**Hello Modules**

JavaScript *modules* are reusable pieces of code that can be exported from one program and imported for use in another program.

Modules are particularly useful for a number of reasons. By separating code with similar logic into files called modules, we can:

* find, fix, and debug code more easily;
* reuse and recycle defined logic in different parts of our application;
* keep information private and protected from other modules;
* and, importantly, prevent pollution of the global namespace and potential naming collisions, by cautiously selecting variables and behavior we load into a program.

**module.exports**

We can get started with modules by defining a module in one file and making the module available for use in another file. Below is an example of how to define a module and how to export it using the statement module.exports.

In **menu.js** we write:

let Menu = {}; Menu.specialty = "Roasted Beet Burger with Mint Sauce"; module.exports = Menu;

Let's consider what this code means.

1. let Menu = {}; creates the object that represents the module Menu. The statement contains an uppercase variable named Menuwhich is set equal to an object.
2. Menu.specialty is defined as a property of the Menu module. We add data to the Menuobject by setting properties on that object and giving the properties a value.
3. "Roasted Beet Burger with Mint Sauce"; is the value stored in the Menu.specialtyproperty.
4. module.exports = Menu; exports the Menuobject as a module. module is a variable that represents the module, and exportsexposes the module as an object.

The pattern we use to export modules is thus:

1. Define an object to represent the module.
2. Add data or behavior to the module.
3. Export the module.

Let's create our first module.

**require()**

To make use of the exported module and the behavior we define within it, we import the module. A common way to do this is with the require() function.

For instance, say we want the module to control the menu's data and behavior, and we want a separate file to handle placing an order. We would create a separate file **order.js** and import the Menu module from **menu.js** to **order.js** using require(): In **order.js** we would write:

const Menu = require('./menu.js'); function placeOrder() { console.log('My order is: ' + Menu.specialty); } placeOrder();

We now have the entire behavior of Menuavailable in **order.js**. Here, we notice:

1. In **order.js** we import the module by creating a variable with const called Menuand setting it equal to the value of the require() function. We can set this variable to any variable name we like, such as menuItems.
2. require() is a JavaScript function that enables a module to load by passing in the module's filepath as a parameter.
3. './menu.js' is the argument we pass to the require() function.
4. The placeOrder() function then uses the Menu module in its function definition. By calling Menu.specialty, we access the property specialty defined in the Menumodule.
5. We can then invoke the function using placeOrder()

Taking a closer look, the pattern to import a module is:

1. Import the module
2. Use the module and its properties within a program.

**module.exports II**

We can also wrap any collection of data and functions in an object, and export the object using module.exports. In **menu.js**, we could write:

let Menu = {}; module.exports = { specialty: "Roasted Beet Burger with Mint Sauce", getSpecialty: function() { return this.specialty; } };

In the above code, notice:

1. module.exports exposes the current module as an object.
2. specialty and getSpecialty are properties on the object.

Then in **order.js**, we write:

const Menu = require('./menu.js'); console.log(Menu.getSpecialty());

Here we can still access the behavior in the Menu module.

**export default**

As of ES6, JavaScript has implemented a new more readable and flexible syntax for exporting modules. These are usually broken down into one of two techniques, *default export* and *named exports*.

We'll begin with the first syntax, default export. The default export syntax works similarly to the module.exports syntax, allowing us to export one module per file.

Let's look at an example in **menu.js**.

let Menu = {}; export default Menu;

1. export default uses the JavaScript exportstatement to export JavaScript objects, functions, and primitive data types.
2. Menu refers to the name of the Menu object, the object that we are setting the properties on within our modules.

When using ES6 syntax, we use export default in place of module.exports.

**import**

ES6 module syntax also introduces the importkeyword for importing objects. In our **order.js**example, we import an object like this:

import Menu from './menu';

1. The import keyword begins the statement.
2. The keyword Menu here specifies the name of the variable to store the default export in.
3. from specifies where to load the module from.
4. './menu' is the name of the module to load. When dealing with local files, it specifically refers to the name of the file without the extension of the file.

We can then continue using the Menu module in the **order.js** file.

**Named Exports**

ES6 introduced a second common approach to export modules. In addition to default export, *named exports* allow us to export data through the use of variables.

Let's see how this works. In **menu.js** we would be sure to give each piece of data a distinct variable name:

let specialty = ''; function isVegetarian() { }; function isLowSodium() { }; export { specialty, isVegetarian };

1. Notice that, when we use named exports, we are not setting the properties on an object. Each export is stored in its own variable.
2. specialty is a string object, while isVegetarian and isLowSodium are objects in the form of functions. Recall that in JavaScript, every function is in fact a function object.
3. export { specialty, isVegetarian };exports objects by their variable names. Notice the keyword export is the prefix.
4. specialty and isVegetarian are exported, while isLowSodium is not exported, since it is not specified.

**Named Imports**

To import objects stored in a variable, we use the import keyword and include the variables in a set of {}.

In the **order.js** file, for example, we would write:

import { specialty, isVegetarian } from './menu'; console.log(specialty);

1. Here specialty and isVegetarian are imported.
2. If we did not want to import either of these variables, we could omit them from the import statement.
3. We can then use these objects as in within our code. For example, we would use specialty instead of Menu.specialty.

**Export Named Exports**

Named exports are also distinct in that they can be exported as soon as they are declared, by placing the keyword export in front of variable declarations.

In **menu.js**

export let specialty = ''; export function isVegetarian() { }; function isLowSodium() { };

1. The export keyword allows us to export objects upon declaration, as shown in export let specialty and export function isVegetarian() {}.
2. We no longer need an export statement at the bottom of our file, since this behavior is handled above.

# Import Named Imports

To import variables that are declared, we simply use the original syntax that describes the variable name. In other words, exporting upon declaration does not have an impact on how we import the variables.

import { specialty, isVegetarian } from 'menu';

**Export as**

Named exports also conveniently offer a way to change the name of variables when we export or import them. We can do this with the askeyword.

Let's see how this works. In our **menu.js**example

let specialty = ''; let isVegetarian = function() { }; let isLowSodium = function() { }; export { specialty as chefsSpecial, isVegetarian as isVeg, isLowSodium };

In the above example, take a look at the exportstatement at the bottom of the of the file.

1. The as keyword allows us to give a variable name an alias as demonstrated in specialty as chefsSpecial and isVegetarian as isVeg.
2. Since we did not give isLowSodium an alias, it will maintain its original name.

**Import as**

To import named export aliases with the askeyword, we add the aliased variable in our import statement.

import { chefsSpecial, isVeg } from './menu';

In **orders.js**

1. We import chefsSpecial and isVeg from the Menu object.
2. Here, note that we have an option to alias an object that was not previously aliased when exported. For example, the isLowSodium object that we exported could be aliased with the as keyword when imported: import {isLowSodium as saltFree} from 'Menu';

Another way of using aliases is to import the entire module as an alias:

import \* as Carte from './menu'; Carte.chefsSpecial; Carte.isVeg(); Carte.isLowSodium();

1. This allows us to import an entire module from **menu.js** as an alias Carte.
2. In this example, whatever name we exported would be available to us as properties of that module. For example, if we exported the aliases chefsSpecial and isVeg, these would be available to us. If we did not give an alias to isLowSodium, we would call it as defined on the Cartemodule.

# Combining Export Statements

We can also use named exports and default exports together. In **menu.js**:

let specialty = ''; function isVegetarian() { }; function isLowSodium() { }; function isGlutenFree() { }; export { specialty as chefsSpecial, isVegetarian as isVeg }; export default isGlutenFree;

Here we use the keyword export to export the named exports at the bottom of the file. Meanwhile, we export the isGlutenFreevariable using the export default syntax.

This would also work if we exported most of the variables as declared and exported others with the export default syntax.

export let Menu = {}; export let specialty = ''; export let isVegetarian = function() { }; export let isLowSodium = function() { }; let isGlutenFree = function() { }; export default isGlutenFree;

Here we use the export keyword to export the variables upon declaration, and again export the isGlutenFree variable using the export default syntax

While it's better to avoid combining two methods of exporting, it is useful on occasion. For example, if you suspect developers may only be interested in importing a specific function and won't need to import the entire default export.

# Combining Import Statements

We can import the collection of objects and functions with the same data.

We can use an import keyword to import both types of variables as such:

import { specialty, isVegetarian, isLowSodium } from './menu'; import GlutenFree from './menu';

**Review**

We just learned how to use JavaScript modules. Let's review what we learned:

*Modules* in JavaScript are reusable pieces of code that can be exported from one program and imported for use in another program.

* module.exports exports the module for use in another program.
* require() imports the module for use in the current program.

ES6 introduced a more flexible, easier syntax to export modules:

* default exports use export default to export JavaScript objects, functions, and primitive data types.
* named exports use the export keyword to export data in variables.
* named exports can be aliased with the askeyword.
* import is a keyword that imports any object, function, or data type.