Module 3

Web APIs and the `global` object

String & Number Methods

Array Methods

Object Methods and ES6 Modules

Object Methods

Object

The Object type represents one of JavaScript's data types. It is used to store various keyed collections and more complex entities. Objects can be created using the `Object()` constructor or the object initializer / literal syntax.

```
const object1 = { a: 'foo', b: 42, c: {} };
console.log(object1.a, object1['b']); // Expected output: "foo" 42

const a = 'foo', b = 42, c = {};
const object2 = { a: a, b: b, c: c };
console.log(object2.b, object2['a']); // Expected output: 42 "foo"

const object3 = { a, b, c };
console.log(object3.a, object3['b']); // Expected output: "foo" 42
```

Source

Object Static Methods

Object.keys()

Returns an array containing the names of all of the given object's own enumerable string properties.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

console.log(Object.keys(object1));
// Expected output: Array ["a", "b", "c"]
```

Object.values()

Returns an array containing the values that correspond to all of a given object's own enumerable string properties.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

console.log(Object.values(object1));
// Expected output: Array ["somestring", 42, false]
```

Object.entries()

Returns an array containing all of the `[key, value]` pairs of a given object's own enumerable string properties.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// Expected output:
// "a: somestring"
// "b: 42"
// "c: false"
```

ES6 Modules

A background on modules

JavaScript programs started off pretty small — most of its usage in the early days was to do isolated scripting tasks, providing a bit of interactivity to your web pages where needed, so large scripts were generally not needed.

Fast forward a few years and we now have complete applications being run in browsers with a lot of JavaScript, as well as JavaScript being used in other contexts (Node.js, for example).

It has therefore made sense in recent years to start thinking about providing mechanisms for splitting JavaScript programs up into separate modules that can be imported when needed.

Source

A background on modules, cont.

Node.js has had this ability for a long time, and there are a number of JavaScript libraries and frameworks that enable module usage (for example, other CommonJS and AMD-based module systems like RequireJS, and more recently Webpack and Babel).

The good news is that modern browsers have started to support module functionality natively. This can only be a good thing — browsers can optimize loading of modules, making it more efficient than having to use a library and do all of that extra client-side processing and extra round trips.

Use of native JavaScript modules is dependent on the `import` and `export` statements.

`import`

The static `import` declaration is used to import read-only live bindings which are **exported** by another module. The imported bindings are called **live bindings** because they are updated by the module that exported the binding, but cannot be modified by the importing module.

In order to use the `import` declaration in a source file, the file must be interpreted by the runtime as a module. In HTML, this is done by adding `type="module"` to the `<script>` tag. Modules are automatically interpreted in strict mode.

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

`import` declarations

There are four forms of import declarations:

- Named import: `import { export1, export2 } from "module-name";`
- Default import: `import defaultExport from "module-name";`
- Namespace import: `import * as name from "module-name";`
- Side effect import: `import "module-name";`

Named import

Given a value named `myExport` which has been exported from the module `my-module` either implicitly as `export * from 'another.js'`, or explicitly using the export statement, this inserts myExport into the current scope.

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

You can import multiple names from the same module.

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

Note: `import { x, y } from "mod"` is not equivalent to import `defaultExport from "mod"` and then destructuring `x` and `y` from `defaultExport`. Named and default imports are distinct syntaxes in JavaScript modules.

Named import, cont.

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';
```

A module may also export a member as a string literal which is not a valid identifier, in which case you must alias it in order to use it in the current module.

```
// /modules/my-module.js
const a = 1;
export { a as "a-b" };
```

```
import { "a-b" as a } from "./modules/my-module.js";
```

Default import

Default exports need to be imported with the corresponding default import syntax. The simplest version directly imports the default:

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

Since the default export doesn't explicitly specify a name, you can give the identifier any name you like.

Default import, cont.

It is also possible to specify a default import with 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.default and myDefault point to the same binding
```

or

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

Importing a name called default has the same effect as a default import. It is necessary to alias the name because default is a reserved word.

```
import { default as myDefault } from '/modules/my-module.js';
```

Namespace import

The following code inserts `myModule` into the current scope, containing all the exports from the module located at `./modules/my-module.js`.

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

Here, `myModule` represents a **namespace** object which contains all exports as properties. For example, if the module imported above includes an export `doAllTheAmazingThings()`, you would call it like this:

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

`myModule` is a sealed object with `null` prototype. All keys are enumerable in lexicographic order (i.e. the default behavior of `Array.prototype.sort()`), with the default export available as a key called default.

Note: JavaScript does not have wildcard imports like `import * from "module-name"`, because of the high possibility of name conflicts.

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 is often used for polyfills, which mutate the global variables.

A **polyfill** is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

Example: Standard Import

```
// getPrimes.js
/**
 * Returns a list of prime numbers that are smaller than `max`.
 */
export function getPrimes(max) {
  const isPrime = Array.from({ length: max }, () => true);
  isPrime[0] = isPrime[1] = false;
  isPrime[2] = true;
  for (let i = 2; i * i < max; i++) {
    if (isPrime[i]) {
      for (let j = i ** 2; j < max; j += i) {
        isPrime[j] = false;
      }
    }
  }
  return [...isPrime.entries()]
    .filter(([, isPrime]) => isPrime)
    .map(([number]) => number);
}
```

Source

In this example, we create a re-usable module that exports a function to get all primes within a given range.

Then, in our `script.js` file (or whatever) we can import the function from `getPrimes.js` and use it:

```
import { getPrimes } from './modules/getPrimes.js';

console.log(getPrimes(10));
// > Array(4) [2, 3, 5, 7]

console.log(getPrimes(100));
// > Array(25) [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … ]

console.log(getPrimes(1000));
// > Array(168) [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … ]

console.log(getPrimes(1000000));
// > Array(78498) [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, … ]
```

export

The `export` declaration is used to export values from a JavaScript module. Exported values can then be **imported** into other programs with the import declaration or dynamic import. The value of an imported binding is subject to change in the module that exports it — when a module updates the value of a binding that it exports, the update will be visible in its imported value.

In order to use the `export` declaration in a source file, the file must be interpreted by the runtime as a module. In HTML, this is done by adding `type="module"` to the `<script>` tag, or by being imported by another module. Modules are automatically interpreted in strict mode.

Description

Every module can have two different types of export, **named export** and **default export**.

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 elsewhere
export { myFunction2, myVariable2 };
```

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

After the `export` keyword, you can use `let`, `const`, and `var` declarations, as well as function or class declarations. You can also use the `export { name1, name2 }` syntax to export a list of names declared elsewhere.

Note that `export {}` does not export an empty object — it's a no-op declaration that exports nothing (an empty name list).

Named exports, cont.

Export declarations are **not** subject to temporal dead zone rules.

You can declare that the module exports X before the name X itself is declared:

```
export { x };
const x = 1;
// This works, because `export` is only a declaration, but doesn't
// utilize the value of `x`.
```

Default exports

```
// export feature declared elsewhere as default
export { myFunction as default };
// This is equivalent to:
export default myFunction;
```

Exports do not need to be named (anonymous):

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

The export default syntax allows any expression.

```
export default 1 + 1;
```

Default exports, cont.

As a special case, functions and classes are exported as declarations, not expressions, and these declarations can be anonymous. This means functions will be hoisted.

```
// Works because `foo` is a function declaration,
// not a function expression
foo();

export default function foo() {
  console.log("Hi");
}
```

Default exports, cont.

Named exports are useful when you need to export several values. When importing this module, named exports must be referred to by the exact same name (optionally renaming it with as), but the default export can be imported with any name. For example:

```
// file test.js
const k = 42;
export default k;
```

```
// some other file
import m from './test';
// note that we have the freedom to use import m
// instead of import k, because k was default export

console.log(m); // will log out: 42
```

Default exports, cont.

You can also rename named exports to avoid naming conflicts:

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

You can rename a name to something that's not a valid identifier by using a string literal.

For example:

```
export { myFunction as "my-function" };
```

Re-exporting / Aggregating

A module can also "relay" values exported from other modules without the hassle of writing two separate import/export statements. This is often useful when creating a single module concentrating various exports from various modules (usually called a "barrel module").

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, except that function1 and function2 do not become available inside the current module:

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

Example: Exporting just the needed functions

partial.js:

```
// partial.mjs
function utility(text) {
    console.log(`Utility logs out ${text}`);
}

function test1(x) {
    console.log(`test1 call utility with ${x}`);
    utility(x);
}

function test2(x) {
    console.log(`test2 call utility with ${x}`);
    utility(x);
}

// We only export functionOne and functionTwo
export { test1, test2 };
```

script.js:

```
import { test1, test2 } from "./modules/partial.js";
// import {utility} from "./modules/partial.js"; // Will not work
// Throws an error: "Uncaught SyntaxError: ambiguous indirect export: utility"

test1("Hallo");
test2("World");

// Expected ooutput:
// test1 call utility with Hallo
// Utility logs out Hallo
// test2 call utility with World
// Utility logs out World
```

index.html:

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

Example: 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;

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

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

Then in the top-level script (type 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();           // 'From graph draw function'
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 recognized 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.

Example: 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
```

Example: 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 modules/childModule1.js
function myFunction() {
  console.log("Hello!");
}
const myVariable = 1;
export { myFunction, myVariable };
```

```
// In modules/childModule2.js
class MyClass {
  constructor(x) {
    this.x = x;
  }
}

export { MyClass };
```

```
// In modules/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, e.g. scripts.js:
// We can consume the exports from a single module since parentModule
// "collected"/"bundled" them in a single source
import { myFunction, myVariable, MyClass } from './modules/parentModule.js'

myFunction(); // Expected output: "Hello!"
const c = new MyClass(myVariable);
console.log(c); // Expected output: > Object { x: 1 }
```

Note: Remember to include `type="module"` when adding `script.js`:

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

Todos

Mollify

Read Object Methods and ES6 Modules, and do the Lesson Task