Notes - JavaScript - ES Modules - Usage

- Dr Nick Hayward

A collection of notes &c. on plain JavaScript modules, in particular usage of ES modules introduced with ES2015.

Contents

Intro

- simpler and easier to work with than CommonJS
  - in most examples…
- JavaScript `strict` mode is enabled by default
- `strict` mode helps with language usage - check for poor usage
  - stops hoisting of variables
  - variables must be declared
  - function parameters must have unique name
  - assignment to read-only properties throws errors
  - …
- modules are exported with `export` statements
- modules are imported with `import` statements

Export - `export` statements

- ES6 modules are individual files
  - expose an API using `export` statements
- declarations are scoped to the local module
- e.g. variables declared inside a module
  - not available to other modules
  - need to be explicitly exported in module API
  - need to be imported for usage in another module
- export statements may only be added to *top-level* of a module
  - e.g. not in function expression *&c.
- cannot dynamically define and expose API using methods
  - unlike CommonJS module system - Node.js &c.

Export - `export default`

- common option is to export a default binding, e.g.

```
export default `hello world`
```

```
export default {
    name: 'Alice',
    place: 'Wonderland'
}
```

```
export default [
    'Alice', 'Wonderland'
]
```

```
export default function name() {
    ...
}
```

Module bindings

- ES modules export **bindings**
  - not values or references
- e.g. an export of `count` variable from a module
  - `count` is exported as a binding
  - export is bound to `count` variable in the module
  - value is subject to changes of `count` in module
- offers flexibility to exported API
  - e.g. `count` might originally be bound to an object
  - then changed to an array...
- other modules consuming this export
  - they would see change as `count` is modified
  - modified in module and exported...
- **n.b.** take care with this usage pattern
  - useful for counters, logs &c.
  - can cause issues with API usage for a module

Export - named export

- we may define bindings for export
- explicit instead of assigning properties to implicit export object
  - e.g.

```
export let counter = 0
export const count = () => counter++
```

- cannot refactor this example for named export
  - syntax error will be thrown
  - e.g.

```
let counter = 0
const count = () => counter++
export counter // this will return syntax error
export count
```

- rigid syntax helps with analysis, parsing
  - static analysis for ES modules

Export - lists

- lists provide a useful solution to previous refactor issue
- syntax for list export easy to parse
- export lists of named *top-level* declarations
  - variables &c.
- e.g.

```
let counter = 0
const count = () => counter++
export { counter, count }
```

- also rename binding for export, e.g.

```
let counter = 0
const count = () => counter++
export { counter, count as increment }
```

- define `default` with export list, e.g.

```
let counter = 0
const count = () => counter++
export { counter as default, count as increment }
```

Export - `export from ...`

- expose another module's API using `export from...`
  - i.e. a kind of pass through...
- e.g.

```
export { increment } from './myCounter.js'
```

- bindings are not imported into module's local scope
- current module acts as conduit, passing bindings along export/import chain...
- module does not gain direct access to `export from ...` bindings
  - e.g. if we call `increment` it will throw a `ReferenceError`
- aliases are also possible for bindings with `export from...`
  - e.g.

```
export { increment as addition } from './myCounter.js'
```

Import - `import` statements

- use `import` to load another module
- `import` statement are only allowed in top level of module definition
  - same as `export` statements
  - helps compilers simplify module loading &c.
- import default exports
  - give default export a name as it is imported
  - e.g.

```
import counter from './myCounter.js'
```

- importing binding to `counter`
- syntax different from declaring a JS variable

Import - `import` named exports

- also imported any named exports
  - import more than just default exports
- named import is wrapped in braces
  - e.g.

```
import { increment } from './myCounter.js'
```

- also import multiple named exports
  - e.g.

```
import { increment, decrement } from './myCounter.js'
```

- import aliases are also supported
  - e.g.

```
import { increment as addition } from './myCounter.js'
```

- combine default with named
  - e.g.

```
import counter, { increment } from './myCounter.js'
```

Import - `import` with wildcard

- we may also import using the *wildcard* operator
  - e.g.

```
import * as counter from './myCounter.js'
counter.increment()
```

- name for wildcard import acts like object for module
- call module exports on wildcard

```
import * as counter from './myCounter.js'
counter.increment()
```

- common pattern for working with libraries &c.

Benefits & practical usage

- offers ability to explicitly publish an API
  - keeps module content local unless explicitly exported
- similar function to *getters* and *setters*
  - explicit way in and out of modules
  - explicit options for reading and updating values...
- code becomes simpler to write and manage
  - module offers encapsulation of code
- import binding to variable, function &c.
  - then use it as normal...
- removes need for encapsulation in main JS code
  - e.g. with patterns such as IIFE...
- *n.b.* need to be careful how we use modules
  - e.g. priority for access, security, testing &c.
  - all now moved to individual modules...