DOM:

- When loading the HTML page, the elements are loaded from top to bottom.
- When the browser loads HTML and comes across a <script> tag, it can't continue building the DOM. It must execute the script right now.
- That leads to two important issues:
    1. Scripts can't see DOM elements below them, so they can't add handlers etc.
    2. If there's a bulky script at the top of the page, it "blocks the page". Users can't see the page content till it downloads and runs.
- That's why, if we have our <script> tag in the head, before the body, and the javascript file requires elements, classes, ids, etc from the <body>, it will cause an error.
- E.g.

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" type="text/css" href="test.css">
        <script type="text/javascript" src="test.js"> </script>
    </head>
    <body>
        <p id="p"> Sample Text </p>
    </body>
</html>
```

```
const test = document.getElementById("p")
test.innerHTML = "New Text"
```

```
Uncaught TypeError: Cannot set property 'innerHTML' of null     test.js:2
    at test.js:2
```

- To bypass this error, you can do:
  document.addEventListener("DOMContentLoaded", function())
- E.g.

```
document.addEventListener("DOMContentLoaded", function(){
    const test = document.getElementById("p")
    test.innerHTML = "New Text"
})
```

New Text

- You can also put the defer or async attribute in the <script> tag.
- The **defer** attribute makes the browser wait to load <script> tags. This Boolean attribute is set to indicate to a browser that the script is meant to be executed after the document has been parsed, but before firing DOMContentLoaded. Scripts with the defer attribute

will execute in the order in which they appear in the document. Scripts with defer never block the page.

- The **async** attribute means that a script is completely independent. The page doesn't wait for async scripts, the contents are processed and displayed. Furthermore, DOMContentLoaded and async scripts don't wait for each other. DOMContentLoaded may happen both before an async script (if an async script finishes loading after the page is complete) or after an async script (if an async script is short or was in HTTP-cache). Other scripts don't wait for async scripts, and async scripts don't wait for them. So, if we have several async scripts, they may execute in any order. Whatever loads first runs first. Async doesn't block.
  I.e. The async attribute will run the JS file(s) while simultaneously loading the HTML file.
- The rule for using defer and async is this: "If you need to wait for a specific DOM to load, use defer. If you don't need to wait for anything in the HTML file to load, use async."

Node.Js:

- Recall that JS needs a runtime environment. So far, we've used the browser, but what if we wanted to run JavaScript without the browser. We will need a different runtime environment. We can use Node.js.
- Node.js is a JavaScript runtime environment.
- Allows us to run JavaScript outside of the browser.
- Implements the ES standard.
- Many modules/libraries have been written for it.
- Node.js still runs the V8 engine.
- In Node.js, there's no document, no window, no JQuery, no DOM.
- We need to use the keyword "require" to use node.js modules.
- E.g.

```
/* Node app.js */
const log = console.log
log('Node intro');

// require() is a built-in function that 'imports'
// modules into a variable
const util = require('util')

const s = util.format('%s %s', 'csc', '309')
log(s)
```

```
C:\Users\rick\Desktop\Basic HTML Code>node test.js
Node intro
csc 309
```

- You can also make your own node.js modules.

- E.g.

```
/* Node app.js */
const log = console.log
log('Node intro');

/// Creating our own module (the module design pattern)
// require() returns module.exports of the module
const course = require('./course')

const courseList = ['csc309', 'csc343']

course.addCourse(courseList, 'csc301')
course.removeCourse(courseList, 'csc343')
log(courseList)

// object destructuring: an easy way to pick off
// the object properties you specifically want
const { addCourse, removeCourse } = course;
addCourse(courseList, 'csc108')
removeCourse(courseList, 'csc301')
log(courseList)
```

```
/* Course selection module */
console.log('Course selection module')

// require() will provide our app with the
// object that is referenced by module.exports:
module.exports = {
    addCourse: function(courseList, course) {
        courseList.push(course)
    },

    removeCourse: function(courseList, course) {
        const i = courseList.indexOf(course)
        courseList.splice(i, 1)
    }
}
```

```
C:\Users\rick\Desktop\Basic HTML Code>node test.js
Node intro
Course selection module
[ 'csc309', 'csc301' ]
[ 'csc309', 'csc108' ]
```

- **Note:** When making your own node.js modules, you must put all functions in module.exports.

Arrow Functions:

- Denoted by "**=>**".
- An **arrow function expression** is a syntactically compact alternative to a regular function expression, although without its own bindings to the this, arguments, super, or new.target keywords. Arrow function expressions are ill suited as methods, and they cannot be used as constructors.
- Syntax: **(param1, param2, …, paramN) => expression**
- E.g.

```
const square = (x) => (x*x)
console.log(square(5))
console.log(square(10))
```

```
25
100
```

Note: The brackets in this case are optional.

```
const square = x => x*x
console.log(square(5))
console.log(square(10))
```

```
25
100
```

- However, if you have no arguments or multiple arguments, then you need the brackets around the parameters.

- E.g. of multiple arguments

```
const multiply = (x, y) => x*y
console.log(multiply(1,3))
```

```
Uncaught SyntaxError: Missing initializer in const declaration    test.js:1
```

However, once we put the brackets around “x, y”, we get:

```
const multiply = (x, y) => x*y
console.log(multiply(1,3))
```

```
3
```

- E.g. of no arguments

```
const six =  => 6
console.log(six())
```

```
Uncaught SyntaxError: Unexpected token '=>'                    test.js:1
```

Once we put the brackets, we get:

```
const six = () => 6
console.log(six())
```

```
6
```

- For arrow functions, “this” is bound based on the lexical function/global scope of where it’s defined.