Modern JavaScript Development: Modules and Tooling"!

This diagram represents the **modern JavaScript development workflow**. It highlights the processes and tools involved in creating, bundling, and optimizing JavaScript code for production. Here's a breakdown:

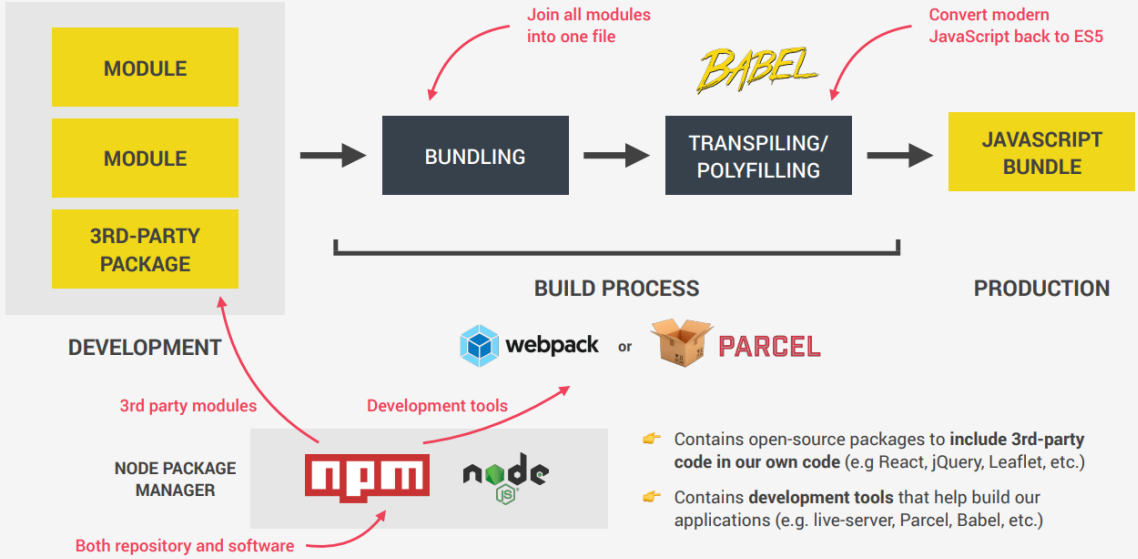
Key Steps in the Process:

1. **Modules & 3rd-Party Packages**:
    - **Modules**: Individual pieces of JavaScript code (often organized into separate files).
    - **3rd-Party Packages**: Libraries and tools downloaded from repositories like **npm** (e.g., React, jQuery).
2. **Node Package Manager (npm)**:
    - Acts as a repository for packages and a tool for managing dependencies.
    - Developers install 3rd-party modules and tools required for their projects using npm.
3. **Bundling**:
    - Tools like **Webpack** or **Parcel** combine all modules (your code and 3rd-party packages) into a single JavaScript file or smaller chunks.
    - Bundling reduces the number of HTTP requests and optimizes the code for browsers.
4. **Transpiling & Polyfilling**:
    - Using tools like **Babel**, modern JavaScript (ES6+) is converted into older versions (e.g., ES5) for compatibility with older browsers.
    - Polyfills are used to add support for missing features in environments that do not natively support them.
5. **JavaScript Bundle**:
    - The final output is an optimized bundle ready for production.

Tools:

- **Build Process**: Webpack and Parcel handle module bundling and optimization.
- **Development Tools**: Include Babel for transpilation, live-server for quick testing, etc.
- **Node.js**: Executes JavaScript on the server and is a runtime for tools like npm.

MODERN JAVASCRIPT DEVELOPMENT



- Contains open-source packages to **include 3rd-party code in our own code** (e.g React, jQuery, Leaflet, etc.)
- Contains **development tools** that help build our applications (e.g. live-server, Parcel, Babel, etc.)

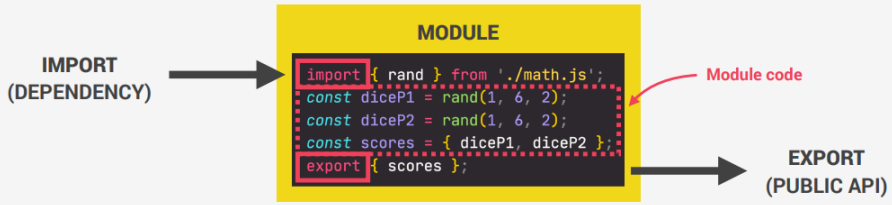
AN OVERVIEW OF MODULES

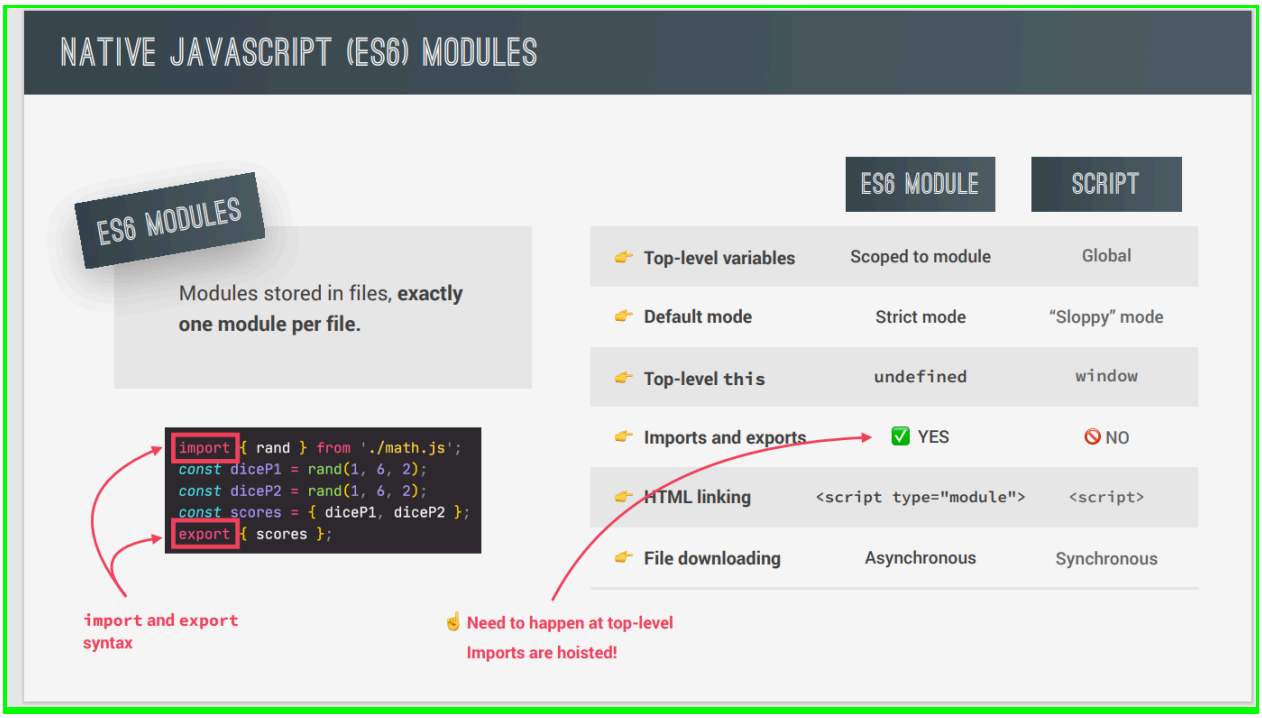
MODULE

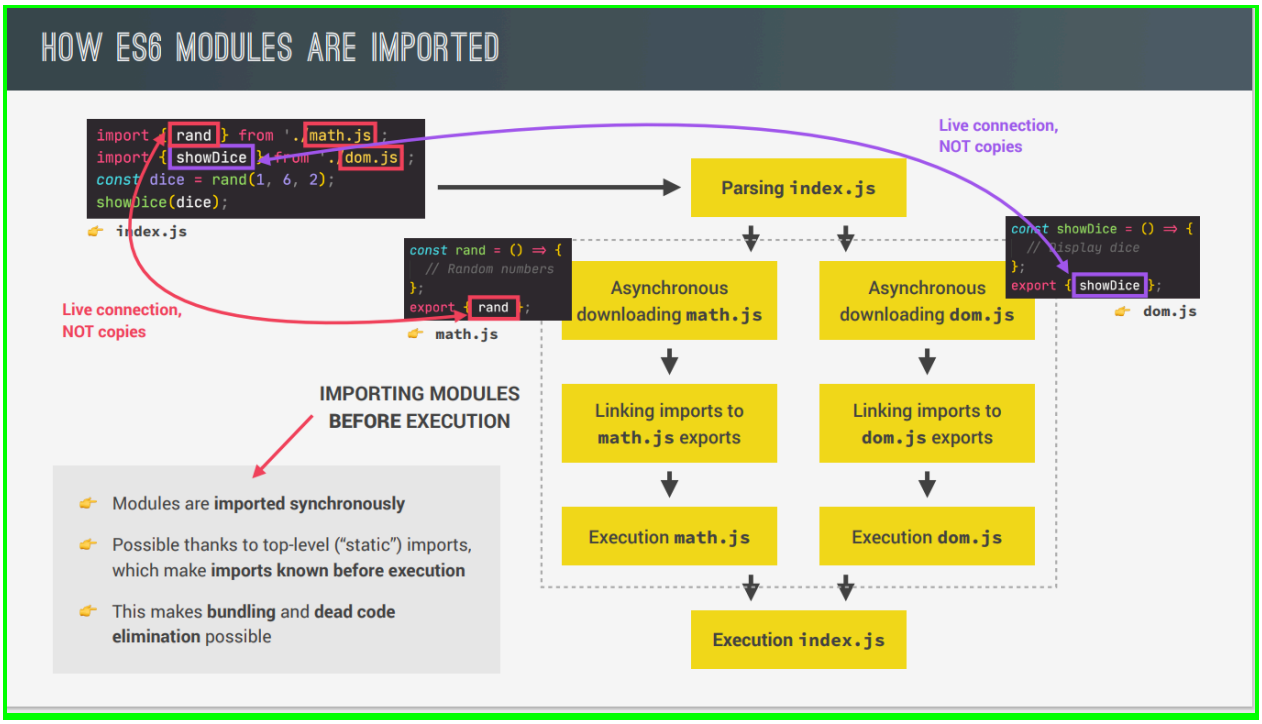
- Reusable piece of code that **encapsulates** implementation details;
- Usually a **standalone file**, but it doesn't have to be.

WHY MODULES?

- **Compose software:** Modules are small building blocks that we put together to build complex applications;
- **Isolate components:** Modules can be developed in isolation without thinking about the entire codebase;
- **Abstract code:** Implement low-level code in modules and import these abstractions into other modules;
- **Organized code:** Modules naturally lead to a more organized codebase;
- **Reuse code:** Modules allow us to easily reuse the same code, even across multiple projects.







Detailed Explanation of ES6 Modules

Modules in JavaScript (introduced in ES6) are a way to organize and structure code for better reusability, maintainability, and modularity. Let's go step by step into their features and how they work.

What Are Modules?

- **Definition**: A module is a file containing JavaScript code that is self-contained. Each module can export pieces of functionality (e.g., variables, functions, or classes) and import functionality from other modules.
- **Encapsulation**: Modules keep details private unless explicitly exposed through export.
- **Standalone Files**: While modules are usually individual files, they can be combined into bundles using tools like Webpack or Parcel.

Why Use Modules?

1. **Compose Software**:
    - Modules act as building blocks that combine to form a full application.
    - For example, a module might handle user authentication, another might fetch data, and another might render UI components.
2. **Isolate Components**:
    - Each module can be developed and tested in isolation without worrying about the entire codebase.
    - For example, you can modify a math.js module without affecting dom.js.
3. **Abstract Code**:
    - Modules allow you to encapsulate low-level logic and expose a high-level interface for other parts of the application to use.
    - For example, a data.js module might provide functions to fetch user data, abstracting the underlying API details.
4. **Organized Code**:
    - Using modules leads to a naturally structured codebase, making it easier to navigate, debug, and extend.
5. **Reuse Code**:
    - Modules allow sharing functionality across projects or within different parts of the same project.
    - For instance, a utility module for date formatting can be reused across multiple applications.

Key Components of ES6 Modules

1. **Exporting**:

    Use the export keyword to expose variables, functions, or classes to other modules.

Named Exports:

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

Multiple named exports are possible.

Default Export:

```
// logger.js
export default function log(message) {
  console.log(message);
}
```

Only one default export is allowed per module.

Importing:

Use the import keyword to bring in functionality from another module.

Named Imports:

```
import { add, subtract } from './math.js';
console.log(add(2, 3)); // Output: 5
```

Default Import:

```
import log from './logger.js';
log('Hello, world!'); // Output: Hello, world!
```

How ES6 Modules Are Imported

1. **Parsing the Importer**:
    - The importing module (index.js) specifies the modules it depends on using import statements.

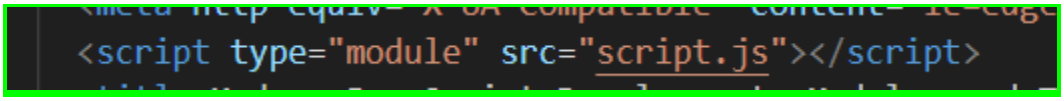
For example:

```
import { rand } from './math.js';
import { showDice } from './dom.js';
```

2. **Asynchronous Loading**:

    - The browser or Node.js downloads the specified modules (math.js, dom.js) asynchronously.
3. **Linking Exports**:
    - The imported modules are linked to their exports. Importing modules do not receive copies of the exported values; they get a live connection to the original module.
4. **Execution**:
    - The imported modules execute in order, resolving dependencies as needed.
    - For example:
        - math.js defines rand and exports it.
        - dom.js defines showDice and exports it.
        - index.js combines these to create a complete application.

type= module in <Script> tag



All modules are executed in ‘strict mode ‘ by default.

Imports are not copy of exports , they are in live-connection

Here's a complete explanation of **exports and imports** in JavaScript, presented in sequence for better understanding:

1. Named Exports

- Named exports allow you to export multiple values from a module.
- Each export must have a unique name, and you must use these exact names during import.

Syntax:

1)Export:

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

Import:

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

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

Renaming during import:

```
import { add as sum, subtract as difference } from './math.js';

console.log(sum(3, 2));       // 5
console.log(difference(5, 3)); // 2
```

2) Renaming Exports

Module: math.js

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

// Renaming during export
export { add as sum, subtract as difference };
```

Importing Renamed Exports:

```
import { sum, difference } from './math.js';

console.log(sum(3, 2));        // 5
console.log(difference(10, 4)); // 6
```

scritp.js

```
import { addTocart, cart, values } from './shoppingCart.js';
//importing module
console.log('importing module');
addTocart('bread', 23);
addTocart('apples', 23);
console.log(cart);
console.log(values);
```

shopppingCart.js

```
//exporting module
console.log('exporting module');
const shippingCost = 18;
//named export
export const values = [1, 2, 3, 4, 5, 6];
export const cart = [];
export const addTocart = function (product, quantity) {
  cart.push(product, quantity);
  console.log(`${quantity} ${product} added to cart`);
};
```

2. Default Exports

- A default export is used when a module has a single primary functionality to export.
- It can be imported with any name of your choice.

Syntax:

Export:

```
// divide.js
export default function divide(a, b) {
    return a / b;
}
```

Import:

```
import divideFunction from './divide.js';

console.log(divideFunction(10, 2)); // 5
```

You are free to import the default export with any name. For example, divideFunction could just as easily be divide, calc, or anything else.

3. Combined Named and Default Exports

A module can have both named exports and a default export.

Syntax:

Export:

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

Import:

```
import divide, { multiply } from './calculator.js';

console.log(multiply(4, 5)); // 20
console.log(divide(20, 4));  // 5
```

```
//exporting module
console.log('exporting module');
const shippingCost = 18;
//named export
export const values = [1, 2, 3, 4, 5, 6];
export const cart = [];
export const addTocart = function (product, quantity) {
  cart.push(product, quantity);
  console.log(`${quantity} ${product} added to cart`);
};
export default function add(a, b) {
  console.log(`${a * b}`);
}
```

```
import add, { addTocart, cart, values } from './shoppingCart.js';
//importing module
console.log('importing module');
addTocart('bread', 23);
addTocart('apples', 23);
```

```
console.log(cart);
console.log(values);
add(2, 3);
```

4. Using * to Import All as an Object

- When you use import * as, all exports (both named and default) are collected into an object.
- Named exports are properties of the object, and the default export is accessed via the default property.

Syntax:

Export:

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

Import:

```
import * as utils from './utils.js';

console.log(utils.add(3, 2));       // 5 (named export)
console.log(utils.subtract(7, 4));  // 3 (named export)
console.log(utils.default(3, 4));   // 12 (default export)
```

```
// exporting module

console.log('exporting module');

const shippingCost = 18;

// Named export

export const values = [1, 2, 3, 4, 5, 6];
```

```
export const cart = [];

export const addTocart = function (product, quantity) {

  cart.push(product, quantity);

  console.log(`${quantity} ${product} added to cart`);

};

// Default export

export default function add(a, b) {

  console.log(`${a * b}`);

}
```

```
// importing module

import * as obj from './shoppingCart.js';

console.log('importing module');

// Using named exports

obj.addTocart('bread', 23);

obj.addTocart('apples', 23);

console.log(obj.cart); // ["bread", 23, "apples", 23]

console.log(obj.values); // [1, 2, 3, 4, 5, 6]

// Using the default export

obj.default(2, 3); // Logs: 6
```

When you import everything as obj using import * as obj, the default export (which is the add function) will be available as a property of the object named default.

So, to access the default export (add), you need to call obj.default(2, 3) instead of obj.add(2, 3).

Top-level Await(ES-22)usage (not recommended sometimes ,we should use depends):

Let's break down the code step by step and then explain the behavior of top-level await in detail, as well as the reasons why it might not be recommended to use in certain cases.

ex:

```
console.log("start");
console.log("importing module");

const res = await fetch('https://jsonplaceholder.typicode.com/posts');
const data = await res.json();

console.log(data);
console.log("end");
```

Step-by-Step Explanation:

console.log("start");

1. The first line runs immediately, and "start" is printed to the console.

Importing Module:
console.log("importing module");

2. The second line runs immediately, and "importing module" is printed to the console.

Top-level await:
const res = await fetch('https://jsonplaceholder.typicode.com/posts');

3. Here, the code encounters the await keyword. This means:

    - The code pauses at this point and waits for the fetch request to complete and resolve a response.
    - fetch() returns a Promise, which is why await is used here. The execution of the remaining code does not continue until this promise resolves.
4. While waiting, other asynchronous code (outside of this specific context) can continue running (for example, other events or timers). However, this specific execution will "pause" here, and the next line won't run until the fetch request is done.

Fetching JSON:
const data = await res.json();

5. After the fetch resolves and a response (res) is returned, this line will execute next.
    - The .json() method is used to parse the response into JSON format, and once again, it returns a Promise. The await pauses execution again until that promise resolves and data is available.

console.log(data);

6. After the await on the .json() promise resolves, this line will execute, and data will be printed to the console.

console.log("end");

7. Finally, after the data is logged, the last console.log("end") will execute, printing "end" to the console.

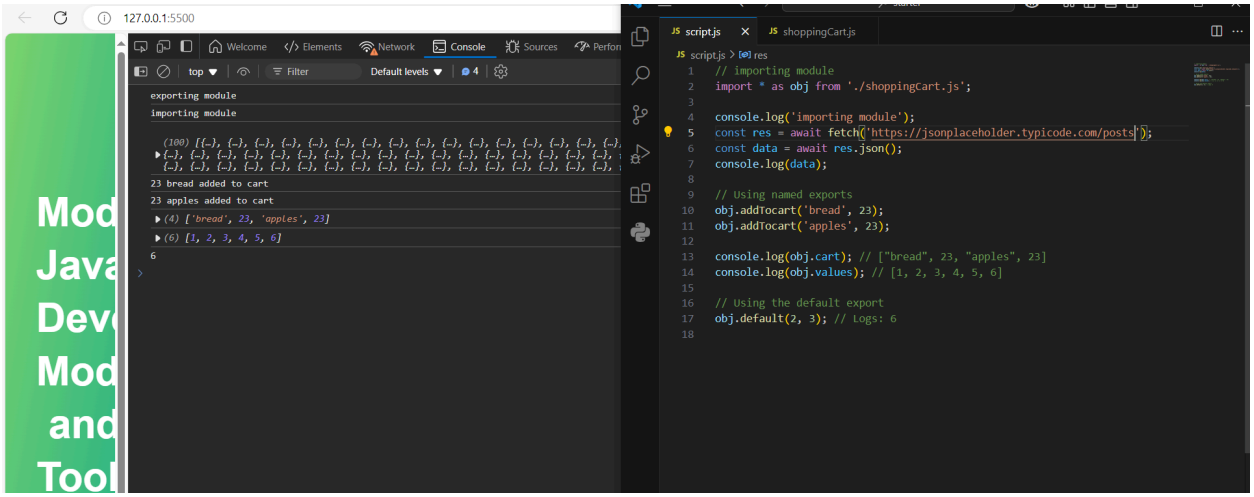
Top-Level await Behavior:

- How it Pauses: The await on the fetch() call pauses the execution of the subsequent lines in the current module until the promise returned by fetch() resolves. This means that the code does not immediately proceed to the next line (like console.log(data)), but instead waits for the response of the fetch request.
- Sequential Execution: After await fetch() resolves, the next line (await res.json()) will also pause until the response is parsed into JSON. Once that promise resolves, the next lines (console.log(data) and console.log("end")) will execute in order.

Top-level await is sometimes not recommended because:

1. Blocking Execution: It pauses the module execution until the promise resolves, which can delay the entire module and prevent parallel execution of asynchronous tasks.
2. Readability and Debugging: It can make the code harder to understand and debug, as the flow of execution isn't as explicit.

3. Error Handling: It requires proper error handling (e.g., try-catch) to avoid unhandled promise rejections, which can complicate error management.
4. Compatibility: Older environments or non-module scripts may not support top-level await, limiting compatibility.



Module Pattern

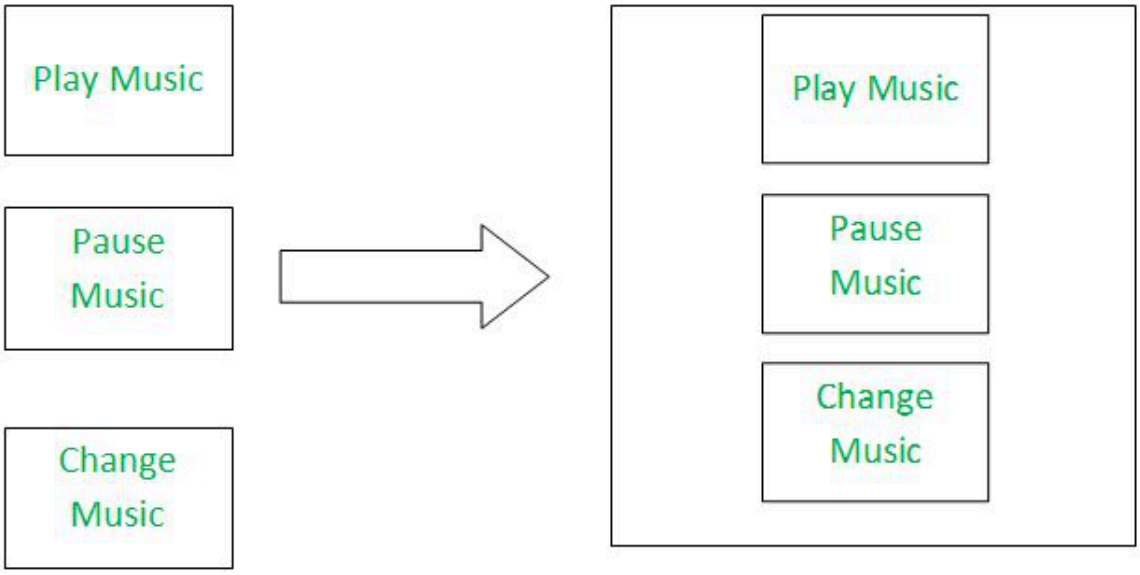
The Module Pattern in JavaScript is a design pattern used to organize code in a way that encapsulates functionality and keeps the global scope clean. It allows you to create private variables and methods that cannot be accessed directly from the outside, exposing only specific parts of the module that you choose to share.

Key Concepts of the Module Pattern:

1. **Encapsulation**: By using the module pattern, you can hide the internal implementation details of the module.
2. **Private and Public Members**: The pattern allows you to define private variables and functions, which can't be accessed from the outside, and public methods, which can be accessed.

Revealing Module Pattern is JavaScript's design pattern that is available for users to actually **organize JavaScript's codes in modules which actually gives better code structure and helps in making things either private or public to users.** This particular design pattern allows the script to be more consistent which helps an individual to identify at the end which method or variable will be privately or publicly accessible which eases readability.

Let's have a look at the illustrated pictorial representation which will help us to understand this design pattern more nicely as well as clearly.



Example of Revealing Module Pattern in JavaScript

As shown in the above pictorial representation (an example of how Music System functions), several methods (including Play Music, Pause Music, and Change Music) combined together (separately) makes a big functional functionality working (which is Music System working). These shown methods would be embedded inside the bigger method which is the main method visible to the user, which will function at first itself.

Syntax: The following syntax gives us a rough idea of how we may declare any working functionality using this design pattern (Note that this syntax is based on the new Arrow function syntax, we may also use simple functions instead of arrow functions):

```
let function_name = () => {

    let first_function = () => {
        // do something...
    }

    let second_function = () => {
        // do something...
```

```
    }

  // More functions we may add on....

  return {
      calling_method_name : original_method_name,
      ...
  }
}
```

Ex1:

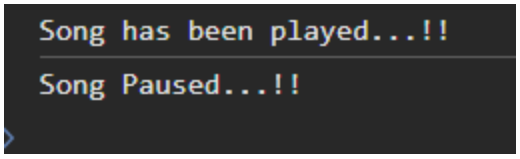
```
const counterModule = (function () {
  // Private variables and functions
  let count = 0;

  function increment() {
    count++;
  }

  function decrement() {
    count--;
  }

  function getCount() {
    return count;
  }

  // Public API (Exposing methods)
  return {
    increment: increment,
    decrement: decrement,
    getCount: getCount,
  };
})();

// Using the module
```

```
counterModule.increment();
counterModule.increment();
console.log(counterModule.getCount()); // Output: 2
counterModule.decrement();
console.log(counterModule.getCount()); // Output: 1
```

The **counterModule** is an IIFE (Immediately Invoked Function Expression), which creates a private scope.
Inside this scope, the variable count and the functions increment and decrement are private and cannot be accessed directly from outside.
The return object exposes the public methods increment, decrement, and getCount, allowing access to the private functionality while keeping the internals hidden.

Ex2:

```
let musicPlayer = () => {
  let playSong = () => {
    console.log('Song has been played...!!');
  };

  let pauseSong = () => {
    console.log('Song Paused...!!');
  };

  return {
    playMusic: playSong,
    pauseMusic: pauseSong,
  };
};

let music_system = musicPlayer();
music_system.playMusic();
music_system.pauseMusic();
```



commonJS Modules

CommonJS modules (often referred to as **CJS** modules) are a specification for how JavaScript modules should be structured and how code can be organized in a reusable way. This module system is commonly used in **Node.js** to enable modularity, which allows JavaScript code to be split into smaller, manageable, and reusable pieces.

Here are the key points about CommonJS modules:

require(): The require() function is used to import modules. This function allows one file to access the functions, objects, or variables exported from another file.

Example:
```
const myModule = require('./myModule');
```

module.exports: This is used to export a module's functionality so that other files can access it via require(). Anything assigned to module.exports is made available for import in other files.

Example:
```
module.exports = function() {
  console.log("Hello, CommonJS!");
};
```

Example of a simple CommonJS module:

myModule.js:

```
module.exports = {
  greet: function(name) {
    console.log(`Hello, ${name}`);
  }
};
```

app.js:

```
const myModule = require('./myModule');

myModule.greet('World');  // Output: Hello, World
```

CommonJS is **not deprecated**. It is still widely used, especially in **Node.js** environments, for building server-side applications. While **ES Modules (ESM)** have become the modern standard for JavaScript module systems, CommonJS continues to be supported and is actively used in many projects.

However, there are some important things to note:

ES Modules (ESM): The ES Module system, introduced in ECMAScript 6 (ES6), has become the preferred module system in modern JavaScript for both client-side (browsers) and server-side (Node.js) applications. It uses import and export syntax instead of require() and module.exports.

Example of ESM:

```
// ES Module
import { greet } from './myModule.js';

greet('World');
```

Command Line

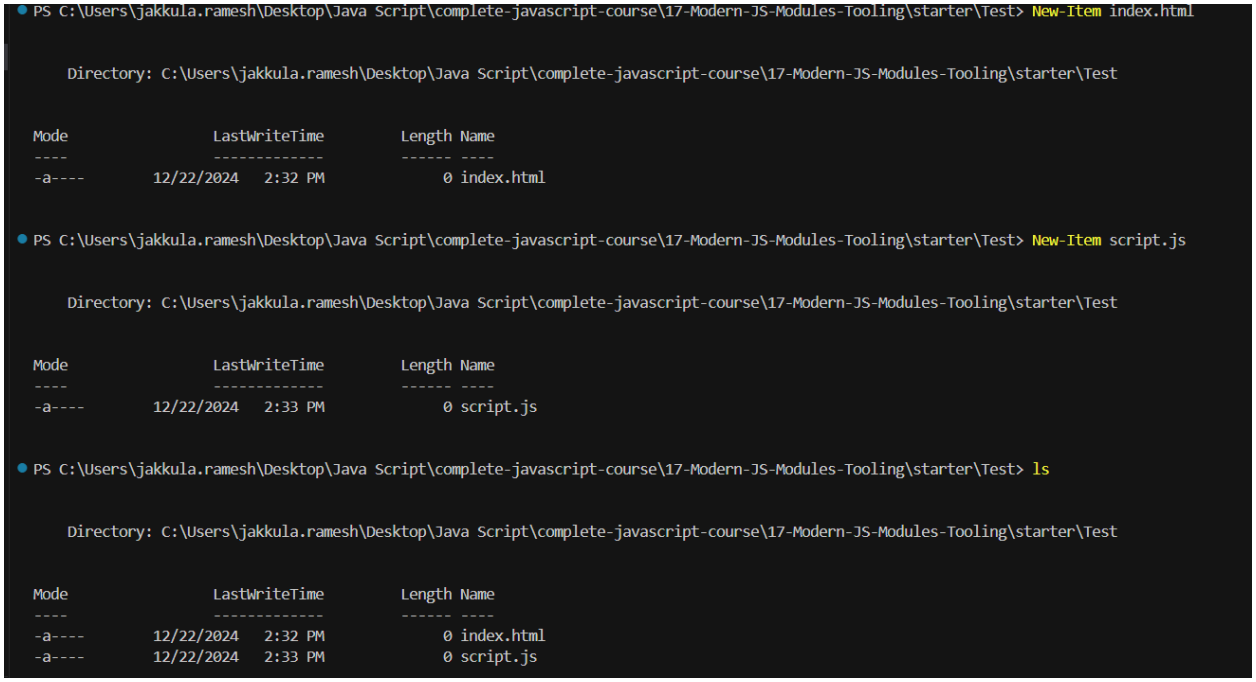
In vs code terminal cmd
1) ls
2) cd ..
3) clear
4) mkdir “folder name”
5) New-Item index.html : **Create empty file**: New-Item <filename> -ItemType File
6) rm index.html script.js

```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test> rm index.html
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test> rm script.js
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test> ls
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test>
```

7) rmdir : you want to **remove a directory,If the directory is not empty,**
Make sure you're not inside the Test directory when trying to delete it. You cannot delete a directory if you're currently in it.

```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test> rmdir Test
rmdir : Cannot find path 'C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test\Test' because it does not exist.
At line:1 char:1
+ rmdir Test
+ ~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\Users\jakkul...arter\Test\Test:String) [Remove-Item], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand

PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter\Test> cd ..
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter> rmdir Test
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter>
```

```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-To
oling\starter> ls


    Directory: C:\Users\jakkula.ramesh\Desktop\Java
    Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter

Mode                 LastWriteTime         Length Name
-a----         12/3/2024   9:57 AM             56 .prettierrc
-a----         12/3/2024   9:57 AM           1610 clean.js
-a----        12/22/2024  11:14 AM            847 index.html
-a----        12/22/2024   2:15 PM           1138 script.js
-a----        12/22/2024  12:41 PM            407 shoppingCart.js


PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-To
oling\starter> cd ..
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-To
oling> ls


    Script\complete-javascript-course\17-Modern-JS-Modules-Tooling

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         12/3/2024   9:57 AM                final
d-----        12/22/2024  11:11 AM                starter


PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-To
oling> cd final
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-To
oling\final> ls
```



NPM

NPM stands for **Node Package Manager**. It is a package manager for the **JavaScript** programming language, primarily used for managing libraries and dependencies for Node.js projects. NPM allows developers to easily install, share, and manage reusable code modules (called packages) that can be included in projects.

NPM (Node Package Manager) is essential because it helps developers efficiently manage and share dependencies, which are libraries or code that your project relies on to function correctly. Without a package manager like NPM, developers would have to manually download, organize, and maintain dependencies, which could become complex and error-prone as projects grow.

Before NPM (Manually Managing Dependencies)

Imagine you're building a JavaScript project that requires a library like **Lodash** for utility functions. Without NPM, you would need to:

1. Download the Lodash(or any other file) library from a website.
2. Manually place the downloaded file in your project.
3. Keep track of the version and any updates yourself.
4. If other developers are working on the project, you would need to share the library files manually.

This approach can quickly become chaotic and hard to maintain, especially as your project grows and you add more libraries.

After NPM (Using NPM to Manage Dependencies)

Now, with NPM, the process becomes much simpler. You can:

1. Use the command npm install lodash to install the Lodash library.
2. NPM will automatically download the library and its dependencies (if any) from the **NPM registry** and place them in your project.
3. NPM keeps track of the version of Lodash you're using, making it easy to update later with a command like npm update lodash.
4. If you're working with a team, you can simply share a file called package.json that lists all the required dependencies, and anyone can run npm install to install the same dependencies automatically.

Simple Example:

Before NPM:

You want to use the **Lodash** library in your project. You would have to manually:

- Download the Lodash JavaScript file.
- Include it in your project folder.
- Manage the version manually.

html

```
<!-- Manually adding Lodash to your project -->
<script src="path/to/lodash.js"></script>
```

After NPM:

1. Initialize a new Node.js project by running npm init -y.
2. Install Lodash with NPM: npm install lodash.
3. Use it in your code like this:

```
// Using Lodash after installing via NPM
const _ = require('lodash');

let array = [1, 2, 3, 4];
let shuffled = _.shuffle(array);
```

```
console.log(shuffled);
```

Key Advantages of NPM:

1. **Simplifies Dependency Management**: With NPM, you don't need to worry about downloading or organizing libraries manually.
2. **Easy Versioning**: NPM ensures you're using the right version of a library, and it handles updates for you.
3. **Collaboration**: NPM makes it easy to share code and dependencies with other developers, ensuring consistency across environments.

Practical example in daily life

Before NPM (Without a Package Manager):

- Imagine you're building a house.
- To complete the house, you need different materials (bricks, cement, wood, etc.).
- Without a package manager, you have to:
    - Go to different stores to buy each material.
    - Keep track of how much of each material you have.
    - Make sure everything is delivered correctly, and organize the materials yourself.

In software development, this is like downloading and managing libraries or tools manually. It becomes time-consuming, error-prone, and hard to keep everything in order.

After NPM (With a Package Manager):

- Now, instead of doing everything yourself, you have a helper.
- You simply tell the helper what materials you need (like "I need bricks, cement, and wood"), and the helper goes to the stores and gets everything for you.
- The helper makes sure everything is delivered on time and keeps track of how much of each material you have.

1. Initialize a Project

npm init

Initializes a new Node.js project and creates a package.json file. This file keeps track of your project's dependencies and scripts.

```
npm init
```

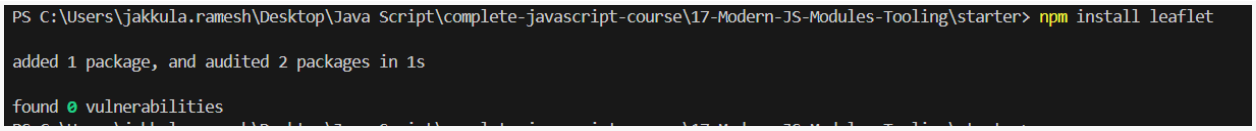
npm init -y

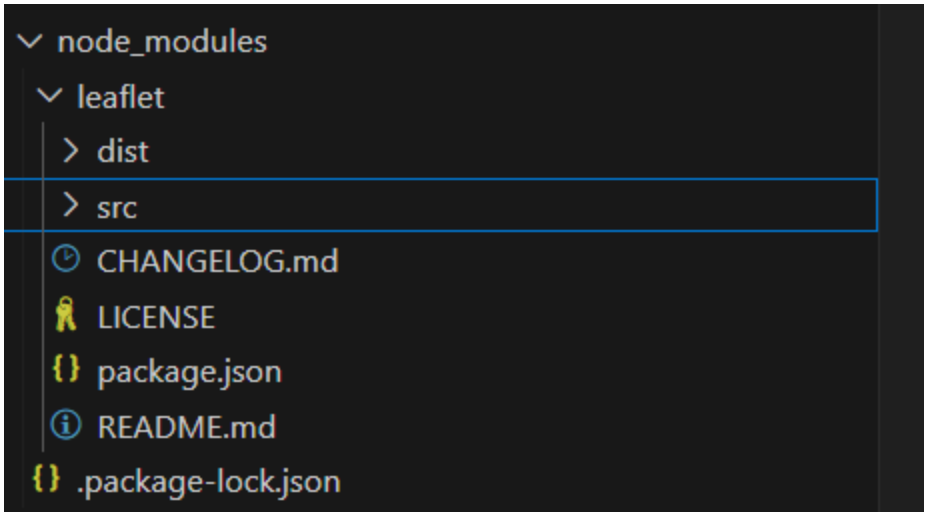
Initializes a new project with default values, skipping the interactive prompts.
npm init -y

2. Install Packages

npm install [package-name]
Installs a package and adds it to the node_modules folder. By default, it installs the latest version.

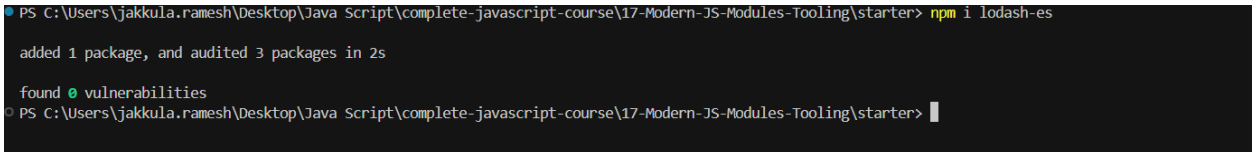
npm install leaflet

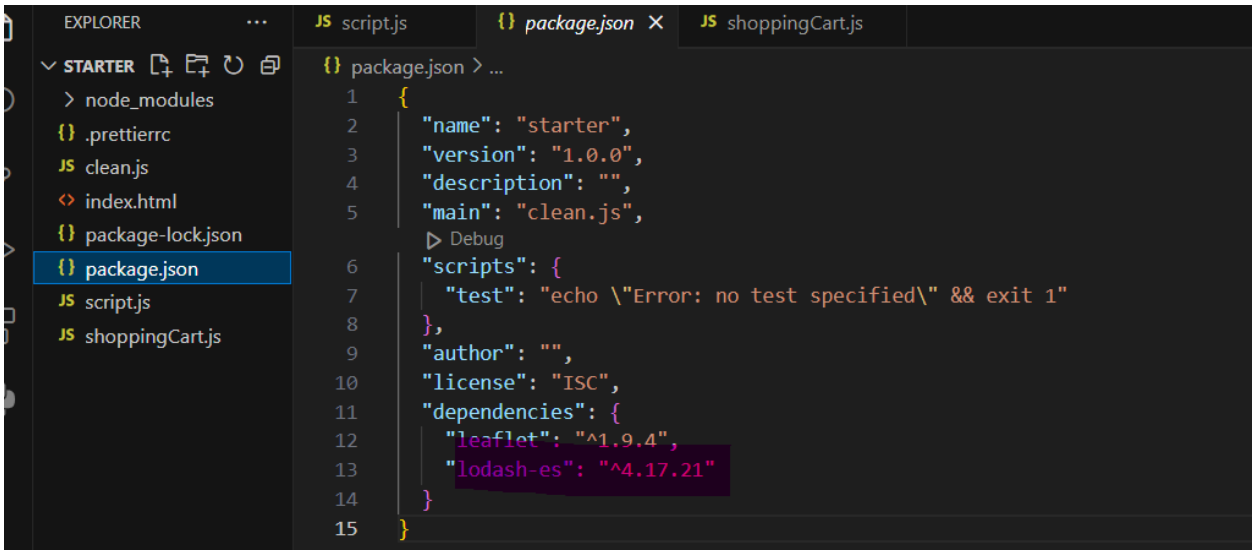


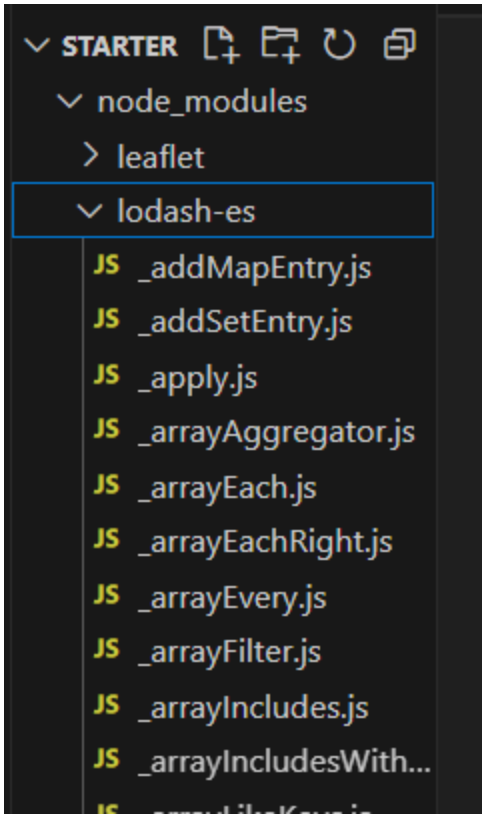


ii) npm i lodash-es

Lodash simplifies JavaScript coding by providing utility functions for common tasks like array manipulation, object handling, and string operations, enhancing code readability and efficiency. It ensures cross-browser compatibility, reducing bugs and inconsistencies across different environments.







npm install [package-name] --save

Installs a package and adds it to the dependencies section of the package.json file. (This is the default in recent versions of NPM.)

npm i

The command npm i is a shorthand for npm install. It is used to install all the dependencies listed in your project's package.json file. When you run npm i in the terminal, npm will:

1. Look for a package.json file in the current directory.
2. Install all dependencies listed in the dependencies and devDependencies sections of the file.
3. Create a node_modules folder in your project directory to store the installed packages.

If you want to share your project with other developers, you shouldn't share the node_modules folder. Instead, other developers can simply run npm i to install the required dependencies listed in the package.json file.

3. Update Packages

npm update

i) Updates all the installed packages to the latest versions according to the version rules in package.json.

npm update

ii) Updates a specific package to the latest version.
npm update [package-name]

Ex: npm update lodash

4. Remove Packages

- **npm uninstall [package-name]**
  Uninstalls a package and removes it from the node_modules folder and the package.json file.

  npm uninstall lodash

5. List Installed Packages

npm list
i) Lists all the installed packages in the current project.

npm list

ii) Lists all globally installed packages. so it can be used across all projects on your system.

npm list --global

6. Check NPM Version

- **npm --version** or **npm -v**
  Shows the version of NPM installed on your system.
  npm --version

Bundling with Parcel and Npm Scripts

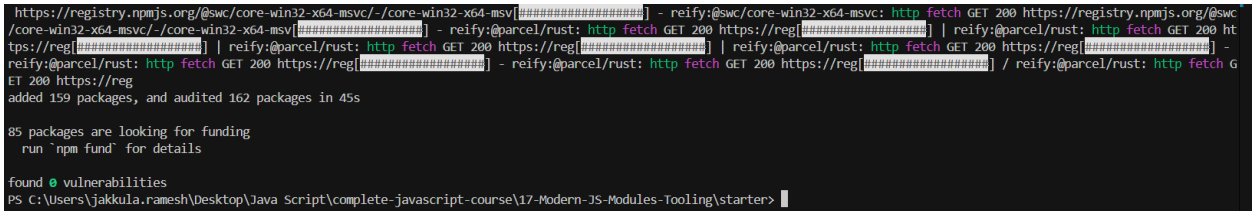
Parcel is a zero-configuration web application bundler that simplifies **the process of building and packaging JavaScript applications**. **It bundles your JavaScript, CSS, HTML, and other assets into optimized files for deployment**. Parcel is known for its simplicity and speed, and it requires minimal setup compared to other bundlers like Webpack.

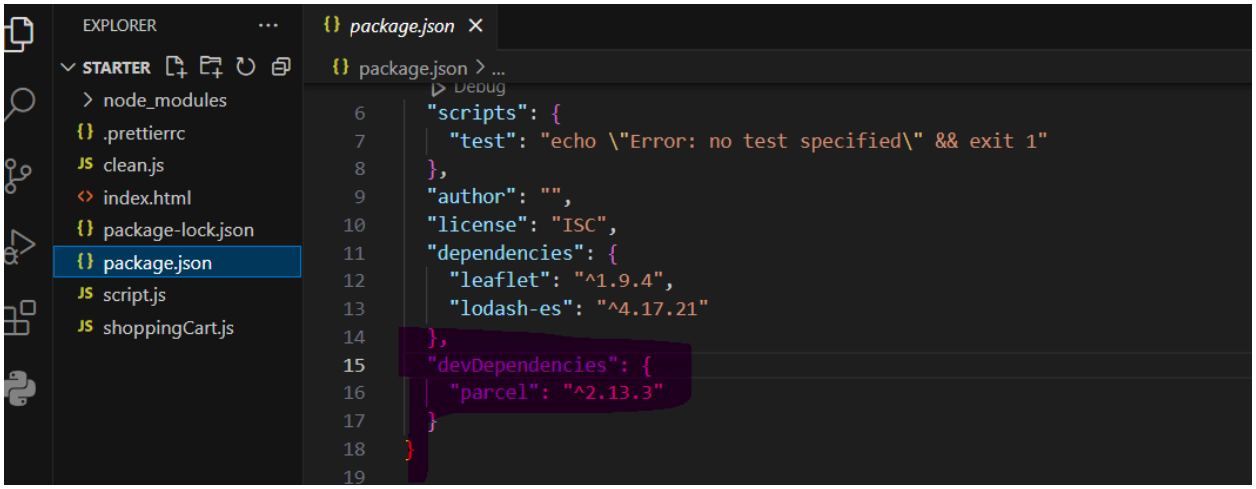
Install Parcel

You can install Parcel globally or as a dev dependency in your project.

Locally (recommended for projects)

npm install --save-dev parcel





Globally

npm install -g parcel-bundler

Key Features of Parcel:

1. **Zero Configuration**: No need for extensive configuration files.
2. **Automatic Code Splitting**: It optimizes your application by splitting it into smaller chunks for faster loading.
3. **Hot Module Replacement (HMR)**: Automatically updates changes in the browser without a full reload.
4. **Out-of-the-box Support**: Supports JavaScript, CSS, images, and other file types without additional configuration.
5. **Tree Shaking**: Removes unused code from the final bundle.

Here's how Parcel bundling works:

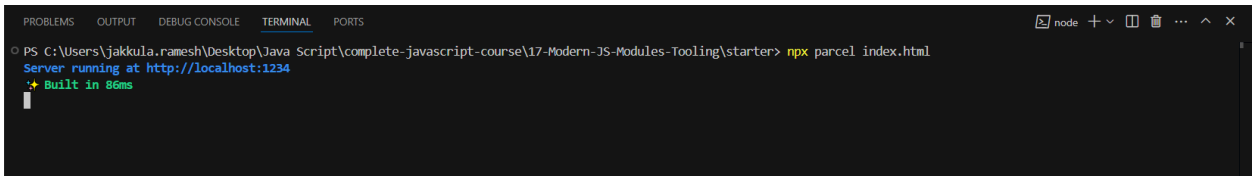
1. **Entry Point**: You define an entry point (e.g., index.html or index.js) which Parcel will use to start the bundling process.
2. **Transformation**: Parcel uses the appropriate transformers (e.g., Babel for JavaScript, PostCSS for CSS, etc.) to process files.
3. **Bundling**: Parcel creates one or more bundles, depending on how your code is structured and how you use dynamic imports.
4. **Output**: The output will be optimized for production with minification and other improvements like caching and code-splitting.

We can run it in two main ways: using npx or npm scripts. Here's how you can do both:

1. Running Parcel with npx

npx is a tool that comes with npm (since version 5.2) and allows you to run binaries from node_modules directly without needing to install them globally.

If you want to run Parcel using npx, you can simply use the following command in your terminal:
npx parcel index.html



When you run the command npx parcel index.html, Parcel automatically analyzes your project starting from the index.html file (or whatever your entry point is) and bundles all the dependencies, including JavaScript, CSS, and other assets. The dist folder is created as Parcel handles the bundling, but it is optimized for **development** speed, not performance.

This will invoke the locally installed Parcel binary (from your node_modules/.bin directory) and start the development server. The index.html file is the entry point, but you can replace it with your desired entry file.

Development (npx parcel index.html): Runs a local server with live-reloading. The dist folder is created but is not optimized (used mainly for development).

Production (npx parcel build index.html): Creates a minified and optimized build in the dist folder, ready for production deployment.

2. Running Parcel with npm Scripts

The other way to run Parcel is by using npm scripts, which is a more common and flexible approach when working with npm.

NPM (Node Package Manager) scripts are a way to automate tasks in your Node.js projects. These scripts are defined in your package.json file under the "scripts" section.

A typical package.json might include something like this:

```
{
  "scripts": {
    "start": "parcel index.html",
    "build": "parcel build index.html"
  }
}
```

```
{} package.json > ...
1   {
2     "name": "starter",
3     "version": "1.0.0",
4     "description": "",
5     "main": "clean.js",
      ▷ Debug
6     "scripts": {
7       "start": "parcel index.html",
8       "build": "parcel build index.html"
9     },
```

We can also use command direct like npm run parcel index.html or npm run parcel build index.html but **defining it in the scripts section** provides better structure, flexibility, and maintainability in the long run. It's especially beneficial for more complex workflows or when working in teams.

- **start**: Runs Parcel in development mode. It starts a local server and watches your files for changes. It's used for development.
- **build**: Bundles the project for production, optimizing the code for performance.

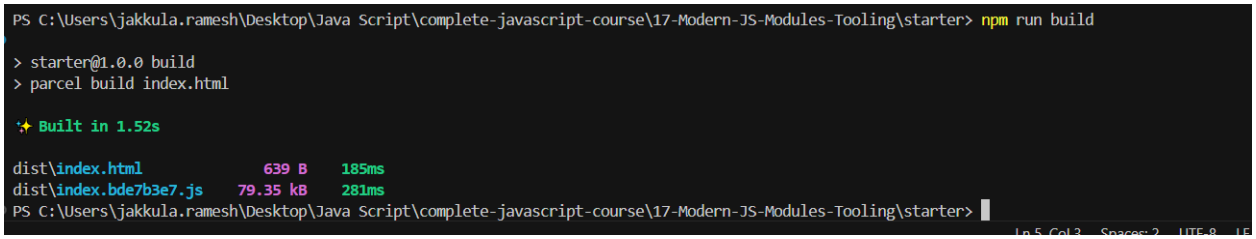
You can execute these scripts using the npm command:

npm run start # Starts Parcel in development mode

```
PS C:\Users\jakkula.ramesh\Desktop\Java Script\complete-javascript-course\17-Modern-JS-Modules-Tooling\starter> npm run start

> starter@1.0.0 start
> parcel index.html

Port "1234" could not be used
Server running at http://localhost:54724
✨ Built in 89ms
```

When you run the command npm run start , Parcel automatically analyzes your project starting from the index.html file (or whatever your entry point is) and bundles all the dependencies, including JavaScript, CSS, and other assets. The dist folder is created as Parcel handles the bundling, but it is optimized for **development** speed, not performance.

npm run build # Builds the project for production



You can customize these scripts to fit your workflow (e.g., adding tasks for linting, testing, or deployment).

Configuring Babel and PolyFilling

Babel ensures modern syntax works in older environments.

Polyfilling ensures missing features are available in environments that don't support them.

Babel :

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.

Or

Babel is a JavaScript compiler that allows you to write modern JavaScript (ES6/ESNext) and then convert it into backward-compatible code that can run on older browsers or environments that don't support the latest JavaScript features. It helps developers use the latest language features without worrying about compatibility.

Babel:

- **Purpose:** Babel is a JavaScript compiler (or transpiler). Its primary job is to convert newer JavaScript syntax (ES6, ESNext) into older versions of JavaScript (like ES5) that can run in older browsers.
- **What it does:**
  - It takes modern JavaScript code (using features like arrow functions, let/const, template literals, async/await, etc.) and converts it into equivalent code that works in environments that don't support those features.
  - Babel can handle syntax transformations but does **not** add missing functionality to the environment.

- **Example:**
    - ES6 code: const add = (a, b) => a + b;

After Babel (converted to ES5):

```
var add = function(a, b) {

  return a + b;

        };
```

PolyFilling: A polyfill is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

- **Purpose:** Polyfilling adds missing functionality for newer JavaScript features that older environments (like old browsers) don't support.
- **What it does:**
    - It includes code libraries (like core-js) that **define new built-in objects or methods** (e.g., Promise, Map, Array.prototype.includes, Object.assign) if they don't exist in the target environment.
    - Polyfills allow your code to use modern JavaScript features even in older environments.

example

Before Polyfill:

In an environment (such as an old browser) that does not support Promise, the code would throw an error because the Promise constructor is undefined.

ES6 Code:

```
const p = new Promise((resolve, reject) => {

  if (/* some condition */) {

    resolve("Success!");

  } else {

    reject("Failure!");

  }

});
```

```
p.then(result => console.log(result)).catch(error => console.log(error));
```

Issue in an old browser:

The Promise constructor is not available, and the browser will throw an error like:
Uncaught ReferenceError: Promise is not defined

After Polyfill:

When a polyfill like core-js is used, it defines the Promise object and its methods, so the code works in the older environment as if it natively supported Promise.

ES6 Code with Polyfill (after adding the polyfill):

```
import 'core-js/stable';  // This imports the polyfill for `Promise` and other features

const p = new Promise((resolve, reject) => {
  if (/* some condition */) {
    resolve("Success!");
  } else {
    reject("Failure!");
  }
});


p.then(result => console.log(result)).catch(error => console.log(error));
```

Result:

- The polyfill ensures that the Promise object and its methods (then, catch, resolve, reject) are available in the older browser, and the code runs successfully.

Other Example: Array.prototype.includes (ES6 feature)

Before Polyfill:

If we use Array.prototype.includes, which is not supported in older browsers, it will result in an error.

ES6 Code:

```
const arr = [1, 2, 3];

console.log(arr.includes(2));  // true

console.log(arr.includes(4));  // false
```

Issue in an old browser:

An old browser will throw an error similar to:
Uncaught TypeError: arr.includes is not a function

After Polyfill:

If you use a polyfill (such as through core-js), the included method becomes available.

ES6 Code with Polyfill:

```
import 'core-js/stable';  // This polyfills includes and other methods


const arr = [1, 2, 3];

console.log(arr.includes(2));  // true

console.log(arr.includes(4));  // false
```

Result:

- The polyfill ensures that Array.prototype.includes works even in older browsers, so the code runs successfully without errors.

REVIEW: MODERN AND CLEAN CODE

READABLE CODE

- Write code so that **others** can understand it
- Write code so that **you** can understand it in 1 year
- Avoid too “clever” and overcomplicated solutions
- Use descriptive variable names: **what they contain**
- Use descriptive function names: **what they do**

GENERAL

- Use DRY principle (refactor your code)
- Don’t pollute global namespace, encapsulate instead
- Don’t use `var`
- Use strong type checks (`===` and `!==`)

FUNCTIONS

- Generally, functions should do **only one thing**
- Don’t use more than 3 function parameters
- Use default parameters whenever possible
- Generally, return same data type as received
- Use arrow functions when they make code more readable

OOP

- Use ES6 classes
- Encapsulate data and **don’t mutate** it from outside the class
- Implement method chaining
- Do **not** use arrow functions as methods (in regular objects)

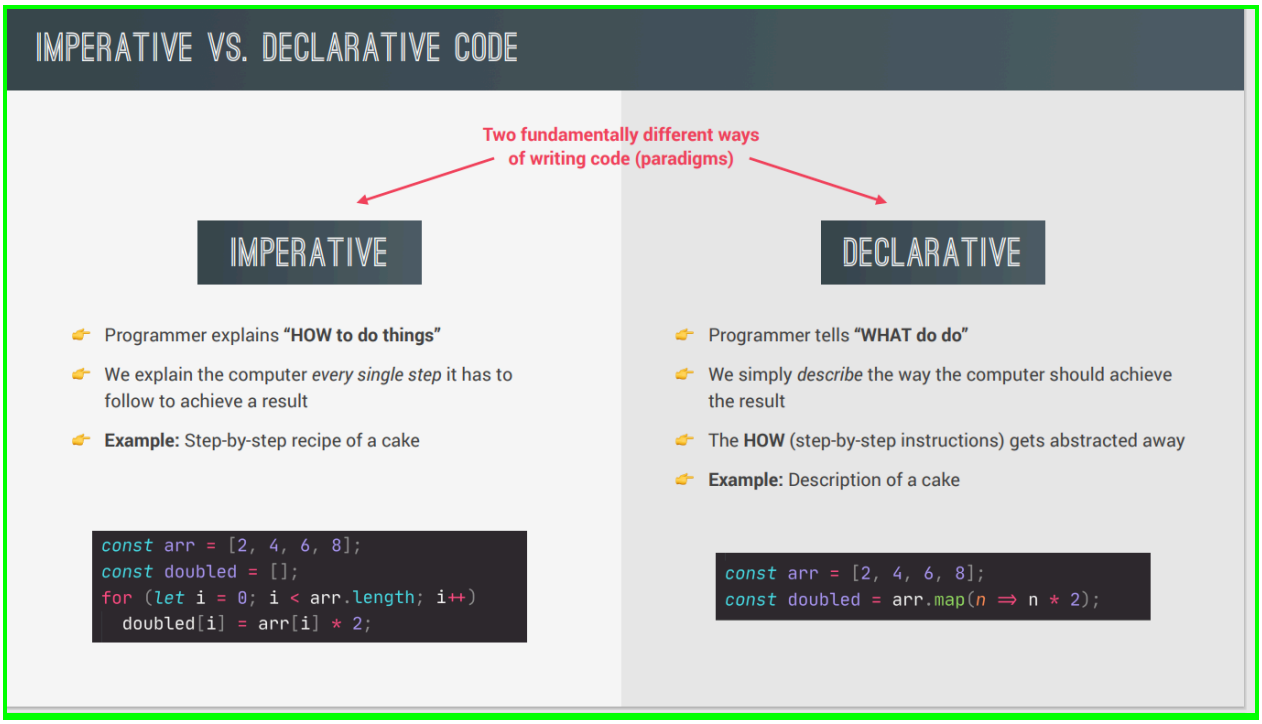
REVIEW: MODERN AND CLEAN CODE

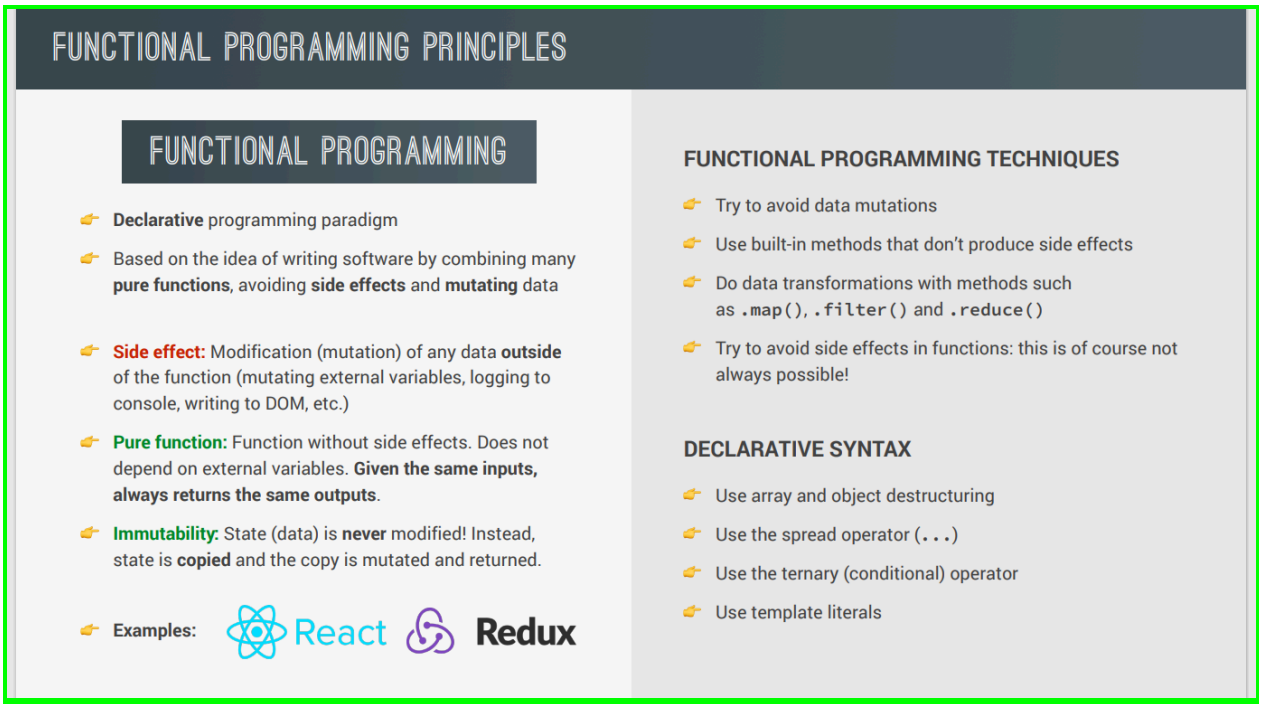
AVOID NESTED CODE

- Use early `return` (guard clauses)
- Use ternary (conditional) or logical operators instead of `if`
- Use multiple `if` instead of `if/else-if`
- Avoid `for` loops, use array methods instead
- Avoid callback-based asynchronous APIs

ASYNCHRONOUS CODE

- Consume promises with async/await for best readability
- Whenever possible, run promises in **parallel** (`Promise.all`)
- Handle errors and promise rejections





Object.freeze()

Object.freeze() is a method in JavaScript that prevents modifications to an object. When you apply Object.freeze() to an object, it:

- Prevents new properties from being added to the object.
- Prevents existing properties from being removed.
- Prevents existing properties from being modified (their values can't be changed).
- Prevents the prototype of the object from being changed.

However, it's important to note that Object.freeze() only applies to the top-level properties of an object. If the object has nested objects, those inner objects will not be frozen unless explicitly frozen using Object.freeze().

Example:

```
const person = {
  name: 'Alice',
  age: 30
};
Object.freeze(person);
person.name = 'Bob';  // This will not change the name property
person.address = '123 Main St';  // This will not add the address property
console.log(person.name);  // Output: Alice
console.log(person.address);  // Output: undefined
```

Modifying Frozen Objects:

- You **cannot modify** the properties of a frozen object (i.e., you can't change their values or add/remove properties).
- If you try to modify a frozen object, the operation will fail silently in non-strict mode or throw an error in strict mode.

However, you **can still modify** the inner properties of an object if they are themselves not frozen.

```
const user = {
  name: 'Alice',
  address: { city: 'Wonderland' }
```

```
};

Object.freeze(user);

user.name = 'Bob';  // This won't work

user.address.city = 'Dreamland';  // This will work, because address is not frozen

console.log(user.address.city);  // Output: Dreamland
```