> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад, присоединившись к русскоязычному сообществу MDN Web Docs.

Модули JavaScript

Это руководство содержит всю необходимую информацию для начала работы с модулями JavaScript

Модули: история вопроса

Сначала программы на JavaScript были небольшими — в прежние времена они использовались для изолированных задач, добавляя при необходимости немного интерактивности веб-страницам, так что большие скрипты в основном не требовались. Прошло несколько лет, и вот мы уже видим полномасштабные приложения, работающие в браузерах и содержащие массу кода на JavaScript; кроме того, язык стал использоваться и в других контекстах (например, Node.js).

Таким образом, в последние годы появились причины на то, чтобы подумать о механизмах деления программ на JavaScript на отдельные модули, которые можно импортировать по мере необходимости. Node.js включал такую возможность уже давно, кроме того, некоторые библиотеки и фреймворки JavaScript разрешали использование модулей (например, CommonJS и основанные на AMD системы модулей типа RequireJS, а позднее также Webpack и Babel).

К счастью, современные браузеры стали сами поддерживать функциональность модулей, о чем и рассказывает эта статья. Этому можно только порадоваться — браузеры могут оптимизировать загрузку модулей, что было бы гораздо эффективнее использования библиотеки, и взять на себя обработку на стороне клиента и прочие накладные расходы.

Встроенная обработка модулей JavaScript связана с инструкциями `import` и `export`, их поддержка браузерами показана в следующих таблицах.

Совместимость с браузерами

javascript.statements.import

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	
import	Chrome 61	Edge 16	Firefox 60	Opera 48	Safari 10.1	Chrome 61 Android	Firefox 60 for Android	Opera 45 Android	
Import attributes (`assert` syntax)	Chrome 91	Edge 91	Firefox No	Opera No	Safari No	Chrome 91 Android	Firefox No for Android	Opera No Android	

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android
import assert {type: json}	Chrome 91	Edge 91	Firefox No	Opera No	Safari No	Chrome 91 Android	Firefox No for Android	Opera No Android
Import attributes	Chrome No	Edge No	Firefox No	Opera No	Safari No	Chrome No Android	Firefox No for Android	Opera No Android
import with {type: 'json'}	Chrome No	Edge No	Firefox No	Opera No	Safari No	Chrome No Android	Firefox No for Android	Opera No Android
Available in service workers	Chrome 91	Edge 91	Firefox 114	Opera 77	Safari 15	Chrome 91 Android	Firefox 114 for Android	Opera 64 Android
Available in workers	Chrome 80	Edge 80	Firefox 114	Opera 67	Safari 15	Chrome 80 Android	Firefox 114 for Android	Opera 57 Android
Available in worklets	Chrome No	Edge No	Firefox 114	Opera No	Safari No	Chrome No Android	Firefox 114 for Android	Opera No Android

Tip: you can click/tap on a cell for more information.

Full support No support Experimental. Expect behavior to change in the future.

Non-standard. Check cross-browser support before using. Deprecated. Not for use in new websites. See implementation notes.

Has more compatibility info.

javascript.statements.export

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
export	Chrome 61	Edge 16	Firefox 60	Opera 48	Safari 10.1	Chrome 61 Android	Firefox 60 for Android	Opera 45 Android	Safari on iOS
default keyword with export	Chrome 61	Edge 16	Firefox 60	Opera 48	Safari 10.1	Chrome 61 Android	Firefox 60 for Android	Opera 45 Android	Safari on iOS
export * as namespace	Chrome 72	Edge 79	Firefox 80	Opera 60	Safari 14.1	Chrome 72 Android	Firefox 80 for Android	Opera 51 Android	Safari on iOS

Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
61	16	60	48	10.1	61	60	45	10.3

Tip: you can click/tap on a cell for more information.

Full support No support See implementation notes.

Пример использования модулей

Для того, чтобы продемонстрировать использование модулей, мы создали простой набор примеров , который вы можете найти на GitHub. В этих примерах мы создаём элемент `<canvas>` на веб-странице и затем рисуем различные фигуры на нём (и выводим информацию об этом).

Примеры довольно тривиальны, но они намеренно сделаны простыми для ясной демонстрации модулей.

> **Примечание:** Если вы хотите скачать примеры и запустить их локально, вам нужно будет запустить их через локальный веб-сервер.

Базовая структура примера

В первом примере (см. директорию basic-modules) у нас следующая структура файлов:

```
index.html
main.js
modules/
    canvas.js
    square.js
```

> **Примечание:** Все примеры в этом руководстве в основном имеют одинаковую структуру.

Давайте разберём два модуля из директории modules:

- `canvas.js` — содержит функции, связанные с настройкой canvas:
  - `create()` — создаёт холст заданной ширины `width` и высоты `height` внутри `<div>`-обертки с указанным `id` и помещённой в родителя `parent`. Результатом выполнения функции будет объект, содержащий 2D-контекст холста и id обертки.
  - `createReportList()` — создаёт неупорядоченный список, добавленный внутри указанного элемента-обёртки, который можно использовать для вывода данных отчёта. Возвращает id списка.
- `square.js` — содержит:
  - `name` — переменная со строковым значением `'square'`.
  - `draw()` — функция, рисующая квадрат на указанном холсте с заданными размером, положением и цветом. Возвращает объект, содержащий размер, положение и цвет квадрата.
  - `reportArea()` — функция, которая выводит посчитанную площадь квадрата в указанный список отчета.
  - `reportPerimeter()` — функция, которая выводи посчитанный периметр квадрата в указанный список отчета.

Взгляд со стороны — `.mjs` против `.js`

В этой статье мы используем расширение `.js` для файлов наших модулей, но в других источниках вы можете встретить расширение `.mjs`. Например, в документации движка V8 используется `.mjs` . Причины следующие:

- Это полезно для ясности, то есть дает понять, какие файлы являются модулями, а какие — обычными JavaScript-файлами.
- Это гарантирует, что файлы вашего модуля будут проанализированы как модуль средами выполнения, такими как Node.js , и инструментами сборки, такими как Babel .

Тем не менее, мы решили продолжать использовать `.js`, по крайней мере на данным момент. Чтобы модули корректно работали в браузере, вам нужно убедиться, что ваш сервер отдаёт их с заголовком `Content-Type`, который содержит JavaScript MIME type такой как `text/javascript`. В противном случае вы получете ошибку проверки MIME type — "The server responded with a non-JavaScript MIME type", и браузер не сможет запустить ваш JavaScript. Большинство серверов уже имеют правильный тип для `.js`-файлов, но ещё не имеют нужного MIME type для `.mjs`-файлов. Серверы, которые уже отдают `.mjs` файлы корректно, включают в себя GitHub Pages и `http-сервер` для Node.js.

Это нормально, если вы уже используете такую среду или ещё нет, но знаете, что делать, и имеете нужные доступы (то есть вы можете настроить свой сервер, чтобы он устанавливал корректный `Content-Type`-заголовок для `.mjs`-файлов). Однако это может вызвать путаницу, если вы не контролируете сервер, с которого отдаются файлы, или публикуете файлы для общего пользования, как мы здесь.

В целях обучения и переносимости на разные платформы мы решили остановится на `.js`.

Если вы действительно видите ценность и ясность использования `.mjs` для модулей по сравнению с использованием `.js` для обычных JavaScript-файлов, но не хотите столкнуться с проблемой описанной выше, вы должны всегда использовать `.mjs` во время разработки и конвертировать их в `.js` во время сборки.

Также стоит отметить, что:

- Некоторые инструменты могут никогда не добавить поддержку `.mjs`, например, TypeScript.
- `<script type="module">` атрибут используется для обозначения того, что файл является модулем. Вы увидите примеры использования данного атрибута ниже.

Экспорт функциональности модуля

Первое, что нужно сделать, чтобы получить доступ к функциональности модуля, — экспортировать его. Это делается с помощью инструкции `export`.

Самый простой способ использовать экспорт — поместить конструкцию `export` перед любыми элементами, которые вы хотите экспортировать из модуля, например:

JS

```
export const name = "square";

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color,
  };
}
```

Вы можете экспортировать `var`-, `let`-, `const`-переменные, и — как мы увидим позже — классы. Они должны быть в верхней области видимости, вы не можете использовать `export` внутри функции, например.

Более удобный способ экспорта всех элементов, которые вы хотите экспортировать,— использовать одну конструкцию `export` в конце файла модуля, где указать переменные, функции, классы, который вы хотите экспортировать, через запятую в фигурных скобках. Например:

JS

```
export { name, draw, reportArea, reportPerimeter };
```

Импорт функциональности в ваш скрипт

После того, как вы экспортировали некоторые части из своего модуля, вам необходимо импортировать их в свой скрипт, чтобы иметь возможность использовать их. Самый простой способ сделать это:

JS

```
import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";
```

Используйте конструкцию `import`, за которой следует разделенный запятыми список функций, которые вы хотите импортировать, заключённый в фигурные скобки, за которым следует ключевое слово from, за которым следует путь к файлу модуля — путь относительно корня сайта, который для нашего примера `basic-modules` будет равен `/js-examples/modules/basic-modules`.

Однако, мы написали путь немного иначе — мы используем (`.`) синтаксис, означающий "текущую директорию", за которым следует путь к файлу, который мы пытаемся найти. Это намного лучше, чем каждый раз записывать весь относительный путь, поскольку он короче и делает URL-адрес переносимым - пример все равно будет работать, если вы переместите его в другое место в иерархии сайта.

Так например:

```
/js-examples/modules/basic-modules/modules/square.js
```

становится

```
./modules/square.js
```

Вы можете найти подобные строки кода в файле `main.js` .

> Примечание: В некоторых модульных системах вы можете опустить расширение файла и начальные `/`, `./`, or `../` (например `'modules/square'`). Это не работает в нативных JavaScript-модулях.

После того, как вы импортировали функции в свой скрипт, вы можете использовать их так же, как если бы они были определены в этом же файле. Следующий пример можно найти в `main.js`, сразу за строками импорта:

JS

```
let myCanvas = create("myCanvas", document.body, 480, 320);
let reportList = createReportList(myCanvas.id);

let square1 = draw(myCanvas.ctx, 50, 50, 100, "blue");
reportArea(square1.length, reportList);
reportPerimeter(square1.length, reportList);
```

> Примечание: Хотя импортированные функции доступны в файле, они доступны только для чтения. Вы не можете изменить импортированную переменную, но вы всё равно можете изменять свойства у `const` -переменных. Кроме того, переменные импортируется как "live bindings" - это означает, что они могут меняться по значению, даже если вы не можете изменить привязку, в отличие от `const`.

Добавление модуля на HTML-страницу

Далее нам необходимо подключить модуль `main.js` на нашу HTML-страницу. Это очень похоже на то, как мы подключаем обычный скрипт на страницу, с некоторыми заметными отличиями.

Прежде всего, вам нужно добавить `type="module"` в `<script>` -элемент, чтобы объявить, что скрипт является модулем. Чтобы подключить модуль `main.js` , нужно написать следующее:

HTML

```
<script type="module" src="main.js"></script>
```

Вы также можете встроить скрипт модуля непосредственно в HTML-файл, поместив JavaScript-код внутрь `<script>` -элемента:

JS

```
<script type="module">/* код JavaScript модуля */</script>
```

Скрипт, в который вы импортируете модуль, в основном действует как модуль верхнего уровня. Если вы упустите это, то Firefox, например, выдаст ошибку "SyntaxError: import declarations may only appear at top level of a module".

Вы можете использовать `import` и `export` инструкции только внутри модулей, внутри обычных скриптов они работать не будут.

Другие отличия модулей от обычных скриптов

- Вы должны быть осторожны во время локального тестирование — если вы попытаетесь загрузить файл HTML локально (то есть по `file://` URL), вы столкнётесь с ошибками CORS из-за требований безопасности JavaScript-модулей. Вам нужно проводить тестирование через сервер.
- Также обратите внимание, что вы можете столкнуться с отличным от обычных файлов поведением кода в модулях. Это происходит из-за того, что модули используют strict mode автоматически.
- Нет необходимости использовать атрибут `defer` (см. атрибуты `<script>` элемента) при загрузке модуля, модули являются deferred по умолчанию.
- Модули выполняются только один раз, даже если на них есть ссылки в нескольких `<script>` тэгах.
- И последнее, но не менее важное: функциональность модуля импортируются в область видимости одного скрипта, она недоступны в глобальной области видимости. Следовательно, вы сможете получить доступ к импортированным частям модуля только в скрипте, в который он импортирован, и, например, вы не сможете получить к нему доступ из консоли JavaScript. Вы по-прежнему будете получать синтаксические ошибки в DevTools, но вы не сможете использовать некоторые методы отладки, которые, возможно, ожидали использовать.

Экспорт по умолчанию против именованного экспорта

Экспорты функций и переменных, которые мы использовали в примерах выше являются именованными экспортами — каждый элемент (будь то функция или `const` -переменная, например) упоминается по имени при экспорте, и это имя также используется для ссылки на него при импорте.

Существует также тип экспорта, который называется экспорт по умолчанию — он существует для удобного экспорта основной функции, а также помогает модулям JavaScript взаимодействовать с существующими модульными системами CommonJS и AMD (это хорошо объясняется в статье Джейсона Орендорфа ES6 в деталях: Модули , ищите по ключевому слову «Default exports»).

Давайте посмотрим на пример и объясним, как это работает. В модуле `square.js` из нашего примера вы можете найти функцию `randomSquare()` , которая создаёт квадрат со случайным цветом, размером и координатами. Мы хотим экспортировать эту функции по умолчанию, поэтому в конце файла пишем следующее:

JS

```
export default randomSquare;
```

Обратите внимание на отсутствие фигурных скобок.

Кстати, можно было бы определить функцию как анонимную и добавить к ней `export default` :

JS

```
export default function(ctx) {
  ...
}
```

В нашем файле `main.js` мы импортируем функцию по умолчанию, используя эту строку:

JS

```
import randomSquare from "./modules/square.js";
```

Снова обратите внимание на отсутствие фигурных скобок. Такой синтаксис допустим, поскольку для каждого модуля разрешен только один экспорт по умолчанию, и мы знаем, что это `randomSquare`. Вышеупомянутая строка является сокращением для:

JS

```
import { default as randomSquare } from "./modules/square.js";
```

> Примечание: «as» синтаксис для переименования экспортируемых элементов поясняется ниже в разделе Переименование импорта и экспорта.

Как избежать конфликтов имён

Пока что наши модули для рисования фигур на холсте работают нормально. Но что произойдёт, если мы попытаемся добавить модуль, который занимается рисованием другой фигуры, например круга или треугольника? С этими фигурами, вероятно, тоже будут связаны такие функции, как `draw()`, `reportArea()` и т.д.; если бы мы попытались импортировать разные функции с одним и тем же именем в один и тот же файл модуля верхнего уровня, мы бы столкнулись с конфликтами и ошибками.

К счастью, есть несколько способов обойти это. Мы рассмотрим их в следующих разделах.

Переименование импорта и экспорта

Можно изменять имя функциональности в целевом модуле с помощью ключевого слова `as` внутри фигурных скобок инструкций `import` и `export`.

Так, например, оба следующих элемента будут выполнять одну и ту же работу, хотя и немного по-разному:

JS

```
// внутри module.js
export { function1 as newFunctionName, function2 as anotherNewFunctionName };

// внутри main.js
import { newFunctionName, anotherNewFunctionName } from "./modules/module.js";
```

JS

```
// внутри module.js
export { function1, function2 };

// внутри main.js
import {
  function1 as newFunctionName,
  function2 as anotherNewFunctionName,
} from "./modules/module.js";
```

Давайте посмотрим на реальный пример. В нашей renaming директории вы увидите ту же модульную систему, что и в предыдущем примере, за исключением того, что мы добавили модули `circle.js` и `triangle.js` для рисования кругов и треугольников и создания отчетов по ним.

Внутри каждого из этих модулей у нас есть функции с одинаковыми именами, которые экспортируются, и поэтому у каждого из них есть один и тот же оператор `export` внизу файла:

```
JS
```

```
export { name, draw, reportArea, reportPerimeter };
```

Если бы в `main.js` при их импорте мы попытались использовать

```
JS
```

```
import { name, draw, reportArea, reportPerimeter } from "./modules/square.js";
import { name, draw, reportArea, reportPerimeter } from "./modules/circle.js";
import { name, draw, reportArea, reportPerimeter } from "./modules/triangle.js";
```

то браузер выдал бы ошибку — "SyntaxError: redeclaration of import name" (Firefox).

Вместо этого нам нужно переименовать импорт, чтобы он был уникальным:

```
JS
```

```
import {
  name as squareName,
  draw as drawSquare,
  reportArea as reportSquareArea,
  reportPerimeter as reportSquarePerimeter,
} from "./modules/square.js";

import {
  name as circleName,
  draw as drawCircle,
  reportArea as reportCircleArea,
  reportPerimeter as reportCirclePerimeter,
} from "./modules/circle.js";

import {
  name as triangleName,
  draw as drawTriangle,
  reportArea as reportTriangleArea,
  reportPerimeter as reportTrianglePerimeter,
} from "./modules/triangle.js";
```

Обратите внимание, что вместо этого вы можете решить проблему в файлах модуля, например.

```
JS
```

```
// внутри square.js
export {
  name as squareName,
  draw as drawSquare,
  reportArea as reportSquareArea,
  reportPerimeter as reportSquarePerimeter,
};
```

```
JS
```

```
// внутри main.js
import {
  squareName,
  drawSquare,
  reportSquareArea,
  reportSquarePerimeter,
} from "./modules/square.js";
```

И это сработало бы точно так же. Какой способ вы будете использовать, зависит от вас, однако, возможно, имеет смысл оставить код модуля в покое и внести изменения в импорт. Это особенно важно, когда вы импортируете из сторонних модулей, над которыми у вас нет никакого контроля.

Создание объекта модуля

Вышеупомянутый способ работает нормально, но он немного запутан и многословен. Существует решение получше — импортировать функции каждого модуля внутри объекта модуля. Для этого используется следующая синтаксическая форма:

```
JS
import * as Module from "./modules/module.js";
```

Эта конструкция берёт все экспорты, доступные внутри `module.js` и делает их доступными в качестве свойств объекта `Module`, фактически давая ему собственное пространство имен. Так например:

```
JS
Module.function1();
Module.function2();
```

и т.д.

Опять же, давайте посмотрим на реальный пример. Если вы посмотрите на нашу директорию [module-objects](), вы снова увидите тот же самый пример, но переписанный с учётом преимуществ этого нового синтаксиса. В модулях все экспорты представлены в следующей простой форме:

```
JS
export { name, draw, reportArea, reportPerimeter };
```

С другой стороны, импорт выглядит так:

```
JS
import * as Canvas from "./modules/canvas.js";

import * as Square from "./modules/square.js";
import * as Circle from "./modules/circle.js";
import * as Triangle from "./modules/triangle.js";
```

В каждом случае теперь вы можете получить доступ к импорту модуля под указанным свойством объекта, например:

```
JS
let square1 = Square.draw(myCanvas.ctx, 50, 50, 100, "blue");
Square.reportArea(square1.length, reportList);
Square.reportPerimeter(square1.length, reportList);
```

Таким образом, теперь вы можете написать код точно так же, как и раньше (при условии, что вы включаете имена объектов там, где это необходимо), и импорт будет намного более аккуратным.

Модули и классы

Как мы намекали ранее, вы также можете экспортировать и импортировать классы — это ещё один способ избежать конфликтов в вашем коде, и он особенно полезен, если у вас уже есть код модуля, написанный в объектно-ориентированном стиле.

Вы можете увидеть пример нашего модуля для рисования фигур, переписанного с помощью классов ES в нашей директории [classes](). В качестве примера, файл [square.js]() теперь содержит всю свою функциональность в одном классе:

```
JS
class Square {
  constructor(ctx, listId, length, x, y, color) {
    ...
  }

  draw() {
    ...
  }

  ...
}
```

который мы затем экспортируем:

```
JS
export { Square };
```

Далее в `main.js` , мы импортируем его так:

```
JS
import { Square } from "./modules/square.js";
```

А затем используем импортированный класс, чтобы нарисовать наш квадрат:

```
JS
let square1 = new Square(myCanvas.ctx, myCanvas.listId, 50, 50, 100, "blue");
square1.draw();
square1.reportArea();
square1.reportPerimeter();
```

Агрегирующие модули

Возможны случаи, когда вы захотите объединить модули вместе. У вас может быть несколько уровней зависимостей, где вы хотите упростить вещи, объединив несколько подмодулей в один родительский модуль. Это возможно с использованием следующего синтаксиса экспорта в родительском модуле:

```
JS
export * from "x.js";
export { name } from "x.js";
```

Для примера посмотрите на нашу директорию `module-aggregation` . В этом примере (на основе нашего предыдущего примера с классами) у нас есть дополнительный модуль с именем `shapes.js` , который собирает функциональность `circle.js` , `square.js` и `triangle.js` вместе. Мы также переместили наши подмодули в дочернюю директорию внутри директории `modules` под названием `shape` . Итак, структура модуля в этом примере:

```
modules/
  canvas.js
  shapes.js
  shapes/
    circle.js
    square.js
    triangle.js
```

В каждом из подмодулей экспорт имеет одинаковую форму, например:

```
JS
```

```
export { Square };
```

Далее идет агрегирование. Внутри `shapes.js` , мы добавляем следующие строки:

```
JS
export { Square } from "./shapes/square.js";
export { Triangle } from "./shapes/triangle.js";
export { Circle } from "./shapes/circle.js";
```

Они берут экспорт из отдельных подмодулей и фактически делают их доступными из модуля `shape.js` .

> Примечание: Экспорты, указанные в `shape.js` , по сути перенаправляются через файл и на самом деле там не существуют, поэтому вы не сможете написать какой-либо полезный связанный код внутри того же файла.

Итак, теперь в файле `main.js` мы можем получить доступ ко всем трём классам модулей, заменив:

```
JS
import { Square } from "./modules/square.js";
import { Circle } from "./modules/circle.js";
import { Triangle } from "./modules/triangle.js";
```

на единственную строку кода:

```
JS
import { Square, Circle, Triangle } from "./modules/shapes.js";
```

Динамическая загрузка модулей

Самая свежая возможность JavaScript-модулей доступная в браузерах, — это динамическая загрузка модулей. Это позволяет вам динамически загружать модули только тогда, когда они необходимы, вместо того, чтобы загружать всё заранее. Это даёт очевидные преимущества в производительности — давайте продолжим читать и посмотрим, как это работает.

Поддержка динамической загрузки модулей позволяет вызывать `import()` в качестве функции, передав ей аргументом путь к модулю. Данный вызов возвращает `Promise` , который резолвится объектом модуля (см. `Создание объекта модуля`), предоставляя вам доступ к экспорту указанного модуля, например:

```
JS
import("./modules/myModule.js").then((module) => {
  // Делаем что-нибудь с импортированным модулем
});
```

Давайте посмотрим на пример. В директории `dynamic-module-imports` у нас есть ещё один пример, основанный на примере наших классов. Однако на этот раз мы ничего не рисуем на холсте при загрузке страницы. Вместо этого мы добавляем на страницу три кнопки — «Circle», «Square» и «Triangle», которые при нажатии динамически загружают требуемый модуль, а затем используют его для рисования указанной фигуры.

В этом примере мы внесли изменения только в наши `index.html` и `main.js` — экспорт модуля остается таким же, как и раньше.

Далее в `main.js` мы взяли ссылку на каждую кнопку, используя вызов `document.querySelector()` :

```
JS
let squareBtn = document.querySelector(".square");
```

Затем мы добавляем обработчик событий на каждую кнопку, чтобы при нажатии соответствующий модуль динамически загружался и использовался для рисования фигуры:

JS

```
squareBtn.addEventListener("click", () => {
  import("./modules/square.js").then((Module) => {
    let square1 = new Module.Square(
      myCanvas.ctx,
      myCanvas.listId,
      50,
      50,
      100,
      "blue",
    );
    square1.draw();
    square1.reportArea();
    square1.reportPerimeter();
  });
});
```

Обратите внимание: поскольку выполнение Promise возвращает объект модуля, класс затем становится подкомпонентом объекта, поэтому теперь для получения доступа к конструктору нам нужно добавить к нему `Module.` , например `Module.Square(...)` .

Устранение проблем

Вот несколько советов, которые могут помочь вам, если вам не удаётся заставить ваши модули работать. Не стесняйтесь дополнять список, если найдете что-то ещё!

- Мы упоминали об этом раньше, но повторяем: `.js` -файлы должны быть загружены с MIME-type равным `text/javascript` (или любым другим JavaScript-совместимым MIME-type, но `text/javascript` является рекомендованным), в противном случае вы получите ошибку проверки MIME-type, например — "The server responded with a non-JavaScript MIME type".
- Если вы попытаетесь загрузить HTML-файл локально (то есть по ссылке `file://`), вы столкнетесь с ошибками CORS из-за требований безопасности JavaScript-модулей. Вам нужно проводить тестирование через сервер. GitHub pages идеально для этого подходят, так как отдают `.js` -файлы с нужным MIME-type.
- Поскольку `.mjs` — нестандартное расширение файла, некоторые операционные системы могут его не распознать или попытаться заменить на что-то другое. Например, мы обнаружили, что macOS незаметно добавляла `.js` в конец файлов `.mjs` , а затем автоматически скрывала расширение файла. Таким образом, все наши файлы на самом деле имели название типа `x.mjs.js` . Когда мы отключили автоматическое скрытие расширений файлов и научили macOS принимать `.mjs` , всё стало в порядке.

Смотрите также

- Использование JavaScript-модулей в вебе , от Эдди Османи и Матиаса Байненса (англ.)
- Глубокое погружение в ES-модули в картинках , от Лина Кларка на Hacks blog (англ.)
- Глубокое погружение в ES-модули в картинках , перевод на русский язык от «Веб-стандартов»
- ES6 в деталях: Модули , статья от Джейсона Орендорфа на Hacks blog (англ.)
- Изучаем JS: Модули (англ) , книга Акселя Раушмайера (англ.)

This page was last modified on 12 дек. 2023 г. by MDN contributors.