можем передать их генератору областей:

```
const area = d3.area()
  .x(d => x(d.data.episode))
  .y0(d => y(d[0]))
  .y1(d => y(d[1]))
  .curve(d3.curveBasis);
```

Затем добавляем пути, пользуясь нашим генератором областей:

```
const stream = this.container.append('g')
  .attr('class', 'streams')
  .selectAll('path')
```

```
    .data(stack(seasons))
    .enter()
    .append('path')
      .attr('d', area)
      .style('fill', (d, i) => color(i));
```

Мы передаем наш объект данных о сезонах стопке, которая передается функции `.data()`. Затем с помощью генератора областей мы задаем форму участка.

Следующим шагом добавляем ось x:

```
this.container.append('g')
  .attr('class', 'axis')
  .attr('transform',
    &grave;translate(0,${this.height - (this.margin.bottom +
      this.margin.top + 30)})&grave;)
  .call(d3.axisBottom(x));
```

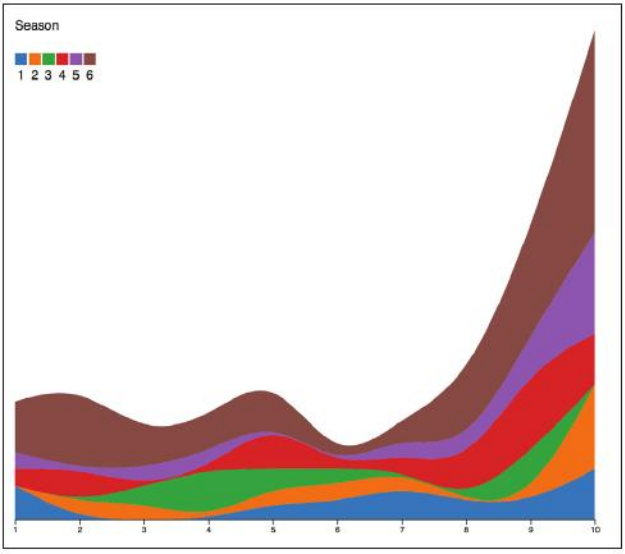
И надпись:

```
const legendOrdinal = legend.legendColor()
  .orient('horizontal')
  .title('Season')
  .labels(d => d.i + 1)
  .scale(color);

const legendTransform = isStream ?
  &grave;translate(50,${this.height -
    (this.margin.bottom + this.margin.top + 130)})&grave; :
    &grave;translate(50,0)&grave;;

this.container.append('g')
  .attr('class', 'legend')
  .attr('transform', legendTransform)
  .call(legendOrdinal);
```

И напоследок всплывающие пояснения:

```
stream.call(tooltip(d => &grave;Season ${d.index + 1}&grave;,
this.container));
```

Мы у цели!

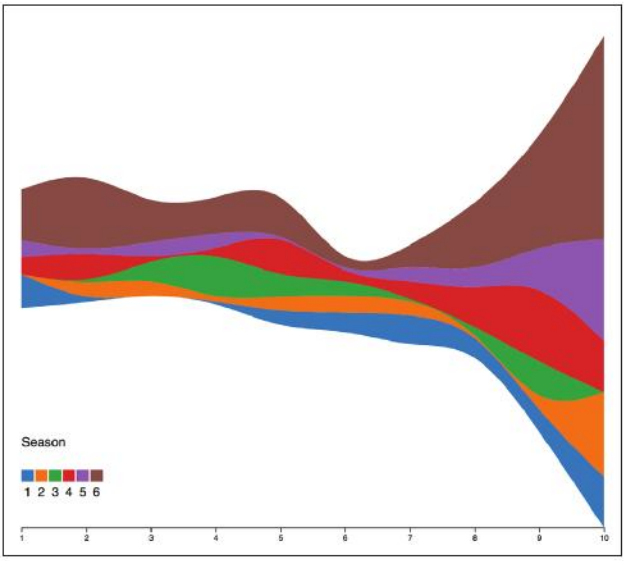


Бонусная диаграмма – сверкающие потоки!

С последней диаграммой в этой главе много возиться не придется – мы, по сути, ее уже сделали. Закомментируйте в файле `main.js` последнюю строку и добавьте вместо нее такую:

```
westerosChart.init('stack', 'data/GoT-deaths-by-season.json', true);
```

Всего-то и надо передать `true`. В результате получается потоковая диаграмма, потому что ранее мы присвоили свойству `offset` значение `d3.stackOffsetWiggle`. Выглядит прикольно, вы не находите?



Резюме

Мы рассмотрели все макеты, уделив каждому достаточно много внимания. Поскольку я указывал имена пакетов, вы можете почерпнуть дополнительные сведения в документации. У каждого макета есть много настраиваемых параметров, что позволяет получить широкое разнообразие диаграмм.

В примерах мы использовали почти все изученные к настоящему моменту приемы. Из написанного нами кода можно даже составить вполне приличную библиотеку утилит. Некоторые функции после незначительного обобщения могли бы стать отдельными макетами. Существует целое сообщество, занимающееся разработкой макетов для различных диаграмм. Неплохой отправной точкой для собственных исследований может стать репозиторий `d3-plugins` на GitHub (https://github.com/d3/d3-plugins).

Теперь вы понимаете, что делают макеты по умолчанию, и, надеюсь, уже подумываете о том, как можно было бы использовать их для решения задач, о которых разработчик даже не помышлял.

В следующей главе мы научимся работать с D3 *вне* браузера – да-да, вы не ослышались, мы направляемся в Серверный город! В этом путешествии мы распотрошим D3 и применим ее для отрисовки объектов, которые даже SVG не являются.

Глава 8

Использование D3 на сервере с применением Canvas, Koa 2 и Node.js

Вот теперь начинается самая неординарная сторона D3. Мало того что с ее помощью мы можем рисовать красивые диаграммы на стороне клиента, так она еще и умеет генерировать изображения до того, как они попадут в браузер пользователя. Это самый передний край библиотеки, но надо понимать, что навыки, освоенные в предыдущих главах, выручат вас в 95% случаев, так что если вы пока предпочитаете ограничиться клиентской частью и SVG, то так и сделайте. А эту главу оставьте до какого-нибудь дождливого вечерка, когда вам захочется узнать, как работает **Heroku**.

Подготовка окружения

Вы когда-нибудь писали приложения на PHP? Если да, то вас ждет приятный сюрприз. Веб-приложения, написанные на JavaScript, развертывать в миллион раз проще благодаря провайдерам, предоставляющим веб-хостинг в виде **платформы как услуги** (Platform-as-a-Service – PaaS), и к тому же вы можете управлять целым парком

серверов с помощью нескольких простых инструментов. Вместо того чтобы сражаться с гигантским монолитным конфигурационным файлом Apache или Nginx, вы можете развернуть новый экземпляр для каждого созданного приложения, который будет исполнять его в песочнице при гораздо более компактной инфраструктуре. Ниже в этой главе мы обсудим, как развертывать приложения в среде Heroku, а пока займемся локальным тестированием.

Что такое Heroku и нужно ли оно вам? Heroku – это технология развертывания приложений, основанная на *принципах 12 факторов* (детали см. на сайте http://12factor.net). Мы не будем глубоко вдаваться в философию двенадцатифакторных приложений, а скажем лишь, что основная идея заключается в создании приложений *без состояния*, в которых вместо монолитной серверной инфраструктуры (например, виртуального Linux-сервера, на котором работают веб-сервер и база данных) используются веб-службы. В данном случае я остановился на Heroku, потому что она бесплатна и допускает простое развертывание в условиях ограниченных ресурсов, но никто не мешает развернуть написанный в этой главе код на любой серверной инфраструктуре, в которой установлен Node.js.

Возможно, вы этого не осознаете, но практически все необходимое для написания серверного приложения уже находится в нашем каталоге проекта. Для тех, кто никогда раньше не разрабатывал для Node.js, скажу, что процесс очень напоминает то, что мы делали в предыдущих главах, но API и концепции несколько иные. И Google Chrome, и Node.js пользуются JavaScript-движком V8, поэтому вам не придется изучать совершенно другой язык или приобретать новые навыки, а можно сразу приступать к созданию серверных приложений.

Установим несколько дополнительных зависимостей с помощью `npm`:

```
$ npm install koa@next koa-bodyparser@next canvas-prebuilt --save
```

`Koa` – одна из ведущих библиотек для веб-сервера Node.js; она изящно абстрагирует способность Node открывать порты и отдавать контент с помощью простого API, очень легкого и быстрого. Концептуально она напоминает другую библиотеку для Node, Express, но претендует на звание более передовой. Мы будем использовать версию `next` программ `koa` и `koa-bodyparser`, потому что в ветви 2.x для управления потоком применяются ключевые слова `async` и `await`, а именно они интересуют нас в этой книге. Мы также установили уже собранную библиотеку `node-canvas` компании Automattic, которая позволяет отрисовывать элементы Canvas в Node.

Поскольку в Node используется стандарт загрузки модулей CommonJS, мы должны добавить дополнительные инструкции сборки в файл `webpack.config.js`, которые позволят преобразовать предложения импорта ES2015 в предложения `require()`, определенные в CommonJS. Для этого мы должны сделать экспортируемый из `webpack.config.js` объект массивом (так чтобы файл содержал обе конфигурации), в результате получится вот что (первая конфигурация для краткости опущена):

```
const path = require('path');
module.exports = [ // Превратить module.exports в массив!
  { ... }, // это первая конфигурация, пусть остается
  {
    entry: './lib/chapter8/index.js',
    target: 'node',
    output: {
      path: path.resolve(__dirname, 'build'),
      filename: 'server.js'
    },
    externals: {
      'canvas-prebuilt': 'commonjs canvas-prebuilt'
    },
    devtool: 'inline-source-map',
    module: {
      loaders: [
        {
          test: /.js?$/,
          exclude: 'node_modules',
          loader: 'babel'
        },
        {
          test: /.json$/,
          loader: 'json-loader'
        }
      ]
    }
  }
]; // Не забудьте закрывающую квадратную скобку!
```

В основном все то же самое, мы только изменили цель target на `node`. Единственное серьезное отличие состоит в том, что мы потребовали от Webpack рассматривать заранее собранную библиотеку `node-canvas` (которая понадобится нам ниже) как внешнюю и не пытаться включить ее в пакет. Собранный пакет будет находиться в файле `build/server.js`.

Отметим, что весь остальной код все равно будет собираться; если вам кажется, что это слишком медленно, временно закомментируйте первую секцию config.

Следует ли использовать Babel для серверных проектов? Это зависит от окружения на сервере, но, скорее всего, нет. Node.js достаточно оперативно поддерживает современные новации в синтаксисе JavaScript, поэтому транслировать код из одной версии синтаксиса в другую нет нужды. Тем не менее поддержка `async/await` появилась только в версии Node 7 (да и тогда до выхода версии 7.7.0 нужно было задавать интерпретатору флаг `-harmony`), поэтому, *не желая требовать от каждого читателя перехода на последнюю версию Node*, мы пропускаем код через Babel и Webpack. Кроме того, мы используем в книге синтаксис модулей ES6, а его во всех текущих версиях Node все-таки надо транслировать. Просто имейте в виду, что если вы работаете с версией Node 7+ и всюду употребляете `require()` вместо `import`, то все относящееся к Webpack и Babel можно спокойно игнорировать.

Всех пассажиров, отправляющихся в Серверный город, просим занять свои места в поезде Koa

Ну что ж, не откладывая в долгий ящик, перейдем к скучным подробностям. Поместите приведенный ниже код в файл `lib/chapter8/index.js`:

```
import * as Koa from 'koa';
import * as bodyParser from 'koa-bodyparser';
const app = new Koa();
const port = process.env.PORT || 5555;
app.use(bodyParser());
app.listen(port, () => console.log(&grave;Listening on port
${port}&grave;));
export default app;
```

Здесь мы импортируем Koa и анализатор Koa Body Parser, создаем экземпляр Koa, задаем номер порта (по умолчанию `5555`), просим Koa использовать ПО промежуточного уровня Body Parser (которое попросту переводит тела `POST`-запросов в объекты JavaScript), после чего Koa начинает прослушивать указанный порт.

Что такое `process.env.PORT`? `Process.env` – это просто глобальный объект, содержащий все переменные окружения, определенные в обо-

лочке, из которой запущен скрипт NodeJS. Мы прослушиваем запросы, поступающие в порт 5555 или *тот, который указан в переменной окружения* `PORT`. Я выделил эту фразу курсивом, потому что ключевой принцип разработки веб-приложений: хранить всю конфигурационную информацию в переменных окружения, чтобы не нужно было беспокоиться о незашифрованных паролях в исходном коде (в данном случае, правда, речь идет не о паролях, а о номере порта, который был сконфигурирован Heroku).

А что это за *переменные окружения*, о которых я все время толкую? Оболочка хранит ряд переменных, служащих разным целям. Наверное, вы слышали о переменной `$PATH`, содержащей список каталогов, в которых оболочка ищет исполняемые файлы. Переменные окружения можно использовать и в интересах веб-серверов для хранения деталей конфигурации, доступных веб-приложениям (в том числе и тому, которое мы собираемся написать).

Например, это полезно для хранения деталей подключения к базе данных – вообще говоря, мы не хотим помещать в систему управления версиями такие секретные сведения, как пароль доступа к базе, поэтому предоставляем их серверу в виде переменных окружения. У такого подхода есть и другие преимущества, но не будем перечислять их, а скажем лишь, что конфигурирование приложений с помощью переменных окружения – важный аспект создания хороших приложений для JavaScript-сервера.

Добавьте в конец такую строку:

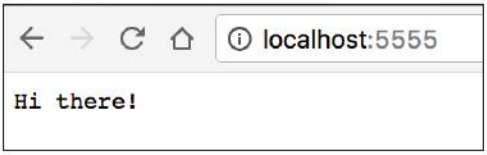
```
app.use(ctx => ctx.body = 'Hi there!');
```

Здесь мы создаем новый элемент промежуточного уровня и добавляем его в стек. Наш элемент принимает один аргумент `ctx` – *контекстный* объект Koa. При каждом поступлении запроса приложению создается новый контекст, который проходит через наш элемент промежуточного уровня. В отличие от Express, контекст Koa обрабатывает как запросы (входящие сообщения от пользовательского браузера приложению), так и ответ (сообщение, отправляемое приложением браузеру). В данном случае мы задали тело ответа – строку *Hi there!* – и передали элемент дальше. Рано или поздно мы дойдем до конца цепочки элементов промежуточного уровня, после чего ответ будет отправлен браузеру.

Теперь введите в командной строке такую команду:

```
$ npm run server
```

Перейдя в браузере по адресу `http://localhost:5555`, вы увидите такой текст:



Примите поздравление со вступлением в ряды разработчиков веб-приложений.

Ну, еще не совсем, но вы *быстро движетесь в этом направлении*.

Отметим, что команды с обратными вызовами в замыкании исполняются одновременно, как и в браузере. В главе 4 мы говорили, что JavaScript – *асинхронный* язык, и при написании кода для сервера мы не можем рассчитывать на какую-то блокировку, как, например, в PHP. Хотя начинающему разработчику для Node привыкнуть к этому поначалу довольно сложно, последствия такого подхода для серверного кода трудно переоценить – он становится очень быстрым и легко масштабируется по горизонтали.

Давайте не будем ограничиваться показом статического сообщения на экране, а рассмотрим совсем новую возможность D3.

Определение близости и диаграммы Вороного

Диаграмма Вороного (пакет `d3-voronoi`) разбивает географическую область на участки, окружающие заданные точки, таким образом, что никакие два участка не пересекаются, а в совокупности участки покрывают всю область. Любая точка внутри участка, связанного с одной из заданных точек, расположена ближе к этой точке, чем к любой другой из числа заданных. По существу, это еще один макет, но используется он обычно в служебных целях, а не для отображения – например, диаграмма Вороного часто применяется для расширения области наведения мыши в диаграммах, так что мышь всегда выделяет ближайшую к ней точку. Мы воспользуемся диаграммой Вороного для нахождения крупного аэропорта, ближайшего к вашему текущему местонахождению, которое будем определять с помощью API геолокации из HTML5.

Замените последнюю написанную нами строку таким кодом:

```
async function renderView(ctx, next) {
if (ctx.method === 'GET') {
ctx.body =
&grave;<!doctype html>
<html>
  <head>
    <title>Find your nearest airport!</title>
  </head>
  <body>
    <form method="POST" action="#">
      <h1>Enter your latitude and longitude, or allow your browser to check./h1>

      <input type="text" name="location" /> <br />
      <input type="submit" value="Check" />
    </form>
    <script type="text/javascript">
    navigator.geolocation.getCurrentPosition(function(pos) {
      var latlng = pos.coords.latitude + ',' + pos.coords.longitude;
      document.querySelector('[name="location"]').value = latlng;
    });
    </script>
  </body>
</html>&grave;;
} else if (ctx.method === 'POST'){
await next(); // Гарантирует, что все остальные элементы промежуточного уровня
              // отработают раньше
const airport = ctx.state.airport.data;
ctx.body = &grave;<!doctype html>
<html>
  <head>
    <title>Your nearest airport is: ${airport.name}</title>
  </head>
  <body style="text-align: center;">
    <h1>
      The airport closest to your location is: ${airport.name}
    </h1>
    <table style="margin: 0 auto;">
      <tr>
        ${Object.keys(airport).map(v =>&grave;<th>${v}</th>&grave;).join('')}
      </tr>
      <tr>
        ${Object.keys(airport).map(v => &grave;<td>${airport[v]}</td>&grave;).
join('')}
      </tr>
    </table>
  </body>
```

```
</html>&grave;;
}
app.use(renderView);
```

Ёлы-палы, кода-то сколько! И что всё это значит?

Сначала мы создаем функцию и назначаем ее элементом промежуточного уровня нашего приложения Koa. Все функции промежуточного уровня получают в качестве аргументов контекст (`ctx`) и функцию `next()`; первый служит для передачи инструкций из одного места приложения в другое, а второй – для управления потоком. В функции `renderView()` мы сначала создаем шаблон HTML-страницы, которая будет отправлена при получении сервером `GET`-запроса, т. е. при первом входе на сайт. Затем создается другой шаблон страницы – для `POST`-запросов. Это уже интереснее. Обратите внимание, что первым делом мы вызываем `next()` и с помощью `await` дожидаемся возврата. Это означает «ждать завершения всех последующих элементов промежуточного уровня, а затем продолжить выполнение этого элемента». Следовательно, наша функция вернет ответ на `GET`-запрос, не вызывая никаких других элементов промежуточного уровня, а в случае `POST`-запроса вызовет все остальные, дав им возможность добавить данные в контекстный объект `state`, а затем продолжит работу.

Мы разместили HTML-код внутри серверной логики, что не назовешь удачным приемом, но для наших целей достаточно. При создании многостраничных приложений с несколькими маршрутами рекомендуется использовать такие модули, как `Koa-Router` и `Koa-Views`. Первый позволяет определять маршруты для запросов, а второй – использовать языки шаблонов, например Handlebars, для определения выходной HTML-разметки.

Остановите сервер нажатием *Ctrl+C* и снова запустите командой

```
$ npm run server
```

Отметим, что, в отличие от клиентских программ, сервер необходимо перезапускать после каждого изменения. Если что-то работает не так, как ожидалось, попробуйте перезапустить сервер.

В результате браузеру будет отправлен простой HTML-документ с предложением ввести широту и долготу через запятую. Или, если пользователь примет предложение использовать API геолокации, координаты будут заполнены автоматически.

Далее необходимо создать функцию для вычисления диаграммы Вороного. Поместите следующий код в начало секции импорта в файле `chapter8/index.js`:

```
import { readFileSync } from 'fs';
import * as d3 from 'd3';
import * as boundaries from '../../data/cultural.json';
import * as land from '../../data/land.json';
```

Здесь мы импортируем D3 и встроенную в Node функцию синхронной загрузки из файловой системы. Кроме того, импортируются файлы TopoJSON из главы 4, а поскольку у нас есть доступ к файловой системе в Node, то мы можем просто прочитать их как обычные JavaScript-файлы, а не делать XHR-запрос.

Затем добавьте такой код:

```
const airportData = readFileSync('data/airports.dat',
  { encoding: 'utf8' });
const points = d3.csvParseRows(airportData)
  .filter(airport => !airport[5].match(/N/) && airport[4] !== '')
  .map(airport => ({
    name: airport[1],
    location: airport[2],
    country: airport[3],
    code: airport[4],
    latitude: airport[6],
    longitude: airport[7],
    timezone: airport[11],
  }));
```

Здесь мы строим из данных об аэропорте объект, который потом можем использовать в приложении. Далее напишем функцию, которая создает макет `voronoi` для вычисления ближайшего к нам аэропорта:

```
export function nearestLocation(location, points) {
  const coords = location.split(/,s?/);
  const voronoi = d3.voronoi()
    .x(d => d.latitude)
    .y(d => d.longitude);
  return voronoi(points).find(coords[0], coords[1]);
}
```

Мы предполагаем, что координаты заданы в виде строки через запятую, и преобразуем эту строку в массив. Затем создается генератор макета `d3.voronoi`, которому мы говорим, как получить широту и долготу каждого элемента данных для вычисления значений `x` и `y` соответственно. Затем мы передаем генератору наш объект, содержа-

щий данные обо всех аэропортах, с помощью метода `.find()` вычисляем ближайший аэропорт и возвращаем его.

Работа с диаграммами Вороного существенно изменилась в версии D3 v4, и особенно следует отметить добавление метода `layout.find()`. Раньше приходилось выполнять заковыристые математические вычисления, из-за чего эта глава была сложнее, чем сейчас. Но это уже в прошлом. Спасибо, Майк!

Это полезно само по себе, но, чтобы связать с нашим Koa-приложением, необходимо написать еще один элемент промежуточного уровня. Добавьте следующую функцию в файл `lib/chapter8/index.js`:

```
function nearestAirport(ctx, next) {
  if (ctx.request.body.location) {
    const { location } = ctx.request.body;
    ctx.state.airport = nearestLocation(location, points);
    next();
  }
}
```

Здесь все просто. Мы получаем координаты из тела запроса, с помощью функции `nearestLocation()` определяем ближайший аэропорт и записываем его в контекстный объект `state`. Затем вызываем метод `next()`, чтобы могли отработать следующие элементы промежуточного уровня, которые увидят только что обновленный контекст `state`. Наша функция `renderView` дождется, когда `nearestAirport` вызовет `next()` и использует объект `state`, чтобы вывести таблицу, содержащую информацию о ближайшем аэропорте.

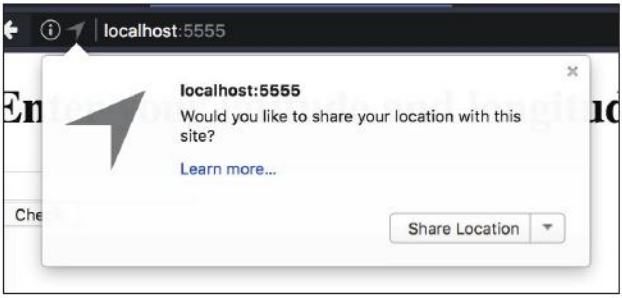
Найдите строки, в которых создаются элементы промежуточного уровня методом `app.use()`. Добавьте в стек новый элемент после `bodyParser` и `renderView`:

```
app.use(bodyParser())
   .use(renderView)
   .use(nearestAirport);
```

После сохранения нажмите *Ctrl+C* в окне терминала, чтобы остановить сервер, и перезапустите его командой

```
npm run server
```

После перехода по адресу `localhost:5555` появится всплывающее окно с просьбой дать разрешение на получение ваших координат:



Так это выглядит в Firefox, но Chrome немногим отличается:

Enter your latitude and longitude, or allow your browser to check.

51.5048775,-0.1145936
Check

После нажатия кнопки **Check** на экране появится информация о ближайшем аэропорте:

name	location	country	code	latitude	longitude	timezone
City	London	United Kingdom	LCY	51.505278	0.055278	Europe/London

Ну, круто же! Заодно мы показали, как можно использовать D3 для задач, вообще не связанных с рисованием.

Теперь вы понимаете, что D3 – не столько набор волшебных инструментов для преобразования данных в красивые зрительные образы, сколько собрание функций для выполнения определенных математических вычислений. И хотя интерес к D3, безусловно, вызван прежде всего ее умением рисовать в браузере, эта библиотека настолько мощная, что способна решать задачи, далеко выходящие за пределы первоначальной цели – отрисовки SVG-разметки, включенной в DOM.

Рисование на холсте на стороне сервера

А что, если заняться одной из таких задач прямо сейчас? Как уже отмечалось, результат работы нашего серверного приложения не вызывает восторга. Давайте нарисуем карту на холсте – Canvas.

Мы пока не затрагивали эту тему, но Canvas – еще один способ визуализации данных в D3. Вместо того чтобы включать элементы SVG в дерево DOM, Canvas рисует пиксели на двумерной плоскости, что обычно гораздо быстрее. Порядок работы здесь совсем не такой, к какому мы привыкли, и я не стану подробно останавливаться на нем в этой книге, поскольку начинающим отлаживать такую программу будет труднее, но, по крайней мере, мы познакомимся с этим подходом на примере.

Кстати говоря, отмечу, что такой способ использования Canvas выглядит очень странно по сравнению с тем, что делается в браузере. Обычно мы полагаемся на встроенный в браузер отрисовщик Canvas, однако на сервере мы этой роскоши лишены. Впрочем, код, который мы напишем для `node-canvas`, будет работать и в браузере, если вы захотите использовать его там.

Добавьте новую функцию в файл `chapter8/index.js`:

```
function canvasMap(ctx, next) {
  const { airport } = ctx.state;
  const scale = ctx.query.scale || 1200;
  const projectionName = d3.hasOwnProperty(ctx.query.projection) ?
    ctx.query.projection : 'geoStereographic';
  const canvas = new Canvas(960, 500);
  const canvasCtx = canvas.getContext('2d');
  const projection = d3[projectionName]()
    .center([airport.data.longitude, airport.data.latitude])
    .scale(scale);
}
```

Сначала мы извлекаем объект аэропорта из контекстного объекта `state`. Мы собираемся использовать указанные в запросе аргументы для задания масштаба и проекции выходной карты, поэтому запишем их в объект `query` внутри контекста, подставив значения по умолчанию вместо отсутствующих аргументов. Затем создаем объект Canvas размера 960 500 пикселей, запрашиваем у него контекст рисования и создаем проекцию (по умолчанию `d3.geoStereographic`) с центром в точке местонахождения пользователя. Чтобы отличить контекст Canvas (описывающий, где и как мы рисуем) от контекста Koa (который связан с обработкой запроса), мы будем обозначать первый `canvasCtx`.

Далее добавьте следующий код в `canvasMap`:

```
const path = d3.geoPath()
  .projection(projection)
  .context(canvasCtx);
```

```
canvasCtx.beginPath();
path(topojson.mesh(land));
path(topojson.mesh(boundaries));
canvasCtx.stroke();
```

Здесь мы с помощью нашей проекции создаем генератор пути и говорим ему, что он должен выводить результат в контекст Canvas. Затем создаем путь, передавая ему страну и границы в виде сеток Topojson, и рисуем его с параметрами по умолчанию. В отличие от того, к чему мы привыкли в D3, методы не сцепляются, а выполняются один за другим – *процедурно*.

Если бы то же самое делалось в браузере, то теперь мы имели бы карту мира с центром в точке местонахождения пользователя. Добавим индикатор, показывающий, где находится ближайший к пользователю аэропорт:

```
const airportProjected = projection([airport.data.longitude, airport.data.
latitude]);
canvasCtx.fillStyle = '#f00';
canvasCtx.fillRect(airportProjected[0] - 5, airportProjected[1] - 5, 10, 10);
```

Это дает нам проекции `x` и `y` аэропорта, которые передаются методу `fillRect` контекста Canvas, рисующему закрашенный прямоугольник (в качестве аргументов ему передаются *координаты x и y левого верхнего угла, ширина и высота*).

Но это еще не все – хотя наш сервер что-то нарисовал на холсте, мы этого не увидим, потому что рисунок существует лишь в памяти веб-сервера. Чтобы отобразить рисунок, мы должны преобразовать его в строку в кодировке base64, которую можно передать тегу `img`. Добавьте следующие строки в конец функции `drawCanvas()`:

```
ctx.state.canvasOutput = canvas.toDataURL();
next();
```

Элемент Canvas преобразуется в кодировку base64 и присоединяется к объекту `state`. Затем мы вызываем `next()`, чтобы дать возможность отработать следующим элементам промежуточного уровня.

Оставим ранее написанную функцию формирования страницы в покое и напишем новую. Продублируйте функцию `renderView`, назовите копию `renderViewCanvas` и добавьте во второй шаблон тег `img`. Ниже я выделил места, на которые нужно обратить внимание:

```
async function renderViewCanvas(ctx, next) {
  if (ctx.method === 'GET') {
    ctx.body =
    &grave;<!doctype html>
      <html>
      <head>
        <title>Find your nearest airport!</title>
      </head>
      <body>
        <form method="POST" action="#">
          <h1>Enter your latitude and longitude, or allow your browser to check.
          </h1>
          <input type="text" name="location" /> <br />
          <input type="submit" value="Check" />
        </form>
        <script type="text/javascript">
        navigator.geolocation.getCurrentPosition(function(position) {
          document.querySelector('[name="location"]').value =
          position.coords.latitude + ',' + position.coords.longitude;
        });
        </script>
      </body>
    </html>&grave;;
  } else if (ctx.method === 'POST') {
    await next(); // This ensures the other middleware runs first!
    const airport = ctx.state.airport.data;
    const { canvasOutput } = ctx.state;
    ctx.body = &grave;<!doctype html>
    <html>
    <head>
      <title>Your nearest airport is: ${airport.name}</title>
    </head>
    <body style="text-align: center;">
      <h1>
        The airport closest to your location is: ${airport.name}
      </h1>
      <img style="width: 480px; height: 250px" src="${canvasOutput}">
      <table style="margin: 0 auto;">
      <tr>
        ${Object.keys(airport).map(v => &grave;<th>${v}</th>&grave;).join('')}
      </tr>
      <tr>
        ${Object.keys(airport).map(v => &grave;<td>${airport[v]}</td>&grave;).
join('')}
```

```
      </tr>
    </body>
    </html>&grave;;
  }
}
```

Теперь, помимо аэропорта, мы извлекаем свойство `canvasOutput` из объекта `state` и выводим его в виде изображения. Вот самая важная часть:

```
<img style="width: 480px; height: 250px;" src="${canvasOutput}" />
```

Отметим, что Canvas порождает *растровые* изображения – в отличие от векторного SVG-изображения, которое может быть увеличено или уменьшено как угодно без ухудшения качества. Размер холста определяется разрешением устройства, на котором он отображается, и при попытке масштабировать его сверх обычных размеров будут видны пиксели. Мы используем популярный прием, гарантирующий, что изображение будет хорошо выглядеть на экранах Retina: при создании объекта Canvas в функции `drawCanvas` мы задали размер вдвое больше того, который хотели бы видеть в конечном итоге. А в функции `renderViewCanvas` мы явно задаем ширину и высоту равными половине первоначально заданных значений, так что изображение получается более резким.

Найдите вызовы `app.use` и замените их таким кодом:

```
app.use(bodyParser())
   .use(renderViewCanvas)
   .use(nearestAirport)
   .use(canvasMap);
```

Если сервер Node работает, остановите его нажатием *Ctrl+C* и перезапустите:

```
$ npm run server
```

Перейдите по адресу `localhost:5555`, введите широту и долготу (или позвольте браузеру определить их) – и появится страница со статической картой:

Если добавить в запрос аргументы, чтобы URL имел вид

```
http://localhost:5555/?scale=400&projection=geoMercator
```

то масштаб будет равен 400, а проекция станет меркаторской. Красиво, правда?

Несмотря на то что использование Canvas в D3 выглядит довольно странно, это чрезвычайно мощная технология, особенно для отрисовки очень больших объемов данных (любая технология отображения, основанная на DOM, начинает серьезно тормозить, когда число элементов превышает 1000, а Canvas просто рисует на двумерной плоскости, поэтому эта проблема ей не страшна).

Развертывание в среде Heroku

Серверное приложение не особенно полезно без сервера.

По счастью, у компании Heroku имеются бесплатные планы для ограниченного использования, а развертывание в этой среде не вызывает никаких трудностей. В настоящий момент компания предлагает 550 (плюс дополнительные 450, если ваша учетная запись верифицирована кредитной картой) *диночасов*, распределенных между всеми вашими серверами, причем неиспользуемые машины не тарифицируются. На деле это означает, что ваш сервер вообще не работает в периоды неактивности, если, конечно, не получает запросов постоянно.

Создайте учетную запись на сервере http://www.heroku.com и установите пакет Heroku Toolbelt с сайта http://toolbelt.heroku.com. Затем перейдите в корневую папку проекта и введите команду

```
$ heroku create
```

В ответ будет создан удаленный репозиторий Git, а вашему приложению будет назначен случайный адрес вида https://calm-dusk-16214.herokuapp.com/.

Далее создайте файл с именем `Procfile`. На этапе развертывания Heroku читает его, чтобы узнать, как запускать ваше приложение (по умолчанию выполняется команда `node start`, но мы в этой книге запускаем сервер разработки Webpack). Поместите в этот файл такую строку:

```
web: node build/server.js
```

Сохраните файл и проверьте, что собрана последняя версия пакета:

```
$ npm run server
```

По завершении сборки нажмите *Ctrl+C*, больше сервер нам не понадобится. И напоследок сохраните весь проект в системе управления версиями:

```
$ git add . && git commit -am "Time for Heroku"
```

Теперь все готово к развертыванию. В предположении, что вы работаете с главной веткой, введите команду

```
$ git push heroku master
```

Heroku всегда производит развертывание из главной ветки. Если вы выгрузили из Git ветку `chapter8`, то вместо показанной выше команды наберите такую: `$ git push heroku chapter8:master`.

Перейдите по URL-адресу, который сообщила команда `heroku create`, – вы увидите свое приложение, развернутое в Сети и доступное любому пользователю Интернета. Примите поздравления! Вы только что написали вполне достойное веб-приложение. А ведь вы и не мечтали пройти ускоренный курс по написанию серверного кода в книге, посвященной визуализации, правда?

Резюме

В этой главе мы сначала настроили Webpack для создания отдельного пакета для сервера, а затем написали простое веб-приложение

с использованием Koa 2, в котором с помощью диаграммы Вороного искали аэропорт, расположенный ближе всего к точке с заданными координатами – широтой и долготой. После этого мы переписали приложение, так чтобы оно рисовало карту с применением D3 и Canvas и выводило ее пользователю в формате PNG в виде строки в кодировке base64.

Все это выглядело не вполне обычно, но увлекательно, не правда ли? С написанием серверного кода всегда так. Однако это может стать настоящим избавлением после бесконечной разработки клиентских приложений, от которых требуется мелочный учет особенностей многочисленных устройств. Более того, хотя некоторое время назад были популярны клиентские *одностраничные приложения*, все чаще применяется технология, когда состояние веб-приложения предварительно генерируется на сервере, а затем *восстанавливается* (rehydrate) на стороне клиента. Никогда еще разработчикам клиентской части не было так важно иметь хотя бы минимальное представление о том, что происходит за кулисами.

Понятно, что на эту тему можно было бы сказать гораздо больше, но не позволяет нехватка места в этой книге. Мы *совсем* не говорили ни о масштабируемости (а это важнейший вопрос, когда речь идет о создании приложений для большой аудитории, например новостных сайтов), ни о том, как правильно выстроить архитектуру нетривиального приложения; интересующимся я рекомендую установить несколько основанных на Express генераторов Yeoman и посмотреть, как они строят заготовки проектов (лично мне особенно нравится `generator-angular-fullstack`), или почитать книгу Alexandru Vladutu «Mastering Web Application Development with Express» (Packt, 2014). Koa очень похожа на Express и усвоила многие его идиомы (в каком-то смысле ее можно назвать духовным наследником Express), поэтому многое, почерпнутое из руководств по Express, окажется полезным (увы, по Koa 2 руководств пока гораздо меньше).

Теперь у вас есть достаточно полный арсенал средств для решения широчайшего круга задач, связанных с визуализацией. В следующей главе мы добавим к нему еще несколько инструментов: проверку синтаксиса, автономное тестирование и статическую типизацию, чтобы помочь вам обрести уверенность в плодах своего труда.

Глава 9

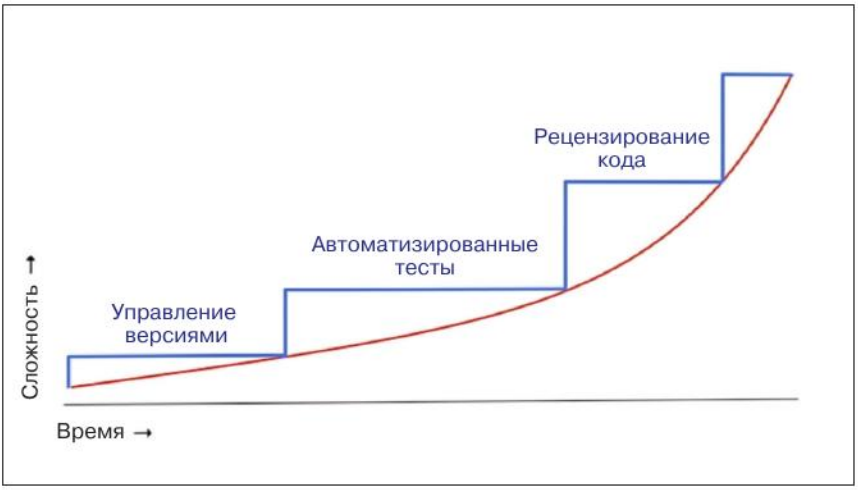
Обретение уверенности в своих визуализациях

При создании чего-то, что увидит огромная аудитория в вебе, очень страшно допустить ошибку, из-за которой данные будут отображаться неправильно. Когда время работы над проектом ограничено, тестированием часто пренебрегают, потому что сроки поджимают, а материал еще должны посмотреть другие люди (редакторы, начальники и т. д.), так что результат важнее процесса. Но хотел бы выразиться предельно ясно – если вам важно высокое качество визуализации, то необходимо позаботиться о том, чтобы она была надлежащим образом протестирована и функционировала, как задумано. На определенном уровне это можно считать упражнением по управлению сложностью.

На рисунке ниже показано изменение сложности проекта со временем. Как видим, сложность возрастает экспоненциально. Увеличение размера группы разработчиков, числа строк кода или количества зависимостей приводит к резкому росту сложности. А наложенная на рисунок шаговая диаграмма показывает, как развивается инструментарий в ответ на рост сложности. Теоретически можно реализовать все эти шаги в начале каждого проекта, но лучше делать это постепенно, сообразуясь с требованиями проекта. Например, обычно любой проект сохраняется в системе управления версиями, поскольку это упрощает совместную работу, позволяет откатить ошибки и получить историю работы над проектом (к тому же система легко настраивается). Допустим, что впоследствии в проект вводятся новые люди или принимается решение открыть исходный код. Теперь изменения

вносятся со всех сторон, и было бы крайне желательно иметь какой-то способ автоматически проверять, что новое изменение ничего не «сломало». Еще чуть позже выясняется, что через фильтр автоматического тестирования все же прошло немало плохо написанного кода, и тогда на помощь приходит взаимное рецензирование кода.

Каждое улучшение процесса требует времени – на внедрение и на использование, – а будет ли от него толк, сильно зависит от размера команды и доверия к каждому сотруднику (а если вы работаете в одиночестве, то от веры в собственную способность не допускать ошибок).



Со временем сложность возрастает,
а состав инструментария изменяется ступенчато.
Источник: Martin Probst, Alex Eagle, https://youtu.be/yy4c0hzNXKw?t=245

Этот рисунок взят из доклада Мартина Пробста и Алекса Игла по языку TypeScript на конференции AngularConnect 2015. В докладе затронуты многие рассматриваемые в этой главе темы, так что имеет смысл посмотреть его запись по адресу https://www.youtube.com/watch?v=yy4c0hzNXKw.

В этом разделе мы акцентируем внимание на нескольких технологиях управления сложностью проекта.

- Мы начнем с **проверки стиля** (linting), т. е. сопоставления кода с предопределенными правилами и стандартами написания и оформления.

- Затем перейдем к **статической проверке типов**, в ходе которой проверяется правильность передачи определенных видов переменных между функциями. Эта проверка производится на этапе сборки пакета.
- Пояснять, что такое **автоматизированное тестирование**, излишне: просто вы пишете тесты, которые обязательно должны проходить. Ошибки при выполнении теста – средство диагностики логических ошибок или иных проблем в коде.

У всех вышеупомянутых инструментов две цели: уменьшить частоту ошибок и повысить квалификацию разработчиков. Для достижения этих целей организуется обратная связь с разработчиком в процессе кодирования: ошибки в написании имен переменных или неправильно переданные аргументы обнаруживаются на этапе разработки, а не на этапе выполнения в производственном режиме.

Проверка стиля

В ходе проверки стиля исходный код сравнивается с заранее заданным набором правил, и в случае несоблюдения выдаются предупреждения. С одной стороны, это устанавливает единые стандарты кодирования в проекте, а с другой – позволяет обнаружить потенциальные ошибки на этапе разработки, особенно очевидные, такие как ошибки в именах переменных.

Стилистические правила часто основаны на передовых стандартах отрасли, но в большинстве проектов с открытым исходным кодом имеются свои наборы правил, выражающие выработанные сообществом рекомендации. Тем самым, с одной стороны, гарантируется единообразие кода, написанного разными людьми, а с другой – каждый разработчик получает уведомление, если напишет что-то сомнительное или чреватое ошибками. Отметим, однако, что все эти правила – не более чем субъективное мнение: вы не обязаны им следовать, но их соблюдение облегчает работу всех остальных участников проекта.

Если, читая книгу, вы извлекали файлы из репозитория на GitHub, то, наверное, обратили внимание на скрытый файл `.eslintrc` или на секцию `eslint` в файле `package.json`. ESLint в настоящее время считается лучшей программой проверки стиля для кода, написанного на ES2017, и работает она очень похоже на своих предшественников: JSHint и JSCS. Обычно конфигурационные данные задаются в файле `.eslintrc`, который содержит конкретные правила (или набор умолчаний, который можно расширить, – я большой поклонник конфигу-

рации Airbnb и в этой книге пользовался слегка модифицированным набором правил этой компании) и информацию об окружении программы (например, в NodeJS глобальные переменные не такие, как в браузере).

Чтобы применить ESLint к текущему проекту, выполните команду

```
$ npm run lint
```

Хотя к моменту выхода книги из печати в репозитории не будет стилистических ошибок, на рисунке ниже показан результат проверки на этапе работы над книгой:

```
/Users/aendrew/Sites/learning-d3-v4/lib/common/index.js
  31:25  error    Assignment to property of function parameter 'd'  no-param-reassign
  41:10  error    Unexpected string concatenation                   prefer-template
  41:10  error    Strings must use singlequote                      quotes
  41:29  error    Strings must use singlequote                      quotes
  42:9   error    Strings must use singlequote                      quotes
  42:29  error    Multiple spaces found before '","'                no-multi-spaces
  42:29  error    Strings must use singlequote                      quotes
  42:33  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  42:61  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  43:9   error    Strings must use singlequote                      quotes
  43:28  error    Strings must use singlequote                      quotes
  43:32  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  43:60  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  44:9   error    Strings must use singlequote                      quotes
  44:28  error    Strings must use singlequote                      quotes
  48:10  error    Unexpected string concatenation                   prefer-template
  48:10  error    Strings must use singlequote                      quotes
  48:29  error    Strings must use singlequote                      quotes
  49:9   error    Strings must use singlequote                      quotes
  49:13  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  49:41  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  49:47  error    Strings must use singlequote                      quotes
  50:9   error    Strings must use singlequote                      quotes
  50:13  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  50:41  error    Unexpected mix of '+' and '/'                     no-mixed-operators
  50:47  error    Strings must use singlequote                      quotes
  51:9   error    Strings must use singlequote                      quotes
  51:28  error    Strings must use singlequote                      quotes
  94:1   warning  Line 94 exceeds the maximum line length of 100    max-len
 223:1   warning  Line 223 exceeds the maximum line length of 100   max-len
 224:1   warning  Line 224 exceeds the maximum line length of 100   max-len
 303:1   warning  Line 303 exceeds the maximum line length of 100   max-len
```

Как мы накосячили-то!

В данном случае мы использовали `npm` для запуска `eslint`, но по существу это то же самое, что такая команда:

```
$ node ./node_modules/.bin/eslint src/*.js
```

Вы также можете установить ESLint глобально и запускать из любого места.

Хотя и такой способ запуска приносит плоды, стилистическая проверка гораздо полезнее, если производится постоянно в процессе разработки. Давайте заставим ESLint вмешиваться всякий раз, как используется `webpack-dev-server`. Сначала установите пакет `eslint-loader`:

```
$ npm install eslint-loader --save-dev
```

Затем пропишите `eslint-loader` в качестве предзагрузчика в конфигурационном файле Webpack, так чтобы он обрабатывал код *до* того, как его увидит Babel. В файле `webpack.config.js` добавьте следующие строки в секцию `module` каждого собираемого элемента:

```
module: {
rules: [
  {
    test: /.js$/,
    loader: 'eslint-loader',
    exclude: /node_modules/,
  },
```

Запустите webpack-dev-server:

```
$ npm start
```

Та-да-да-дам! Теперь вы будете получать информацию о своих косяках при каждом сохранении! Я даже отсюда вижу, в каком вы восторге!

Стилистическая проверка – очень слабый способ управления сложностью; найденная ошибка не приводит ни к чему, кроме появления сообщения на мониторе разработчика. Вы сами и команда в целом должны соблюдать дисциплину – не допускать записи в репозиторий кода, не прошедшего проверку. Однако именно благодаря слабости стилистическую проверку проще всего внедрить – даже если все остальные члены команды считают ее никчемной глупостью, вы все равно можете поместить файл `.eslintrc` в свой домашний каталог и наслаждаться благами стилистической проверки в одиночестве.

Что может быть полезнее получения сообщений о стилистических ошибках на консоли? Получение их *прямо в IDE*! Практически каждый современный текстовый редактор имеет подключаемый модуль ESLint, для Atom и Sublime Text такие модули просто великолепны. Сделайте себе **большой** подарок – установите такой модуль; он даст о себе знать сразу, как только вы неправильно наберете имя переменной или воспользуетесь переменной вне ее

области видимости. Одно это сэкономит вам бесчисленные часы просмотра журналов браузера. Вам даже не придется устанавливать в проект ESlint – все и так будет работать!

Статическая проверка типов: TypeScript или Tern.js

В процессе статической проверки типов анализируется использование переменных и выдается сообщение, если обнаружены какие-то странности. Точнее, анализируются типы переменных и с помощью *аннотаций типов* (текстового указания типа переменной в момент ее определения) или *выведения типа* (установления типа по первому использованию переменной) проверяется, что функция не изменяет переменную неожиданным способом. Этот механизм *статической типизации* встроен во многие языки, в т. ч. C++ и Java. И хотя *динамическая типизация* в JavaScripts (и во многих других языках, применяемых для веб-программирования, в частности PHP и Ruby) в некоторых отношениях полезна и лежит в основе определенного стиля программирования, она также может быть источником горького разочарования из-за возможности внесения незаметных ошибок.

Поскольку мы все равно собираемся преобразовывать JavaScript-код из одной версии синтаксиса в другую (мы пользовались транслятором Babel, но есть и много других), то можем попутно выполнить и статическую проверку типов.

Это лишь часть того, чем полезен статический анализ типов; вторая часть – интеграция с IDE, которая в числе прочего предоставляет простой доступ к документации и весьма точное автоматическое автозавершение. Кроме того, интегрированная среда разработки немедленно реагирует на ошибки разработчика, благодаря чему ошибки просачиваются в производственный код гораздо реже, а потому и исправлять их проще. К тому же статическая проверка помогает составлять внутреннюю документацию программы, так что новым членам команды проще войти в курс дела.

TypeScript отлично работает с D3, т. к. располагает высококачественным определением типов, что позволяет быстро обращаться к документации из редактора. В сочетании со способностью компилятора TypeScript мгновенно откликаться на неправильно функционирующий код это означает, что писать высококачественный код с использованием D3 гораздо проще.

Для начала предупреждение: а стоит ли вообще заморачиваться с TypeScript? Недавно я сделал несколько проектов на TypeScript и, хотя я искренне старался полюбить его и полагаю, что с его помощью немного улучшил качество своего кода, не могу не признать, что он все же требует от разработчика ряда компромиссов. Если вы пишете на TypeScript в режиме отключения «неявного any» (о чем мы поговорим чуть ниже), то можно потратить уйму времени на поиск (или написание) правильного интерфейса, который удовлетворил бы конкретную функцию.

Хотя дополнительные метаданные немного упрощают чтение кода и в конечном итоге делают более предсказуемым его поведение в производственных условиях, ряд фактов говорит о том, что их эффект в плане предотвращения ошибок даже рядом не лежал с эффектом автоматизированного тестирования и методики разработки через тестирование (TDD). Кроме того, существенно замедляется использование некоторых беглых стилей программирования, применяемых теми, кто уже в достаточной мере освоил библиотеку, и это может вызвать неприятие больше, чем что-либо другое. Более полное обсуждение см. в статье Эрика Эллиота «You Might Not Need TypeScript (Or Static Types)» по адресу https://medium.com/javascript-scene/you-might-not-need-typescript-or-static-types-aa7cb670a77b.

Лично мне нравится писать на TypeScript и работать с подключаемым модулем TypeScript для редактора Atom, пусть даже временами он несколько замедляет мою работу (и если уж совсем честно, то последние версии TypeScript не такие многословные и гораздо лучше выводят правильный тип, не заставляя писать явных аннотаций). Определенно его стоит попробовать, но только не ценой отказа от тестирования, о котором пойдет речь в следующем разделе.

И еще раз повторю сказанное во введении – отдача от применения любых технологий целиком и полностью зависит от вашей команды и сложности проекта: если вы создаете единичные страницы, содержащие статический HTML и JavaScript, для публикации в новостях, то, пожалуй, можно и обойтись без изучения TypeScript. Если же вы пишете крупные интерактивные приложения с несколькими представлениями и десятками модулей, то пользы от TypeScript будет гораздо больше.

Что, если вы хотите улучшить внутреннюю документацию, не подключая всяких разных инструментов? Возможное решение – программа `Tern.js`, которая предлагает похожие на TypeScript средства Intellisense без статической типизации. Но это полезно лишь в том случае, когда имеется поддержка в IDE, поэтому проверьте, сущест-

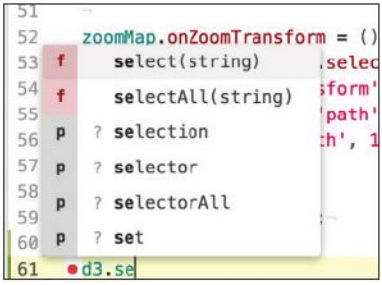
вует ли подключаемый модуль для вашего редактора (для самых лучших такие модули есть). Мы рассмотрим `Tern.js` первой, потому что ее проще настроить и запустить, и дополнительных усилий почти не потребуется.

Анализ кода с помощью Tern.js

Как уже отмечалось, проверка типов на этапе сборки полезна для нахождения ошибок в коде, поскольку вы получаете сообщения об ошибках на этапе разработки. Но при этом приходится тратить дополнительные усилия на конфигурирование. У определений типов есть и другая сторона – они являются средством документирования и подмогой при анализе кода.

Значительную часть этой функциональности можно получить и без статической типизации с помощью инструмента `Tern.js`. Это движок анализа кода, который просматривает модули и сообщает информацию другим инструментам. Он редко используется в одиночестве, обычно мы устанавливаем подключаемый модуль к редактору, а тот общается с Tern.

Для Atom, Sublime и других IDE такие модули существуют, порядок установки зависит от конкретной IDE. Лично я предпочитаю Atom с atom-ternjs:

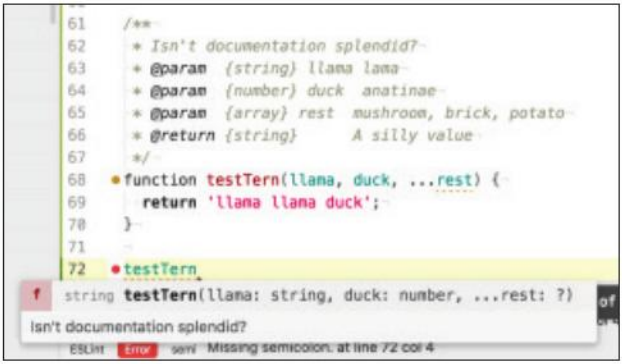


Если Tern добавлен в Atom, то появляется автозавершение кода, как показано на рисунке выше.

Обратите внимание, что после ввода строки `d3.se` появилось всплывающее окно, содержащее варианты `d3.select` и `d3.selectAll` вместе с сигнатурами. Это помогает проверить, правильный ли тип передается функции. Но Tern может сделать и больше, если функция снабжена документацией в следующем формате:

```
/**
* Isn't documentation splendid?
* @param  {string} llama lama
* @param  {number} duck anatinae
* @param  {array} rest mushroom, brick, potato
* @return {string} A silly value
*/
function testTern(llama, duck, ...rest) {
  return 'llama llama duck';
}
```

Тогда механизм автозавершения предоставит куда более подробную информацию:



Для установки Tern глобально выполните следующую команду:

```
$ npm install -g tern
```

Обычно это нужно сделать до того, как Tern начнет работать в IDE. Кроме того, нужно будет создать конфигурационный файл `Tern`. Поместите в корневой каталог проекта файл `.tern-project` со следующим содержимым:

```
{
  "ecmaVersion": 7,
  "libs": [
    "browser"
  ],
  "plugins": {
    "doc_comment": {
      "fullDocs": true,
      "strong": false
    },
```

```
    "node": {
      "dontLoad": "",
      "load": "",
      "modules": ""
    },
    "modules": {
      "dontLoad": "",
      "load": "",
      "modules": ""
    },
    "es_modules": {}
  }
}
```

После установки подключаемого модуля Tern в IDE должны появляться формируемые им детальные окна автозавершения.

К сожалению, в D3.js нет никакой внутренней документации, поэтому пользы от Tern.js немного. В следующем разделе мы рассмотрим TypeScript, для которого имеются отличные определения типов D3, т. е. в редактор «подпихивается» обширная документация.

Мощная связка TypeScript – D3

У использования TypeScript много плюсов, и я к ним скоро перейду. Но сначала хочу предупредить, что для настройки TypeScript придется потрудиться, потому что это самостоятельный транслятор, а значит, мы больше не будем использовать Babel.

Но есть и хорошая новость – TypeScript транслирует JavaScript почти так же, как Babel, и перед его использованием нужно задать всего несколько параметров. На практике вы вряд ли вообще заметите разницу между тем и другим. Более того, мы будем использовать их одновременно, импортируя модули TypeScript в Babel и наоборот.

Для начала установим дополнительные пакеты. Разработка на современном JavaScript на 50% состоит из кодирования, на 20% – из поиска нужных библиотек и на 30% – из установки этих библиотек. Приступим:

```
$ npm install typescript ts-loader --save-dev
```

Здесь устанавливаются транслятор TypeScript и загрузчик Webpack. Далее введите команду

```
$ npm install @types/d3 d3-sankey aendrew/d3-sankey --save
```

Она устанавливает определения типов D3, подготовленные сообществом. Раньше для установки определений нужно было использовать отдельную утилиту, но в версии TypeScript 2.0 достаточно директивы `@types` в npm. Мы сохраняем определения как обычную зависимость, потому что это будет часть нашего кода наряду с TypeScript. Устанавливаются также пакет `d3-sankey`, который не является частью основной библиотеки D3, и `aendrew/d3-sankey`, содержащий определения типов для него.

Том Ванзек – тот герой, который создал великолепные определения типов TypeScript для D3 v4. Мне не хватает слов, чтобы выразить ему благодарность за это, поскольку работа была адовой. Я дополнил его работу определениями типов для пакета `d3-sankey` в пакете `aendrew/d3-sankey`. Техника написания определений для TypeScript выходит за рамки этой главы, но не нужно быть гуру в TypeScript, чтобы внести свой вклад в проекты сообщества, в частности в DefinitelyTyped, откуда организация `@types npm` черпает определения.

Теперь внесем изменения в конфигурационный файл Webpack. Под секцией правил поместите такие строки:

```
{
  test: /.ts$/,
  loader: 'ts-loader'
},
```

Теперь для загрузки всех файлов с расширением `.ts` будет использоваться TypeScript. Можете переименовать все ранее написанные файлы, заменив расширение `.js` на `.ts`, а можете поступить, как мы: использовать *одновременно* Babel и TypeScript. Привыкнув к TypeScript, вы, возможно, захотите использовать его с самого начала, но на этот раз мы для пущей крутизны смешаем ES2015+ и TypeScript вместе.

Это еще не все. Создайте файл `.tsconfig` в корневом каталоге проекта и поместите в него такой код:

```
{
  "compilerOptions": {
    "target": "ESNext",
    "module": "ES6",
    "moduleResolution": "node",
    "isolatedModules": false,
    "jsx": "react",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
```

```
    "declaration": false,
    "noImplicitAny": false,
    "noImplicitUseStrict": false,
    "removeComments": true,
    "noLib": false,
    "preserveConstEnums": true,
    "suppressImplicitAnyIndexErrors": true,
    "allowJs": true,
    "lib": [
      "es2015",
      "es2015.iterable",
      "es2015.symbol",
      "es2015.promise",
      "dom"
    ]
  },
  "exclude": [
    "node_modules"
  ],
  "compileOnSave": false,
  "buildOnSave": false,
  "atom": {
    "rewriteTsconfig": false
  }
}
```

Теперь закомментируйте весь код в файле `lib/main.js` и добавьте такой:

```
import sankeyChart from './chapter9/index.ts';
sankeyChart();
```

Создайте папку `lib/chapter9` и в ней файл `index.ts`:

```
import {
  sankey,
  SankeyLink,
  SankeyNode,
  SankeyData } from 'd3-sankey';
import * as d3 from 'd3';
import chartFactory from '../common/index';
```

Здесь мы включаем определения типов TypeScript и импортируем D3. Начиная с этого момента, TypeScript можно использовать как обычно.

Если вы до сих пор не поняли, мы собираемся создать диаграмму Сэнкея. Как и диаграмма действующих сил, диаграмма Сэнкея изображает связи в виде графа, однако ее основное назначение – по-

казать силу этих связей. На нашей диаграмме Сэнкея будет показано распределение мест в парламенте по результатам последних всеобщих выборов в Великобритании. Полагаю, что выборы президента США 2016 года были бы более актуальны, но диаграммы Сэнкея становятся интересны, когда узлов больше двух.

Начнем с нашего доброго знакомца – функции `chartFactory`, которая создаст заготовку диаграммы:

```
const chart = chartFactory({
  margin: { left: 40, right: 40, top: 40, bottom: 40 },
  padding: { left: 10, right: 10, top: 10, bottom: 10 },
});
```

Затем создадим объект, содержащий цвета основных партий, отнеся более мелкие к категории «Прочие» (Other).

```
const partyColors = {
  CON: '#0087DC',
  LAB: '#DC241F',
  SNP: '#FFFF00',
  LIB: '#FDBB30',
  UKIP: '#70147A',
  Green: '#6AB023',
  Other: '#CCCCCC',
};
```

Чтобы упростить пример, я заранее разобрал и переформатировал данные, файл `uk-election-sankey.json` находится в репозитории в каталоге `data/`. Напишем новую асинхронную функцию для загрузки данных методом `fetch`:

```
async function typescriptSankey() {
  const sankeyData: SankeyData = await (
    await fetch('data/uk-election-sankey.json')).json();
  const width = chart.width;
  const height = chart.height;
  const svg = chart.container;
}
```

Если хотите посмотреть, как были сгенерированы данные, загляните в скрипт `getSankeyData.ts` в каталоге `data/scripts/`. Он тоже написан на TypeScript!

Мы не будем аннотировать большинсто переменных в этой главе, потому что определения типов уже проделали за нас трудную работу. Но результат `fetch` аннотировать нужно, потому что в противном случае TypeScript будет рассматривать его как нетипизированный объ-

ект. Для этого воспользуемся ранее импортированным интерфейсом `SankeyData`, который является частью определения TypeScript для пакета `d3-sankey`. *Интерфейс* – это, по существу, просто группа объявлений типов наподобие объекта, содержащего тип. Полный интерфейс `SankeyData` выглядит так:

```
export interface SankeyData {
  nodes: Array<SankeyNode>;
  links: Array<SankeyLink>;
}
```

Несмотря на необычный синтаксис, все здесь достаточно просто – мы создаем объект с двумя свойствами: `nodes` и `links`, которые являются массивами, содержащими объекты типа `SankeyNode` и `SankeyLink` соответственно. Несложно посмотреть, как выглядят сами эти интерфейсы (если вы установили подключаемый модуль `atom-typescript`, то щелкните правой кнопкой мыши по слову `import` и выберите из контекстного меню команду **Go to declaration** (Перейти к объявлению)), но достаточно будет сказать, что они не вызывают никаких сложностей и похожи на структуры данных, с которыми мы уже много раз встречались на страницах этой книги (точнее, у объекта `SankeyNode` есть свойства `name` и `value`, а у объекта `SankeyLink` – свойства `source`, `target` и `value`). Эти типы, в свою очередь, расширяют еще один объект, который содержит все свойства, которыми `d3-sankey` декорирует наши данные, но это происходит за кулисами, и вам об этом и знать-то не нужно, за исключением тех случаев, когда что-то идет не так и компилятор TypeScript ругается на отсутствие свойства или еще на что-то. Во многих отношениях это заменяет стилистическую проверку – в настоящее время TypeScript несовместим с ESLint, для него есть собственное средство проверки – tslint, но оно в основном предназначено для проверки форматирования и в меньшей степени для выявления проблем в коде (компилятор TypeScript сам скажет, если что-то не в порядке).

Мы пока еще не встречались с градиентами SVG, так давайте исправим это упущение в последнем примере. Для создания градиента нужно подготовить объект с его определением и сохранить его в скрытом теге *defs*. Впоследствии мы сошлемся на него по идентификатору. Немного потерпите, поначалу это кажется запутанным, но скоро все прояснится.

Добавьте в функцию `typescriptSankey()` такой код:

```
const defs = svg.append('defs');
Object.keys(partyColors).forEach((d, i, a) => {
  a.forEach(v => {
```

```
    defs.append('linearGradient')
      .attr('id', &grave;${d}-${v}&grave;)
      .call((gradient) => {
        gradient.append('stop')
          .attr('offset', '0%')
          .attr('style', &grave;stop-color:${partyColors[d]};
stopopacity:0.8&grave;);
        gradient.append('stop')
          .attr('offset', '100%')
          .attr('style', &grave;stop-color:${partyColors[v]};
stopopacity:0.8&grave;);
      });
  });
});
```

В результате создается примерно такая структура:

```
<defs>
  <linearGradient id="CON-LAB" >
    <stop
      offset="0%"
      style="stop-color: #0087DC; stop-opacity: 0.8"
    />
    <stop
      offset="100%"
      style="stop-color: #DC241F; stop-opacity: 0.8"
    />
  </linearGradient>
</defs>
```

Линейный градиент может иметь произвольное число стоп-цветов. Мы переходим от цвета исходной партии к цвету конечной партии и назначаем этой связи идентификатор.

Вот и пришло время для генератора макета! Все еще оставаясь внутри `typescriptSankey`, добавьте такой код:

```
const sankeyGenerator = sankey()
  .size([width - 100, height - 100])
  .nodeWidth(15)
  .nodePadding(10)
  .nodes(sankeyData.nodes)
  .links(sankeyData.links)
  .layout(1);

const path = sankeyGenerator.link();
```

Здесь размер генератора макета Сэнкея задается равным размеру окна браузера, ширина каждого узла 15, промежуток между узлами 10. И наконец, мы передаем генератору макета все наши узлы и связи. Затем метод `Sankey.link()` создает генератор путей, который рисует симпатичные изгибающиеся пути между узлами.

Все подготовив, мы переходим к изображению этого добра на странице. Нарисуем связи:

```
const link = svg.selectAll('.link')
  .data(sankeyData.links)
  .enter()
  .append('g')
  .classed('link', true);

link.append('path')
  .attr('d', path)
  .attr('fill', 'none')
  .attr('stroke', d => {
    const source = d.source.name.replace(/(2010|2015)/, '');
    const target = d.target.name.replace(/(2010|2015)/, '');
    return &grave;url(#${source}-${target})&grave;;
  })
  .style('stroke-width', d => Math.max(1, d.dy));
```

Единственное, что здесь может вызвать вопросы: атрибуты `stroke-width` и `stroke`; в качестве цвета обводки мы генерируем строку, отражающую названия начальной и конечной партий, и используем ее для ссылки на созданные ранее градиенты. Так, в приведенном выше примере переход от синего к красному (означающий, что поддержка избирателей теперь на стороне консерваторов, а не лейбористов) обозначается `#CON-LAB`. Чтобы построить ссылку на предыдущее определение градиента, мы обертываем этот идентификатор функцией `url()`.

И напоследок задаем `stroke-width` равным относительному значению `y` каждой связи.

Теперь, когда все связи подготовлены, нарисуем узлы и метки.

Сначала создадим группу, содержащую все наши прямоугольники и метки:

```
const node = svg.selectAll('.node')
  .data(sankeyData.nodes)
  .enter()
  .append('g')
  .classed('node', true)
  .attr('transform', d => &grave;translate(${d.x},${d.y})&grave;);
```

Это просто. Добавим прямоугольники:

```
node.append('rect')
  .attr('height', d => d.dy)
  .attr('width', sankeyGenerator.nodeWidth())
  .style('fill', d => partyColors[d.name.replace(/(2010|2015)/, '')])
  .append('title')
  .text(d => &grave;${d.name}nSeats: ${d.value}&grave;);
```

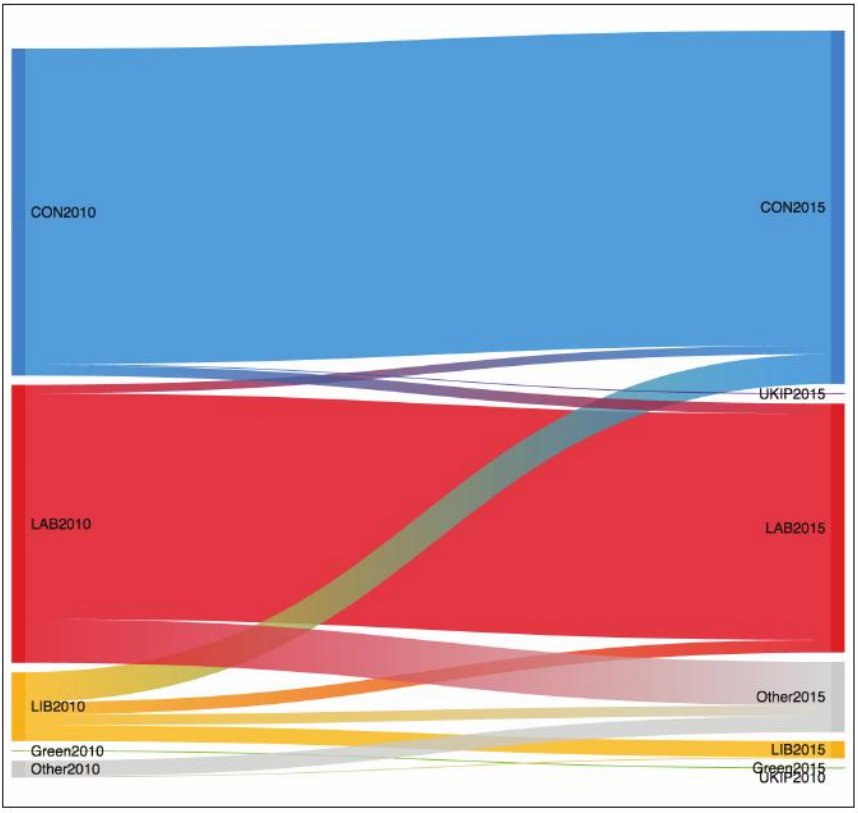
Мы получаем ширину прямоугольников методом `sankey.nodeWidth()` без аргумента, т. е. он возвращает заданную ранее ширину. Цвет заливки берется из объекта `partyColors`, но мы удаляем из названий год методом `String.prototype.replace()`, оставляя только аббревиатуру партии.

Наконец, мы добавляем метки, чтобы не пришлось идентифицировать партии только по цвету:

```
node.append('text')
  .attr('x', -6)
  .attr('y', d => d.dy / 2)
  .attr('dy', '.35em')
  .attr('text-anchor', 'end')
  .attr('transform', null)
  .text(d => d.name)
  .filter(d => d.x < width / 2)
  .attr('x', 6 + sankeyGenerator.nodeWidth())
  .attr('text-anchor', 'start');
```

Тут все понятно; мы увеличиваем ширину узлов на 6, чтобы немного отодвинуть их друг от друга. Я взял именно 6, потому что так получается красиво. Да здравствуют магические числа!

Сохранитесь и зайдите в браузер, должна появиться такая диаграмма Сэнкея:



Видите странную метку «UKIP 2010», в одиночестве прозябающую в правом нижнем углу? Запомните, это будет важно, когда ниже мы перейдем к автономному тестированию.

Неплохо, да? Мы сразу же понимаем, что либеральные демократы и лейбористы потеряли голоса, причем значительная часть сторонников либеральных демократов теперь проголосовала за лейбористов и консерваторов, а многие голосовавшие за лейбористов переметнулись в стан *прочих* (хотя это и не очевидно, поскольку мы объединили национальные и альтернативные партии в одну группу «прочие», но большая часть потерь лейбористов в 2015 году приходится на Шотландию, и эти голоса отошли к Шотландской национальной партии,

которая в ходе предвыборной борьбы обещала голосовать, как лейбористы, но отдавать предпочтение интересам Шотландии). Консерваторы очень беспокоились насчет возможных успехов UKIP (Партия независимости Соединенного королевства), но в итоге этой партии отошло только одно их место. Уверен, что бывший премьер-министр Дэвид Кэмерон был бы очень не прочь узнать этот результат заранее, прежде чем обещать проведение референдума по брекситу до выборов!

Ну и хватит о британской политике. Надеюсь, эта демонстрация убедила вас в том, что TypeScript очень похож на обычный ES2017-код с немногими незначительными отличиями. Если пользоваться популярными библиотеками, содержащими высококачественные определения типов, то дополнительные усилия, необходимые для включения TypeScript в проект, могут показаться не столь уж обременительными по сравнению с результатом: мгновенной обратной связью и доступом к внутренней документации.

А что, если вы пользуетесь библиотекой, для которой нет определений типов TypeScript? Тогда у вас имеются два варианта: объявить ее с типом `any` или присвоить свойству `noImplictAny` значение `false` в файле `tsconfig.json`. Это, пожалуй, проще, чем самостоятельно писать определения типов для библиотеки, но тогда теряются все преимущества, которые дает TypeScript. Стоит посмотреть, какие библиотеки вы используете, перед тем как включать в проект TypeScript, поскольку иметь дело с нетипизированной библиотекой – лишняя головная боль; я испытал это, когда создавал первую версию этого примера!

Это был крайне поверхностный обзор TypeScript, но, надеюсь, вы все-таки дали ему шанс проявить себя в хорошем редакторе с подключаемым модулем и теперь понимаете, какой потенциал в нем скрыт. Чтобы работать с TypeScript эффективно, нужно, по крайней мере, знать, как приводить типы переменных и как определять интерфейсы. Если технология TypeScript вас заинтересовала, настоятельно рекомендую познакомиться со справочником по адресу `http://www.typescriptlang.org/Handbook`.

Mocha и Chai – разработка через поведение

Все вышеописанное завело нас довольно далеко по пути обретения уверенности в правильности визуализаций, но следующий шаг еще важнее – добавить в проект автоматизированное тестирование.

Есть много причин для написания автоматизированных тестов: если ваш продукт должен надежно отрисовывать диаграммы, а диаграмма – всего лишь часть гораздо более крупного приложения, то, наверное, вы хотели бы, чтобы автоматизированные тесты гарантировали, что изменения, внесенные где-то в приложение, не поломали диаграммы. С другой стороны, если вы работаете над проектом с открытым исходным кодом и получаете много запросов на включение добавлений и изменений от пользователей своей библиотеки, то с помощью тестов проверяется, что сторонний код не привносит ошибок. Ко всему прочему, автоматизированные тесты очень полезны, когда нужно убедить редактора в правильности своей диаграммы или когда вы просто хотите сами увериться, что ваша визуализация данных работает.

Есть два принципиально разных подхода к тестированию: разработать проект и добавить тесты постфактум и при этом, возможно, переработать какие-то части, чтобы они стали тестопригодными, или писать тесты в самом начале проекта, еще до кода, а только *потом* приступать к разработке, проверяя на каждом шаге, что тесты по-прежнему проходят.

Второй подход называется **разработкой через тестирование** (Test-Driven Development – TDD), и его следует рассматривать как знак достижения определенного уровня владения JavaScript. Обобщение этого подхода носит название **разработка через поведение** (Behavior-Driven Development – BDD), она больше ориентирована на взаимодействие с пользователем. Тесты BDD обычно менее хрупкие, поскольку внимание в них акцентируется на том, как некоторая возможность функционирует в целом, а не на особенностях ее работы.

Лично мне синтаксис, принятый в BDD-системах тестирования, кажется проще и для чтения, и для написания, его я и буду использовать – в виде библиотеки Chai.

Существует несколько типов автоматизированных тестов, но мы сосредоточимся на автономном и функциональном тестировании.

Под автономным понимается изолированное тестирование каждой функции в проекте, для этого код следует писать так, чтобы *побочные эффекты* были сведены к минимуму. Напомним (см. главу 4), что одна из целей функционального программирования – избавить функции от побочных эффектов, и автономные тесты дают способ проверить, что так оно и есть. TDD предполагает написание автономных тестов для каждой части приложения в самом начале процесса разработки; впоследствии мы одновременно пишем код и контролируем его качество.

Одно из последствий такого подхода состоит в том, что мы меньше полагаемся на ситуативные методы тестирования, т. е., вместо того чтобы переключаться между редактором и браузером после каждого сохранения, чтобы понять, правильно ли работает очередная версия кода, мы переходим к исполнителю тестов, запускаемому из командной строки, который явно сообщает, прошли тесты или нет. Зачастую это оказывается гораздо быстрее, и, стало быть, время, потраченное на написание тестов заранее, с лихвой окупается. Можно даже настроить исполнитель тестов (мы будем пользоваться программой Mocha) так, чтобы он запускался прямо из редактора.

С другой стороны, функциональное тестирование лучше сравнить с анализом того, как приложение ведет себя в потребительском контексте. Допустим, вы купили новый телефон. Все компоненты телефона тщательно протестированы на заводе, и в этот процесс вы не можете вмешаться; просто предполагается, что раз телефон покинул стены завода, значит, он прошел все тесты.

А функциональный тест проверяет, успешно ли выполняются определенные действия. Можете ли вы открыть браузер и зайти на свой любимый сайт? А после нажатия кнопки увеличения громкости громкость действительно увеличивается? А можно ли поставить будильник на следующее утро в приложении «Часы»? Из таких вот ориентированных на пользователя последовательностей действий и состоят хорошие функциональные тесты.

Методика BDD больше ориентирована на второй подход и особенно полезна в случае визуализации данных. В каком-то смысле D3 уже проделала за вас работу по автономному тестированию; коль скоро в вашем проекте используется версия D3, прошедшая все тесты (как телефон, покинувший территорию завода), вам не нужно переживать, что в самой D3 что-то не так. А думать нужно о том, как предотвратить незаметные ошибки из-за динамической типизации (например, конкатенацию чисел *1* и *1* с получением строки *11* вместо их сложения и получения результата *2*), и о том, чтобы пользовательские операции над данными не привели к ошибке. Вот тут-то и приходит на помощь BDD.

Конфигурирование проекта для работы с Mocha

Для написания автономных тестов нам понадобятся две вещи: Mocha и Chai. Mocha – процесс, запускающий тесты, и Chai – библиотека, позволяющая писать удобные утверждения. Утверждение описывает

желаемое поведение; чаще всего мы пишем утверждения *equals*, имеющие вид *фактический результат equals ожидаемый результат*. Все остальное – синтаксический сахар, цель которого – сделать утверждения и тесты более удобочитаемыми (о том, достигается эта цель или нет, можно поспорить).

Установите эти пакеты вместе с дополнительными примочками в свой проект:

```
$ npm install mocha chai mocha-loader --save-dev
```

Почему мы в этой главе все время используем `--save-dev`, а не просто `--save`? Потому что это инструменты разработчика, которые работают только в среде разработки. Включать их в распространяемый код не нужно.

Нам понадобится также новый конфигурационный файл Webpack для тестирования. Создайте файл `webpack.test.config.js` в корне проекта и поместите в него такой код:

```
const path = require('path');
module.exports = [
  {
    output: {
      path: path.resolve(__dirname, 'build'),
      publicPath: '/assets/',
      filename: 'bundle.js',
    },
    devtool: 'inline-source-map',
    module: {
      rules: [
        {
          test: /.ts?$/,
          exclude: [/(node_modules|bower_components)/],
          loader: 'ts-loader',
        },
        {
          test: /.js?$/,
          exclude: /(node_modules|bower_components)/,
          loader: 'babel-loader',
        },
      ],
    },
  },
];
```

Составим некоторое представление о том, как происходит разработка с применением Mocha и Chai, для чего добавим в созданный ра-

нее класс TypeScript несколько новых поведений. Попутно добавьте следующее предложение в секцию `scripts` файла `package.json`:

```
"start-tests": "webpack-dev-server --config webpack.test.config.js 'mocha-loader!./
lib/chapter9/index.spec.js' --inline"
```

Теперь для разработки и запуска тестов на работающем в фоновом режиме сервере введите команду

```
$ npm run start-tests
```

Тестирование поведений – BDD и Mocha

Мы не можем по-настоящему продемонстрировать подход BDD, потому что в книге уже нет места, а сколько-нибудь подробное описание автоматизированного тестирования заняло бы больше страниц, чем у нас осталось. Мы изменим функцию `typescriptSankey`, так чтобы происходила фильтрация по определенной связи при щелчке мышью по узлу, однако сделаем это в духе BDD, чтобы хоть приблизительно понять, как все это работает. Только помните, что тестирование обычно дает оптимальные результаты, когда тесты пишутся *раньше* кода!

Для начала создайте файл `lib/chapter9/index.spec.js`. По соглашению файл с тестом называется так же, как тестируемый файл, только перед расширением имени добавляется суффикс `.spec`. Прежде чем писать код, полезно изложить цели на естественном языке.

- Диаграмма должна уменьшить степень непрозрачности нерелевантных участков при щелчке по узлу.
- Релевантные участки должны остаться полностью непрозрачными.
- Повторный щелчок по узлу должен восстановить полную непрозрачность всех частей диаграммы.

Как мы увидим, утверждения в Mocha очень похожи на эти предложения[1]. Добавьте в файл `index.spec.js` следующий код:

```
import * as d3 from 'd3';
import chai from 'chai';
import sankey from './index.ts';

chai.should();

describe('functional tests for UK election sankey', () => {
  describe('select() method', () => {
```

[1] В их англоязычном написании. – *Прим. перев.*

```
    it('should set change opacity when supplied an argument',
    async () => {
      const chart = await sankey();
    });
  });
});
```

Сначала мы импортируем библиотеку D3 и настраиваем родительский блок `describe`. Блок `describe` служит для организации тестов. По соглашению в каждом `spec`-файле принято заводить один родительский блок `describe`, который содержит дополнительные логические группы. Затем идут заготовки утверждений в виде блоков `it`, аналогичных `describe`; каждый блок `it` группирует утверждения, которые мы вскоре напишем в присоединенной к `it` функции обратного вызова. Можно считать каждое предложение `it` тестом, а собранные внутри утверждения – частями теста, нуждающимися в проверке.

Мы проверим каждую часть диаграммы и убедимся, что она возвращает правильные данные. Сначала напишем вспомогательную функцию, которая сэкономит немного работы. Поместите следующий код внутрь блока `it`:

```
function getOpacities(source = null) {
  chart.select(source);

  const links = d3.selectAll('.link[opacity="1"]')
    .data()
    .map(d => d.target.name)
    .sort();

  const nodes = d3.selectAll('.node[opacity="1"]')
    .data()
    .map(d => d.name)
    .sort();

  return {
    links,
    nodes,
  };
}
```

Пора уже и тесты написать. Поскольку наша диаграмма – асинхронная функция, то для тестирования ее частей мы напишем еще несколько асинхронных функций. После `getOpacities()` добавьте такой код:

```
await (async () => {
  const { links, nodes } = getOpacities('CON2010');
```

```
  links.should.eql(['CON2015', 'LAB2015', 'UKIP2015']);
  nodes.should.eql(['CON2010', 'CON2015', 'Green2015',
                    'LAB2015', 'LIB2015', 'Other2015', 'UKIP2015']);
})();
```

Эта функция получает два массива от `getOpacities()` и сравнивает их с ожидаемыми результатами. Обратите внимание, что в `getOpacities()` массивы сортируются, поэтому отсортированными должны быть и *ожидаемые* массивы.

Ниже показаны все остальные тесты, поместите их после написанного ранее:

```
await (async () => {
  const { links, nodes } = getOpacities('LAB2010');
  links.should.eql(['CON2015', 'LAB2015', 'Other2015']);
  nodes.should.eql(['CON2015', 'Green2015', 'LAB2010',
                    'LAB2015', 'LIB2015', 'Other2015', 'UKIP2015']);
})();

await (async () => {
  const { links, nodes } = getOpacities('LIB2010');
  links.should.eql(['CON2015', 'LAB2015', 'LIB2015', 'Other2015']);
  nodes.should.eql(['CON2015', 'Green2015', 'LAB2015',
                    'LIB2010', 'LIB2015', 'Other2015', 'UKIP2015']);
})();

await (async () => {
  const { links, nodes } = getOpacities('Green2010');
  links.should.eql(['Green2015']);
  nodes.should.eql(['CON2015', 'Green2010', 'Green2015',
                    'LAB2015', 'LIB2015', 'Other2015', 'UKIP2015']);
})();

await (async () => {
  const { links, nodes } = getOpacities('Other2010');
  links.should.eql(['LIB2015', 'Other2015']);
  nodes.should.eql(['CON2015', 'Green2015', 'LAB2015',
                    'LIB2015', 'Other2010', 'Other2015', 'UKIP2015']);
})();
```

Все тесты похожи. Напоследок добавим тест, который проверяет, что происходит, когда не выбрано ни одного узла:

```
await (async () => {
  const { links, nodes } = getOpacities(null);
  links.should.have.length(13);
  nodes.should.have.length(11);
})();
```

Тут мы немного поленились и проверяем только длины результирующих массивов. Но если угодно, можно проверить и их состав.

Итак, тесты собраны, пора проверить, проходят ли они. Запустите сервер следующей командой:

```
$ npm run start-server
```

Появится сообщение о том, что тесты не прошли:

На самом деле это прекрасно – значит, мы не получаем ложноположительного результата о том, что все якобы хорошо, хотя тесты не прошли. Ложноположительный результат – угроза для тестирования: если асинхронный тест написан неправильно, он может пройти. По счастью, здесь мы пользуемся паттерном `async/await`, который абстрагирует различные потенциальные проблемы. Обычно первым аргументом блоку `it()` передается функция с именем `done()`, которая вызывается, если асинхронные операции завершились и все утверждения справедливы. Тем не менее при написании тестов обязательно проверяйте, не завершились ли они с ошибкой!

Еще один результат ошибочного завершения тестов состоит в том, что мы получаем информацию о том, *где* произошла ошибка, и это позволяет быстро находить и устранять ошибки. В этом и состоит суть тестов – точно указать, где сломалось, если что-то в процессе разработки пошло наперекосяк.

Допишем недостающее. В конец функции `typescriptSankey()` в файле `index.ts` добавьте такой код:

```
const select = (item: string|null) => {
  if (item) {
    const filteredLinks = sankeyData.links.filter(d => d.source.name === item);
    const filteredNodes = sankeyData.nodes
```

```
        .filter(d => d.name === item || d.name.match(/2015$/));

    svg.selectAll('.link')
      .attr('opacity', d => d.source.name === item ? 1 : .3);
    svg.selectAll('.node')
      .attr('opacity', d => (d.name === item || d.name.match('2015')) ? 1 :
.3);
  } else {
    svg.selectAll('.link')
      .attr('opacity', 1);

    svg.selectAll('.node')
      .attr('opacity', 1);
  }
}
```

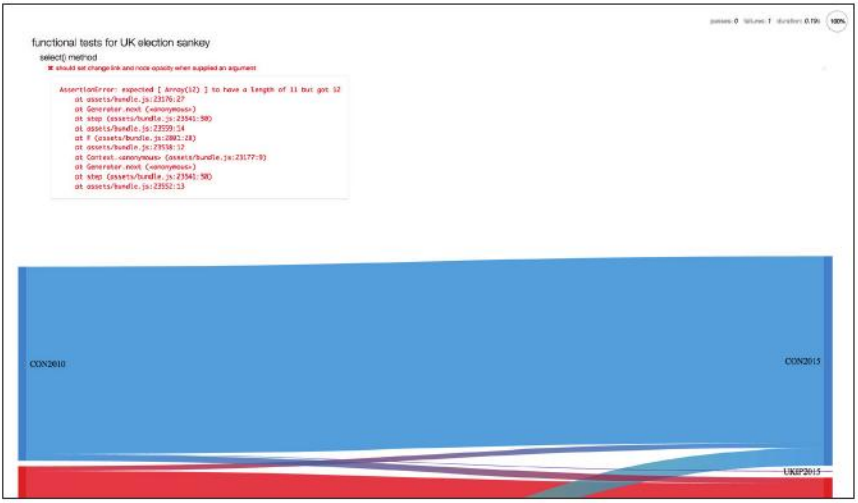
Затем добавьте обработчики событий:

```
let current = null;
node.on('click', e => {
  if (current === e.name) {
    current = null;
  } else {
    current = e.name;
  }
  select(current);
});
```

И наконец, наши методы должны что-то возвращать, чтобы выполнить обещание асинхронной функции:

```
return {
  select,
  node,
  link,
  data: sankeyData,
};
```

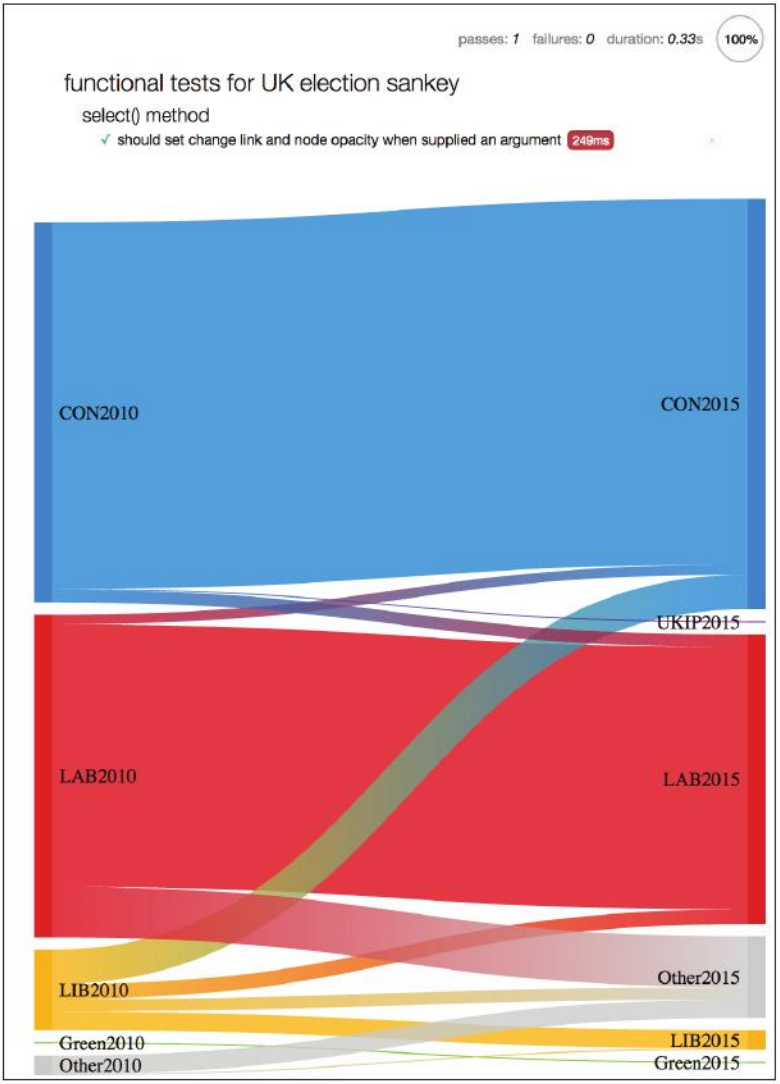
Сохранитесь – и в добрый путь!



Вот черт! Тест по-прежнему падает. Все из-за этого лишнего узла *Ukip2010*, который мы заметили раньше. Я напортачил при создании набора данных и не мог понять, как это исправить. Тем не менее заставим программу работать. Поместите следующий код перед методом `select()` в файле `index.ts`:

```
node.selectAll('rect[height="0"]')
  .each(function(d){
    this.parentNode.remove();
  });
```

Это не совсем честно, ну да ладно. Мы выбираем все прямоугольные узлы с нулевой высотой (такие как узел «UKIP2010») и удаляем их родителей. В результате мы избавимся от неправильно помещенной метки «UKIP2010». Сохранитесь.



Вот теперь все наши тесты прошли, и можно радостно потирать руки!

Повторю еще раз, у меня нет никакой возможности детально осветить эту тему в одной главе, но, надеюсь, вы поняли, как приступить к тестированию своих проектов. Если вас это заинтересовало, то горячо рекомендую почитать дополнительную литературу о философии

автоматизированного тестирования и BDD; писать тесты довольно легко, но вот для написания *хороших* тестов требуется изрядное мастерство. И хотя автоматизированное тестирование – удачный инструмент для обретения уверенности в собственных визуализациях данных, доверять ему на все сто процентов не стоит. Можно написать кучу тестов, которые в действительности ничего не проверяют, а кроме того, есть опасность уклониться в сторону составления тестов, подтверждающих правильность вашего кода, а не проверяющих ее. При всем при этом тестирование может стать исключительно полезным инструментом написания кода мирового класса.

В следующий раз, как почувствуете непреодолимую тягу к написанию автоматизированных тестов, попробуйте для забавы двухминутную задачку от «Нью-Йорк таймс», и если потерпите неудачу, то пишите больше отрицательных тестовых утверждений. Задачку вы найдете по адресу http://www.nytimes.com/interactive/2015/07/03/upshot/a-quick-puzzle-to-test-your-problem-solving.html.

Резюме

Эта глава была посвящена инструментальным средствам. Начали мы со стилистической проверки с помощью ESLint, затем перешли к TypeScript и статической типизации, а закончили тестированием средствами Mocha. Материал обширнейший! Не переживайте, если чувствуете некоторое ошеломление, это просто потому, что обзор был очень поверхностным. Существует великое множество ресурсов, на которых можно продолжить знакомство, я настоятельно рекомендую почитать еще о тестировании, потому что из всех рассмотренных тем эта, пожалуй, самая обширная и важная.

В заключение мы рассмотрим несколько особенно впечатляющих примеров визуализации данных. Не волнуйтесь, кода больше не будет; откиньтесь на спинку стула, расслабьтесь и получайте удовольствие, а мы пока расскажем, из чего состоит качественная визуализация.

Глава 10

Проектирование хорошей визуализации данных

Визуализация данных – инструмент, который можно использовать по-разному. В этой книге мы видели, что иногда визуализация служит для представления информации необычными или интересными способами, иногда она проясняет суть дела, а иногда нужна просто ради красивого внешнего эффекта.

Кем бы вы ни были – журналистом, стремящимся наглядно продемонстрировать изменение ВВП; ученым, которому нужно сообщить коллегам о результатах эксперимента; программистом, ищущим способы интегрировать визуализацию в программу, – скорее всего, вы хотели бы, чтобы визуализация была ясной, лаконичной и не вводящей в заблуждение. Примеры в этой главе взяты в основном из области новостных СМИ, но многие обсуждаемые моменты относятся и к визуализации вообще.

В этой главе мы рассмотрим несколько общих принципов, о которых следует помнить при создании визуализаций, и я проиллюстрирую их примерами хорошей визуализации. Замечу, что сам я ни в коем случае не являюсь профессионалом в сфере визуализации, я разработчик и журналист, приобретший некоторые познания в области дизайна, и на мои представления о том, что называть *хорошей визуализацией данных*, сильно повлиял опыт построения объяснительной графики для таких изданий, как *Financial Times*, *Times*, *Economist* и *Guardian*. Отделы новостей в них очень динамичны, а цель обычно – сообщить публике важные аспекты набора данных, а не дать чи-

тателям возможность исследовать данные. Хотя ваш способ использования D3 может отличаться от присущего журналисту, многое из сказанного ниже относится также к представителям академических кругов, издательского бизнеса и другим; умение быстро и лаконично передавать информацию весьма ценно в любой профессии.

А теперь, покончив со вступлением, перейдем к обсуждению деталей того, что составляет хорошую визуализацию данных.

Выбор правильных характеристик данных и типа диаграммы

Первый шаг на пути к достойной визуализации – выбор подходящего типа диаграммы. Очень часто у данных много различных характеристик, и ваша задача – решить, какие их них визуализировать. А потом определиться с тем, *как именно* их визуализировать.

Очень легко впасть в заблуждение, будто для всего на свете нужна карта. И особенно это относится к освещению выборов; все представляют себе гигантские фоновые диаграммы, перед которыми стоит телекомментатор с указкой и сообщает о поступающей информации о ходе выборов, причем каждый регион окрашивается в цвет партии, за которую проголосовал. Однако карты полезны в основном для изображения географических данных и чтобы дать представление о близости предметов в пространстве. В США штаты Айдахо и Нью-Гэмпшир дают по четыре выборщика, но второй в несколько раз меньше первого. Штат Нью-Йорк по площади гораздо меньше Монтаны, но число выборщиков от него 29, тогда как от Монтаны всего три. Если цель состоит не в том, чтобы показать, что все штаты, находящиеся в определенной части страны, голосуют определенным образом, то не имеет особого смысла представлять эти данные в виде карты (если не считать узнаваемости этого формата отображения страны).

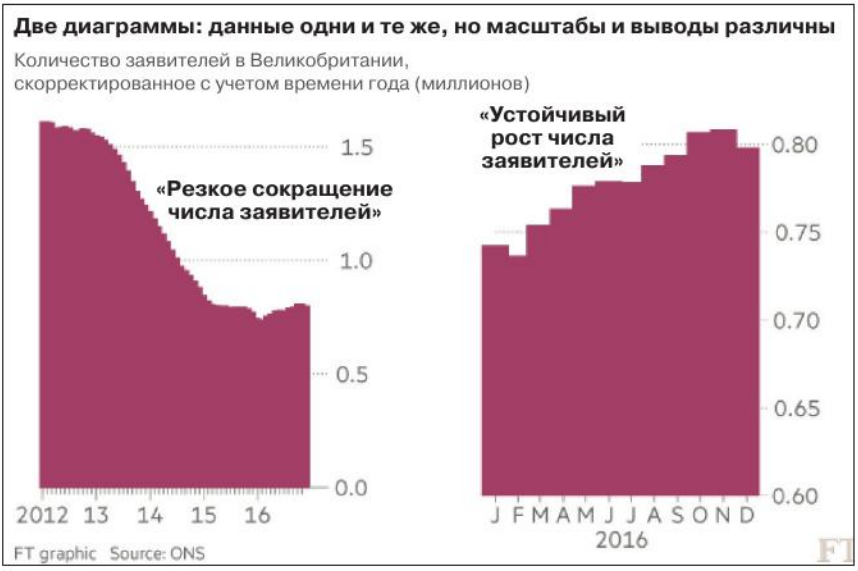
Моя группа в газете *Financial Times* пользуется инструментом Visual Vocabulary, основанным на идее Graphic Continuum Йона Швабиша (Jon Schwabish) и Северино Рибекки (Severino Ribecca). Чтобы воспользоваться им, нужно выбрать интересующие характеристики данных, после чего будет предложено несколько подходящих для их представления типов диаграмм. К тому же репозиторий на GitHub содержит примеры D3-диаграмм каждого типа. Если вас это заинтересовало, зайдите по адресу ft-interactive.github.io/visual-vocabulary и разветвите репозиторий примеров (github.com/ft-interactive/visual-vocabulary).

Ясность, честность и чувство цели

В настоящее время существуют две школы визуализации данных: ультраминималисты, возглавляемые Альберто Каиро (Alberto Cairo) и Эдуардом Тафтом (Edward Tufte), которые считают, что главная цель визуализации данных – свести к минимуму возможность двоякого толкования, и те, кто использует данные для создания изящных вещиц, в которых дизайн важнее передачи голой информации. Если это непонятно из названия раздела, поясню, что я, вообще говоря, считаю первый подход более подходящим в большинстве случаев.

Если ваша цель – наглядно представить данные, то худшее, что можно сделать, – это ввести аудиторию в заблуждение, намеренно или случайно. Мало того что вы утратите доверие публики, как только она обнаружит, что стала жертвой обмана, так еще поспособствуете распространению скептического отношения к способности данных сообщать правду.

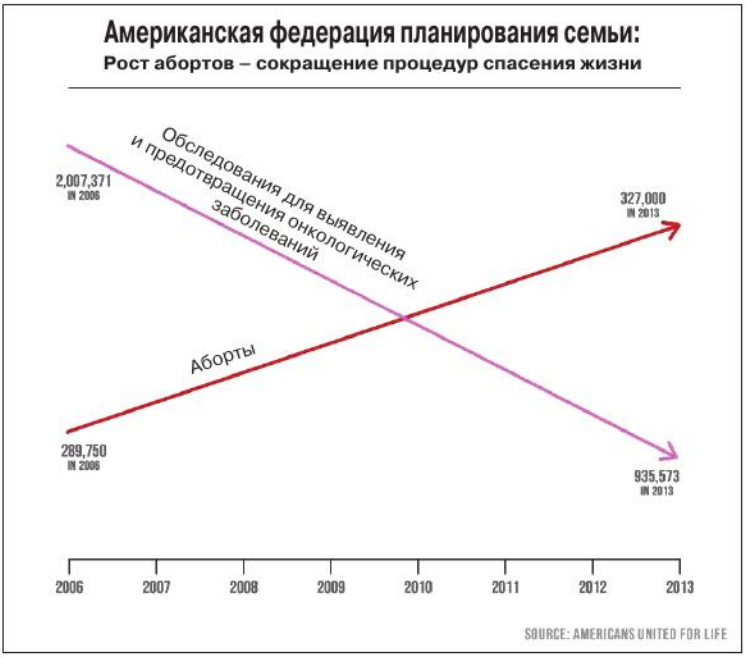
Проще всего напортачить с осями и масштабами. Обычно следует начинать отсчет с нуля, потому что в противном случае форма диаграммы сильно исказится, и это скроет важную информацию от зрителя. В случае временных рядов объем данных также может повлиять на восприятие диаграммы. Вот пример из серии статей Алана Смита (Alan Smith) *Chart Doctor*:



Взято из статьи Алана Смита «How alternative facts rewrite history» (https://www.ft.com/content/3062d082-e3da-11e6-8405-9e5580d6e5fb)

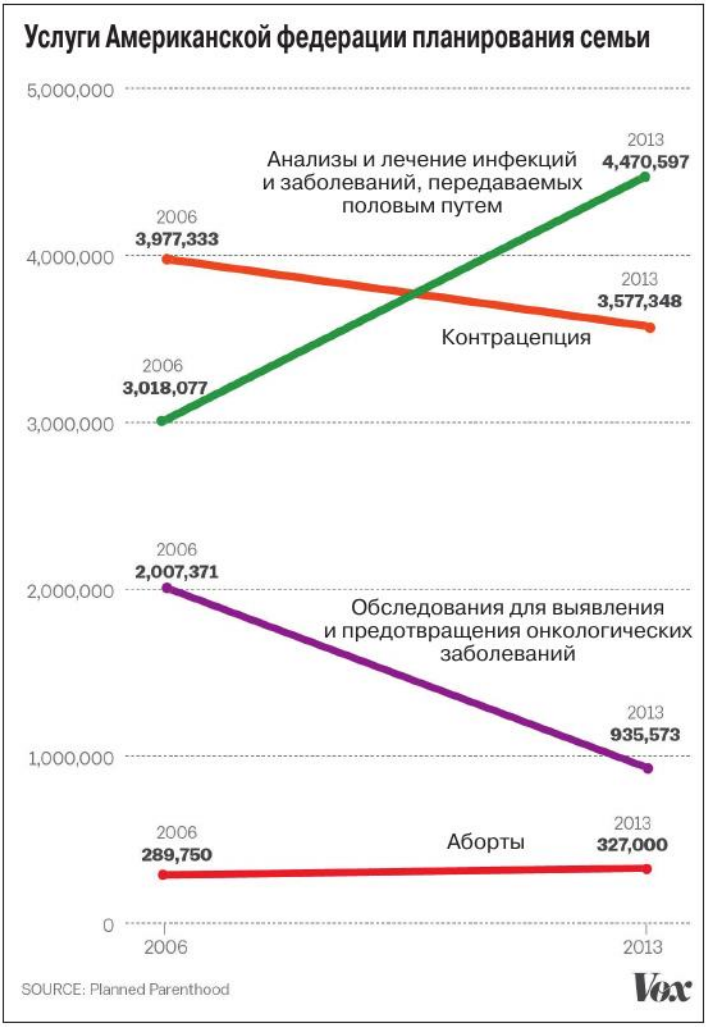
На этом рисунке показано, как простое изменение периода времени может изменить смысл сказанного. Изображен один и тот же набор данных – число граждан Великобритании, подавших заявку на получение пособия по безработице и представивших доказательства, что они действительно ищут работу. В какой-то степени это отражает уровень безработицы в стране. Правая диаграмма особенно дезориентирует, потому что ось *y* начинается не с нуля, так что рост на 0,2 млн в 2016 году сильно преувеличен. Напротив, левая диаграмма начинается с нуля, и, хотя на ней виден рост, изображенный на второй диаграмме, расширение периода на годы, начиная с 2012-го, показывает, что с 2015 года кривая практически плоская, а количество заявителей гораздо меньше, чем несколькими годами ранее, в 2012-м.

А вот еще один современный пример. В сентябре 2015 года в конгрессе США прошли слушания по инициативе Американской федерации планирования семьи, организации, занимающейся репродукцией и женским здоровьем. На слушаниях конгрессмен от республиканцев Джейсон Чаффец (Jason Chaffetz) показал следующую диаграмму, созданную группой противников абортов «Союз американцев за жизнь» (Americans United For Life):



На этой диаграмме много проблем, и не последняя из них – полное отсутствие оси *y*. Сотрудники проекта «Политифакт» перерисовали диаграмму с исправленными осями, и вот что получилось:

Организация Vox пошла еще дальше и дорисовала другие услуги Американской федерации планирования семьи:



Как видим, хотя количество обследований для выявления и предотвращения онкологических заболеваний, безусловно, уменьшилось (как, впрочем, и применение контрацептивов) и имеет место небольшое увеличение числа абортов, одновременно наблюдается резкий рост расходов на лечение и предотвращение инфекций и заболеваний, передаваемых половым путем. Вот как Альберто Каиро прокомментировал исходную диаграмму:

> *Эта картинка – ложь от начала до конца... Вне зависимости от того, что люди думают по этому вопросу, такое искажение истины неприемлемо с точки зрения этики.*

Публичная реакция на эту вводящую в заблуждение диаграмму привела к тому, что издание Quartz назвало ее «самой лживой диаграммой 2015 года». И не важно, каковы были изначальные намерения ее создателей, всякая надежда на достижение поставленной цели испарилась, когда читатели поняли, что их обманывают.

Более подробное обсуждение всех несообразностей в диаграмме Чаффеца, найденных Политифакт и Vox, см. соответственно на страницах http://www.politifact.com/truth-o-meter/statements/2015/oct/01/jason-chaffetz/chart-shown-planned-parenthood-hearing-misleading-/ и http://www.vox.com/2015/9/29/9417845/planned-parenthood-terrible-chart.

В приведенной выше цитате Каиро сделал интересное замечание о том, что при доведении данных до аудитории необходимо соблюдать определенные этические нормы. Именно этим *визуализация данных* отличается от *искусства данных*; во втором случае к *художнику* предъявляется этическое требование осознанно и честно передать свои эмоции, верования, страхи и т. д. Это совсем не то, что требуется с точки зрения этики от *визуализатора*, а именно: передать конкретные качественные характеристики данных визуальными методами. Можно сделать следующий шаг и сказать, что этика журналистики данных требует от журналиста объяснять аудитории, что *означают* данные и какое отношение они имеют к этой конкретной аудитории.

В своих проектах точно определяйте поставленную цель. Вы играете роль журналиста данных, который хочет провести читателя по морю цифр? Или роль визуализатора данных, например занятого созданием контрольной панели, позволяющей быстро и эффективно построить выжимку из очень большого многомерного набора данных? Или же вы намереваетесь создать нечто развлекательное, а данные используются только как способ достижения этой цели? Любая из этих ролей вполне допустима, и поле деятельности простирается от скрупулезного объяснения смысла линейных графиков до создания *предметов искусства*, позволяющих лучше понять нашу значимость и место во Вселенной. Но что бы вы ни делали, необходимо четко представлять свои цели и никогда не вводить аудиторию в заблуждение.

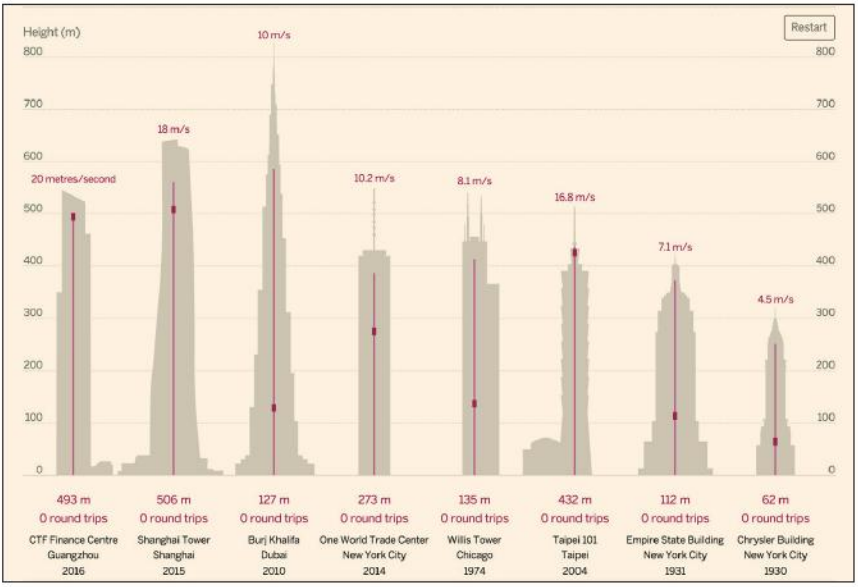
Однако иногда необходимо обращать внимание на то, какой смысл несут данные и в чем состоит послание. Вот хороший пример: диаграмму не следует начинать с нуля, если данные вообще не приближаются к нулю. Если построить стопочную диаграмму распределения голосов избирателей по партиям на федеральных выборах в США за двадцать лет, начав ось *y* с нуля, то получится фактически плоская прямая. Аналогично если построить график индекса Доу-Джонса начиная с 1900 года в линейном масштабе, то вся Великая депрессия покажется плоской и совершенно незначительной (вообще, при изображении данных, различающихся на несколько порядков, часто имеет смысл использовать логарифмический масштаб; хороший вводный материал о том, когда и почему следует применять логарифмический масштаб, см. в серии статей «Chart Doctor», упомянутой в начале этой главы).

Помогайте аудитории понять масштаб

Важная часть визуализации данных – передача величины масштаба и различий. В приведенных ниже примерах это сделано образцово.

Для начала рассмотрим график скоростных лифтов, подготовленный Джоном Бэрн-Мэрдоком для газеты *Financial Times* (http://www.ft.com/cms/s/2/1392ab72-64e2-11e4-ab2d-00144feabdc0.html).

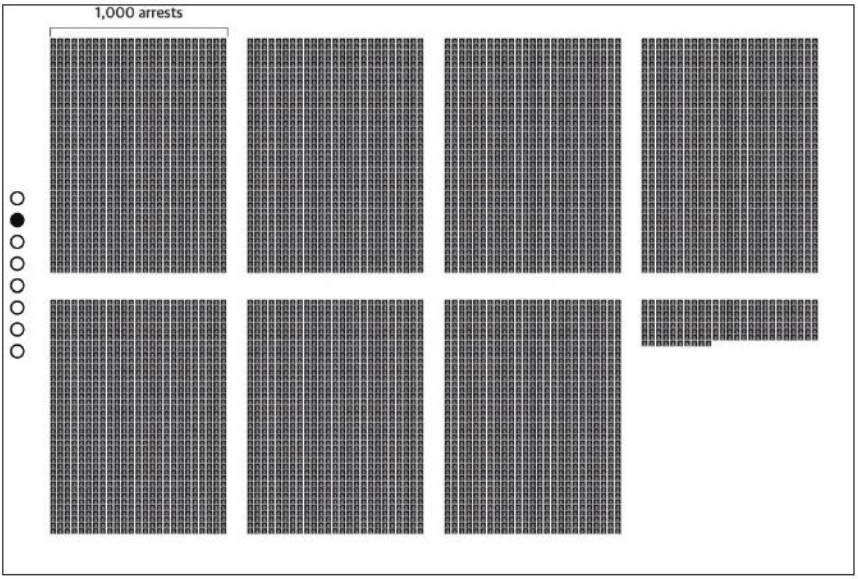
Следующий рисунок не позволяет составить о нем полное представление:



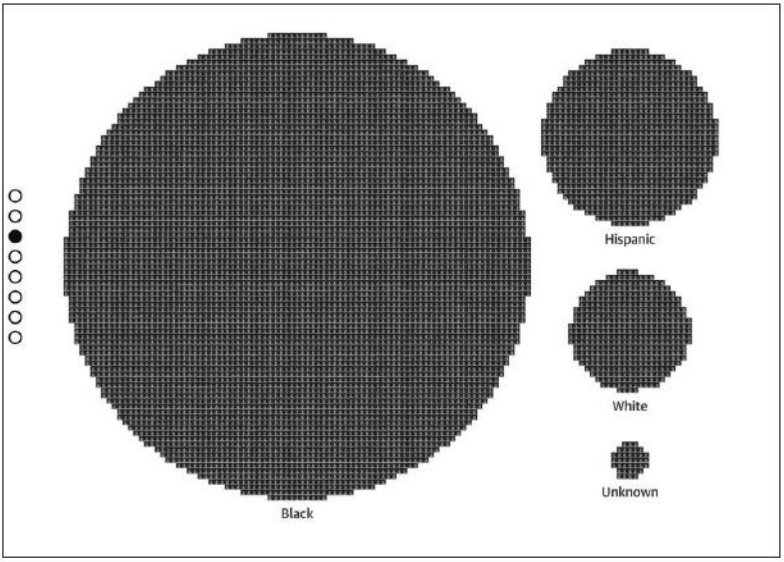
На динамической визуализации лифты в каждом здании бесконечно поднимаются и опускаются, а счетчик внизу показывает, сколько полных циклов подъем–спуск лифт совершил за время, пока вы смотрите на страницу. Небольшое замедление в верхней и нижней точках создает впечатление, что крохотный сиреневый квадратик ведет себя как настоящий лифт, подчиняясь тем же законам физики, что и большая металлическая клетка, движущаяся вверх-вниз по самым высоким в мире зданиям. В печатной версии можно передать только одну характеристику – относительную высоту лифта и здания, а в интерактивной – еще и скорость лифта. Это пример превосходства цифровых СМИ над печатными в плане представления данных. Для демонстрации масштаба визуальный контент увязывается со временем: просто сказав, что *лифт совершает полную поездку вверх-вниз за две минуты*, мы не получим того эффекта, какой дает прямое восприятие этих двух минут читателем.

Еще более красноречивый пример использования возможностей цифровых СМИ для демонстрации масштаба дает интерактивная визуализация «Хоман сквер: портрет заключенных в Чикаго», опубликованная газетой *Guardian*. Если вы ее не видели, загляните на страницу http://www.theguardian.com/us-news/ng-interactive/2015/oct/19/homan-square-chicago-police-detainees и приготовьтесь к сильному душевному переживанию.

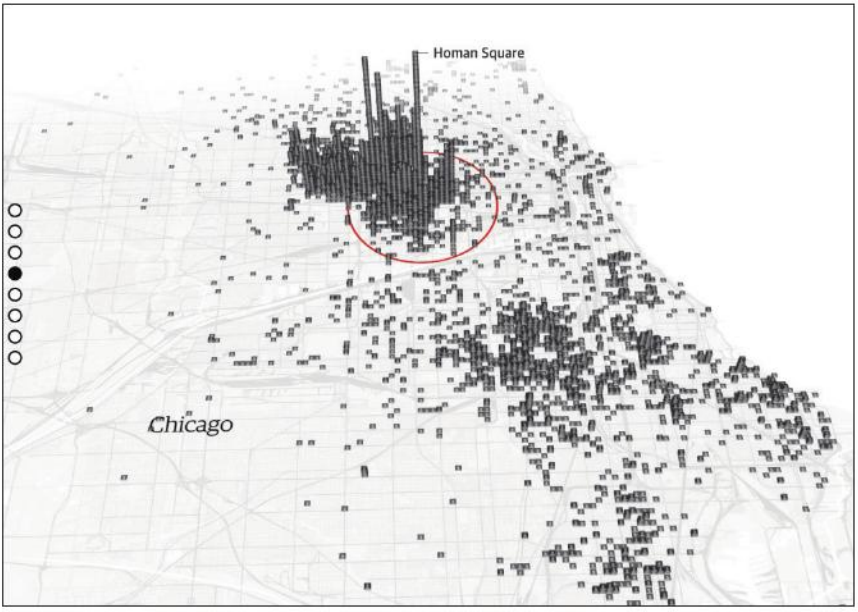
Визуализация начинается с огромного набора безликих серых фигур, каждая из которых представляет одного заключенного в секретной полицейской тюрьме в Чикаго:



По мере прокрутки по экрану начинают перемещаться лица, образуя различные конфигурации, например такую пузырьковую диаграмму:



Они могут сложиться также в изоморфную карту:



Группа интерактивной графики газеты *Guardian* опубликовала увлекательный перечень вопросов и ответов касательно процесса создания этой визуализации, который стоит почитать. Его можно найти на странице https://source.opennews.org/en-US/articles/how-we-made-homan-square-portrait/.

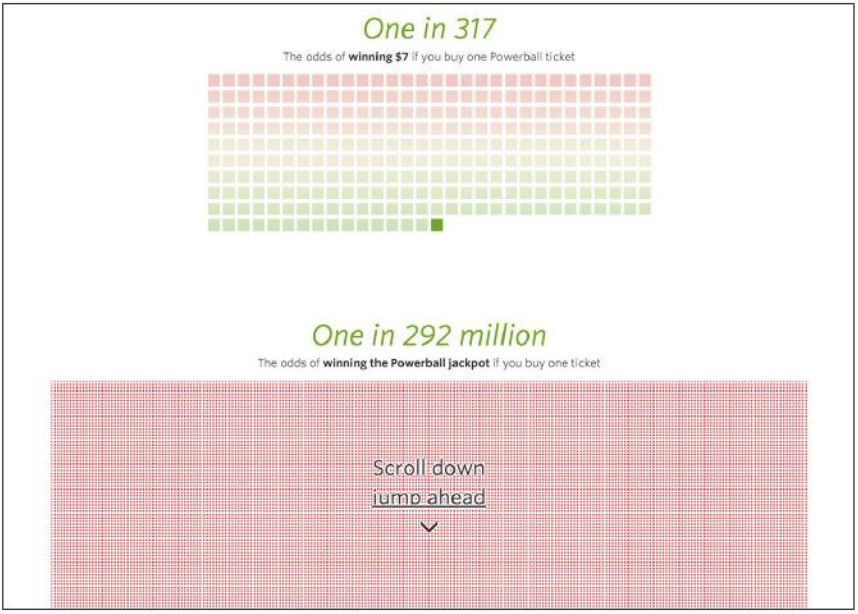
Хотя данные принимают разные формы, вы ни на минуту не забываете, как они выглядели вначале: это не просто пиксели на экране, каждая представляет собой конкретного человека со сломанной судьбой (это ощущение усиливается, когда позже в визуализации появляются портреты отдельных людей).

Впрочем, чтобы быть убедительным, ваш проект не должен быть таким сложным, как *Хоман сквер*. Еще один пример удачного использования анимации для объяснения масштаба – *проект Gapminder* великого, ныне покойного, *Ханса Рослинга* (http://www.gapminder.org/world/), который показывает, как наш мир изменялся со временем. В многочисленных выступлениях на тему понимания мира через данные Рослинг говорил о том, что если судить по таким показателям, как ожидаемая продолжительность жизни и ВВП на душу населения, мир становится значительно лучше, а наше восприятие других стран часто коренится в крайнем непонимании того, насколько мы все похожи:

Пузырьки на этой диаграмме постепенно смещаются из левого нижнего квадранта в правый верхний, демонстрируя рост ожидаемой продолжительности жизни и душевого дохода. Стоит сравнить эту довольно сухую объяснительную визуализацию с выступлением Рослинга на конференции TED (https://www.ted.com/talks/hans_rosling_shows_the_best_stats_you_ve_ever_seen). Хотя демонстрируются примерно те же данные, впечатление гораздо ярче благодаря повествованию Рослинга, в голосе которого ясно слышится волнение, когда он рассказывает, как изменяется и улучшается мир с развитием общества. Хорошую объяснительную визуализацию данных необязательно ограничивать тем, что можно изобразить на экране.

Та же диаграмма, воспроизведенная средствами D3 Майком Бостоком, имеется по адресу http://bost.ocks.org/mike/nations/.

И последний пример – визуализация ваших шансов сорвать джекпот в лотерее Powerball, подготовленная Wall Street Journal; увы, и в этом случае рисунки не передают всего, что происходит на экране, зайдите на страницу http://graphics.wsj.com/lottery-odds/. И, оказавшись там, постарайтесь хотя бы некоторое время воздержаться от нажатия кнопки *jump ahead*:



После довольно продолжительной непрерывной прокрутки экран будет выглядеть так:

В этот момент вы осознаете, что нет никаких шансов прокрутить до конца, и нажимаете *jump ahead*. Если вы по-прежнему убеждены, что можете планировать жизнь на пенсии на доходы от выигрышей в лотерею, то вас ничто не исправит.

В этом случае масштаб подчеркивается сравнением вероятности выпадения орла при подбрасывании моменты с вероятностью угадать выигрышную комбинацию в лотерее Powerball. Сравнить шансы 1 к 2 и 1 к 292 201 337 на одном экране (и тем более на печатной странице) невозможно, поскольку при любом масштабе первая величина несоизмеримо меньше второй. А вот использование ничем не ограниченного размера экрана по вертикали – весьма эффективный способ связать визуальные элементы с физическим свойством (вспомните интерактивную скорость лифта в примере выше), поскольку физические усилия, необходимые для того, чтобы пройти хотя бы 1% пути (не говоря уже о количестве точек, составляющем этот путь), – на-

глядная демонстрация всей огромности числа «292 миллиона» в контексте сравнения вероятностей.

Эти примеры великолепно передают читателю ощущение масштаба. Визуализация данных является столь мощным инструментом не в последнюю очередь потому, что позволяет ответить на вопрос, *насколько велико «большое» и насколько мало «малое»*. Поскольку работа репортера все чаще опирается на данные, очень легко ввести читателя в заблуждение, недооценив или переоценив масштаб (именно поэтому диаграмма Чаффеца, описанная в предыдущем разделе, была сочтена обманом). Часто проблему можно смягчить, проектируя визуальное представление так, чтобы зритель чувствовал некоторый эмоциональный контакт с визуальными раздражителями на экране.

Эффективное использование цвета

Одно из главных решений, которые приходится принимать при создании визуализации данных, – выбор цветов. Конечно, и монохромные диаграммы способны передать немало информации, но задействование широкого диапазона цветов, представимых на цифровом дисплее, может оказаться чрезвычайно эффективным способом изобразить дополнительную характеристику визуализируемых данных.

Если вы планируете использовать цвет для передачи информации, то следует учесть несколько моментов. Во-первых, сможет ли пользователь понять, что именно представляет каждый цвет. Если имеется надпись, объясняющая назначение цветов, то попробуйте убрать ее – сможете ли вы тогда разобраться, что означают комбинации цветов. Передается ли смысл увеличением яркости, контрастными значениями или просто выбором конкретного цвета (в последнем случае без пояснительной надписи точно не обойтись)?

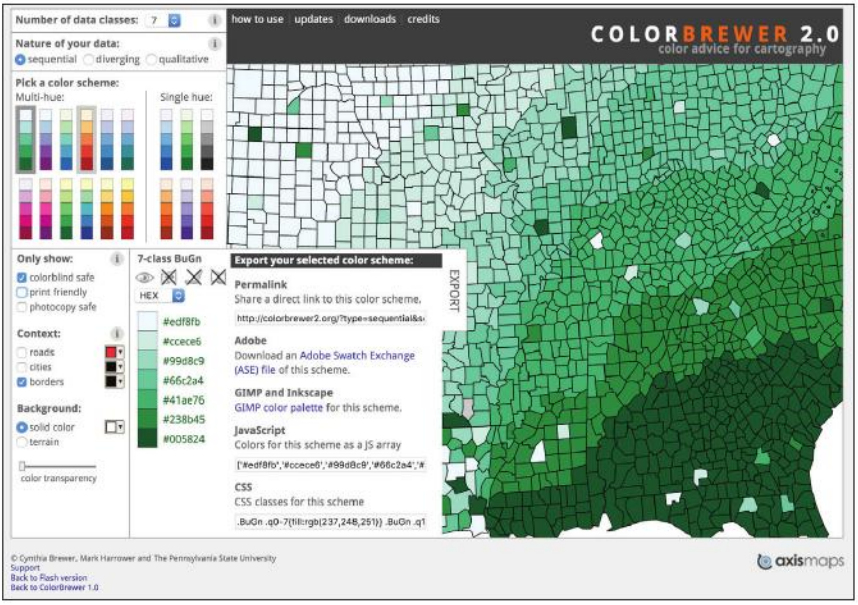
Во-вторых, не забывайте о людях, страдающих цветовой слепотой. В фоновых диаграммах чаще всего применяются *цвета светофора*: зеленым цветом обозначаются низкие значения, желтым – средние, красным – высокие. Проблема в том, что дальтоникам, не различающим зеленый и красный цвета, минимальные и максимальные значения будут казаться одинаковыми.

И раз уж мы заговорили о цветах светофора, при обсуждении таких деликатных тем, как иммиграция, следует хорошенько подумать, выбирая для цветового кодирования зеленый или красный цвет: если красный означает «плохой» или «опасный», так ли это на самом деле? Или красный цвет использован только потому, что цветовой масштаб

был ограничен значениями в наборе данных? Цвета несут значительный эмоциональный заряд, и при их использовании очень легко непреднамеренно выразить некоторую пристрастность. В общем случае безопаснее остановиться на последовательной цветовой схеме (масштабированной на среднее ожидаемое значение).

Очень хороший способ подбора цветовой схемы для карты (и много чего еще) предлагает **ColorBrewer** (http://colorbrewer2.org) – программа, разработанная Синтией Брюер (Cynthia Brewer) и Марком Хэрроуэром (Mark Harrower) из Университета штата Пенсильвания. Она предоставляет ряд предопределенных цветовых схем для представления данных, и вы можете указать, какой тип связей хотите визуализировать.

Кроме того, она позволяет отбросить цветовые схемы, не подходящие для дальтоников, а также плохо воспроизводимые устройством печати или фотокопирования.



Схемы ColorBrewer можно использовать прямо в D3! Масштаб `d3.schemeCategory20c` из модуля `d3-color` как раз содержит предлагаемые ей цвета. Еще более простой способ их использования дает пакет `colorbrewer`. Выполните команду

```
$ npm install colorbrewer --save
```

а затем включите пакет директивой `require('colorbrewer')`, как обычно. Чтобы применить показанный на предыдущем рисунке масштаб 7-Class BuGn (его название находится слева от панели экспорта в пользовательском интерфейсе Colorbrewer2), нужно написать:

```
import BuGn from 'colorbrewer';
const sevenColorBlueGreen = BuGn[7];
```

Наконец, учитывайте естественные особенности данных: если вы визуализируете данные, относящиеся к политическим партиям, то имеет смысл кодировать их цветами, похожими на цвета партий (конечно, использовать цвета партий скучно, но новации, идущие вразрез с этой традицией, могут стать причиной когнитивного диссонанса, а это хуже скуки).

Оцените свою аудиторию

Потенциальная аудитория – один из важнейших факторов, которые необходимо принимать во внимание, начиная новый проект визуализации. Тут есть два аспекта: первый – редакторский (что аудитория знает о рассматриваемой теме? диаграммы каких типов аудитория сможет воспринять и правильно прочитать? как эти диаграммы соотносятся с более широким контекстом рассказываемой истории и другими опубликованными на эту тему работами?), второй – технический (какие платформы и устройства будут использоваться для ознакомления с материалом?).

Очень важно сделать примерный эскиз заказанной визуализации данных, прежде чем приступать к написанию кода, и подойти к этому можно по-разному. С одной стороны, никогда не повредит прикинуть форму данных перед выбором конкретного стиля визуализации. Меня часто просят нарисовать секторные диаграммы, выделив нетипичные малые значения (выбросы). Но эта идея совершенно неработоспособна (на фоне остальной диаграммы эти выбросы будут просто незаметны). Для прикидки вам даже не понадобятся карандаш и бумага; загрузите данные в Excel и поэкспериментируйте со встроенными в эту программу диаграммами, перед тем как воплощать свои идеи в коде.

Второй способ – сделать набросок визуализации на уровне компонентов или взаимодействия. Здесь важно понимать, какими устройствами пользуется ваша аудитория. Если у вас уже есть предыдущие работы, познакомьтесь с их аналитикой. Чтобы узнать в *Google*

Analytics долю читателей, пользующихся определенным браузером, зайдите в раздел **Audience ⇒ Technology**. Обратите внимание также на раздел **Audience ⇒ Mobile ⇒ Overview**, где сообщается о том, какая часть аудитории пользуется мобильными устройствами, а какая – настольными. Если аналитических данных нет (например, вы начинаете новый проект), то обычно разумно предположить, что половина аудитории мобильна, и соответственно подходить к проектированию.

Разработка для нескольких разрешений экрана носит название «**отзывчивый дизайн**» (responsive design), и вести ее можно двумя способами: «сначала мобильные» и «сначала настольные». Традиционно вначале рассматривается размер экрана настольного компьютера, а затем – уменьшенный экран мобильного устройства, но это зачастую означает, что поддержка мобильных устройств изначально не закладывается, и тогда возникают проблемы с уменьшением больших элементов. Если принят подход «сначала мобильные», то мы начинаем с минимально возможного размера и затем увеличиваем его. Это проще, потому что самые трудные варианты страницы проектируются вначале, а по мере увеличения разрешения размер элементов может возрастать, ничему не мешая.

Нужно ли реализовывать оба варианта дизайна, зависит от аудитории – вполне возможно, что она предпочитает какую-то одну платформу, и тогда можно не подстраиваться под обе. И в этом случае может помочь аналитика. Но хочу добавить – хотя для обеспечения правильной работы на всех устройствах и во всех браузерах требуется приложить немало дополнительных усилий, это должно стать стандартом качества, к которому следует стремиться при создании визуализаций для широкой публики.

Если вы сначала создаете дизайн для мобильных устройств, возьмите бумагу и черный маркер и нарисуйте несколько фигур размером с телефон. Нарисуйте прямоугольники, обозначающие расположение всех элементов интерфейса (кнопок, раскрывающихся списков, переключателей и т. д.) или диаграмм. Как отличить один элемент от другого, решать вам, важно только, чтобы все члены команды использовали один и тот же набор обозначений. Никаких художественных изысков не требуется, картинка должна быть детальной лишь настолько, чтобы было понятно, как, на ваш взгляд, должно *выглядеть* каждое взаимодействие с пользователем (щелчок/касание, щипок, буксировка и т. д.). Нарисуйте примерный эскиз того, куда должен *перетекать* каждый элемент в зависимости от устройства. Затем сделайте то же самое для большого экрана настольного компьютера:

Если есть возможность, попросите нескольких человек посмотреть ваши рукописные прототипы и оценить естественность взаимодействия.

Несколько принципов дизайна для мобильных и настольных устройств

Мобильные и настольные устройства имеют ряд важных отличий, и понимать их необходимо, если вы хотите, чтобы визуализация данных достигала желаемого эффекта на обеих платформах.

Мобильное устройство:

- имеет несколько ориентаций экрана;
- размеры экрана невелики и изменяются в узком диапазоне;
- отсутствует указательное устройство – взаимодействие достигается жестами, а состояния «наведен курсор» нет;
- часто рассчитано на непостоянную доступность данных;
- обладает значительно меньшей вычислительной мощностью, по сравнению с ноутбуком;
- клавиатура присутствует на экране, только когда используется; для навигации она не применяется.

Настольный компьютер:

- как правило, имеет только одну ориентацию экрана;
- размеры экрана велики и изменяются в широком диапазоне (часто к компьютеру подключены огромные дисплеи);
- для взаимодействия применяется указательное устройство (и клавиатура), доступно состояние «наведен курсор»;

- обычно на доступность данных можно надежно полагаться;
- вычислительная мощность гораздо выше, чем у телефонов и планшетов;
- клавиатуру можно использовать, не изменяя изображения на экране, в том числе для навигации.

У меня здесь нет места для полного курса по отзывчивому дизайну, но несколько важных принципов следует иметь в виду, приступая к работе.

Столбцы для настольных, строки для мобильных

На определенном уровне безопасно предположить, что большинство пользователей мобильных устройств будет просматривать вашу работу в книжном режиме. Это значит, что, по сути дела, в вашем распоряжении один столбец, и каждый элемент в этом столбце – это полная строка, перпендикулярная направлению столбца. Абзацы всегда должны течь вертикально (а не горизонтально в горизонтальных столбцах). Как и в демонстрации вероятности выигрыша в лотерею на сайте Wall Street Journal, используйте бесконечную высоту экрана для прокрутки.

С другой стороны, на настольном компьютере один абзац, занимающий всю ширину экрана, очень трудно читать, поскольку приходиться много бегать глазами. Лучше организовать несколько столбцов, т. к. при этом не только рачительнее используется пространство в горизонтальном направлении, но и читать удобнее. В частности, элементы формы часто выглядят лучше, когда сгруппированы в несколько столбцов.

Есть и хорошая новость: flex-контейнеры позволяют легко менять ориентацию группы элементов, если только не стоит задача поддержать старые версии Internet Explorer. Если вы еще не пользуетесь flex-контейнерами, приступайте, не откладывая; это заметно облегчит вам жизнь (по крайней мере, после того как вы поймете, как ими пользоваться).

Хотите выучить flex-контейнеры, играючи? Попробуйте поиграть в Flexbox Froggy (http://flexboxfroggy.com/), и вы научитесь позиционировать элементы с применением flex-контейнеров и лягушек. Все еще обескуражены flex-контейнерами? Вы не одиноки. Попробуйте Flexbox Grid, где для создания отзывчивых сеток с помощью flex-контейнеров используются классы наподобие применяемых в Bootstrap от Twitter (http://flexboxgrid.com/).

Не злоупотребляйте анимацией на мобильных

Анимация на мобильных устройствах – дело сложное не только потому, что средства работы с графикой менее мощные, но и потому, что события прокрутки традиционно плохо уживаются с хронометражем в JavaScript. Если вы не хотите полностью отказаться от анимации на мобильных устройствах, то постарайтесь обойтись только CSS-переходами, потому что они более производительны, чем обход значений свойств средствами JavaScript. Если сомневаетесь, отключайте анимацию вообще.

Помните, что схожие элементы по-разному ведут себя на разных платформах

Переключатели, ползунки и флажки имеются и на мобильных, и на настольных устройствах, но одни использовать на мобильных устройствах проще, другие – сложнее. Вообще говоря, браузеры для мобильных устройств рисуют эти элементы настолько мелко, что пользоваться ими неудобно. Если есть возможность (например, в случае раскрывающихся списков `select`), растягивайте элементы формы на всю ширину мобильного устройства, а для флажков и переключателей применяйте свойство `for` элемента `<label>`, чтобы сделать элемент восприимчивым к касанию, и располагайте эти элементы по горизонтали.

Избегайте таинственной навигации

У указательных устройств есть возможность использовать состояние «наведен курсор», чтобы показать какую-то информацию (например, метку на кнопке). Для настольных компьютеров это дает шанс реализовать минималистский дизайн, но в случае мобильных устройств является чудовищным антипаттерном, который получил название «таинственная навигация» (mystery meat navigation[1]). Наводя курсор на кнопку, мы не обязуемся нажать ее. Однако у мобильных устройств нет состояния «наведен курсор», поэтому, чтобы понять, для чего предназначена кнопка, пользователь вынужден вступить с ней во взаимодействие. А учитывая, каким медленным бывает чтение с экрана мобильного устройства, пользователь дважды подумает, прежде чем совершить действие, которое может привести к перезагрузке страницы.

---

[1] Mystery meat – мясо неизвестного происхождения. – *Прим. перев.*

Решение простое – если не можете похвастаться не допускающими двояких толкований значками, снабжайте метками все кнопки в интерфейсе для мобильного устройства.

Великолепные значки для мобильных устройств можно найти на сайте **Font Awesome** (http://fortawesome.github.io) и среди Material Icons от Google (https://design.google.com/icons).

Остерегайтесь прокрутки

Распространенная идиома взаимодействия с пользователем на настольных компьютерах (завоевавшая популярность после проекта *Snowfall* газеты *New York Times*, где впервые появился формат лонгрид) – привязать анимацию к событию прокрутки браузера. На мобильных устройствах это часто сопряжено с опасностью, поскольку прокрутка инициирует перерисовку, которая потребляет много памяти и зачастую блокирует выполнение JavaScript. И хотя последние версии мобильных операционных систем справляются с этим лучше прежних (это я в основном о тебе, iOS), в общем случае небезопасно предполагать, что основанная на прокрутке анимация будет работать нормально, а не выглядеть медленной и противной. Еще раз повторю, что в новых устройствах ситуация улучшилась, но все равно разумнее привязывать анимацию к событиям касания (скажем, касания кнопки).

Резюме

Надеюсь, к этому моменту вы хорошо понимаете характер работы, в которой хотели бы применить D3 для визуализации данных. Мы рассмотрели несколько примеров и сформулировали базовые правила создания высококачественной визуализации данных, которая не только доставит аудитории информацию, но и будет при этом иметь привлекательный вид. Мы также обсудили, как обеспечить функционирование плодов вашего труда на мобильных устройствах.

Как ни прискорбно, настало время прощаться. Я всей душой надеюсь, что вы узнали кое-что полезное и с удовольствием прочитали все 10 глав; написание книги – это поиск хрупкого равновесия между грубым и скорым сообщением фактов и увлекательной формой изложения. Как бы то ни было, если вам удалось прочесть книгу от начала до конца, снимаю перед вами шляпу, поскольку мы охватили ряд умопомрачительных технологий и подходов к разработке программного обеспечения.

Мы начали с самых основ – DOM и CSS. Затем мы перешли к SVG и научились создавать весьма симпатичную векторную графику для веба, после чего занялись настоящим делом, взяв на вооружение D3, – помните главу о макетах, где мы насоздавали кучу диаграмм? Мы проделали интересные вещи с Node.js и Canvas, построили полезные карты и даже развернули проект в среде Heroku, поскольку стали настоящими ниндзя в веб-разработке! Ийя-я! Ай да мы!

И наконец, мы остановились на одном из важнейших аспектов технологии – как обрести уверенность в результатах своей работы. Не важно, прибегаете вы к помощи других людей для проверки или пишете солидный комплект тестов, визуализация данных – ремесло, значимость которого будет возрастать по мере того, как данных в мире становится все больше, однако оно требует точности. Не страшно, если вы где-то допустите ошибку, но приложите все силы, чтобы до этого не дошло.

Мы живем в поразительную эпоху веба, когда ограничения, удерживавшие разработчиков от создания истинных бриллиантов, постепенно снимаются благодаря появлению все более изощренных подходов, когда принимаются более смелые решения, поддержанные улучшенными инструментальными средствами. Все это движется вперед с *бешеной* скоростью, но пусть это вас не останавливает: пробуйте новое, экспериментируйте с кодом, находящимся на самом переднем крае. Нет другой области компьютерных технологий, где можно создавать такие *универсальные* вещи так *моментально*. Хотя веб-разработка может оказаться нелегким делом (это я о тебе, отзывчивый дизайн!), никогда еще ни один набор технологий не имел таких критически важных последствий для способов потребления информации миром. Прочитав эту книгу, вы получили в свое распоряжение обширный арсенал орудий, готовых к использованию. Понятно, что о некоторых из них еще нужно почитать (особенно о тех, по которым я прошелся лишь мимоходом: Node, Canvas и TypeScript), чтобы научиться применять эффективно, но, надеюсь, я снабдил вас кое-чем, что пригодится в этом путешествии и поможет приспособить все это добро к делу.

Как минимум, у вас есть все необходимое, чтобы приступить к визуализации своих данных с помощью D3 и разделить плоды своего труда со всем миром посредством *магии Интернета*.

Наконец, если вы разочарованы и нуждаетесь в помощи или месте, где можно излить свое раздражение тем, что вы уже два дня ваяете эту *хрень*, а она все никак не хочет работать, то приходите к нам в канал D3 Slack; там обычно всегда найдется кто-нибудь, готовый помочь.