the shortcut

Fullstack Web Development Tutorial Lesson 18

Today’s lesson will cover

- Modules
- Export and Import
- Dynamic imports

the shortcut

JavaScript fundamentals

Modules

- As our application grows bigger, we want to split it into multiple files, so called “modules”. A module may contain a class or a library of functions for a specific purpose.
- Modules can load each other and use special directives `export` and `import` to interchange functionality, call functions of one module from another one:
  - `export` keyword labels variables and functions that should be accessible from outside the current module.
  - `import` allows the import of functionality from other modules.
- As modules support special keywords and features, we must tell the browser that a script should be treated as a module, by using the attribute `<script type="module">`.
- **Modules work only via HTTP(s), not in local files**
  - If you try to open a web-page locally, via `file://` protocol, you’ll find that `import/export` directives don’t work. Use a local web-server, such as static-server or use the “live server” capability of your editor, such as VS Code Live Server Extension to test modules.
- Build tools:
  - In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as Webpack and deploy to the production server.
  - One of the benefits of using bundlers – they give more control over how modules are resolved, allowing bare modules and much more, like CSS/HTML modules.
  - Unreachable code removed.
  - Unused exports removed (“tree-shaking”).
  - Development-specific statements like `console` and `debugger` removed.

Export and Import

- We can put `import/export` statements at the top or at the bottom of a script, that doesn't matter. In practice imports are usually at the start of the file, but that's only for more convenience.

- **Common types of `export`:**
- Before declaration of a class/function/…: `export [default] class/function/variable ...`
- Standalone export: `export {x [as y], ...}`.
- Re-export:
  - `export {x [as y], ...} from "module"`
  - `export * from "module"` (doesn't re-export default).
  - `export {default [as y]} from "module"` (re-export default).
- **Imports:**
- Named exports from module: `import {x [as y], ...} from "module"`
- Default export:
  - `import x from "module"`
  - `import {default as x} from "module"`
- Everything: `import * as obj from "module"`
- Import the module (its code runs), but do not assign it to a variable: `import "module"`
- **Please note that import/export statements don't work if inside `{...}`.**
- A conditional import, like this, won't work:

```
if (something) {
 import {sayHi} from "./say.js"; // Error: import must be at top level}
```

Dynamic imports

- Dynamic `import()` introduces a new function-like form of `import` that caters to those use cases. `import(moduleSpecifier)` returns a promise for the module namespace object of the requested module, which is created after fetching, instantiating, and evaluating all of the module's dependencies, as well as the module itself.

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

- Dynamic imports work in regular scripts, they don't require `script type="module"`.

- Although `import()` looks like a function call, it's a special syntax that just happens to use parentheses (similar to `super()`).

- Static `import` and dynamic `import()` are both useful. Each have their own, very distinct, use cases. Use static `import`s for initial paint dependencies, especially for above-the-fold content. In other cases, consider loading dependencies on-demand with dynamic `import()`.

Summary: Modules

- A module is a file. To make `import/export` work, browsers need `<script type="module">`. Modules have several differences:
  - Deferred by default.
  - Async works on inline scripts.
  - To load external scripts from another origin (domain/protocol/port), CORS headers are needed.
  - Duplicate external scripts are ignored.
- Modules have their own, local top-level scope and interchange functionality via `import/export`.
- Modules always `use strict`.
- Module code is executed only once. Exports are created once and shared between importers.
- When we use modules, each module implements the functionality and exports it. Then we use `import` to directly import it where it's needed. The browser loads and evaluates the scripts automatically.
- In production, people often use bundlers such as Webpack to bundle modules together for performance and other reasons.

Exercise

- Create a file called `library.js` which contains an area function and a perimeter function
- The `area` function calculates area and provides an alert output stating `'Area of the rectangle is GIVENAREA square unit'`
- The `perimeter` function calculates area and provides an alert output stating '`Perimeter of the rectangle is GIVENAREA unit'`
- Export the `area` and `perimeter` functions `library.js`.
- Create another file called `script.js` which imports the area and perimeter functions from `library.js`
- Create a length and width variable with certain values and call the functions in this file to find the area and perimeter for the given length and width
- Link the `script.js` on an HTML doc and run the file using liveserver

Exercise

- Create a file called `math.js` which exports a class called `Math`
- The class constructor contains `this.sum` with a function to add two arguments, and `this.sub` with a function to subtract value of two arguments
- The class also contains `findSum(a,b)` method which returns the `sum()`
- And `findSub(a,b)` method which returns the `sub()`
- `main.js` file imports the Math class and creates a new math class
- It calls the functions `findSum()` and `findSub()` with given arguments and provides output using browser alert
- Link the `main.js` on an HTML doc and run the file using liveserver

the shortcut

Self Study Assignments

To Dos

- Create a game of Rock, Paper and Scissors using JS which works on console, or with interactive UI using HTML, CSS and JS however you prefer ***(If you are working on your own project where you are using JS already, feel free to ignore this task but please share the project update with Lena.)***
- Continue freecodecamp (FCC) Javascript. Ideally finish before we resume after summer.
- Continue with FCC HTML, CSS lessons. Ideally finish all the lessons by end of this month.
- If you believe FCC exercises aren't the best for you as in if you are quite advanced already, please start working on your own project and reach out to mentors for help if needed.