The **`export`** statement is used when creating JavaScript modules to export live bindings to functions, objects, or primitive values from the module so they can be used by other programs with the `import` statement. Bindings that are exported can still be modified locally; when imported, although they can only be read by the importing module the value updates whenever it is updated by the exporting module.

Exported modules are in strict mode whether you declare them as such or not. The export statement cannot be used in embedded scripts.

Syntax

There are two types of exports:

1. Named Exports (Zero or more exports per module)
2. Default Exports (One per module)

```
// Exporting individual features
export let name1, name2, …, nameN; // also var, const
export let name1 = …, name2 = …, …, nameN; // also var, const
export function functionName(){...}
export class ClassName {...}

// Export list
export { name1, name2, …, nameN };

// Renaming exports
export { variable1 as name1, variable2 as name2, …, nameN };

// Exporting destructured assignments with renaming
export const { name1, name2: bar } = o;

// Default exports
export default expression;
export default function (…) { … } // also class, function*
export default function name1(…) { … } // also class, function*
export { name1 as default, … };

// Aggregating modules
```

```
export * from …; // does not set the default export
export * as name1 from …; // Draft ECMAScript® 2021
export { name1, name2, …, nameN } from …;
export { import1 as name1, import2 as name2, …, nameN } from …;
export { default } from …;
```

nameN
Identifier to be exported (so that it can be imported via `import` in another script).

Description

There are two different types of export, **named** and **default**. You can have multiple named exports per module but only one default export. Each type corresponds to one of the above syntax:

Named exports:

```
// export features declared earlier
export { myFunction, myVariable };

// export individual features (can export var, let,
// const, function, class)
export let myVariable = Math.sqrt(2);
export function myFunction() { ... };
```

Default exports:

```
// export feature declared earlier as default
export { myFunction as default };

// export individual features as default
export default function () { ... }
export default class { .. }
```

Named exports are useful to export several values. During the import, it is mandatory to use the same name of the corresponding object.

But a default export can be imported with any name for example:

```
// file test.js
let k; export default k = 12;
```

```
// some other file
import m from './test'; // note that we have the freedom to use import
console.log(m);         // will log 12
```

You can also rename named exports to avoid naming conflicts:

```
export { myFunction as function1,
         myVariable as variable };
```

Re-exporting / Aggregating

It is also possible to "import/export" from different modules in a parent module so that they are available to import from that module. In other words, one can create a single module concentrating various exports from various modules.

This can be achieved with the "export from" syntax:

```
export { default as function1,
         function2 } from 'bar.js';
```

Which is comparable to a combination of import and export:

```
import { default as function1,
         function2 } from 'bar.js';
export { function1, function2 };
```

But where `function1` and `function2` do not become available inside the current module.

> **Note:** The following is syntactically invalid despite its import equivalent:

```
import DefaultExport from 'bar.js'; // Valid
```

```
export DefaultExport from 'bar.js'; // Invalid
```

The correct way of doing this is to rename the export:

```
export { default as DefaultExport } from 'bar.js';
```

Examples

Using named exports

In a module `my-module.js`, we could include the following code:

```
// module "my-module.js"
function cube(x) {
  return x * x * x;
}

const foo = Math.PI + Math.SQRT2;

var graph = {
  options: {
      color:'white',
      thickness:'2px'
  },
  draw: function() {
      console.log('From graph draw function');
  }
}

export { cube, foo, graph };
```

Then in the top-level module included in your HTML page, we could have:

```
import { cube, foo, graph } from './my-module.js';

graph.options = {
    color:'blue',
    thickness:'3px'
};

graph.draw();
console.log(cube(3)); // 27
console.log(foo);    // 4.555806215962888
```

It is important to note the following:

- You need to include this script in your HTML with a `<script>` element of type="module", so that it gets recognised as a module and dealt with appropriately.
- You can't run JS modules via a `file://` URL — you'll get CORS errors. You need to run it via an HTTP server.

Using the default export

If we want to export a single value or to have a fallback value for your module, you could use a default export:

```
// module "my-module.js"

export default function cube(x) {
  return x * x * x;
}
```

Then, in another script, it is straightforward to import the default export:

```
import cube from './my-module.js';
console.log(cube(3)); // 27
```

Using export from

Let's take an example where we have the following hierarchy:

- `childModule1.js`: exporting `myFunction` and `myVariable`
- `childModule2.js`: exporting `myClass`
- `parentModule.js`: acting as an aggregator (and doing nothing else)
- top level module: consuming the exports of `parentModule.js`

This is what it would look like using code snippets:

```
// In childModule1.js
let myFunction = ...; // assign something useful to myFunction
let myVariable = ...; // assign something useful to myVariable
export {myFunction, myVariable};
```

```
// In childModule2.js
let myClass = ...; // assign something useful to myClass
export myClass;
```

```
// In parentModule.js
// Only aggregating the exports from childModule1 and childModule2
// to re-export them
export { myFunction, myVariable } from 'childModule1.js';
export { myClass } from 'childModule2.js';
```

```
// In top-level module
// We can consume the exports from a single module since parentModule
// "collected"/"bundled" them in a single source
import { myFunction, myVariable, myClass } from 'parentModule.js'
```