The static **`import`** statement is used to import read only live bindings which are exported by another module. Imported modules are in `strict mode` whether you declare them as such or not. The `import` statement cannot be used in embedded scripts unless such script has a `type="module"`. Bindings imported are called live bindings because they are updated by the module that exported the binding.

There is also a function-like dynamic **`import()`**, which does not require scripts of `type="module"`.

Backward compatibility can be ensured using attribute `nomodule` on the `<script>` tag.

Syntax

```
import defaultExport from "module-name";
import * as name from "module-name";
import { export1 } from "module-name";
import { export1 as alias1 } from "module-name";
import { export1 , export2 } from "module-name";
import { foo , bar } from "module-name/path/to/specific/un-exported/file";
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export1 [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
import "module-name";
var promise = import("module-name");
```

`defaultExport`
Name that will refer to the default export from the module.

`module-name`
The module to import from. This is often a relative or absolute path name to the `.js` file containing the module. Certain bundlers may permit or require the use of the extension; check your environment. Only single quoted and double quoted Strings are allowed.

`name`
Name of the module object that will be used as a kind of namespace when referring to the imports.

`exportN`
Name of the exports to be imported.

`aliasN`
Names that will refer to the named imports.

Description

The `name` parameter is the name of the "module object" which will be used as a kind of namespace to refer to the exports. The `export` parameters specify individual named exports, while the `import * as name` syntax imports all of them. Below are examples to clarify the syntax.

Import an entire module's contents

This inserts `myModule` into the current scope, containing all the exports from the module in the file located in `/modules/my-module.js`.

```
import * as myModule from '/modules/my-module.js';
```

Here, accessing the exports means using the module name ("myModule" in this case) as a namespace. For example, if the module imported above includes an export `doAllTheAmazingThings()`, you would call it like this:

```
myModule.doAllTheAmazingThings();
```

Import a single export from a module

Given an object or value named `myExport` which has been exported from the module `my-module` either implicitly (because the entire module is exported) or explicitly (using the `export` statement), this inserts `myExport` into the current scope.

```
import {myExport} from '/modules/my-module.js';
```

Import multiple exports from module

This inserts both `foo` and `bar` into the current scope.

```
import {foo, bar} from '/modules/my-module.js';
```

Import an export with a more convenient alias

You can rename an export when importing it. For example, this inserts `shortName` into the current scope.

```
import {reallyReallyLongModuleExportName as shortName}
  from '/modules/my-module.js';
```

Rename multiple exports during import

Import multiple exports from a module with convenient aliases.

```
import {
  reallyReallyLongModuleExportName as shortName,
  anotherLongModuleName as short
} from '/modules/my-module.js';
```

Import a module for its side effects only

Import an entire module for side effects only, without importing anything. This runs the module's global code, but doesn't actually import any values.

```
import '/modules/my-module.js';
```

This works with dynamic imports as well:

```
(async () => {
  if (somethingIsTrue) {
    // import module for side effects
    await import('/modules/my-module.js');
  }
})();
```

If your project uses packages that export ESM, you can also import them for side effects only. This will run the code in the package entry point file (and any files it imports) only.

Importing defaults

It is possible to have a default `export` (whether it is an object, a function, a class, etc.). The `import` statement may then be used to import such defaults.

The simplest version directly imports the default:

```
import myDefault from '/modules/my-module.js';
```

It is also possible to use the default syntax with the ones seen above (namespace imports or named imports). In such cases, the default import will have to be declared first. For instance:

```
import myDefault, * as myModule from '/modules/my-module.js';
// myModule used as a namespace
```

or

```
import myDefault, {foo, bar} from '/modules/my-module.js';
// specific, named imports
```

When importing a default export with dynamic imports, it works a bit differently. You need to destructure and rename the "default" key from the returned object.

```
(async () => {
  if (somethingIsTrue) {
    const { default: myDefault, foo, bar } = await import('/modules/my-
  }
})();
```

Dynamic Imports

The standard import syntax is static and will always result in all code in the imported module being evaluated at load time. In situations where you wish to load a module conditionally or on demand, you can use a dynamic import instead. The following are some reasons why you might need to use dynamic import:

- When importing statically significantly slows the loading of your code and there is a low likelihood that you will need the code you are importing, or you will not need it until a later time.
- When importing statically significantly increases your program's memory usage and there is a low likelihood that you will need the code you are importing.
- When the module you are importing does not exist at load time
- When the import specifier string needs to be constructed dynamically. (Static import only supports static specifiers.)
- When the module being imported has side effects, and you do not want those side effects unless some condition is true. (It is recommended not to have any side effects in a module, but you sometimes cannot control this in your module dependencies.)

Use dynamic import only when necessary. The static form is preferable for loading initial dependencies, and can benefit more readily from static analysis tools and tree shaking.

To dynamically import a module, the `import` keyword may be called as a function. When used this way, it returns a promise.

```
import('/modules/my-module.js')
  .then((module) => {
    // Do something with the module.
  });
```

This form also supports the `await` keyword.

```
let module = await import('/modules/my-module.js');
```

Examples

Standard Import

The code below shows how to import from a secondary module to assist in processing an AJAX JSON request.

The module: file.js

```
function getJSON(url, callback) {
  let xhr = new XMLHttpRequest();
  xhr.onload = function () {
    callback(this.responseText)
  };
  xhr.open('GET', url, true);
  xhr.send();
}

export function getUsefulContents(url, callback) {
  getJSON(url, data => callback(JSON.parse(data)));
}
```

The main program: main.js

```
import { getUsefulContents } from '/modules/file.js';

getUsefulContents('http://www.example.com',
    data => { doSomethingUseful(data); });
```

Dynamic Import

This example shows how to load functionality on to a page based on a user action, in this case a button click, and then call a function within that module. This is not the only way to implement this functionality. The `import()` function also supports `await`.

```
const main = document.querySelector("main");
for (const link of document.querySelectorAll("nav > a")) {
  link.addEventListener("click", e => {
    e.preventDefault();

    import('/modules/my-module.js')
      .then(module => {
        module.loadPageInto(main);
      })
```

```
        .catch(err => {
          main.textContent = err.message;
        });
    });
  }
```