

14. Módulos

Introducción

JavaScript también especifica un sistema de módulos compatible con la mayoría de los **entornos de ejecución** (*runtimes*).

- un **módulo** suele ser un archivo, identificado por su ruta de archivo o URL.
- puedes usar las sentencias de **import** y **export** para intercambiar datos entre módulos:

```
import { foo } from "./foo.js";

// Las variables no exportadas son locales para el módulo
const b = 2;

export const a = 1;
```

- A diferencia de Haskell, Python, Java, etc., la **resolución del módulo** de JavaScript está completamente definida por el host: generalmente se basa en URL o rutas de archivos, por lo que las rutas de archivos relativas "simplemente funcionan" y son relativas a la ruta del módulo actual en lugar de alguna **ruta raíz del proyecto**.

Sin embargo, el lenguaje JavaScript no ofrece **módulos de biblioteca estándar**.

- todas las funcionalidades principales funcionan gracias a variables globales de la misma forma que **Math** e **Intl**.
- esto se debe a
 - la larga historia en la que JavaScript careció de un sistema de módulos
 - y al hecho de que optar por el sistema de módulos implica algunos cambios en la configuración del **entorno de ejecución**.
- cada entorno de ejecución puede llegar a usar un sistema de módulos diferente; por ejemplo:
 - **Node.js** utiliza el administrador de paquetes **npm** y se basa principalmente en el sistema de archivos,
 - **mientras que Deno y los navegadores están completamente basados en URL y los módulos se pueden resolver desde URLs HTTP**.

Antecedentes de los módulos

Los programas de JavaScript comenzaron siendo bastante pequeños: la mayor parte de su uso en los primeros días era para realizar tareas de secuencias de comandos aisladas, brindando un poco de **interactividad** a sus páginas web cuando era necesario, por lo que generalmente no se necesitaban secuencias de comandos grandes.

Tras el paso de los años, ahora tenemos aplicaciones completas que se ejecutan en navegadores con una gran cantidad de JavaScript, así como JavaScript que se utiliza en otros contextos (Node.js, por ejemplo).

Por lo tanto, en los últimos años tuvo sentido comenzar a pensar en proporcionar mecanismos para dividir los programas de JavaScript en **módulos separados** que se pueden importar cuando sea necesario.





- **Node.js** ha tenido esta capacidad durante mucho tiempo, y hay una serie de **frameworks** y **bibliotecas de JavaScript** que permiten el uso de módulos (por ejemplo, otros sistemas basados en módulos como **CommonJS** y **AMD** como **RequireJS** y, más recientemente, **Webpack** y **Babel**).

La buena noticia es que los navegadores modernos han comenzado a admitir la funcionalidad de los módulos de forma nativa:


- esto solo puede ser algo bueno: los navegadores pueden optimizar la carga de módulos, haciéndolo más eficiente que tener que usar una biblioteca y hacer todo ese procesamiento adicional del lado del cliente y viajes de ida y vuelta adicionales.

El uso de **módulos JavaScript nativos** depende de las sentencias de **import** y **export** ; estos son compatibles con los navegadores, como se muestra en la siguiente tabla de compatibilidad.

javascript.statements.import

















													
	Chrome 	Edge 	Firefox 	Opera 	Safari 	Chrome Android 	Firefox for Android 	Opera Android 	Safari on iOS 	Samsung Internet 	WebView Android 	Deno 	Node.js 
import	✓ 61	✓ 16	✓ 60	✓ 48	✓ 10.1	✓ 61	✓ 60	✓ 45	✓ 10.3	✓ 8.0	✓ 61	✓ 1.0	✓ 13.2.0 *...
Available in workers	✓ 80	✓ 80	✗ No *	✗ No	✓ 15	✓ 80	✗ No *	✗ No	✓ 15	✓ 13.0	✓ 80	✓ 1.0	✗ No

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


✓ Full support ✗ No support * See implementation notes.  User must explicitly enable this feature.

... Has more compatibility info.

javascript.statements.export

													
	Chrome 	Edge 	Firefox 	Opera 	Safari 	Chrome Android 	Firefox for Android 	Opera Android 	Safari on iOS 	Samsung Internet 	WebView Android 	Deno 	Node.js 
export	✓ 61	✓ 16	✓ 60	✓ 48	✓ 10.1	✓ 61	✓ 60	✓ 45	✓ 10.3	✓ 8.0	✓ 61	✓ 1.0	✓ 13.2.0 *...
default keyword with export	✓ 61	✓ 16	✓ 60	✓ 48	✓ 10.1	✓ 61	✓ 60	✓ 45	✓ 10.3	✓ 8.0	✗ No	✓ 1.0	✓ 13.2.0 *...
export * as namespace	✓ 72	✓ 79	✓ 80	✓ 60	✓ 14.1	✓ 72	✓ 80	✓ 51	✓ 14.5	✓ 11.0	✗ No	✓ 1.0	✓ 13.2.0 *...

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

✓ Full support ✗ No support * See implementation notes.  User must explicitly enable this feature.

... Has more compatibility info.

Introducción a los ejemplos

Para demostrar el uso de módulos, veremos un conjunto simple de ejemplos (código original en: <https://github.com/mdn/js-examples/tree/main/module-examples>):

- estos ejemplos muestran un conjunto simple de módulos que crean un elemento `<canvas>` en una página web y luego dibujan (y entregan información sobre) diferentes formas en el lienzo (*canvas*).
- son bastante triviales, pero se han mantenido deliberadamente simples para ilustrar claramente las ideas básicas sobre módulos.

Ejemplo de estructura básica

En el primer ejemplo tenemos una estructura de archivos de la siguiente manera:

`index.html`

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8">
    <title>Ejemplo básico de módulos en JavaScript</title>
    <style>
      canvas {
        border: 1px solid black;
      }
    </style>
    <script type="module" src="main.js"></script>
  </head>
  <body>

  </body>
</html>
```

`main.js`

```
import { crear, crearListaInforme } from './modulos/canvas.js';
import { nombre, dibujar, informeArea, informePerimetro } from './modulos/cuadrado.js';
import cuadradoAleatorio from './modulos/cuadrado.js';

let miCanvas = crear('miCanvas', document.body, 480, 320);
let listaInforme = crearListaInforme(miCanvas.id);

let cuadrado1 = dibujar(miCanvas.ctx, 50, 50, 100, 'blue');
informeArea(cuadrado1.length, listaInforme);
crearListaInforme(cuadrado1.longitud, listaInforme);

// Use the default
let cuadrado2 = cuadradoAleatorio(miCanvas.ctx);
```

`modulos/`

`canvas.js`

```
function crear(id, parent, ancho, alto) {
  let envoltorioDiv = document.createElement('div');
  let canvasElem = document.createElement('canvas');
  parent.appendChild(envoltorioDiv);
  envoltorioDiv.appendChild(canvasElem);

  envoltorioDiv.id = id;
  canvasElem.width = ancho;
  canvasElem.height = alto;

  let ctx = canvasElem.getContext('2d');

  return {
    ctx: ctx,
    id: id
  };
}

function crearListaInforme(envoltorioId) {
  let lista = document.createElement('ul');
  lista.id = envoltorioId + '-informante';
```

```

    let envoltorioCanvas = document.getElementById(envoltorioId);
    envoltorioCanvas.appendChild(lista);

    return lista.id;
}

export { crear, crearListaInforme };

```

cuadrado.js

```

const nombre = 'cuadrado';

function dibujar(ctx, longitud, x, y, color) {
    ctx.fillStyle = color;
    ctx.fillRect(x, y, longitud, longitud);

    return {
        longitud: longitud,
        x: x,
        y: y,
        color: color
    };
}

function random(min, max) {
    let num = Math.floor(Math.random() * (max - min)) + min;
    return num;
}

function informeArea(longitud, idLista) {
    let elementoLista = document.createElement('li');
    elementoLista.textContent = `El area de ${nombre} es ${ longitud * longitud}px cuadrados.`

    let lista = document.getElementById(idLista);
    lista.appendChild(elementoLista);
}

function informePerimetro(longitud, idLista) {
    let elementoLista = document.createElement('li');
    elementoLista.textContent = `El perimetro de ${nombre} es ${ longitud * longitud}px.`

    let lista = document.getElementById(idLista);
    lista.appendChild(elementoLista);
}

function cuadradoAleatorio(ctx) {
    let color1 = random(0, 255);
    let color2 = random(0, 255);
    let color3 = random(0, 255);
    let color = `rgb(${color1},${color2},${color3})`;
    ctx.fillStyle = color;

    let x = random(0, 480);
    let y = random(0, 320);
    let longitud = random(10, 100);
    ctx.fillRect(x, y, longitud, longitud);

    return {
        longitud: longitud,
        x: x,
        y: y,
        color: color
    };
}

export { nombre, dibujar, informeArea, informePerimetro };
export default cuadradoAleatorio;

```

Los dos módulos del directorio de módulos se describen a continuación:

- `canvas.js`: contiene funciones relacionadas con la configuración del lienzo:
 - `crear()`:
 - crea un lienzo con un ancho y una altura específicos dentro de un envoltorio `<div>` con una ID específica, que a su vez se agrega dentro de un elemento principal específico.

- devuelve un objeto que contiene el contexto 2D del lienzo y el ID del envoltorio.
- `crearListaInformacion()` :
 - crea una lista desordenada adjunta dentro de un elemento contenedor específico, que se puede usar para generar datos de informes.
 - devuelve el ID de la lista.
- `cuadrado.js` — contiene:
 - `nombre` :
 - una constante que contiene la cadena 'cuadrado'.
 - `dibujar()` :
 - dibuja un cuadrado en un lienzo específico, con un tamaño, posición y color específicos.
 - devuelve un objeto que contiene el tamaño, la posición y el color del cuadrado.
 - `informeArea()`
 - escribe el área de un cuadrado en una lista de informes específica, dada su longitud.
 - `informePerimetro()` :
 - escribe el perímetro de un cuadrado en una lista de informes específica, dada su longitud.

Nota: `.mjs` frente a `.js`

Usaremos extensiones `.js` para nuestros **archivos de módulo**, pero en otros recursos podrás ver que se usa la extensión `.mjs` en su lugar.

- la documentación de V8 recomienda esto, por ejemplo.
- las razones dadas son:
 - es bueno para la claridad, es decir, deja claro qué archivos son módulos y cuáles son JavaScript normal.
 - garantiza que los archivos de su módulo sean analizados como un módulo por entornos de ejecución como Node.js y herramientas de compilación como Babel.

Sin embargo, en estos ejemplos seguiremos usando `.js`, al menos por el momento.

- para que los módulos funcionen correctamente en un navegador, debe asegurarse de que su servidor los proporcione con un **encabezado de tipo de contenido** que contenga un **tipo MIME** de JavaScript, como `text/javascript`.
- si no lo hace, obtendrá un error de verificación de tipo MIME estricto del tipo "El servidor respondió con un tipo MIME que no es de JavaScript" y el navegador no ejecutará su JavaScript.
- la mayoría de los servidores ya establecieron el tipo correcto para los archivos `.js`, pero aún no para los archivos `.mjs`.
- entre los servidores que ya entregan archivos `.mjs` correctamente se incluyen [GitHub](#) y [el servidor http](#) para Node.js.
- sin embargo, podría causar confusión si no controlas el servidor desde el que entregas los archivos o si publicas archivos para uso público, como estamos aquí.
- por motivos de aprendizaje y portabilidad, decidimos quedarnos por ahora con la extensión `.js`.

También vale la pena señalar que:

- es posible que algunas herramientas nunca admitan `.mjs`.
- utilizamos el atributo `type` `<script type="module">` para indicar cuándo se apunta a un módulo, como verás a continuación.

Exportación de prestaciones desde un módulo

Lo primero que debes hacer para obtener acceso a las prestaciones que ofrece un módulo es exportarlas desde dicho módulo.

- esto se hace usando la sentencia **export**.
- la forma más sencilla de usarla es colocar **export** delante de cualquier elemento que desee exportar fuera del módulo, por ejemplo:

```
export const nombre = 'cuadrado';

export function dibujar(ctx, longitud, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, longitud, longitud);

  return { longitud, x, y, color };
}
```

Puedes exportar declaraciones **function**, **var**, **let**, **const** y **class**

- deben ser elementos de nivel superior; no puede usar exportar dentro de una función, por ejemplo.

Una forma más conveniente de exportar todos los elementos que desea exportar es usar una única declaración de exportación al final de tu **archivo de módulo**, seguida de una lista separada por comas de las prestaciones que deseas exportar entre llaves.

- por ejemplo:

```
export { nombre, dibujar, informeArea, informePerimetro };
```

Importación de características en su secuencia de comandos

Una vez que hayas exportado algunas funciones de tu módulo, si las quieres usar en otro script tendrás que **importarlas**:

- la forma más sencilla de hacer esto es la siguiente:

```
import { nombre, dibujar, informeArea, informePerimetro } from './modulos/cuadrado.js';
```

La declaración **import** se usa:

- seguida de una lista separada por comas de las funciones que deseas importar, entre llaves,
- seguido de la palabra clave **from**,
- seguido de
 - la ruta al archivo del módulo,
 - o una ruta relativa a la raíz del sitio,que para nuestro ejemplo de módulos básicos sería `/ejemplos-js /ejemplos-modulos /modulos-basicos`

Sin embargo, la ruta anterior aparece un poco diferente: usamos la sintaxis de punto (`.`) para indicar "la ubicación actual", seguida de la ruta al archivo que estamos tratando de encontrar.

- es más práctico que escribir la ruta relativa completa cada vez, ya que es más corta y hace que la URL sea portátil.
- el ejemplo seguirá funcionando si lo mueves a una ubicación diferente en la jerarquía del sitio web.
- así por ejemplo:

```
/ejemplos-js/ejemplos-modulos/modulos-basicos/modulos/cuadrado.js
```

se convierte en

```
/modulos/cuadrado.js
```

Puede ver esas líneas en acción en **main.js**.

Nota: en algunos sistemas de módulos, puedes omitir la extensión de archivo y la barra `/`, `.` / o `../` inicial (p. ej., `'modulos/cuadrado'`).

- esto no funciona en el entorno del navegador, ya que puede dar lugar a múltiples viajes de ida y vuelta a la red.

Una vez que haya importado las prestaciones de otro módulo en tu script, puede usarlas tal y como se hubieran definido dentro del archivo importador.

- lo siguiente se encuentra en `main.js`, debajo de las líneas de importación:

```
const miCanvas = create('miCanvas', document.body, 480, 320);
const listaInforme = crearListaInformacion(miCanvas.id);

const cuadrado1 = dibujar(miCanvas.ctx, 50, 50, 100, 'blue');
informeArea(cuadrado1.longitud, listaInforme);
informePerimetro(cuadrado1.longitud, listaInforme);
```

Nota: aunque las funciones importadas están disponibles en el archivo, son **vistas de solo lectura** de la función que se exportó.

- no puedes cambiar la variable que se importó, pero aún puede modificar propiedades de forma similar a lo que pasa con **const**.
- además, estas características se importan como **ligaduras vivas** (*live bindings*), lo que significa que pueden cambiar de valor incluso si no puede modificar el enlace a diferencia de lo que ocurre con una declaración **const**

Aplicando el módulo a tu HTML

Ahora solo necesitamos aplicar el módulo `main.js` a nuestra página HTML.

- esto es muy similar a cómo aplicamos un script regular a una página, con algunas diferencias notables.
- en primer lugar, debe incluir `type="module"` en el elemento `<script>` para declarar este script como un módulo.
 - para importar el script `main.js`, usamos esto:

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

También puedes incrustar el script del módulo directamente en el archivo HTML colocando el código JavaScript dentro del cuerpo del elemento `<script>`:

```
<script type="module">
  /* Código del módulo JavaScript aquí */
</script>
```

La secuencia de comandos en la que importas las funciones del módulo básicamente actúa como el **módulo de nivel superior**.

- si lo omite, Firefox, por ejemplo, le da un error de "SyntaxError: las declaraciones de importación solo pueden aparecer en el nivel superior de un módulo".
- **sólo puede usar sentencias de `import` y `export` dentro de módulos, no scripts regulares.**

Otras diferencias entre módulos y scripts estándar

- Debes prestar atención a las pruebas locales: si intentas cargar el archivo HTML localmente (es decir, con una URL `file://`), te encontrarás con **errores CORS** debido a los requisitos de seguridad del módulo JavaScript: tienes que hacer tus pruebas a través de un servidor.

Intercambio de recursos de origen cruzado (CORS) es un estándar que permite a un servidor relajar la política del mismo origen.

- esto se usa para permitir explícitamente algunas solicitudes de origen cruzado mientras se rechazan otras.
- por ejemplo, si un sitio ofrece un servicio embebible, puede ser necesario relajar ciertas restricciones.
- establecer una configuración CORS de este tipo no es necesariamente fácil y puede presentar algunos desafíos.
- más adelante, veremos algunos mensajes de error comunes de CORS y cómo resolverlos.

- además, ten en cuenta que puedes obtener un comportamiento diferente de las secciones del script definidas dentro de los módulos en comparación con los scripts estándar.
 - **esto se debe a que los módulos usan el modo estricto automáticamente**
- no es necesario usar el atributo `defer` (ver atributos de `<script>`) al cargar un script de módulo: a carga de **módulos se aplaza automáticamente.**
- **los módulos solo se ejecutan una vez, incluso si se les ha hecho referencia en varias etiquetas `<script>`.**
- por último, pero no menos importante, aclaremos esto: las características del módulo se importan al **ámbito** de un solo script; no están disponibles en el **ámbito global**.
 - por lo tanto, solo podrá acceder a las funciones importadas en la secuencia de comandos en la que se importaron y no podrá acceder a ellas desde la consola de JavaScript, por ejemplo.
 - seguirás viendo los errores de sintaxis que se produzcan en DevTools, pero no podrás usar algunas de las técnicas de depuración que podrías haber esperado usar.
- las variables definidas por el módulo están en el **ámbito del módulo** a menos que se adjunten explícitamente al **objeto global**.
- las variables definidas globalmente están disponibles dentro del módulo.

Por ejemplo, dado el siguiente código:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="UTF-8" />
    <title></title>
    <link rel="stylesheet" href="" />
  </head>
  <body>
    <div id="principal"></div>
    <script>
      // Una declaración var crea una variable global.
      var texto = "Hola";
    </script>
```

```
<script type="module" src="./renderizar.js"></script>
</body>
</html>
```

```
/* renderizar.js */
```

```
document.getElementById("principal").innerText = texto;
```

La página aún mostraría Hola, porque el `texto` y el `document` de las variables globales están disponibles en el módulo.

- (también ten en cuenta en este ejemplo que un módulo no necesita necesariamente una sentencia de `import/export`: lo único que se necesita es que el **punto de entrada** (*entry point*) tenga `type="module"`).

Exportaciones predeterminadas comparadas con exportaciones con nombre

La funcionalidad que hemos exportado hasta ahora se compone de **exportaciones con nombre**: cada elemento (ya sea una función, const, etc.) se ha mencionado por su identificador en la exportación, y ese mismo identificador se ha utilizado para referirse a él en la importación también.

También hay un tipo de exportación llamado **exportación predeterminada** (*default export*)

- está diseñada para que sea fácil tener una función predeterminada proporcionada por un módulo, y también ayuda a los módulos de JavaScript a interoperar con los sistemas de módulos **CommonJS** y **AMD existentes**

Veamos un ejemplo mientras explicamos cómo funciona.

- en nuestro ejemplo básico de módulos `cuadrado.js` podremos encontrar una función llamada `cuadradoAleatorio()` que crea un cuadrado con un color, tamaño y posición aleatorios.
- queremos exportar esto como nuestro valor predeterminado, por lo que en la parte inferior del archivo escribimos esto:

```
export default cuadradoAleatorio;
```

- ten en cuenta la falta de llaves.

Otra opción sería poner como prefijo la declaración `export default` a la declaración de la función y definirla como una función anónima, así:

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

Por otro lado, en el archivo `main.js`, importamos la función predeterminada usando esta línea:

```
import cuadradoAleatorio from './modulos/cuadrado.js';
```

Nuevamente, ten en cuenta la falta de llaves.

- esto se debe a que solo se permite una exportación predeterminada por módulo, y sabemos que `cuadradoAleatorio` lo es.
- la línea anterior es básicamente una abreviatura de:

```
import {default as cuadradoAleatorio} from './modulos/cuadrado.js';
```

Evitar conflictos de nombres

Hasta ahora, nuestros módulos de dibujo de formas de lienzo parecen funcionar bien.

- pero, ¿qué sucede si tratamos de agregar un módulo que trata de dibujar otra forma, como un círculo o un triángulo?
- estas formas probablemente también tendrían funciones asociadas como `dibujar()`, `informeArea()`, etc;
- si intentáramos importar diferentes funciones del mismo nombre en el mismo módulo de nivel superior archivo, terminaríamos con **conflictos** y **errores**.

Afortunadamente, hay varias maneras de evitar esto. Los veremos en las siguientes secciones.

Cambio de nombre de importaciones y exportaciones

Dentro de las llaves de las declaraciones de importación y exportación, puedes usar la palabra clave `as` junto con identificador de prestación para cambiar el nombre de identificación que usarás para la prestación dentro del módulo de nivel superior.

- entonces, por ejemplo, los dos siguientes harían el mismo trabajo, aunque de una manera ligeramente diferente:

```
// dentro de modulo.js
export {
  function1 as nuevoNombreFuncion,
  function2 as otroNuevoNombreFuncion
};

// dentro de main.js
import { nuevoNombreFuncion, otroNuevoNombreFuncion } from './modulos/modulo.js';
```

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

// dentro de main.js
import {
  function1 as nuevoNombreFuncion,
  function2 as otroNuevoNombreFuncion,
} from './modulos/modulo.js';
```

Veamos un ejemplo real.

- ahora añadimos 2 módulos nuevos al ejemplo `circulo.js` y `triangulo.js` para dibujar e informar sobre círculos y triángulos.
- dentro de cada uno de estos módulos, tenemos funciones con los mismos nombres que se exportan y, por lo tanto, cada una tiene la misma declaración de exportación en la parte inferior:

```
export { nombre, dibujar, informeArea, informePerimetro };
```

Al importarlos a `main.js`, si intentamos usar

```
import { nombre, dibujar, informeArea, informePerimetro } from './modulos/cuadrado.js';
import { nombre, dibujar, informeArea, informePerimetro } from './modulos/circulo.js';
import { nombre, dibujar, informeArea, informePerimetro } from './modulos/triangulo.js';
```

El navegador mostraría un error como `"SyntaxError: redeclaration of import name "` (Firefox).

En su lugar, debemos cambiar el nombre de las importaciones para que sean únicas:

```
import {
  nombre          as nombreCuadrado,
  dibujar          as dibujarCuadrado,
  informeArea      as informeAreaCuadrado,
  informePerimetro as informePerimetroCuadrado,
} from './modulos/cuadrado.js';

import {
  nombre          as nombreCirculo,
  dibujar          as dibujarCirculo,
  informeArea      as informeAreaCirculo,
  informePerimetro as informePerimetroCirculo,
} from './modulos/circulo.js';

import {
  nombre          as nombreTriangulo,
  dibujar          as dibujarTriangulo,
  informeArea      as informeAreaTriangulo,
  informePerimetro as informePerimetroTriangulo,
} from './modulos/triangulo.js';
```

Ten en cuenta que podrías resolver el problema en los archivos del módulo, por ejemplo, y funcionaría también bien:

```
// En cuadrado.js
export {
  nombre          as nombreCuadrado,
  dibujar         as dibujarCuadrado,
  informeArea     as informeAreaCuadrado,
  informePerimetro as informePerimetroCuadrado,
};

// En main.js
import { nombreCuadrado, dibujarCuadrado, informeAreaCuadrado, informePerimetroCuadrado } from
'./modulos/cuadrado.js';
```

El estilo que uses depende de ti,

- sin embargo, podría decirse que tiene más sentido no tocar el código de tu módulo y realizar los cambios en las importaciones.
- esto tiene sentido especialmente cuando estás importando desde módulos de terceros sobre los que no tienes ningún control.

Creación de un objeto de módulo

El método anterior funciona bien, pero es un poco complicado y prolijo.

Una solución aún mejor es importar las prestaciones de cada módulo dentro de un **objeto de módulo**.

- la siguiente forma de sintaxis hace eso:

```
import * as Modulo from './modulos/modulo.js';
```

- esto toma todas las exportaciones disponibles dentro de `modulo.js` y las pone a disposición como miembros de un **módulo de objeto**, dándole efectivamente su propio **espacio de nombres**.
- así por ejemplo:

```
Modulo.funcion1();
Modulo.funcion2();
```

Una vez más, veamos un ejemplo real.

- podemos cambiar nuestros módulos anteriores, para utilizar esta nueva sintaxis
- en los módulos, las exportaciones están todas en la siguiente forma simple:

```
export { nombre, dibujar, informeArea, informePerimetro };
```

Las importaciones, por otro lado, se ven así:

```
import * as Canvas    from './modulos/canvas.js';
import * as Cuadrado  from './modulos/cuadrado.js';
import * as Circulo   from './modulos/circulo.js';
import * as Triangulo from './modulos/triangulo.js';
```

En cada caso, ahora puede acceder a las importaciones del módulo debajo del nombre de objeto especificado, por ejemplo:

```
const cuadrado1 = Cuadrado.dibujar(miCanvas.ctx, 50, 50, 100, 'blue');
Cuadrado.informeArea(cuadrado1.longitud, listaInforme);
Cuadrado.informePerimetro(cuadrado1.longitud, listaInforme);
```

Así que ahora puedes escribir el código igual que antes (siempre y cuando incluyas los nombres de los objetos donde sea necesario), y las importaciones son mucho más claras.

Módulos y clases

También puede exportar e importar clases: esta es otra opción para evitar conflictos en su código, y es especialmente útil si ya tienes el código de tu módulo escrito en un estilo orientado a objetos.

Si tenemos un módulo que implementa como una clase por cada forma Cuadrado, Circulo, Triangulo y el Canvas, en los módulos que implementen las clases pondremos las sentencias export:

./modulos/canvas.js

```
class Canvas {
  ....
}

export { Canvas };
```

./modulos/cuadrado.js

```
class Cuadrado {
  ....
}

export { Cuadrado };
```

./modulos/circulo.js

```
class Circulo {
  ....
}

export { Circulo };
```

./modulos/triangulo.js

```
class Triangulo {
  ....
}

export { Triangulo };
```

En main.js, lo importamos así:

```
import {Canvas}    from './modulos/canvas.js';
import {Cuadrado}  from './modulos/cuadrado.js';
import {Circulo}   from './modulos/circulo.js';
import {Triangulo} from './modulos/triangulo.js';
```

Agregar módulos

Alguna vez necesitarás **agregar módulos** juntos.

- es posible que tenga múltiples niveles de **dependencias**, donde desees simplificar las cosas, combinando varios submódulos en un módulo principal.
- esto es posible utilizando la siguiente sintaxis de exportación en el **módulo principal**:

```
export * from 'x.js'
```

```
export { nombre } from 'x.js'
```

Por ejemplo, podemos tener la siguiente estructura de módulos:

```
modulos/
  canvas.js
  formas.js <---- agrega la funcionalidad de circulo.js, cuadrado.js y triangulo.js
  formas/
    circulo.js
    cuadrado.js
    triangulo.js
```

En cada uno de los **submódulos**, la exportación es de la misma forma, por ejemplo

```
export { Canvas };
export { Cuadrado };
export { Circulo };
export { Triangulo };
```

A continuación viene la parte de **agregación**:

- dentro de `formas.js`, incluimos las siguientes líneas:

```
export {Cuadrado} from './formas/cuadrado.js';
export {Circulo} from './formas/circulo.js';
export {Triangulo} from './formas/triangulo.js';
```

- estas declaraciones toman las exportaciones de los submódulos individuales y los hacen disponibles desde el módulo `formas.js`.

Nota: las exportaciones a las que se hace referencia en `shape.js` básicamente se redireccionan a través del archivo y en realidad no existen allí, por lo que no podrá escribir ningún código relacionado útil dentro del mismo archivo.

Entonces, ahora en el archivo `main.js`, podemos obtener acceso a las tres clases de módulos simplemente con:

```
import { Canvas } from './modulos/canvas.js';

import { Cuadrado, Circulo, Triangulo } from './modulos/formas.js';
```

Carga dinámica de módulos

Una adición reciente a la funcionalidad de los módulos de JavaScript es la **carga dinámica de módulos**.

- esto te permite cargar módulos dinámicamente solo cuando y si llegan a ser necesarios, en lugar de tener que cargar todo por adelantado.
- esto tiene algunas ventajas de rendimiento obvias.

Esta nueva funcionalidad te permite llamar a `import()` como una función, pasándole la ruta al módulo como parámetro.

- la llamada devuelve una **Promise**, que se completa con un **objeto módulo el cuál** le da acceso a las exportaciones de ese objeto, por ejemplo

```
import('./modulos/miModulo.js')
  .then((modulo) => {
    // Hacer algo con el módulo
  });
```

Otra ventaja de las importaciones dinámicas es que siempre están disponibles, incluso en entornos de script.

- por lo tanto, si tienes una etiqueta `<script>` existente en tu HTML que no tiene `type="module"`, aún puede reutilizar el código distribuido como módulos importándolo dinámicamente.

```
<script>
  import("./modulos/cuadrado.js").then((modulo) => {
    // Hacer algo con el módulo.
  });
  // Otro código que opera en el ámbito global y no está
  // listo para ser refactorizado en módulos todavía.
  var boton = document.querySelector(".cuadrado");
</script>
```

El nivel superior espera

La **espera de nivel superior** (*top level await*) es una característica disponible dentro de los módulos.

- esto significa que se puede usar la palabra clave **await**.
- permite que los módulos actúen como grandes **funciones asíncronas**, lo que significa que el código se puede evaluar antes

de usarlo en los módulos principales, pero sin bloquear la carga de los módulos hermanos.

Por ejemplo, supongamos que tenemos un archivo `colores.json` con nuestra paleta de colores:

```
{
  "amarillo": "#F4D03F",
  "verde": "#52BE80",
  "azul": "#5499C7",
  "rojo": "#CD6155",
  "naranja": "#F39C12"
}
```

Luego, crearemos un módulo llamado `obtenerColores.js` que usa una **solicitud fetch** para cargar el archivo `colores.json` y devolver los datos como un objeto.

```
// buscar solicitud
const colores = fetch ('../datos/colores.json').then((respuesta) => respuesta.json());

export default await colores;
```

Observa la última línea de exportación:

- estamos usando la palabra clave `await` antes de especificar la constante `colores` para exportar.
- esto significa que cualquier otro módulo que incluya este, antes de usarlo, esperará hasta que los colores se hayan descargado y analizado antes de usarlo.

Podemos incluir este módulo en nuestro archivo `main.js`:

```
import colores from './modulos/obtenerColores.js';
import { Canvas } from './modulos/canvas.js';

const botonCirculo = document.querySelector('.circulo');
```

Para hacer uso de los valores del módulo `obtenerColores.js` pondremos como prefijo `colores` así:

```
colores.amarillo
colores.verde
```

Esto es útil porque el código dentro de `main.js` no se ejecutará hasta que se haya ejecutado el código en `obtenerColores.js`.

- sin embargo, no bloqueará la carga de otros módulos.
- por ejemplo, nuestro módulo `canvas.js` continuará cargándose mientras se recuperan los colores.

Creación de módulos "isomorfos"

La introducción de módulos anima al **ecosistema de JavaScript** a distribuir y reutilizar el código de forma modular.

- sin embargo, eso no significa necesariamente que un fragmento de código JavaScript pueda ejecutarse en todos los entornos.
 - suponga que descubrió un módulo que genera hashes SHA de la contraseña de su usuario.
 - ¿puedes usarlo en la interfaz del navegador?
 - ¿puedes usarlo en tu servidor Node.js?La respuesta es, depende.
- los módulos aún tienen acceso a las variables globales, como se demostró anteriormente.
 - si el módulo hace referencia a elementos globales como `window`, puede ejecutarse en el navegador, pero arrojará un error en su servidor Node.js, porque `window` no está disponible allí.
 - de manera similar, si el código requiere acceso a `process` para ser funcional, solo se puede usar en **Node.js**.

Para maximizar la reutilización de un módulo, a menudo se recomienda hacer que el código sea "isomorfo".

- es decir, exhibe el mismo comportamiento en cada **entorno de ejecución (runtime)**.

Esto se logra comúnmente de tres maneras:

- 1) Separa tus módulos en "**núcleo**" (*core*) y "**enlace**" (*binding*).

- para el "núcleo", concéntrate en la lógica de JavaScript puro, como calcular el hash, sin ningún DOM, red, acceso al sistema de archivos y funciones de utilidad expuestas.
 - para la parte de "enlace", puede leer y escribir en el contexto global.
 - por ejemplo, el "enlace del navegador" puede optar por leer el valor de un input HTML, mientras que el "enlace Node" puede leerlo desde `process.env`, pero los valores leídos desde cualquier lugar se canalizarán a la misma función central y se manejarán en de la misma manera
 - el núcleo se puede importar en todos los entornos y usarse de la misma manera, mientras que solo el enlace, que generalmente es liviano, debe ser **específico de la plataforma**.
- 2) Detectar si existe un **global** particular antes de usarlo.
- por ejemplo, si realizas la prueba `typeof window === "undefined"`, sabes que probablemente te encuentras en un entorno Node.js y no deberías leer DOM.

```
// miModulo.js
let contrasena;
if (typeof process !== "undefined") {
  // Este script está corriendo en Node.js; leer la contraseña de `process.env`
  contrasena = process.env.PASSWORD;
} else if (typeof window !== "undefined") {
  // Este script está corriendo en un navegador; leer la contraseña de una caja de texto
  contrasena = document.getElementById('contrasena').value;
}
```

- Esto es preferible si las dos ramas terminan con el mismo comportamiento (" **isomorfo** ").
 - si es imposible proporcionar la misma funcionalidad, o si hacerlo implica cargar cantidades significativas de código mientras una gran parte permanece sin usar, entonces es mejor usar varios archivos de "enlace", uno para cada entorno de ejecución.
- 3) Usa un **polyfill** para proporcionar un mecanismo de respaldo para las funciones que falten:
- por ejemplo, si deseas utilizar la función `fetch`, que solo se admite en Node.js desde v18, puedes utilizar una API similar, como la proporcionada por `node-fetch`.
 - puedes hacerlo condicionalmente a través de **importaciones dinámicas**:

```
// miModulo.js
if (typeof fetch === "undefined") {
  // Estamos corriendo en Node.js: usar node-fetch
  globalThis.fetch = (await import("node-fetch")).default;
}
// ...
```

- la variable `globalThis` es un **objeto global** que **está disponible en todos los entornos** y es útil si desea leer o crear variables globales dentro de los módulos.

Estas prácticas no son exclusivas para los módulos.

- aún así, con la tendencia de la **reutilización del código** y la **modularización del código**, es muy recomendable hacer que tu código sea **multiplataforma** para que pueda ser disfrutado por tantas personas como sea posible.
- los **entornos de ejecución** como **Node.js** también están implementando APIs web de forma activa con el objetivo de mejorar la interoperabilidad con la web.

Solución de problemas

Aquí hay algunos consejos que pueden ayudarte si tienes problemas para que tus módulos funcionen.

- como ya se mencionó, pero para reiterar: los archivos `.mjs` deben cargarse con un **tipo MIME** de `texto/javascript` (u otro tipo MIME compatible con JavaScript, pero se recomienda `texto/javascript`), de lo contrario obtendrá un estricto Error de verificación de tipo MIME como "El servidor respondió con un tipo MIME que no es de JavaScript".
- si intentas cargar el archivo HTML localmente (es decir, con una URL `file://`), te encontrarás con errores CORS debido a los requisitos de seguridad del módulo JavaScript.
 - necesitas hacer tus pruebas a través de un servidor.
 - las páginas de GitHub son ideales ya que también sirven archivos `.mjs` con el tipo MIME correcto.
- porque `.mjs` es una extensión de archivo no estándar, es posible que algunos sistemas operativos no la reconozcan o intenten reemplazarla por otra o que la extensión aparezca oculta.