Web System Development

Module 03: Javascript

Javascript

- Javascript is a programming language that allows you to implement complex features on web pages.
- It is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions.
- While it is most well-known as the scripting language for Web pages, many non-browser environments also use it, such as Node.js, Apache CouchDB and Adobe Acrobat.
- Javascript is a prototype-based, multi-paradigm, dynamic language, supporting object-oriented, imperative, and declarative (e.g. functional programming) styles.

ECMAScript

- The official name of the JavaScript standard is ECMAScript .
- At this moment, the latest version is the one released in June of 2025 with the name ECMAScript®2025 , otherwise known as ES16.

Transpiling

- **Transpiling** is the process of converting code from one language to another.
- Browsers do not yet support all of JavaScript's newest features. Due to this fact, a lot of code run in browsers has been *transpiled* from a newer version of JavaScript to an older, more compatible version.
- Today, the most popular way to do transpiling is by using Babel . Transpilation is automatically configured in React applications created with Vite.

Running Javascript in Node.js

- Node.js is a JavaScript runtime environment based on Google's Chrome V8 JavaScript engine and works practically anywhere - from servers to mobile phones.
- Let's practice writing some JavaScript using Node. The latest versions of Node already understand the latest versions of JavaScript, so **the code does not need to be transpiled** .
- The code is written into files ending with `.js` that are run by issuing the command `node name_of_file.js`
- It is also possible to write JavaScript code into the Node.js console, which is opened by typing *node* in the command line, as well as into the browser's developer tool console.
- Alternatively, you can use a tool like JS Bin .

Variables

In JavaScript there are a few ways to go about defining variables:

```
// variables.js

// To run this code, execute in the terminal:
// node variables.js
const x = 1
let y = 5

console.log(x, y)   // 1 5 are printed
y += 10
console.log(x, y)   // 1 15 are printed
y = 'sometext'
console.log(x, y)   // 1 sometext are printed
x = 4               // causes an error
```

- const does not define a *variable* but a *constant* for which the value can no longer be changed. On the other hand, let defines a normal variable.
- We also see that the variable's data type can change during execution. At the start, `y` stores an integer; at the end, it stores a string.
- It is also possible to define variables in JavaScript using the keyword var :
  - JavaScript Variables - Should You Use let, var or const? on Medium
  - var, let and const - ES6 JavaScript Features
  - Keyword: var vs. let on JS Tips

Arrays

- The Array object enables storing a collection of multiple items under a single variable name, and has members for performing common array operations.
- In JavaScript, arrays aren't primitives but are instead Array objects with the following core characteristics:
  - JavaScript arrays are **resizable** and can contain a mix of different data types .
  - JavaScript arrays **are not associative arrays** and so, array elements cannot be accessed using arbitrary strings as indexes, but must be accessed using nonnegative integers (or their respective string form) as indexes.
  - JavaScript arrays are zero-indexed : the first element of an array is at index `0` , the second is at index `1` , and so on — and the last element is at the value of the array's `length` property minus `1` .
  - JavaScript array-copy operations create shallow copies . (All standard built-in copy operations with *any* JavaScript objects create shallow copies, rather than deep copies).

Array Example

```
// arrays01.js
const t = [1, -1, 3]

t.push(5)

console.log(t.length) // 4 is printed
console.log(t[1])     // -1 is printed

t.forEach(value => {
  console.log(value)  // numbers 1, -1, 3, 5 are printed, each on its own line
})
```

- Notable in this example is the fact that although a variable declared with `const` cannot be reassigned to a different value, the contents of the object it references can still be modified.
- This is because the const declaration ensures the immutability of the reference itself, not the data it points to. Think of it like changing the furniture inside a house, while the address of the house remains the same.
- One way of iterating through the items of the array is using `forEach` :
  - `forEach` receives a *function* defined using the arrow syntax as a parameter.
  - `forEach` calls the function *for each of the items in the array* , always passing the individual item as an argument. .
  - The function as the argument of `forEach` may also receive other arguments

Immutable Arrays

- In the previous example, a new item was added to the array using the method push . When using **React** , techniques from functional programming are often used.
- One characteristic of the functional programming paradigm is the use of immutable data structures. In React code, it is preferable to use the method concat , which creates a new array with the added item. This ensures the original array **remains unchanged** .
- The method call `t.concat(5)` does not add a new item to the old array but returns **a new array** which, besides containing the items of the old array, also contains the new item.

```
// arrays02.js
const t = [1, -1, 3]

const t2 = t.concat(5)  // creates new array

console.log(t)  // [1, -1, 3] is printed
console.log(t2) // [1, -1, 3, 5] is printed
```

Destructuring

Individual items of an array are easy to assign to variables with the help of the destructuring assignment :

```
// arrays03.js
const t = [1, 2, 3, 4, 5]

const [first, second, ...rest] = t

console.log(first, second)  // 1 2 is printed
console.log(rest)           // [3, 4, 5] is printed
```

- Above, the variable `first` is assigned the first integer of the array and the variable `second` is assigned the second integer of the array.
- The variable `rest` "collects" the remaining integers into its own array.

Array Map Method

There are plenty of useful methods defined for arrays. Let's look at a short example of using the map method.

```
// arrays04.js
const t = [1, 2, 3]

const m1 = t.map(value => value * 2)
console.log(m1)   // [2, 4, 6] is printed
```

Based on the old array, map creates a **new array** , for which the function given as a parameter is used to create the items. In the case of this example, the original value is multiplied by two.

Map can also transform the array into something completely different:

```
// arrays04.js
const m2 = t.map(value => '<li>' + value + '</li>')
console.log(m2)
// [ '<li>1</li>', '<li>2</li>', '<li>3</li>' ] is printed
```

Here an array filled with integer values is transformed into an array containing strings of **HTML** using the map method.

Useful Array Methods

```
// Example array to demonstrate array methods
const arr = [1, 2, 3, 4, 5];

arr.push(6)            // [1, 2, 3, 4, 5, 6]
arr.pop()              // [1, 2, 3, 4, 5]
arr.shift()            // [2, 3, 4, 5]
arr.unshift(0)         // [0, 2, 3, 4, 5]
arr.concat([6, 7])     // [0, 2, 3, 4, 5, 6, 7]
arr.slice(1, 3)        // [2, 3]
arr.splice(1, 2)       // [0, 4, 5]
arr.reverse()          // [5, 4, 0]
arr.sort()             // [0, 4, 5]
arr.indexOf(4)         // 1
arr.includes(3)        // false
arr.find(x => x > 3)   // 4
arr.filter(x => x > 3) // [4, 5]
arr.map(x => x * 2)    // [10, 8, 0]
arr.reduce((sum, x) => sum + x, 0) // 9
arr.join('-')          // "5-4-0"
arr.every(x => x > 2)  // false
arr.some(x => x > 4)   // true
arr.findIndex(x => x > 4) // 0
arr.fill(9)            // [9, 9, 9]
arr.copyWithin(0, 1)   // [9, 9, 9]
arr.flatMap(x => [x, x * 2]) // [18, 18, 18]
arr.entries()          // Array Iterator with key/value pairs
arr.keys()             // Array Iterator with keys
arr.values()           // Array Iterator with values
```

Functions

We have already become familiar with defining arrow functions. The complete process, without cutting corners, of defining an arrow function is as follows:

```
// functions01.js
const sum = (p1, p2) => {
  console.log(p1); // 1 is printed
  console.log(p2); // 5 is printed
  return p1 + p2
}

const result = sum(1, 5)
console.log(result) // 6 is printed
```

If there is just a single parameter, we can exclude the parentheses from the definition:

```
// functions01.js
const square = p => {
  console.log(p)
  return p * p
}

console.log(square(2)); // 4 is printed
```

Shortening Function Definitions

If the function only contains a single expression then the braces are not needed. In this case, the function only returns the result of its only expression. Now, if we remove console printing, we can further shorten the function definition:

```
const square = p => p * p
```

This form is particularly handy when manipulating arrays - e.g. when using the map method:

```
const t = [1, 2, 3]
const tSquared = t.map(p => p * p)
// tSquared is now [1, 4, 9]
```

Objects

There are a few different ways of defining objects in JavaScript. One very common method is using object literals , which happens by listing its properties within braces:

```
// objects01.js
const object1 = {
  name: 'Arto Hellas',
  age: 35,
  education: 'PhD',
}

const object2 = {
  name: 'Full Stack web application development',
  level: 'intermediate studies',
  size: 5,
}

const object3 = {
  name: {
    first: 'Dan',
    last: 'Abramov',
  },
  grades: [2, 3, 5, 3],
  department: 'Stanford University',
}

console.log(object1.name)          // Arto Hellas is printed
const fieldName = 'age'
console.log(object1[fieldName])    // 35 is printed
```

- The values of the properties can be of any type, like integers, strings, arrays, objects...
- The properties of an object are referenced by using the "dot" notation, or by using brackets

Adding Properties

```
object1.address = 'Helsinki'
object1['secret number'] = 12341
```

- You can **add properties** to an object on the fly by either using dot notation or brackets.
- The latter of the additions has to be done by using brackets because when using dot notation, *secret number* is not a valid property name because of the space character.

Classes

There is no class mechanism in JavaScript like the ones in object-oriented programming languages. There are, however, features to make "simulating" object-oriented classes possible.

Let's take a quick look at the *class syntax* that was introduced into JavaScript with ES6, which substantially simplifies the definition of classes (or class-like things) in JavaScript.

In the following example we define a *class* called `Person` and two `Person` objects:

```
// classes01.js
class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
  }
  greet() {
    console.log('hello, my name is ' + this.name)
  }
}

const adam = new Person('Adam Ondra', 29)
adam.greet()

const janja = new Person('Janja Garnbret', 23)
janja.greet()
```

Running JavaScript in the Browser

- All modern browsers come with a built-in JavaScript engine.
- You can run JavaScript in the browser using:
  1. The browser console
  2. A `<script>` tag inside an HTML file
  3. An external `.js` file linked from HTML

Using the Browser Console

- Open DevTools with *F12* or *Cmd+Opt+I* / *Ctrl+Shift+I*
- Go to the *Console* tab
- Type any JavaScript expression and press Enter

```
console.log("Hello from the console");
2 + 2
```

Using Script Tags

- You can embed JavaScript in HTML using `<script>` tags.
- JS code runs when the browser reaches the `<script>` tag

```
<!DOCTYPE html>
<html>
  <body>
    <h1>My Page</h1>
    <script>
      console.log("Hello from script tag");
    </script>
  </body>
</html>
```

External JavaScript Files

- Save your JS code in a file (e.g., `main.js`) and link it from your HTML.
- Make sure the script is loaded **after** the HTML content (it ensures the DOM is loaded before JS runs). Alternatively you can use defer

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>

  ...

  <script src="main.js"></script>

</body>
</html>
```

DOM

- We can think of HTML pages as implicit tree structures.
- The same treelike structure can be seen on the browser's console's Elements tab.
- Document Object Model, or DOM , is an Application Programming Interface (*API*) for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content.
- The DOM is not a programming language, but a way to represent and interact with the HTML document.

```
html
head
  link
  script
body
  div
    h1
    div
      ul
        li
        li
        li
    form
      input
      input
```

DOM Manipulation

- JavaScript can dynamically change the structure and content of a web page.
- The topmost node of the DOM tree of an HTML document is called the `document` object.
- The `document` object gives access to the DOM (Document Object Model).
- For instance, the following code creates a new node, assigns it to the variable ul, and adds some child nodes to it.
- Finally, the tree branch of the `ul` variable is connected to its proper place in the HTML tree of the whole page:

```
var ul = document.createElement('ul')

// data is an array of objects (notes)
data.forEach(function(note) {
  var li = document.createElement('li')

  ul.appendChild(li)
  li.appendChild(document.createTextNode(note.content))
})

document.getElementById('notes').appendChild(ul)
```

Selecting and Modifying Elements

- You can select elements using methods like:
  - `getElementById()`
  - `querySelector()`
  - `querySelectorAll()`
- Change text with `element.textContent` or `element.innerHTML`
- Modify style using `element.style`
- Add or remove classes with `element.classList`

```
const heading = document.getElementById('title');
heading.textContent = 'Hello World';
heading.style.color = 'blue';
```

Query Selectors

```
// Basic Syntax
document.querySelector('selector');       // Selects the first matching element
document.querySelectorAll('selector');    // Selects all matching elements (returns a NodeList)

// Examples of CSS Selectors

// ID Selector
document.querySelector('#myId');          // Select element with id="myId"

// Class Selector
document.querySelector('.myClass');       // Select the first element with class="myClass"
document.querySelectorAll('.myClass');    // Select all elements with class="myClass"

// Tag Selector
document.querySelector('div');            // Select the first <div> element
document.querySelectorAll('div');         // Select all <div> elements

// Attribute Selector
document.querySelector('[type="text"]');  // Select first element with attribute type="text"

// Descendant Selector (e.g., nested elements)
document.querySelector('.parent .child');         // Select first .child inside .parent
document.querySelectorAll('.parent .child');      // Select all .child elements inside .parent

// Pseudo-Classes
document.querySelector('a:hover');                // Select the first <a> element on hover
document.querySelector('li:nth-child(2)');        // Select the second <li> element
document.querySelectorAll('li:nth-child(odd)');   // Select all odd-numbered <li> elements
```

Manipulating Selected Elements

```
// Change text content
let elem = document.querySelector('.myClass');
elem.textContent = 'New Text';                    // Change text content of first matching element

// Change inner HTML
elem.innerHTML = '<p>New HTML Content</p>';       // Set inner HTML of first matching element

// Change attributes
elem.setAttribute('src', 'newImage.jpg');         // Change an attribute value

// Change styles
elem.style.color = 'blue';                        // Change the style of the first matched element

// Looping Through NodeList from querySelectorAll
let elems = document.querySelectorAll('.myClass');
elems.forEach((el) => {
  el.style.backgroundColor = 'yellow';            // Change background color for all matched elements
});

// Selecting Direct Descendants
document.querySelector('.parent > .child');       // Select the first direct child .child of .parent
```

Example

```
<html>
  <body>
    <div id="app"></div>
    <script type="text/javascript">
      // Select the div element with 'app' id
      const app = document.getElementById('app');

      // Create a new H1 element
      const header = document.createElement('h1');

      // Create a new text node for the H1 element
      const text = 'Develop. Preview. Ship.';
      const headerContent = document.createTextNode(text);

      // Append the text to the H1 element
      header.appendChild(headerContent);

      // Place the H1 element inside the div
      app.appendChild(header);
    </script>
  </body>
</html>
```

The browser will then render the page with the `h1` element inside the div with the id *app* .

Event Handling

Reacting to User Interaction

- Use `addEventListener` to attach events to elements
- Common events: `click` , `input` , `submit`

```
const button = document.getElementById('submit');
button.addEventListener('click', () => {
  alert('Button clicked!');
});
```

Javascript 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).
- Complex projects necessitate a mechanism for splitting JavaScript programs into separate modules that can be imported when needed.

Why Do We Need Modules?

- JavaScript files can grow large and become hard to maintain
- Furthermore, the HTML on a page is loaded in the order in which it appears, which means we cannot load scripts before the content inside the `<body>` element has finished loading.
- If you try to access an element within the `<body>` tag using `document.getElementById("id-name")` and the element is not loaded yet, then you get an undefined error.

The Old Fashioned Way

The old fashioned way of solving such issue was to load the scripts right before the `</body>` element.

```
<!DOCTYPE html>
  <head>
    <!-- Head content goes here-->
  </head>
  <body>

    <!--HTML content goes here-->
    ...

    <script src="js/jquery.js"></script>
    <script src="js/script2.js"></script>
    <script src="js/script3.js"></script>
    <script src="js/script4.js"></script>

  </body>
</html>
```

But in the long run, the number of scripts adds up and we may end up with 10+ scripts while trying to maintain version and dependency conflicts.

What Are ES6 Modules?

- The solution to these problems is to utilize ES6's syntax , import and export statements.
- It's an elegant and maintainable approach that allows us to keep things separated, and only available when we need it.
- ES6 Modules are introduced in ECMAScript 2015 (ES6).
- Each module has its own scope.
- Works natively in browsers (with `<script type="module" src="/src/main.js"></script>`)

The import and export statements

- The `export` keyword is used when we want to make something available somewhere, and the `import` is used to access what `export` has made available.
- The thumb rule is, in order to `import` something, you first need to `export` it.
- And what can we actually `export` ?
  - A variable
  - An object literal
  - A class
  - A function

Example

```html
<!-- index.html -->

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <title>ES6 Modules Demo</title>
  </head>
  <body>
    <script type="module" src="/src/main.js"></script>
  </body>
</html>
```

```js
// file: /src/main.js

import { add, subtract } from './math.js';
import multiply from './math.js';

console.log(add(2, 3));        // 5
console.log(subtract(5, 2));   // 3
console.log(multiply(3, 4));   // 12
```

```js
// file: /src/math.js

export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// You can also export a default
export default function multiply(a, b) {
  return a * b;
}
```

Exporting in ES6 Modules

There are two main types of exports:

- **Named Exports** : export multiple values with names
- **Default Export** : export a single default value

Named Exports

```
// file: math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

```
// file: main.js
import { add, subtract } from './math.js';

console.log(add(2, 3));        // 5
console.log(subtract(5, 2));   // 3
```

Renaming Named Exports

You can rename the imported values to avoid conflicts:

```
// Import with alias
import { add as sum } from './math.js';

console.log(sum(1, 2));  // 3
```

Default Export

A default export is a single value that is exported from a module. It is the value that is imported when no other name is specified.

```
// file: greet.js
export default function greet(name) {
  return `Hello, ${name}!`;
}
```

```
// file: main.js
import greet from './greet.js';

console.log(greet('Alice'));  // Hello, Alice!
```

Mixing Named and Default Exports

You can mix named and default exports in the same module.

```
// file: helpers.js
export const capitalize = (str) => str[0].toUpperCase() + str.slice(1);
export default function log(msg) {
  console.log(`[LOG]: ${msg}`);
}
```

```
// file: main.js
import log, { capitalize } from './helpers.js';

log(capitalize('hello'));  // [LOG]: Hello
```

Why Use Braces in Imports?

- Braces `{}` are required when importing **named exports** .
- No braces are needed when importing the **default export** .

```
// math.js
export const add = (a, b) => a + b;
export default function multiply(a, b) {
  return a * b;
}
```

```
// main.js
import multiply, { add } from './math.js';

console.log(add(2, 3));       // Named import → needs braces
console.log(multiply(2, 3));  // Default import → no braces
```

Summary

- `export` lets you export multiple values by name
- `export default` allows exporting a single main value
- `import` syntax must match export type (named or default)
- You can combine both in one file

Transpilation

- Node.js has had this ability for a long time, and there are a number of JavaScript libraries and frameworks that enable module usage (CommonJS RequireJS , webpack , Babel , etc.)
- Currently, all modern browsers support module features natively without needing transpilation .
- 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.
- Bundlers still do a good job at partitioning code into reasonably sized chunks and do other optimizations like minification, dead code elimination, and tree-shaking .

What is JSON?

- JSON (JavaScript Object Notation) is a lightweight data format used for storing and exchanging data
- It is easy to read and write for humans, and easy to parse for machines
- Used extensively in web development (APIs, config files, etc.)
- JSON looks like JavaScript objects, but it's a **string** .

Example of JSON

```
{
  "name": "Alice",
  "age": 30,
  "isStudent": false,
  "skills": ["HTML", "CSS", "JavaScript"]
}
```

- Keys are always in double quotes
- Values can be strings, numbers, booleans, arrays, or objects
- jsoncrack.com

Working with JSON in JavaScript

You can use the JSON.parse() and JSON.stringify() methods to convert between JSON and JavaScript objects:

```
const jsonString = '{"name": "Alice", "age": 30}';
const obj = JSON.parse(jsonString); // JSON → JS object

console.log(obj.name); // "Alice"
```

```
const newObj = { name: "Bob", age: 25 };
const newJson = JSON.stringify(newObj); // JS object → JSON string

console.log(newJson); // '{"name":"Bob","age":25}'
```

JSON in Web Development

- JSON is the most common format used in API communication
- Frontend apps use fetch() to retrieve JSON data from servers

```
fetch('/data/profile.json')
  .then(response => response.json())
  .then(data => {
    console.log(data.name); // Access data from JSON
  });
```

Console API

- The Console API provides functionality to allow developers to perform debugging tasks, such as *logging* messages or the *values* of variables at set points in your code, or *timing* how long an operation takes to complete.
- Usage is very simple — the console object contains many methods that you can call to perform rudimentary debugging tasks, generally focused around logging various values to the browser's Web Console.

```
let myString = "Hello world";

// Output "Hello world" to the console
console.log(myString);
```

log(), warn() and error()

- console.log() is used to log messages to the console
- console.warn() is used to log warning messages to the console
- console.error() is used to log error messages to the console

```
const logMessage = "This is a log message";
const warnMessage = "This is a warning message";
const errorMessage = "This is an error message";

console.log(logMessage);
console.warn(warnMessage);
console.error(errorMessage);
```

console.assert()

The `console.assert()` static method writes an error message to the console if the assertion is false. If the assertion is true, nothing happens.

```
const errorMsg = "the # is not even";
for (let number = 2; number <= 5; number++) {
  console.log(`the # is ${number}`);
  console.assert(number % 2 === 0, "%o", { number, errorMsg });
}
// output:
// the # is 2
// the # is 3
// Assertion failed: {number: 3, errorMsg: "the # is not even"}
// the # is 4
// the # is 5
// Assertion failed: {number: 5, errorMsg: "the # is not even"}
```

console.count()

- The `console.count(label)` static method logs the number of times that this particular call to count() has been called.
- If a label is supplied, `count()` outputs the number of times it has been called with that label. If omitted, `count()` behaves as though it was called with the *default* label.

```
function greet(user) {
  console.count();
  return `hi ${user}`;
}

greet("bob");
greet("alice");
greet();
console.count();
```

```
function greet(user) {
  console.count(user);
  return `hi ${user}`;
}

greet("bob");
greet("alice");
greet("alice");
console.count("alice");
```

console.group()

The `console.group()` static method creates a new inline group in the Web console log, causing any subsequent console messages to be indented by an additional level, until `console.groupEnd()` is called.

```
console.log("This is the outer level");
console.group();
console.log("Level 2");
console.group();
console.log("Level 3");
console.warn("More of level 3");
console.groupEnd();
console.log("Back to level 2");
console.groupEnd();
console.log("Back to the outer level");
```

console.trace()

The `console.trace()` static method outputs a stack trace to the console.

```
function foo() {
  function bar() {
    console.trace();
  }
  bar();
}

foo();
```

Package Managers

- Package managers are tools that help install, update, and remove software pieces (packages).
- They manage versions and what other packages are needed. Examples are *npm* for JavaScript and *pip* for Python.
- They make it easy to share and reuse code by keeping packages in one place.
- Package managers simplify project setup and help keep things consistent. They are very important for modern software development by making work smoother and improving teamwork.
- Using *npm* effectively is a cornerstone of modern web development, no matter if it's exclusively with Node.js, as a package manager or build tool for the front-end, or even as a piece of workflows in other languages and on other platforms.

package.json

- As a general rule, any project that's using Node.js will need to have a `package.json` file. What is a `package.json` file?
- At its simplest, a `package.json` file can be described as a manifest of your project that includes the packages and applications it depends on, information about its unique source control, and specific metadata like the project's name, description, and author.
- A `package.json` file is *always* structured in the JSON format, which allows it to be easily read as metadata and parsed by machines.

```
{
  "name": "metaverse", // The name of your project
  "version": "0.92.12", // The version of your project
  "description": "The Metaverse virtual reality. The final outco
  "main": "index.js"
  "license": "MIT" // The license of your project
}
```

Dependencies

- The other majorly important aspect of a `package.json` is that it contains a collection of any given project's *dependencies* . These dependencies are the modules that the project relies on to function properly.
- Having dependencies in your project's `package.json` allows the project to install the versions of the modules it depends on. By running an install command (see the instructions for `npm install` below) inside of a project, you can install *all* of the dependencies that are listed in the project's `package.json` - meaning they don't have to be (and almost never should be) bundled with the project itself.

Dependencies

- Second, it allows the separation of dependencies that are needed for production and dependencies that are needed for development.
- In production, you're likely not going to need a tool to watch your CSS files for changes and refresh the app when they change.
- But in both production and development, you'll want to have the modules that enable what you're trying to accomplish with your project - things like your web framework, API tools, and code utilities.

Example

```
{
  "name": "metaverse",
  "version": "0.92.12",
  "description": "The Metaverse virtual reality. The final outcome of all virtual worlds, augmented reality, and the Internet.",
  "main": "index.js"
  "license": "MIT",
  "devDependencies": {
    "mocha": "~3.1",
    "native-hello-world": "^1.0.0",
    "should": "~3.3",
    "sinon": "~1.9"
  },
  "dependencies": {
    "fill-keys": "^1.0.2",
    "module-not-found-error": "^1.0.0",
    "resolve": "~1.1.7"
  }
}
```

One key difference between the dependencies and the other common parts of a `package.json` is that they're both objects, with multiple key/value pairs. Every key in both `dependencies` and `devDependencies` is a name of a package, and every value is the version range that's acceptable to install.

Using npm init to Initialize a Project

The `npm init` command is a step-by-step tool to scaffold out your project. It will prompt you for input for a few aspects of the project in the following order:

- The project's name,
- The project's initial version,
- The project's description,
- The project's entry point (meaning the project's main file),
- The project's test command
- The project's git repository (where the project source can be found)
- The project's keywords (basically, tags related to the project)
- The project's license (this defaults to ISC - most open-source Node.js projects are MIT)

If you want to get on to building your project, and don't want to spend the (albeit brief) time answering the prompts that come from `npm init` , you can use the `--yes` flag on the `npm init` command to automatically populate all options with the default `npm init` values.

Install Modules with npm install

Installing modules from npm is one of the most basic things you should learn to do when getting started with npm.

```
npm install <module>
```

In the above command, you'd replace `<module>` with the name of the module you want to install. For example, if you want to install Express (the most used and most well known Node.js web framework), you could run the following command:

```
npm install express
```

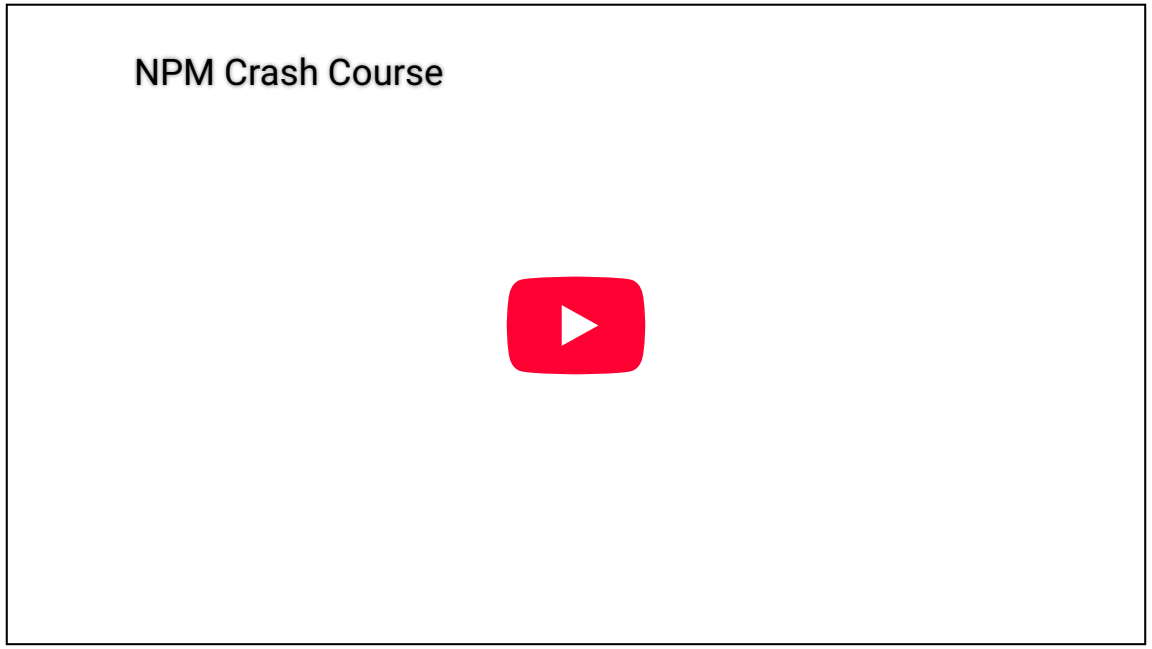
The above command will install the `express` module into `/node_modules` in the current directory. Whenever you install a module from npm, it will be installed *into* the `node_modules` folder.

Install Modules with npm install

In addition to triggering an install of a single module, you can actually trigger the installation of *all* modules that are listed as `dependencies` and `devDependencies` in the `package.json` in the current directory. To do so, you'll simply need to run the command itself:

```
npm install
```

npm crash course



References

- MDN Web Docs
  - Language Overview
- Hello JavaScript
- W3Schools
- Full Stack Open
- Youtube series - Functional Programming in JavaScript
- Eloquent JavaScript
- Namaste JavaScript
- JavaScript.info
- You Don't Know JS

Module 03 Exercise

JavaScript Fundamentals: Budget Tracker

Create a simple budgeting page using vanilla JavaScript. Your page must:

- Include an input for entering an expense name with `name="expense-name"`
- Include an input for entering the amount with `name="expense-amount"`
- A button to add the expense to a list
- Show the total amount spent, calculated as new items are added. Display the total in an element with `id="total"`
- Prevent adding empty or invalid values (name must not be empty, amount must be a positive number)

Dynamically render the list of expenses using JavaScript and calculate the total. Do not use frameworks.

Add this exercise to your repository under:

`/exercises/03/`

Deadline: To be announced