# 10 Модули

> Пишите код, который легко удалить,
> но трудно растянуть.
>
> *Tef. Программирование — ужасная вещь*

Идеальная программа имеет кристально ясную структуру. Можно легко объяснить, как она работает, и роль каждой ее части четко определена.

Типичная реальная программа растет как дерево. По мере того как появляются новые потребности, добавляются функциональные возможности. Структурирование — и сохранение структуры — дополнительная работа, которая окупится только в будущем, когда кто-нибудь в *очередной* раз будет работать над программой. А пока есть сильный соблазн пренебречь этим и позволить частям программы сильно запутаться.

На практике подобное приводит к двум проблемам. Во-первых, понять такую систему непросто. Если любые изменения могут затронуть все остальное, трудно рассмотреть какой-то фрагмент в отдельности. Приходится разбираться во всей программе целиком. Во-вторых, если вы захотите использовать какую-либо функциональность такой программы в другой ситуации, может оказаться, что ее проще переписать заново, чем пытаться отделить от контекста.

Подобные большие, бесструктурные программы часто описывают фразой «большой ком грязи»: все слиплось, и стоит попытаться выделить что-то одно, как оно разваливается и вы стоите с грязными руками.

## Зачем нужны модули

*Модули* дают возможность избежать подобных проблем. Модулем называется часть программы, которая определяет, на какие другие компоненты она

опирается и какие функциональные возможности предоставляет другим модулям (*интерфейс* модуля).

Интерфейсы модулей имеют много общего с интерфейсами объектов, описанными в главе 6. Интерфейс делает часть модуля доступной для внешнего мира и закрывает остальное. Ограничивая способы взаимодействия модулей между собой, система превращается в подобие конструктора «Лего», части которого соединяются посредством четко определенных разъемов, и становится уже менее похожей на кусок грязи, где все смешалось в один большой ком.

Отношения между модулями называются *зависимостями*. Когда модулю требуется фрагмент из другого модуля, говорят, что он зависит от этого модуля. Когда данный факт четко описан в самом модуле, его можно использовать с целью выяснить, какие другие модули необходимы для того, чтобы можно было применять данный модуль и автоматически загружать зависимости.

Для того чтобы разделить модули подобным образом, каждому из них нужна своя собственная область видимости.

Простое размещение кода JavaScript в разных файлах не удовлетворяет этим требованиям. Файлы по-прежнему используют общее глобальное пространство имен. Они могут, намеренно или случайно, изменять привязки друг друга. Остается также неясной структура зависимостей. Как мы увидим в данной главе, это можно улучшить.

Разработать подходящую структуру модулей программы бывает непросто. На этапе предварительного исследования проблемы, подбирая для нее различные варианты решения, об этом можно особо не беспокоиться, поскольку это может сильно отвлекать. Но, когда у вас уже есть что-то, что представляется надежным решением, самое время сделать шаг назад и привести это в порядок.

## Пакеты

Одним из преимуществ построения программы из отдельных частей и возможности запускать их по отдельности является то, что вы можете использовать один и тот же фрагмент в разных программах.

Но как это сделать? Предположим, я хочу применить функцию `parseINI` из главы 9 в другой программе. Если известно, от чего зависит эта функция

(в данном случае ни от чего), то я могу просто скопировать весь необходимый код в мой новый проект и использовать его. Но потом, если я найду ошибку в этом коде, я, вероятно, исправлю ее в той программе, с которой сейчас работаю, но забуду исправить ее в другой программе.

Как только вы начнете дублировать код, вы быстро обнаружите, что тратите время и силы на перемещение копий и поддержание их в актуальном состоянии.

Именно здесь нам помогут *пакеты*. Пакет — это кусок кода, который можно распространять (копировать и устанавливать). Пакет может содержать один или несколько модулей, а также информацию о том, от каких других пакетов он зависит. Пакет обычно поставляется в комплекте с документацией, объясняющей, что он делает, чтобы тот, кто его не писал, мог его использовать.

Если в пакете обнаружена проблема или добавляется новая функция, пакет обновляется. После этого программы, которые от него зависят (и которые также могут быть пакетами), могут обновиться до новой версии.

Такой стиль работы требует инфраструктуры. Нам нужно место для хранения и поиска пакетов, а также удобный способ их установки и обновления. В мире JavaScript данная инфраструктура предоставляется NPM (https://www.npmjs.com/).

NPM — это два в одном: онлайн-сервис, откуда можно загружать (и где можно размещать) пакеты, и программа (поставляется в комплекте с Node.js), которая помогает устанавливать пакеты и управлять ими.

На момент написания этой книги в NPM было доступно более полумиллиона различных пакетов. Следует признать, что большая часть из них — просто мусор. Но почти все полезные, общедоступные пакеты там также можно найти. Например, в пакете с именем `ini` есть синтаксический анализатор INI-файлов, аналогичный тому, который мы создали в главе 9.

В главе 20 вы узнаете, как устанавливать такие пакеты локально с помощью программы командной строки `npm`.

Наличие качественных пакетов, доступных для скачивания, чрезвычайно важно. Это зачастую означает возможность не переписывать заново программу, которую уже написали до нас 100 человек, вместо чего получить надежную, проверенную реализацию, нажав всего нескольких клавиш.

Программное обеспечение легко копируется, поэтому распространение того, что кто-то уже написал ранее, среди других людей является эффективным процессом. Но написать исходную программу — *немалый* труд, а реагировать на сообщения от людей, которые обнаружили проблемы в коде или хотят предложить новые функции, — труд еще больший.

По умолчанию авторские права на написанный вами код принадлежат вам, и другие люди могут использовать его только с вашего разрешения. Но, поскольку существуют просто щедрые люди, а публикация хорошего программного обеспечения способна сделать вас немного популярнее среди программистов, многие пакеты публикуются по лицензии, которая явно позволяет другим применять эти пакеты.

Именно так лицензируется бóльшая часть кода на NPM. Некоторые лицензии требуют, чтобы вы тоже публиковали код, созданный на основе пакета, под той же лицензией. Другие менее требовательны, они лишь требуют сохранять лицензию вместе с кодом при его распространении. Сообщество JavaScript в основном использует последний тип лицензии. При применении пакетов других разработчиков ознакомьтесь с их лицензией.

## Импровизированные модули

До 2015 года в языке JavaScript не было встроенной системы модулей. Тем не менее к тому времени программисты уже более десяти лет создавали большие системы на JavaScript, и им были *необходимы* модули.

Поэтому они разработали собственные модульные системы как надстройку над языком. Вы можете использовать функции JavaScript для создания локальных областей видимости и объектов, имитирующих интерфейсы модулей.

Ниже показан модуль для преобразования названий дней недели в их номера (как возвращает метод `Date getDay`). Его интерфейс состоит из `weekDay.name` и `weekDay.number`, а локальная привязка `names` скрыта внутри области видимости функции, которая вызывается немедленно.

```
const weekDay = function() {
  const names = ["Понедельник", "Вторник", "Среда", "Четверг",
                 "Пятница", "Суббота", "Воскресенье"];
  return {
    name(number) { return names[number]; },
    number(name) { return names.indexOf(name); }
  };
```

```
}());

console.log(weekDay.name(weekDay.number("Воскресенье")));
// → Воскресенье
```

Такой стиль модулей в определенной степени обеспечивает изоляцию, но не объявляет зависимости. Вместо чего он просто размещает свой интерфейс в глобальной области и ожидает, что его зависимости, если таковые имеются, будут поступать так же. Долгое время данный подход был основным в веб-программировании, но сейчас он практически полностью устарел.

Если мы хотим сделать отношения зависимостей частью кода, то мы должны взять загрузку зависимостей под контроль. Для этого необходимо уметь выполнять строки как код. JavaScript позволяет это сделать.

## Выполнение данных как кода

Существует несколько способов взять данные (строку кода) и выполнить их как часть текущей программы.

Наиболее очевидным способом является специальный оператор `eval`, который выполняет строку в *текущей* области видимости. Обычно это плохая идея, потому что при этом нарушаются отдельные свойства, которыми обычно обладают области видимости, — например, предсказуемость того, какое имя относится к какой привязке.

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
  return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1
```

Менее неудачный способ интерпретации данных как кода — использовать конструктор `Function`. Он принимает два аргумента: строку, содержащую список имен аргументов через запятую, и строку, содержащую тело функции; он обертывает код в функцию-значение, чтобы получить собственную область видимости и не делать странных вещей с другими областями.

```
let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5
```

Это именно то, что нам нужно для модульной системы. Мы можем обернуть код модуля в функцию и использовать область видимости этой функции в качестве области видимости модуля.

## CommonJS

Наиболее широко применяемый подход для построения системы модулей JavaScript называется *модулями CommonJS*. Эта система используется в Node.js, а также в большинстве NPM-пакетов.

Основная концепция модулей CommonJS — функция, именуемая `require`. Если вызвать ее с именем модуля зависимости, то функция проверит, загружен ли этот модуль, и вернет его интерфейс.

Поскольку загрузчик обертывает код модуля в функцию, модули автоматически получают собственную локальную область видимости. Для того чтобы получить доступ к их зависимостям и поместить их интерфейс в объект, связанный с `exports`, остается лишь вызвать `require`.

В следующем примере показан модуль, содержащий функцию форматирования даты. Он использует два пакета из NPM — `ordinal` для преобразования чисел в строки, такие как `"1st"` и `"2nd"`, и `date-names`, чтобы получить английские названия дней недели и месяцев. Модуль экспортирует отдельную функцию `formatDate`, которая принимает объект `Date` и строку шаблона.

Строка шаблона может содержать коды, определяющие формат, такие как `YYYY` для полного года и `Do` для номера дня месяца. Например, если задать в качестве формата строку `"MMMM Do YYYY"`, то получим результат вида November 22nd 2017.

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");

exports.formatDate = function(date, format) {
  return format.replace(/YYYY|M(MMM)?|Do?|dddd/g, tag => {
    if (tag == "YYYY") return date.getFullYear();
    if (tag == "M") return date.getMonth();
    if (tag == "MMMM") return months[date.getMonth()];
```

```
    if (tag == "D") return date.getDate();
    if (tag == "Do") return ordinal(date.getDate());
    if (tag == "dddd") return days[date.getDay()];
  });
};
```

Интерфейс `ordinal` — это отдельная функция, тогда как `date-names` экспортирует объект, содержащий несколько свойств: `days` и `months` являются массивами имен. Деструктуризация очень удобна при создании привязок для импортируемых интерфейсов.

Модуль добавляет к `exports` свою функцию интерфейса, чтобы модули, зависящие от него, получали к ней доступ. Мы могли бы использовать этот модуль следующим образом:

```
const {formatDate} = require("./format-date");

console.log(formatDate(new Date(2017, 9, 13),
                       "dddd the Do"));
// → Friday the 13th
```

Мы можем определить `require` в самой минималистической форме, например так:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name);
    let module = {exports: {}};
    equire.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module);
  }
  return require.cache[name].exports;
}
```

В этом коде `readFile` представляет собой готовую функцию, которая читает файл и возвращает его содержимое в виде строки. В стандартном JavaScript такой функциональности нет, но различные среды JavaScript, такие как браузеры и Node.js, предоставляют собственные способы доступа к файлам. В этом примере просто предполагается, что `readFile` существует.

Во избежание загрузки одного и того же модуля несколько раз функция `require` хранит (кэширует) уже загруженные модули. При вызове она сначала

проверяет, был ли уже загружен запрошенный модуль, и если нет, то загружает его — читает код модуля, обертывает его в функцию и вызывает эту функцию.

Интерфейс пакета `ordinal`, который нам уже встречался, — это не объект, а функция. У модулей CommonJS есть одна странность: система модулей создает пустой интерфейсный объект (связанный с `exports`), но его можно заменить любым значением, перезаписав `module.exports`. Во многих модулях так и сделано, чтобы экспортировать одно значение вместо объекта интерфейса.

Определяя `require`, `export` и `module` в качестве параметров для генерируемой функции-оболочки (и передавая соответствующие значения при ее вызове), загрузчик обеспечивает доступность этих привязок в области видимости модуля.

Строка, переданная в `require`, преобразуется в фактическое имя файла или в веб-адрес разными способами, в зависимости от системы. Если строка начинается с `"./"` или `"../"`, то она обычно интерпретируется как путь относительно имени файла текущего модуля. Таким образом, `"./format-date"` — это файл с именем `format-date.js` в той же директории.

Если же имя не является относительным, Node.js будет искать установленный пакет с таким именем. В примере кода, рассмотренном в данной главе, мы будем интерпретировать такие имена как ссылки на NPM-пакеты. Подробнее о том, как устанавливать и использовать NPM-модули, я расскажу в главе 20.

Теперь, вместо того чтобы писать собственный анализатор INI-файлов, мы можем задействовать соответствующий NPM-пакет.

```
const {parse} = require("ini");

console.log(parse("x = 10\ny = 20"));
// → {x: "10", y: "20"}
```

## Модули ECMAScript

Модули CommonJS работают вполне приемлемо — в сочетании с NPM они позволили сообществу JavaScript начать широкомасштабное распространение кода.

Но они все еще остаются чем-то вроде решения «на коленке». Их нотация не очень удобна — например, то, что вы добавляете в `exports`, недоступно