Содержание

Отзывы о первом издании 13

От автора 14

Благодарности 16

Об этой книге 18

Кому адресована книга 18
Структура книги 18
Условные обозначения, принятые в книге 21
Загружаемый исходный код 21
Как связаться с авторами 21

Об авторах 22

Об иллюстрации на обложке книги 24

От издательства 25

Часть I. Разминка 27

Глава 1. JavaScript повсюду 29

1.1. Общее представление о языке JavaScript 30
1.1.1. Дальнейшее развитие JavaScript 32
1.1.2. Транспиляторы, обеспечивающие доступ к будущему JavaScript сегодня 33
1.2. Общее представление о браузере 33
1.3. Нормы передовой практики 35
1.3.1. Отладка 36
1.3.2. Тестирование 36
1.3.3. Анализ производительности 37
1.4. Усиление переносимости приобретенных навыков 38
Резюме 39

Глава 2. Создание страницы в динамическом режиме 41

2.1. Общее представление о жизненном цикле веб-приложения 42
2.2. Стадия создания страницы 45
2.2.1. Синтаксический анализ кода HTML и построение модели DOM 46
2.2.2. Выполнение кода JavaScript 48
2.3. Обработка событий 52
2.3.1. Общее представление об обработке событий 53

Резюме 59
Упражнения 59

Часть II. Представление о функциях 61

Глава 3. Функции высшего порядка для начинающих: определения и аргументы 63

3.1. Главное отличие JavaScript как языка функционального программирования 64
3.1.1. Функции в качестве объектов высшего порядка 65
3.1.2. Функции обратного вызова 66
3.2. Особенности применения функций в качестве объектов 70
3.2.1. Сохранение функций 71
3.2.2. Самозапоминающиеся функции 73
3.3. Определение функций 75
3.3.1. Объявления функций и функциональные выражения 77
3.3.2. Стрелочные функции 82
3.4. Аргументы и параметры функций 84
3.4.1. Оставшиеся параметры 86
3.4.2. Стандартные параметры 88
Резюме 91
Упражнения 92

Глава 4. Функции для ученика мастера: представление об их вызове 95

4.1. Использование неявных параметров функции 96
4.1.1. Параметр `arguments` 96
4.1.2. Параметр `this`, представляющий контекст функции 101
4.2. Вызов функций 102
4.2.1. Вызов как функции 103
4.2.2. Вызов как метода 104
4.2.3. Вызов как конструктора 107
4.2.4. Вызов через методы `apply()` и `call()` 113
4.3. Разрешение затруднений, связанных с контекстами функций 120
4.3.1. Обращение с контекстами функций с помощью стрелочных функций 120
4.3.2. Применение метода `bind()` 124
Резюме 125
Упражнения 126

Глава 5. Функции для мастера: замыкания и области видимости 129

5.1. Общее представление о замыканиях 130
5.2. Применение замыканий на практике 134
5.2.1. Имитация закрытых переменных 134
5.2.2. Применение замыканий при обратных вызовах 136

5.3. Отслеживание выполнения кода с помощью контекстов выполнения 139
5.4. Отслеживание идентификаторов с помощью лексических сред 142
5.4.1. Вложение кода 143
5.4.2. Вложение кода и лексические среды 144
5.5. Общее представление о типах переменных в JavaScript 147
5.5.1. Изменяемость переменных 147
5.5.2. Ключевые слова для определения переменных и лексические среды 150
5.5.3. Регистрация идентификаторов в лексических средах 154
5.6. Исследование принципа действия замыканий 159
5.6.1. Еще раз об имитации закрытых переменных с помощью замыканий 159
5.6.2. Разъяснение по поводу закрытых переменных 164
5.6.3. Еще раз о замыканиях и обратных вызовах 165
Резюме 167
Упражнения 168

Глава 6. Функции на перспективу: генераторы и обещания 171

6.1. Достижение изящности асинхронного кода с помощью генераторов и обещаний 172
6.2. Использование функций-генераторов 174
6.2.1. Управление генератором с помощью итератора 176
6.2.2. Применение генераторов 179
6.2.3. Обмен данными с генератором 183
6.2.4. Исследование внутреннего механизма действия генераторов 186
6.3. Работа с обещаниями 193
6.3.1. Затруднения, связанные с простыми обратными вызовами 195
6.3.2. Углубленное исследование обещаний 197
6.3.3. Отклонение обещаний 200
6.3.4. Создание первого настоящего обещания 202
6.3.5. Связывание обещаний в цепочку 204
6.3.6. Ожидание ряда обещаний 205
6.4. Сочетание генераторов и обещаний 207
6.4.1. Асинхронные функции в перспективе 211
Резюме 212
Упражнения 213

Часть III. Исследование объектов и упрочение кода 215

Глава 7. Объектная ориентация с помощью прототипов 217

7.1. Общее представление о прототипах 218
7.2. Создание объектов и прототипы 221
7.2.1. Свойства экземпляров 224
7.2.2. Побочные эффекты динамического характера JavaScript 227

7.2.3. Типизация объектов через конструкторы 230
7.3. Достижение наследования 232
7.3.1. Трудности переопределения свойства `constructor` 236
7.3.2. Операция `instanceof` 240
7.4. Применение “классов” в стандарте ES6 языка JavaScript 242
7.4.1. Применение ключевого слова `class` 243
7.4.2. Реализация наследования 246
Резюме 249
Упражнения 250

Глава 8. Управление доступом к объектам 253

8.1. Управление доступом к свойствам объектов с помощью методов получения и установки 254
8.1.1. Определение методов получения и установки 256
8.1.2. Применение методов получения и установки для проверки достоверности значений свойств 262
8.1.3. Применение методов получения и установки для определения вычисляемых свойств 264
8.2. Применение прокси-объектов для управления доступом 266
8.2.1. Применение прокси-объектов для протоколирования 270
8.2.2. Применение прокси-объектов для измерения производительности 272
8.2.3. Применение прокси-объектов для автоматического заполнения свойств 274
8.2.4. Применение прокси-объектов для реализации отрицательных индексов массивов 275
8.2.5. Проблемы с производительностью при использовании прокси-объектов 278
Резюме 279
Упражнения 280

Глава 9. Работа с коллекциями 283

9.1. Массивы 284
9.1.1. Создание массивов 284
9.1.2. Добавление и удаление элементов с обоих концов массива 287
9.1.3. Добавление и удаление элементов в любом месте массива 289
9.1.4. Наиболее употребительные операции над массивами 291
9.1.5. Повторное использование встроенных методов обработки массивов 303
9.2. Отображения 305
9.2.1. Объекты непригодны в качестве отображений 306
9.2.2. Создание первого отображения 309
9.2.3. Перебор элементов отображений 313

9.3. Множества 314
9.3.1. Создание первого множества 315
9.3.2. Объединение множеств 317
9.3.3. Пересечение множеств 318
9.3.4. Разность множеств 319
Резюме 319
Упражнения 320

Глава 10. Овладение регулярными выражениями 323

10.1. Достоинства регулярных выражений 324
10.2. Основные сведения о регулярных выражениях 325
10.2.1. Назначение регулярных выражений 326
10.2.2. Члены и операции 328
10.3. Компиляция регулярных выражений 333
10.4. Фиксация совпадающих частей 336
10.4.1. Выполнение простых фиксаций 336
10.4.2. Проверка на совпадение с помощью глобальных регулярных выражений 337
10.4.3. Ссылки на фиксации 339
10.4.4. Нефиксируемые группы 340
10.5. Замена текста с помощью функций 341
10.6. Решение типичных задач с помощью регулярных выражений 344
10.6.1. Учет символов перехода на новую строку 344
10.6.2. Сопоставление с символами Юникода 346
10.6.3. Сопоставление с экранированными символами 346
Резюме 347
Упражнения 348

Глава 11. Методики модуляризации кода 351

11.1. Модуляризация кода JavaScript до появления стандарта ES6 352
11.1.1. Определение модулей с помощью объектов, замыканий и немедленно вызываемых функций 353
11.1.2. Модуляризация приложений на JavaScript по стандартам AMD и CommonJS 360
11.2. Модули по стандарту ES6 364
11.2.1. Функциональные возможности экспорта и импорта 365
Резюме 371
Упражнения 372

Часть IV. Исследование браузеров 375

Глава 12. Работа с моделью DOM 377

12.1. Вставка HTML-кода в объектную модель документа 378
12.1.1. Преобразование из формата HTML в структуру DOM 379

12.1.2. Вставка элементов разметки в документ 384
12.2. Атрибуты и свойства модели DOM 386
12.3. Трудности обращения с атрибутами стилевого оформления 388
12.3.1. Местонахождение стилей 389
12.3.2. Именование свойств стилевого оформления 391
12.3.3. Извлечение вычисленных стилей 393
12.3.4. Преобразование значений, указываемых в пикселях 397
12.3.5. Указание размеров по высоте и ширине 398
12.4. Сведение к минимуму перегрузки верстки 403
Резюме 406
Упражнения 407

Глава 13. Особенности обработки событий 409

13.1. Углубленное исследование цикла ожидания событий 410
13.1.1. Пример только с очередью макрозадач 414
13.1.2. Пример с обеими очередями макро- и микрозадач 419
13.2. Овладение таймерами: тайм-ауты и интервалы времени 424
13.2.1. Запуск обработчиков таймера в цикле ожидания событий 425
13.2.2. Преодоление трудностей затратной обработки вычислений 432
13.3. Обработка событий 436
13.3.1. Распространение событий по модели DOM 437
13.3.2. Специальные события 444
Резюме 448
Упражнения 449

Глава 14. Стратегии разработки кросс-браузерного кода 451

14.1. Соображения по поводу кросс-браузерной разработки 452
14.2. Пять самых насущных задач разработки 455
14.2.1. Программные ошибки и отличия в браузерах 456
14.2.2. Преодоление программных ошибок в браузерах 456
14.2.3. Внешний код и разметка 458
14.2.4. Регрессии 463
14.3. Стратегии реализации 465
14.3.1. Надежное устранение ошибок в кросс-браузерном коде 465
14.3.2. Обнаружение функциональных средств и полифиллы 467
14.3.3. Области непроверяемых ошибок в браузерах 469
14.4. Сокращение допущений 472
Резюме 473
Упражнения 474

Часть V. Приложения 475

Приложение А. Дополнительные средства стандарта ES6 477

Шаблонные литералы 477

Деструктурирование 479
Усовершенствованные литералы объектов 480

Приложение Б. Средства тестирования и отладки 483

Инструментальные средства для разработки и отладки веб-приложений 484
Firebug 484
Firefox Developer Tools 485
F12 Developer Tools 486
WebKit Inspector 487
Chrome DevTools 488
Отладка кода на JavaScript 488
Протоколирование 489
Точки останова 490
Заход в функцию 491
Создание тестов 494
Основы организации среды тестирования 497
Наиболее распространенные среды тестирования 499
QUnit 500
Jasmine 501
Измерение покрытия кода 502

Приложение В. Ответы на упражнения 505

Глава 2. Создание страницы в динамическом режиме 505
Глава 3. Функции высшего порядка для начинающих: определения и аргументы 506
Глава 4. Функции для ученика мастера: представление об их вызове 507
Глава 5. Функции для мастера: замыкания и области видимости 510
Глава 6. Функции на перспективу: генераторы и обещания 512
Глава 7. Объектная ориентация с помощью прототипов 514
Глава 8. Управление доступом к объектам 517
Глава 9. Работа с коллекциями 520
Глава 10. Овладение регулярными выражениями 522
Глава 11. Методики модуляризации кода 525
Глава 12. Работа с моделью DOM 527
Глава 13. Особенности обработки событий 528
Глава 14. Стратегии разработки кросс-браузерного кода 530

Предметный указатель 533

11 Методики модуляризации кода

В этой главе...

- Применение проектного шаблона `Модуль`
- Написание модульного кода согласно текущим стандартам AMD и CommonJS
- Работа с модулями в стандарте ES6

До сих пор мы исследовали такие основные примитивы JavaScript, как функции, объекты, коллекции и регулярные выражения. Но в арсенале средств разработчика имеются и другие инструментальные средства для решения текущих задач написания прикладного кода на JavaScript. По мере расширения веб-приложений возникает ряд других задач, связанных со структурированием прикладного кода и его управлением. Как было неоднократно доказано, крупные монолитные кодовые базы намного труднее понять и сопровождать, чем более мелкие и хорошо организованные. Поэтому вполне естественно, что единственный способ усовершенствовать структуру и организацию прикладных программ состоит в том, чтобы разделить их на более мелкие, относительно слабо связанные части, называемые *модулями*.

Модули являются более крупными единицами организации прикладного кода, чем объекты и функции. Они позволяют разделять прикладные программы на связанные вместе блоки. При создании модулей следует стремиться образовать согласованные абстракции и инкапсулировать подробности реализации. Благодаря этому упрощается осмысление прикладной программы, поскольку, используя функциональные возможности модулей, можно не беспокоиться о малозначащих подробностях. Кроме того, наличие модулей означает возмож-

ность без особого труда повторно использовать функциональные средства в разных частях одной прикладной программы и даже разных прикладных программ, значительно ускоряя процесс разработки.

Как пояснялось ранее, в JavaScript повсеместно применяются глобальные переменные. Всякий раз, когда переменная определяется в главном коде, она автоматически становится глобальной и может быть доступна в любой другой части прикладного кода. В небольших программах это обычно не вызывает особых затруднений, но по мере расширения прикладных программ и включения в них стороннего кода заметно увеличивается вероятность конфликта имен. В большинстве других языков программирования подобное затруднение разрешается с помощью пространств имен (в C++ и C#) или пакетов (в Java). Они позволяют изолировать все используемые в них имена и назначить им другое имя. Благодаря этому значительно снижается вероятность появления конфликта имен.

До появления стандарта ES6 в языке JavaScript отсутствовало высокоуровневое встроенное средство, позволяющее группировать связанные вместе переменные в модуль, пространство имен или пакет. Поэтому в качестве выхода из подобного затруднительного положения программирующие на JavaScript выработали усовершенствованные методики модуляризации, в которых выгодно используются преимущества существующих языковых конструкций JavaScript, в том числе объектов, немедленно вызываемых функций и замыканий. Такие методики будут исследованы в этой главе.

Правда, только время покажет, как долго придется пользоваться подобными обходными приемами, поскольку в стандарт ES6 наконец-то были внедрены собственные модули. Но, к сожалению, браузеры не поспевают за этими нововведениями, и поэтому мы выясним, каким образом модули должны действовать в стандарте ES6, несмотря на то, что проверить их пока еще нельзя из-за отсутствия конкретной реализации в отдельных браузерах.

Знаете ли вы?

Каким из имеющихся механизмов можно воспользоваться для приближенного представления модулей в коде JavaScript до появления стандарта ES6?

Чем отличаются стандарты определения модулей AMD и CommonJS?

Какие операторы потребуются в коде, начиная со стандарта ES6, чтобы вызвать в модуле `guineaPig` функцию `tryThisOut()` из модуля `test`?

Итак, начнем с рассмотрения методик модуляризации, которыми можно пользоваться в настоящее время.

11.1. Модуляризация кода JavaScript до появления стандарта ES6

До появления стандарта ES6 в языке JavaScript допускались лишь следующие две области видимости: глобальная и локальная (область видимости функции).

В языке не существовало промежуточной области видимости (например, пространства имен или модуля), которая бы позволяла сгруппировать определенные функциональные возможности. Чтобы написать модульный код, разработчики веб-приложений на JavaScript были вынуждены творчески подходить к применению имеющихся языковых средств JavaScript.

Выбирая подходящие языковые средства, необходимо иметь в виду, что каждая модульная система должна быть способна как минимум на следующее.

- *Определить интерфейс*, через который можно получить доступ к функциональным возможностям, которые предоставляет модуль.
- *Скрыть внутренние подробности реализации модуля*, чтобы не обременять ими излишне пользователей модуля. Скрывая внутренние подробности реализации модуля, можно также защитить их от вмешательства извне, предотвратив тем самым ненужные модификации, способные породить всевозможные побочные эффекты и программные ошибки.

В этом разделе мы сначала покажем, как создавать модули, пользуясь стандартными языковыми средствами JavaScript, рассмотренными ранее в данной книге, в том числе объектами, замыканиями и немедленно вызываемыми функциями. Продолжая рассмотрение особенностей модуляризации, мы исследуем два наиболее распространенных стандарта определения модулей в JavaScript: AMD (Asynchronous Module Definition — асинхронное определение модуля) и CommonJS, которые построены на немного разных принципах. По ходу изложения материала этого раздела вы узнаете, как определять модули по этим стандартам и каковы их достоинства и недостатки. Но начнем мы с того, для чего была подготовлена почва в предыдущих главах.

11.1.1. Определение модулей с помощью объектов, замыканий и немедленно вызываемых функций

Итак, вернемся к перечисленным выше минимальным требованиям к модульной системе, которые состоят в том, чтобы скрыть внутренние подробности реализации и определить интерфейсы модулей. И теперь рассмотрим те языковые средства, которыми можно выгодно воспользоваться для реализации этих требований.

- **Сокрытие внутренних подробностей реализации модулей.** Как известно, при вызове функции в JavaScript образуется новая область видимости, в которой можно определить переменные, доступные только в теле данной функции. Следовательно, одна из возможностей скрыть внутренние подробности реализации модуля заключается в том, чтобы воспользоваться функциями в качестве модулей. Благодаря этому все переменные функции становятся внутренними переменными модуля, скрытыми от внешнего мира.

- **Определение интерфейсов модулей.** Реализация внутренних подробностей функционирования модулей через переменные функций означает, что эти переменные доступны только в самом модуле. Но если модули предполагается использовать в другом коде, то придется определить ясный интерфейс, через который можно получить доступ к функциональным возможностям, предоставляемым модулем. Чтобы добиться этого, можно, например, воспользоваться преимуществами объектов и замыканий. Идея состоит в том, чтобы из функции модуля возвращался объект, представляющий открытый интерфейс этого модуля. У этого объекта должны быть методы, которые через замыкания сохраняют активными внутренние переменные модуля — даже после того, как функция модуля завершит свое выполнение.

Описав в общих чертах, каким образом модули реализуются в JavaScript, перейдем к постепенному рассмотрению способов реализации требований к модулям, начав с применения функций для сокрытия внутренних подробностей реализации модулей.

Функции в качестве модулей

При вызове функции образуется новая область видимости, в которой определяются переменные, недоступные за пределами текущей функции. В качестве примера рассмотрим следующий фрагмент кода, в котором подсчитывается количество щелчков левой кнопкой мыши на веб-странице:

```
(function countClicks(){
  let numClicks = 0;
  document.addEventListener("click", () => {
    alert( ++numClicks );
  });
})();
```

Определить локальную переменную для хранения подсчета количества щелчков (← `let numClicks = 0;`)

Всякий раз, когда пользователь щелкает кнопкой мыши на веб-странице, инкрементируется счетчик и сообщается его текущее значение (← `alert( ++numClicks );`)

В приведенном выше фрагменте кода определяется функция `countClicks()`, где создается переменная `numClicks` и регистрируется обработчик событий от щелчка кнопкой мыши на всем документе. Всякий раз, когда на нем выполняется щелчок кнопкой мыши, инкрементируется значение переменной `numClicks`, а результат отображается для пользователя в окне оповещения. В данном коде обращает на себя внимание следующее.

- Внутренняя переменная `numClicks` функции `countClicks()` поддерживается активной через замыкание функции-обработчика событий от щелчка кнопкой мыши. На эту переменную можно ссылаться только в самом обработчике событий и *нигде больше*! Переменная `numClicks` изолирована от кода за пределами функции `countClicks()`. В то же время глобальное пространство имен прикладной программы не засорено переменной, которая вряд ли представляет интерес для остального кода.
- Функция `countClicks()` вызывается только в одном конкретном месте, и поэтому она определена как немедленно вызываемая (подробнее об

этом см. в главе 3), а не как функция, определяемая и далее вызываемая с помощью отдельного оператора.

Текущее состояние прикладного кода можно проанализировать в отношении того, каким образом внутренняя переменная функции (или модуля) поддерживается активной через замыкания (рис. 11.1).

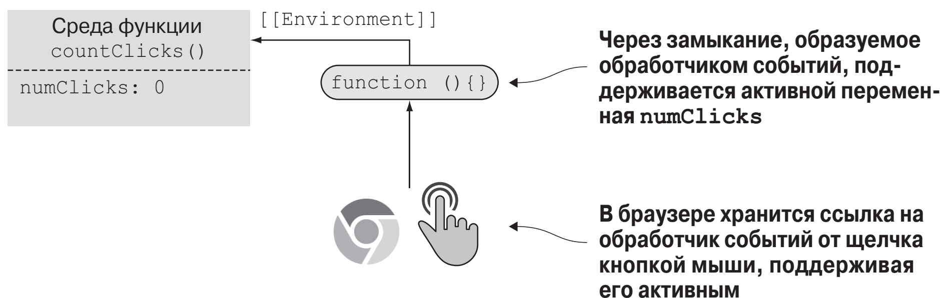

Рис. 11.1. Локальная переменная **numClicks** поддерживается активной через замыкания в обработчике событий от щелчка кнопкой мыши

А теперь, когда стало понятно, как скрывать подробности реализации модуля и поддерживать их активными столько, сколько потребуется, перейдем к рассмотрению второго минимального требования к модулям: определения их интерфейсов.

Проектный шаблон Модуль: расширение функций как модулей объектами как интерфейсами

Как правило, интерфейс модуля состоит из ряда переменных и функций, предоставляемых модулем внешнему миру. Чтобы создать такой интерфейс, проще всего воспользоваться обычным объектом в JavaScript.

В качестве примера рассмотрим создание интерфейса для упомянутого выше модуля, в котором подсчитывается количество щелчков кнопкой мыши на веб-странице, как демонстрируется в коде из листинга 11.1.

Листинг 11.1. Проектный шаблон Модуль

```
const MouseCounterModule = function() {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
```

Создать глобальную переменную модуля и присвоить ей результат выполнения немедленно вызываемой функции

Создать “закрытую” переменную модуля

Создать “закрытую” функцию модуля

Возвратить объект, представляющий интерфейс модуля. “Закрытые” переменные и функции могут быть доступны через замыкания

```
}();
```

К свойствам можно обратиться из вне через их интерфейс доступа

```
assert(typeof MouseCounterModule.countClicks === "function",
       "We can access module functionality");
assert(typeof MouseCounterModule.numClicks === "undefined"
       && typeof MouseCounterModule.handleClick === "undefined" ,
       "We cannot access internal module details");
```

При этом внутренние элементы модуля недоступны

Сначала в данном примере кода модуль реализуется с помощью немедленно вызываемой функции. В теле этой функции определяются подробности реализации модуля: одна локальная переменная `numClicks` и одна локальная функция `handleClick()`, причем они доступны только в самом модуле. Затем в данном коде создается и немедленно возвращается объект, который послужит в качестве "открытого интерфейса" модуля. Этот интерфейс содержит метод `countClicks()`, который может быть вызван за пределами модуля для доступа к его функциональным возможностям.

В то же время внутренние элементы модуля поддерживаются активными через замыкания, образуемые интерфейсом модуля, поскольку этот интерфейс становится доступным извне. Так, в методе `countClicks()` интерфейса модуля поддерживаются активными внутренние переменные модуля `numClicks` и `handleClick`, как показано на рис. 11.2.

```
const MouseCounterModule = function(){
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();
```

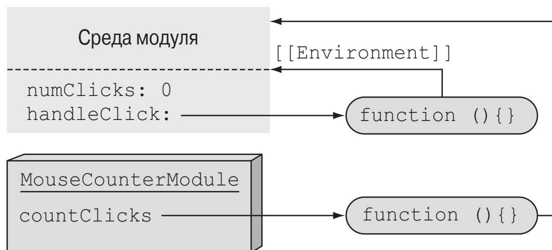

Рис. 11.2. Доступ к интерфейсу модуля через возвращаемый объект. Внутренняя реализация модуля ("закрытые" переменные и функции) поддерживается активной через замыкания, образуемые открытыми методами интерфейса

И, наконец, объект, представляющий интерфейс модуля, возвращаемого немедленно вызываемой функцией, сохраняется в переменной `MouseCounter Module`. И через нее можно легко обратиться к функциональным возможностям модуля, написав следующую строку кода:

```
MouseCounterModule.countClicks()
```

Вот, собственно, и все, что требуется знать о требованиях к реализации модуля. Воспользовавшись преимуществами немедленно вызываемых функций, можно скрыть некоторые подробности реализации модуля. А вводя объекты и замыкания в обычный объект, можно указать интерфейс модуля, раскрывающий функциональные возможности, предоставляемые модулем внешнему миру.

Такой порядок применения немедленно вызываемых функций, объектов и замыканий для создания модулей в JavaScript и составляет то, что называется проектным шаблоном `Модуль`. Этот шаблон был популяризирован Дугласом Крокфордом (Douglas Crockford) и стал одним из первых массово распространенных способов модуляризации кода JavaScript.

Таким образом, получив возможность определять модули, было бы неплохо распределить их код по нескольким файлам, чтобы можно было легче управлять ими, или же определять дополнительные функциональные возможности в существующих модулях, не видоизменяя их исходный код. Выясним далее, как это реализовать непосредственно в коде.

Расширение модулей

Попробуем теперь расширить модуль `MouseCounterModule` из предыдущего примера так, чтобы он мог дополнительно подсчитывать количество прокруток мышью. При этом первоначальный код данного модуля не должен видоизменяться, как показано в коде из листинга 11.2.

Листинг 11.2. Расширение модулей

```
const MouseCounterModule = function(){   ← Первоначальный модуль MouseCounterModule
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();

(function(module) {   ← Немедленный вызов функции, которой в качестве аргумента передается модуль, требующий расширения
  let numScrolls = 0;
  const handleScroll = () => {
    alert(++numScrolls);
  }
```

Определить новые закрытые переменные и функции

```
  module.countScrolls = () => {
    document.addEventListener("wheel", handleScroll);
  };
})(MouseCounterModule);

assert(typeof MouseCounterModule.countClicks === "function",
       "We can access initial module functionality");
assert(typeof MouseCounterModule.countScrolls === "function",
       "We can access augmented module functionality");
```

Расширить интерфейс модуля

Передать модуль в качестве аргумента

При расширении модуля обычно используется процедура, подобная созданию нового модуля. С этой целью организуется немедленный вызов функции, но на сей раз в качестве аргумента ей передается расширяемый модуль:

```
(function(module){
  ...
  return module;
})(MouseCounterModule);
```

В теле немедленно вызываемой функции создаются все закрытые переменные и функции, требующиеся для ее выполнения. В данном случае определяются закрытая переменная и локальная функция для подсчета и сообщения о количестве прокруток:

```
let numScrolls = 0;
const handleScroll = () => {
  alert(++numScrolls);
}
```

И, наконец, модуль, доступный через параметр `module` немедленно вызываемой функции, расширяется подобно любому другому объекту следующим образом:

```
module.countScrolls = ()=> {
  document.addEventListener("wheel", handleScroll);
};
```

После выполнения этой простой операции модуль `MouseCounterModule` позволяет также подсчитывать количество прокруток документа мышью. В открытом интерфейсе этого модуля теперь имеются два метода, а самим модулем можно воспользоваться следующими способами:

```
MouseCounterModule.countClicks();
MouseCounterModule.countScrolls();
```

Метод, первоначально ставший частью интерфейса модуля

Новый метод, введенный путем расширения модуля

Как упоминалось ранее, расширение модуля происходит через немедленно вызываемую функцию аналогично созданию нового модуля. Такое расширение имеет ряд интересных побочных эффектов, связанных с замыканиями, поэтому проанализируем подробнее состояние прикладного кода после расширения модуля, как показано на рис. 11.3.

```
const MouseCounterModule = function() {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };
  return {
    countClicks: () => {
      document.addEventListener("click", handleClick);
    }
  };
}();

(function (module){
  let numScrolls = 0;
  const handleScroll = () => {
    alert(++numScrolls);
  }

  module.countScrolls = () => {
    document.addEventListener("wheel", handleClick);
  };
})(MouseCounterModule);
```

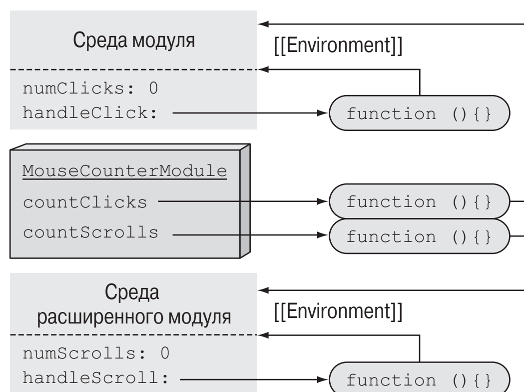

Рис. 11.3. При расширении модуля его внешний интерфейс дополняется новыми функциональными возможностями, как правило, посредством передачи модуля другой немедленно вызываемой функции. В данном примере модуль **MouseCounterModule** дополняется функцией **countScrolls()**. Обратите внимание на то, что две отдельные функции определяются в разных средах и не могут иметь доступ к внутренним переменным друг друга

Если внимательно проанализировать рис. 11.3, то можно также обнаружить одно из ограничений проектного шаблона `Модуль`, которое заключается в невозможности обращаться к закрытым переменным модуля из его расширений. Например, функция `countClicks()` поддерживает замыкание вокруг переменных `numClicks` и `handleClick`, и эти закрытые внутренние элементы модуля могут быть доступны только внутри данной функции.

К сожалению, расширение модуля функцией `countScrolls()` произведено в отдельной области видимости с совершенно новым набором закрытых переменных `numScrolls` и `handleScroll`. Функция `countScrolls()` образует замыкание только вокруг переменных `numScrolls` и `handleScroll`, поэтому ей недоступны переменные `numClicks` и `handleClick`.

На заметку

Если расширения модуля производятся через отдельные немедленно вызываемые функции, закрытые внутренние элементы модуля нельзя сделать общедоступными, поскольку каждый вызов функции приводит к созданию новой области видимости. Несмотря на это ограничение, проектный шаблон `Модуль` все же позволяет поддерживать модульность приложений на JavaScript.

Следует заметить, что в проектном шаблоне `Модуль` сами модули являются объектами как и любые другие элементы, а следовательно, их можно расширять как угодно. Например, функциональные возможности могут быть введены путем расширения объекта модуля новыми свойствами:

```
MouseCounterModule.newMethod = ()=> {...}
```

По тому же самому принципу можно без особого труда создать и подмодули:

```
MouseCounterModule.newSubmodule = () => {
  return {...};
}();
```

Однако следует иметь в виду, что все эти способы страдают одним и тем же главным недостатком, присущим проектному шаблону `Модуль`. В частности, всем последующим расширениям модуля будут недоступны определенные ранее внутренние элементы модуля.

К сожалению, у проектного шаблона `Модуль` имеются и другие недостатки. Приступая к написанию модульных приложений, следует иметь в виду, что функциональные возможности одних модулей зачастую будут зависеть от других модулей. К сожалению, проектный шаблон `Модуль` не позволяет управлять этими зависимостями. Поэтому разработчикам приходится самим упорядочивать зависимости, чтобы предоставить все необходимое для выполнения модульного кода. И если в небольших и средних приложениях это не вызывает особых трудностей, то в крупных приложениях со многими взаимозависимыми модулями в связи с этим могут возникнуть серьезные затруднения. Для разрешения подобных затруднений были созданы два конкурирующих стандарта: AMD (Asynchronous Module Definition) и CommonJS.

11.1.2. Модуляризация приложений на JavaScript по стандартам AMD и CommonJS

AMD и CommonJS — два конкурирующих стандарта определения модулей в JavaScript. Помимо некоторых синтаксических и принципиальных отличий, главное их отличие заключается в том, что стандарт AMD разработан явно с

учетом браузеров, тогда как стандарт CommonJS предназначен для универсальной среды JavaScript (например, серверов на основе Node.js) и не привязан к ограничениям, накладываемым браузерами. В этом разделе дается относительно краткий обзор обеих стандартов определения модулей, а описание их установки и включения в проекты выходит за рамки данной книги. За более подробными сведениями по данному вопросу рекомендуется обратиться к книге *JavaScript Application Design* Николя Г. Бевакуа (Nicolas G. Bevacqua; издательство Manning, 2015 г.).

Стандарт AMD

Стандарт AMD возник на основе Dojo (`https://dojotoolkit.org/`) — одного из самых распространенных наборов инструментальных средств для написания клиентских веб-приложений на JavaScript. Стандарт AMD позволяет без особого труда определять модули и их зависимости. В то же время он был изначально создан специально для браузеров. В настоящее время наиболее распространенной реализацией стандарта AMD является загрузчик модулей RequireJS (`http://requirejs.org/`).

В качестве примера рассмотрим определение небольшого модуля, у которого имеется зависимость от библиотечного модуля jQuery, как демонстрируется в коде из листинга 11.3.

Листинг 11.3. Определение модуля с зависимостью от библиотечного модуля jQuery по стандарту AMD

```
define('MouseCounterModule',['jQuery'], $ => {
  let numClicks = 0;
  const handleClick = () => {
    alert(++numClicks);
  };

  return {
    countClicks: () => {
      $(document).on("click", handleClick);
    }
  };
});
```

С помощью функции `define()` определяется модуль, его зависимости и фабричная функция, создающая модуль

Открытый интерфейс модуля

В стандарте AMD определена функция `define()`, которой передаются следующие аргументы.

- Идентификатор вновь созданного модуля. Этим идентификатором можно будет воспользоваться в дальнейшем, чтобы затребовать модуль из других частей системы.
- Список идентификаторов модулей, от которых зависит текущий модуль, т.е. обязательных модулей.
- Фабричная функция, инициализирующая модуль, которой передаются в качестве аргументов используемые модули.

В данном примере кода используется функция `define()` по стандарту AMD, чтобы создать модуль с идентификатором `MouseCounterModule` и зависимостью от библиотечного модуля jQuery. В силу этой зависимости по стандарту AMD функция сначала запрашивает библиотечный модуль jQuery, на что может уйти время, если файл должен быть загружен из удаленного сервера. Такое действие выполняется асинхронно во избежание блокировки. Как только все зависимости будут загружены и проинтерпретированы, вызывается фабричная функция модуля с одним аргументом для каждого запрашиваемого модуля. В данном случае указан один аргумент, поскольку новому модулю требуется только библиотечный модуль jQuery. Новый модуль создается в теле фабричной функции таким же образом, как и при использовании проектного шаблона `Модуль`, т.е. путем возврата объекта, делающего видимым открытый интерфейс модуля.

Как видите, стандарт AMD предоставляет ряд интересных преимуществ, в том числе следующие.

- Автоматическое разрешение зависимостей, избавляющее от необходимости думать о порядке, в котором следует включать модули.
- Модули могут загружаться асинхронно, благодаря чему исключаются блокировки.
- В одном файле может быть определено несколько модулей.

Итак, пояснив основной принцип стандарта AMD, перейдем к рассмотрению CommonJS — другого, не менее распространенного стандарта определения модулей.

Стандарт CommonJS

Если стандарт AMD был создан специально для браузеров, то стандарт CommonJS предназначен для определения модулей в универсальной среде JavaScript. В настоящее время он получил наибольшее распространение в сообществе пользователей платформы Node.js.

В стандарте CommonJS применяются файловая структура модулей, что позволяет сохранять модули в виде отдельных файлов. Для каждого модуля согласно стандарту CommonJS доступна переменная `module` со свойством `exports`, которое нетрудно расширить дополнительными свойствами. В конечном счете содержимое свойства `module.exports` используется в качестве открытого интерфейса модуля.

Если требуется воспользоваться модулем в других частях приложения, его сначала нужно запросить. Файл модуля будет загружен синхронно, после чего станет доступен его открытый интерфейс. Именно поэтому стандарт CommonJS чаще всего используется на стороне сервера, где модули загружаются относительно быстро, поскольку для этого достаточно лишь выполнить операцию чтения в файловой системе, а не на стороне клиента, где модули при-

ходится загружать из удаленного сервера и их синхронная загрузка зачастую приводит к блокировке.

В качестве примера рассмотрим снова определение модуля `MouseCounterModule`, но на этот раз по стандарту CommonJS, как демонстрируется в коде из листинга 11.4.

Листинг 11.4. Определение модуля по стандарту CommonJS

```
// MouseCounterModule.js
const $ = require("jQuery");       ← Запрос на синхронную загрузку
let numClicks = 0;                   библиотечного модуля jQuery
const handleClick = () => {
  alert(++numClicks);
};

                                   Изменить свойство module.exports,
module.exports = {            ←    чтобы указать открытый интерфейс модуля
  countClicks: () => {
    $(document).on("click", handleClick);
  }
};
```

Чтобы включить модуль в другой файл, достаточно написать следующий фрагмент кода:

```
const MouseCounterModule = require("MouseCounterModule.js");
MouseCounterModule.countClicks();
```

Как видите, все очень просто. Принцип действия стандарта CommonJS предполагает распределение модулей по отдельным файлам, и поэтому любой код, размещаемый в файловом модуле, становится частью этого модуля. Следовательно, отпадает необходимость заключать переменные в оболочку немедленно вызываемых функций. Все переменные, определяемые в модуле, безопасно содержатся в области видимости текущего модуля, и никак не влияют на глобальную область видимости. Например, переменные (`$`, `numClicks` и `handleClick`) оказываются в области видимости рассматриваемого здесь модуля, несмотря на то, что они определены в коде верхнего уровня (т.е. за пределами всех функций и блоков), что, по существу, делает их глобальными переменными в стандартных файлах JavaScript.

И в этом случае очень важно заметить, что из внешнего модуля доступны только переменные и функции, видимые через свойство объекта `module.exports`. Сама процедура — такая же, как и при использовании проектного шаблона `Модуль`, только вместо возврата совершенно нового объекта в среде заранее создается объект, который может быть расширен новыми методами и свойствами интерфейса модуля.

Определение модулей по стандарту CommonJS имеет следующие преимущества.

- Простой синтаксис. Достаточно указать только свойства `module.exports`, оставив остальную часть модуля практически такой же, как

и при написании стандартного кода JavaScript. Запрос модулей также осуществляется очень просто, для чего достаточно вызвать функцию `require()`.

- Формат CommonJS является стандартным для платформы Node.js. Благодаря этому тысячи пакетов становятся доступными через утилиту `npm` — диспетчер пакетов среды Node.js.

Самый крупный недостаток стандарта CommonJS заключается в том, что он разработан без учета среды браузеров. В интерпретаторе JavaScript браузера отсутствует поддержка переменной `module` и свойства `export`, и поэтому модули, определяемые по стандарту CommonJS, приходится упаковывать в формате, удобочитаемом для браузера. И этого можно добиться с помощью инструментального средства Browserify (`http://browserify.org/`) или RequireJS (`http://requirejs.org/docs/commonjs.html`).

Наличие стандартов AMD и CommonJS для определения модулей привело к разделению разработчиков на два (иногда противоположных) лагеря. Если речь идет о работе над относительно закрытыми проектами, то достаточно выбрать наиболее подходящий стандарт. Но трудности могут появиться, когда потребуется повторное использование кода из противоположного лагеря, и тогда придется преодолеть немало препятствий. В качестве выхода из столь затруднительного положения можно, например, воспользоваться проектным шаблоном `UMD` (Universal Module Definition — универсальное определение модулей; `https://github.com/umdjs/umd`) с несколько замысловатым синтаксисом, позволяющим применять один и тот же модульный файл согласно обоим стандартам, AMD и CommonJS. Рассмотрение этого вопроса выходит за рамки данной книги, но если он заинтересует вас, то в Интернете вы сможете найти немало полезных ресурсов.

Правда, комитет ECMAScript, отвечающий за стандартизацию JavaScript, признал необходимость поддержки единообразного синтаксиса модулей во всех средах JavaScript. И с этой целью в ES6 был определен новый стандарт поддержки модулей, который призван окончательно разрешить все упомянутые выше недоразумения.

11.2. Модули по стандарту ES6

В ES6 модули определяется таким образом, чтобы можно было использовать преимущества обоих стандартов, AMD и CommonJS. В частности,

- подобно стандарту CommonJS, модули обладают относительно простым синтаксисом и распределяются по отдельным файлам;
- подобно стандарту AMD, модули поддерживают асинхронную загрузку.

На заметку

Встроенные модули являются частью стандарта ES6. Как станет ясно в дальнейшем, в синтаксис определения модулей, принятый в стандарте ES6, включена дополнительная семантика и ключевые слова (например, **`export`** и **`import`**), которые пока еще не поддерживаются в современных браузерах. Чтобы пользоваться модулями в настоящее время, придется выполнить транспиляцию модульного кода средствами Traceur (**`https://github.com/google/traceur-compiler`**), Babel (**`http://babeljs.io/`**) или TypeScript (**`www.typescriptlang.org/`**). Можно также воспользоваться библиотекой SystemJS (**`https://github.com/systemjs/systemjs`**), обеспечивающей поддержку загрузки модулей по всем ныне действующим стандартам: AMD, CommonJS и даже ES6. Подробные инструкции по применению библиотеки SystemJS можно найти в хранилище данного проекта (**`https://github.com/systemjs/systemjs`**).

Принцип определения модулей по стандарту ES6 основывается на том, что из модуля явным образом экспортируются только идентификаторы, доступные за его пределами. А все остальные идентификаторы, даже те, что определены в области видимости верхнего уровня (т.е. глобальной области видимости в стандартном коде JavaScript), доступны только в самом модуле. И такое решение было навеяно стандартом CommonJS.

Чтобы обеспечить подобные функциональные возможности, в стандарт ES6 были введены следующие ключевые слова:

- **`export`** — для доступа к определенным идентификаторам за пределами модуля;
- **`import`** — для импорта экспортированных идентификаторов модулей.

Синтаксис экспорта и импорта функциональных возможностей модулей довольно прост. Но у него имеется немало едва заметных особенностей, которые мы постепенно рассмотрим далее.

11.2.1. Функциональные возможности экспорта и импорта

Начнем с простого примера кода из листинга 11.5, где демонстрируется, каким образом функциональные возможности экспортируются из одного модуля и импортируются в другой.

Листинг 11.5. Экспорт из библиотечного модуля Ninja.js

```
const ninja = "Yoshi";
export const message = "Hello";

export function sayHiToNinja() {
  return message + " " + ninja;
}
```

Определить в модуле переменную верхнего уровня

Определить переменную и функцию и экспортировать их из модуля с помощью ключевого слова `export`

Получить доступ к внутренней переменной модуля через открытый интерфейс API

Сначала в данном примере кода определяется переменная `ninja`, доступная только в самом модуле, хотя это и сделано в коде верхнего уровня (в коде до стандарта ES6 такая переменная стала бы глобальной).

Затем в данном примере кода определяется еще одна переменная `message` верхнего уровня, доступная за пределами модуля благодаря ключевому слову `export`. И, наконец, создается и экспортируется функция `sayHiToNinja()`.

Вот и все! Это минимальный синтаксис, который следует знать для определения модулей. Чтобы экспортировать функциональные возможности из модуля, не нужно вызывать промежуточные функции или запоминать какой-нибудь замысловатый синтаксис. Достаточно написать код в привычном для JavaScript стиле, указав лишь перед некоторыми идентификаторами (например, переменных, функций или классов) ключевое слово `export`.

Прежде чем выяснять, каким образом импортируются функциональные возможности модулей, рассмотрим другой способ экспорта идентификаторов. Все, что требуется экспортировать из модуля, перечисляется в его конце, как выделено полужирным шрифтом в примере кода из листинга 11.6.

Листинг 11.6. Экспорт в конце модуля

```
const ninja = "Yoshi";
const message = "Hello";

function sayHiToNinja() {
  return message + " " + ninja;
}

export { message, sayHiToNinja };
```

Определить все идентификаторы модуля

Экспортировать некоторые идентификаторы модуля

Такой способ экспорта идентификаторов модуля чем-то напоминает проектный шаблон `Модуль`, где немедленно вызываемая функция возвращает объект, представляющий открытый интерфейс данного модуля. И особенно он напоминает стандарт CommonJS, где открытый интерфейс модуля расширяет свойство объекта `module.exports`.

Но независимо от способа экспорта идентификаторов определенного модуля, для их импорта придется воспользоваться ключевым словом `import`, как выделено полужирным шрифтом в примере кода из листинга 11.7.

Листинг 11.7. Импорт из модуля Ninja.js

```
import { message, sayHiToNinja} from "Ninja.js";

assert(message === "Hello",
       "We can access the imported variable");
assert(sayHiToNinja() === "Hello Yoshi",
       "We can say hi to Yoshi from outside the module");

assert(typeof ninja === "undefined",
       "But we cannot access Yoshi directly");
```

Импортировать привязку идентификатора из модуля с помощью ключевого слова `import`

Теперь можно обратиться к импортированной переменной и вызвать импортированную функцию

Неэкспортированные переменные недоступны напрямую

В данном примере кода ключевое слово import применяется для импорта переменной message и функции sayHiToNinja() из модуля ninja следующим образом:

```
import { message, sayHiToNinja} from "Ninja.js";
```

Таким образом, мы получаем доступ к этим двум идентификаторам, определенным в модуле ninja. И, наконец, возможность доступа к переменной message и функции sayHiToNinja() проверяется в следующем фрагменте кода:

```
assert(message === "Hello",
       "We can access the imported variable");
assert(sayHiToNinja() === "Hello Yoshi",
       "We can say hi to Yoshi from outside the module");
```

Но недоступными оказываются переменные, которые не были экспортированы и импортированы. Например, переменная ninja недоступна потому, что ее определение не помечено ключевым словом export, что и подтверждается в приведенном ниже фрагменте кода.

```
assert(typeof ninja === "undefined",
       "But we cannot access Yoshi directly");
```

Благодаря поддержке модулей в стандарте ES6 наконец-то появилась возможность меньше злоупотреблять глобальными переменными. Все, что не помечено явно ключевым словом export, остается надежно изолированным в модуле.

В данном примере использован *именованный экспорт*, позволяющий экспортировать несколько идентификаторов из модуля, как это было сделано с идентификаторами message и sayHiToNinja. Но в связи с возможностью экспортировать большое количество идентификаторов перечислять их полностью в операторе импорта не очень удобно. Поэтому для этой цели употребляется сокращенное обозначение, как демонстрируется в примере кода из листинга 11.8.

Листинг 11.8. Импорт всего именованного экспорта из модуля Ninja.js

```
import * as ninjaModule from "Ninja.js";

assert(ninjaModule.message === "Hello",
       "We can access the imported variable");
assert(ninjaModule.sayHiToNinja() === "Hello Yoshi",
       "We can say hi to Yoshi from outside the module");

assert(typeof ninjaModule.ninja === "undefined",
       "But we cannot access Yoshi directly");
```

Импортировать все экспортированные идентификаторы, используя сокращенное обозначение *

Обратиться к именованному экспорту, используя синтаксис свойства

Неэкспортированные идентификаторы по-прежнему недоступны

Как показано в коде из листинга 11.8, для импорта всех экспортированных идентификаторов из модуля употребляется сокращенная запись import * в сочетании с идентификатором, который затем будет использоваться для ссылки

на весь модуль (в данном случае это идентификатор `ninjaModule`). После этого к экспортированным идентификаторам можно обращаться, используя синтаксис свойства, например: `ninjaModule.message`, `ninjaModule.sayHiToNinja`. Следует, однако, иметь в виду, что те переменные верхнего уровня модуля, которые не были экспортированы, по-прежнему остаются недоступными, (например, переменная `ninja`).

Экспорт по умолчанию

Нередко вместо экспорта из модуля взаимосвязанных идентификаторов, нужно представить весь модуль в виде единственной операции экспорта. Нечто подобное нередко происходит в том случае, если модули содержат единственный класс, как показано в примере кода из листинга 11.9.

Листинг 11.9. Экспорт по умолчанию из модуля Ninja.js

```
export default class Ninja {
  constructor(name){
    this.name = name;
  }
}

export function compareNinjas(ninja1, ninja2){
  return ninja1.name === ninja2.name;
}
```

Указать привязку к модулю по умолчанию, используя ключевые слова `export` и `default`

Наряду с экспортом по умолчанию можно по-прежнему пользоваться именованным экспортом

В данном примере кода после ключевого слова `export` указывается ключевое слово `default`, обозначающее привязку к модулю по умолчанию. Такой привязкой в данном случае служит класс `Ninja`. Но несмотря на указание привязки по умолчанию, для экспорта дополнительных идентификаторов можно по-прежнему пользоваться именованным экспортом, как это было показано на примере функции `compareNinjas()`.

И теперь для импорта функциональных возможностей из модуля Ninja.js можно воспользоваться упрощенным синтаксисом (выделено полужирным в примере кода из листинга 11.10).

Листинг 11.10. Импорт экспорта по умолчанию

Чтобы импортировать экспорт по умолчанию, не нужно указывать фигурные скобки; при этом можно использовать любое удобное имя

Именованный экспорт можно по-прежнему импортировать

```
import ImportedNinja from "Ninja.js";
import {compareNinjas} from "Ninja.js";

const ninja1 = new ImportedNinja("Yoshi");
const ninja2 = new ImportedNinja("Hattori");

assert(ninja1 !== undefined
      && ninja2 !== undefined, "We can create a couple of Ninjas");
```

Создать пару объектов ниндзя и проверить их существование

```
assert(!compareNinjas(ninja1, ninja2),
       "We can compare ninjas");
```

Именованный экспорт по-прежнему доступен

Сначала в данном примере кода импортируется экспорт по умолчанию. С этой целью используется более простой синтаксис без фигурных скобок, которые обязательны для импорта именованного экспорта. Следует также заметить, что для обозначения экспорта по умолчанию можно выбрать произвольное имя, и это совсем не обязательно должно быть имя, использованное при экспорте. В данном примере именем `ImportedNinja` обозначается класс `Ninja`, определенный в модуле Ninja.js.

Далее в рассматриваемом здесь примере кода импортируется именованный элемент, чтобы продемонстрировать, как и в предыдущих примерах, возможность как экспорта по умолчанию, так и именованного экспорта элементов из одного и того же модуля. Наконец, в данном примере кода получаются два экземпляра объекта `ninja` и вызывается функция `compareNinjas()`, чтобы убедиться, что весь импорт действует должным образом.

Поскольку в данном случае оба элемента импортированы из одного и того же файла, в стандарте ES6 реализован специальный сокращенный синтаксис, как показано ниже.

```
import ImportedNinja, {compareNinjas} from "Ninja.js";
```

Здесь в одном операторе импорта из модуля Ninja.js через запятую (`,`) указывается как экспорт по умолчанию, так и именованный экспорт.

Переименование экспорта и импорта

При необходимости экспорт и импорт можно переименовать. Начнем с переименования экспорта, как в приведенном ниже примере кода, где в комментариях указана принадлежность кода к определенному файлу.

```
//************* Greetings.js ************/
function sayHi(){
  return "Hello";
}

assert(typeof sayHi === "function"
       && typeof sayHello === "undefined",
       "Within the module we can access only sayHi");

export { sayHi as sayHello }

//************* main.js *****************/
import {sayHello } from "Greetings.js";

assert(typeof sayHi === "undefined"
       && typeof sayHello === "function",
       "When importing, we can only access the alias");
```

Определить функцию `sayHi()`

Проверить возможность доступа к функции `sayHi()` только по идентификатору, но не по псевдониму!

Указать псевдоним идентификатора с помощью ключевого слова `as`

При импорте доступен только псевдоним `sayHello`

В приведенном выше примере кода определяется функция `sayHi()` и проверяется возможность доступа к ней только по идентификатору `sayHi`, но не по псевдониму `sayHello`, который определяется в конце модуля с помощью ключевого слова `as`:

```
export { sayHi as sayHello }
```

Переименовать экспорт можно только в такой форме, а не предваряя объявление функции или переменной ключевым словом `export`. А когда выполняется импорт переименованного экспорта, то обращение к импорту осуществляется по указанному псевдониму:

```
import { sayHello } from "Greetings.js";
```

И, наконец, возможность доступа к импортированной функции по псевдониму, но не по первоначальному ее идентификатору проверяется следующим образом:

```
assert(typeof sayHi === "undefined"
       && typeof sayHello === "function",
       "When importing, we can only access the alias");
```

Аналогичная ситуация возникает при переименовании импорта, как в следующем примере кода:

```
/************* Hello.js *************/
export function greet(){        // Экспортировать функцию под именем greet() из модуля Hello.js
  return "Hello";
}

/************* Salute.js *************/
export function greet(){        // Экспортировать функцию под тем же самым именем greet() из модуля Salute.js
  return "Salute";
}

/************* main.js *************/
import { greet as sayHello } from "Hello.js";   // Указать псевдонимы импорта
import { greet as salute } from "Salute.js";    // с помощью ключевого слова as, исключив конфликт имен

assert(typeof greet === "undefined",            // Функция недоступна по
       "We cannot access greet");               // первоначальному имени

assert(sayHello() === "Hello" && salute() === "Salute",   // Но она доступна через
       "We can access aliased identifiers!");             // псевдонимы
```

Как и при экспорте идентификаторов, для указания псевдонимов при импорте идентификаторов из других модулей можно воспользоваться ключевым словом `as`. Это удобно в том случае, если требуется предоставить имя, более подходящее для текущего контекста, или избежать конфликта имен, как и в приведенном выше небольшом примере кода.

На этом исследование синтаксиса определения модулей по стандарту ES6 завершается. А краткие его итоги подведены в табл. 11.1.

Таблица 11.1. Краткий обзор синтаксиса определения модулей по стандарту ES6

Код	Назначение
`export const ninja = "Yoshi";`	Экспорт именованной переменной
`export function compare(){}`	Экспорт именованной функции
`export class Ninja{}`	Экспорт именованного класса
`export default class Ninja{}`	Экспорт класса по умолчанию
`export default function Ninja(){}`	Экспорт функции по умолчанию
`const ninja = "Yoshi";`	
`function compare(){};`	
`export {ninja, compare};`	Экспорт существующих переменных
`export {ninja as samurai, compare};`	Экспорт переменной по новому имени
`import Ninja from "Ninja.js";`	Импорт экспорта по умолчанию
`import {ninja, Ninja} from "Ninja.js";`	Импорт именованного экспорта
`import * as Ninja from "Ninja.js";`	Импорт всего именованного экспорта из модуля
`import {ninja as iNinja} from "Ninja.js";`	Импорт именованного экспорта по новому имени

Резюме

Подведем краткий итог тому, что вы узнали из этой главы.

- Крупные и монолитные кодовые базы намного более трудны для понимания и сопровождения, чем более мелкие и хорошо организованные кодовые базы. Чтобы усовершенствовать структуру и организацию прикладных программ, можно, в частности, разбить их на более мелкие, слабо связанные части, или модули.
- Модули являются более крупными единицами организации кода, чем объекты и функции. Они позволяют разделять прикладные программы на принадлежащие друг другу части.
- В общем, модули помогают лучше понимать, проще сопровождать исходный код и усовершенствовать его повторное использование.
- До принятия стандарта ES6 в языке JavaScript отсутствовали встроенные модули, поэтому разработчикам приходилось творчески подходить к применению имеющихся языковых средств JavaScript, чтобы добиться модуляризации прикладного кода. Одним из самых распространенных

способов создания модулей стало сочетание немедленно вызываемых функций с замыканиями.

- Немедленно вызываемые функции применяются потому, что они образуют новую область видимости для определения в модуле переменных, недоступных за пределами этой области видимости.
- Замыкания применяются потому, что они позволяют поддерживать активными переменные в модуле.
- Наибольшее распространение при этом нашел проектный шаблон `Модуль`, где вызов немедленно вызываемой функции обычно сочетается с возвратом нового объекта, представляющего открытый интерфейс модуля.

- Помимо проектного шаблона `Модуль`, существуют два распространенных стандарта: AMD (Asynchronous Module Definition) и CommonJS. Первый из них предназначен для поддержки модулей в браузерах, а второй чаще всего применяется в серверных приложениях JavaScript.
  - Стандарт AMD позволяет автоматически разрешать зависимости, асинхронно загружать модули, исключая тем самым блокировку.
  - Стандарт CommonJS обладает простым синтаксисом, позволяет синхронно загружать модули и поэтому в большей степени пригоден для применения на стороне сервера. А с помощью диспетчера пакетов Node.js `npm` можно сделать доступными многие модули.
- Модули, введенные в стандарт ES6, призваны вобрать в себя все преимущества стандартов AMD и CommonJS. Такие модули обладают простым синтаксисом, навеянным стандартом CommonJS, а также допускают асинхронную загрузку, как предусмотрено в стандарте AMD.
  - Модули в стандарте ES6 распределяются по отдельным файлам.
  - Идентификаторы можно экспортировать, используя ключевое слово `export`, чтобы обращаться к другим модулям.
  - Идентификаторы можно импортировать из других модулей, используя ключевое слово `import`.
  - У модуля может быть единственный экспорт по умолчанию (`default`), с помощью которого можно представить модуль в целом.
  - Экспорт и импорт модулей можно переименовывать с помощью ключевого слова `as`.

Упражнения

1. Какой из перечисленных ниже механизмов допускает наличие закрытых переменных модуля в проектном шаблоне `Модуль`?

   а) Прототипы

б) Замыкания

в) Обещания

2. В приведенном ниже фрагменте кода применяются модули, внедренные в стандарт ES6. Какие из перечисленных ниже идентификаторов могут быть доступны, если импортировать модуль?

```
const spy = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
export const general = "Minamoto";
```

а) `spy`

б) `command`

в) `general`

3. В приведенном ниже фрагменте кода применяются модули, внедренные в стандарт ES6. Какие из перечисленных ниже идентификаторов могут быть доступны, если импортировать модуль?

```
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general = "Minamoto";

export {ninja as spy};
```

а) `spy`

б) `command`

в) `general`

г) `ninja`

4. Какой из перечисленных ниже видов импорта модулей допустим?

```
// Файл модуля: personnel.js
const ninja = "Yagyu";
function command(){
  return general + " commands you to wage war!";
}
const general = "Minamoto";

export {ninja as spy};
```

а) `import {ninja, spy, general} from "personnel.js"`

б) `import * as Personnel from "personnel.js"`

в) `import {spy} from "personnel.js"`

5. Какой из перечисленных ниже операторов импортирует класс `Ninja` в приведенном ниже фрагменте кода?

```
// Ninja.js
export default class Ninja {
  skulk(){ return "skulking"; }
}
```

а) `import Ninja from "Ninja.js"`

б) `import * as Ninja from "Ninja.js"`

в) `import * from "Ninja.js"`