1. Introduction to Node modules

1.1. Getting started with Node.js (installing and running our script on Node)

- **Understanding modules to master Node**: Modules are the main thing that you need to understand if you want to do anything using Node. To understand what modules are, we need to first understand these:
  - What is meant by a modular structure
  - Why a modular structure is needed
  - How modules are structured
  - How are modules used
  - How are modules created and included
- Let us go through the above concepts one by one to understand modules.

1.2. What is back-end development?

- In this class, we will discuss how to use Node.js as a back-end framework. Let us revise our knowledge of backend development first.
- **Backend development (also known as server-side development)**: It is the development of the logic/code that receives requests from the clients/browsers and contains the logic to get the appropriate data from the database and send the data back to the clients/browsers. The back-end development also includes the database, which includes the logic written to store data into the database for the website.
- **Backend**: It is part of the website that the users do not see, which contains behind-the-scenes activities that occur when performing any action on a website. It focuses primarily on databases, backend logic, APIs, and servers. That is why it is called a backend.
- **Examples of backend processes**: These processes include processing an incoming webpage request and running a server-side script to generate HTML, accessing data (such as an article) from a website's database using SQL queries.

- **Your major responsibilities as a backend developer** are building server-side script/logic for a website, writing queries to store or retrieve data from a database, and organizing and managing communications between browser and server through the TCP/IP network.
- **Node as the backend Framework:** The framework we chose to handle backend development is the Node.js framework.

1.3. Modular structure: avoiding problems of global namespace collision

- **Modular structure**: Modular structure is a general design principle, not special to Node or programming languages.
- **Modular design**: It is a design theory and practice that subdivides a system into smaller parts called modules, which can be independently created, modified, replaced, or exchanged between different systems
- **Advantages of modular design**:
  - Separation of concerns: A modular structure allows different parts of a design to be developed and improved independently of the other parts.
  - Instead of starting from scratch, a complex system can be built by bringing together multiple pieces with specific functionality.

    Example: Car manufacturers that follow the modular design principle can only focus on assembling the pieces together in a way that they think best fits the customer's demand.
  - Reusing independently created parts/designs for other projects is possible.
  - The pieces/modules are developed independently.

    Example: A company producing the engine of a car is only focused in producing the best engine only.

- **Why is a modular structure needed in Node.js?** To solve the problem of global name-space collisions
  - **Problem of global name-space collisions**
    - **Namespace**: Namespace is basically the place/space where the name of a variable or function is accessible.
    - **Name collision**: Name collision is the accidental overriding of the value of a variable/function because the same variable/function name is used in different places.
  - **Name collision in JavaScript**: In JavaScript, using the same variable name or function name in a space will override the first/earlier function/variable and return the value of the latest if we use the same variable/function name in our JavaScript file. However, we can use different name-spacing ways in JavaScript to escape the collision while using the same variable or function name.

```
Example 1:
 var eat = () => {
  console.log("Kebede ate");
};

Example 2:
var eat = () => {
  console.log("Hana ate");
};
eat(); // This will print "Hana ate" by overriding the function
   under Example 1
```

    - Whatever we write in JavaScript becomes the properties of the global window object. This window object is our global namespace.

```
Example: var a = 3;
console.log(window);
// You will see our variable "a" and its assigned value under the window object.
console.log(window.a);
// will print 3. This means our var "a" has now become one of the properties of the window object.
```

- If we want to use the same variable or function name more than once in the window object space, there will be a naming collision, which will result in overriding the earlier function/variable. We have learned under phase 2 that there is a variable scoping method in JavaScript whereby the rule is that variables declared inside of a function are only available inside of that function. Other than this very limited measure JavaScript takes to avoid collision, it does not have a default mechanism to avoid name collisions fully. However, we can create measures to avoid name collisions in JavaScript. Let us look at the measures below

- **Ways to avoid name collisions in JavaScript**:
  - **Creating a Namespace using a global object**: Do you remember the eat() functions we used above? Now, let us create a global object containing the functions as properties of this object. We use this object to restrict the functions from polluting the global window environment. Let us create a global object for the eat() functions we used above as an example:

```
Example 1:
var Kebede = {
   gender: "male",
```

```
  eat: function () {
  console.log("Kebede ate");
  },
};
Example 2:
var Hana = {
  gender: "female",
  eat: function () {
  console.log("Hana ate");
  },
};
Hana.eat(); // prints "Hana ate"
Kebede.eat(); // prints "Kebede ate"
```

- The above 2 functions are not accessible outside of their respective objects. That is why we can use this name spacing method to hide our function/code

- **Wrapping your code inside braces**:

```
Example:
(eat = () => {
console.log("Kebede ate");
})
Example:
(eat = () => {
console.log("Hana ate");
});
eat(); // This will print both "Kebede ate" and "Hana ate"
```

- **Using the IIFE (Immediately Invoked Function Expression) way**: This is wrapping variables/functions inside of another function to limit the scope of these variables and functions to a local scope.
- **Modularity**: Here, you will set up local variables/functions using modules where these variables/functions are treated local to the modules.
- Let us discuss IIFE and modularity in the next section.

1.4. Approaches to avoid namespace collision in Node: the IEFE Way

- **The IIFE to solve global name-space collisions**: IIFE a function defined as an expression and executed immediately after creation. We said earlier that when we declare a function or a variable (outside of a function), the JavaScript engine adds the variable or function to the global window object and makes them properties of this super object. If you have many global variables and functions with the same name, this causes name collisions. One way to prevent the functions and variables from polluting the global object is to use immediately invoked function expressions (IIFE).

```
Syntax of defining an IIFE:
(function(){
  //...
})();
```

Note: You can use an arrow function to define an IIFE

- **How IIFE way solves global name-space collisions**: The IIFE way approach avoids name-space collision by wrapping variables/functions inside of another function that is executed on load. Because variables/functions will be wrapped in a function, they

will have a local scope, meaning they won't be globally available. Let us look at the following examples:

Assume this is your HTML:

```
<body>
    <script src="first.js"></script>
    <script src="second.js"></script>
    <script src="third.js"></script>
</body>
```

Assume this is the script in your first.js:

```
(function () {
function eat() {
console.log("Hana ate");
}
    return eat();
})();
```

Assume this is the script in your second.js (IIFE):

```

function eat() {
console.log("Kebede ate");
}

```

Assume this is the script in your third.js:

```
eat ();
```

- **Now run your HTML in the browser**: We will see “Hana ate” printed first as this function gets immediately executed because it is IIFE. “Kebede ate” is also printed after that. If we did not use a function to wrap one of the eat() functions, one of the functions would have been overridden by the last one as they have the same function name. That is how IIFE avoids name collision.

- **Drawbacks of the IIFE method**:
  - We need to wrap everything inside an IIFE.
  - We must pay attention to the order of the script tags as they could be dependent on the output of the other. Meaning, if you take the above example script files, the IIFE function, first.js, got executed and “Hana ate” was printed before our second.js, even if our second.js by default should have priority as it is linked in our HTML file after our first.js. If one script is dependent on the other, we need to make sure we arrange our script files in our HTML file accordingly before applying the IIFE method.

1.5. Basic concept of Node module

- **Evolution of modular development to solve global name-space collisions**:
  - Modules are the fundamental building blocks of the code structure. The module system allows you to organize your code, hide information, and only expose the public aspect of a module by exporting.
  - Without module systems like CommonJS, if we have JavaScript files that depend on one another, they will have to be loaded in our HTML with <script> tags.

- We have seen earlier that there were various ways that were used to avoid name collision. Before modularity was conceived, JavaScript implemented a function block (limiting the scope of variables by declaring them in a function), encapsulation of variables/functions in an object, and encapsulation of variables/functions in an IIFE function to prevent global variables/functions from colliding. Then came modularity. We will discuss modularity in detail, especially how Node.js implements modularization.

- **Node module**: It is a single or multiple JavaScript files with specific functionality organized in a way that can be reused throughout the Node.js application.
- **How do we access JS objects in Node?** Node.js does not have the document, window, and all the other objects that are provided by the browser. There is no browser, window, or URL location in NodeJS. Instead, Node has a variable called global.
  - In Node, we do not have the object called window as we do in a browser. Instead, we have another object called global.
  - We can access the standard JavaScript objects that are already defined in JavaScript through the global object. To see what the global object in Node does, go to one of your Node modules and write this code:

```
console.log(global); // will print all the available JS objects
console.log(global.console); // will print the console object
```

  - But the global variables we declare in our code, say for example var test = "abebe", are not added to the global object as each module is wrapped by a function in Node. The ones we define in a file are only scoped in our file. They are not available outside of the file. This is because of the node Modular system.

- Every file inside of node is considered a module. The functions and variables defined in that file are scoped only to that file (these functions are private).
- In order to be able to use the variable and function defined in a module outside of the module, we need to export it explicitly (make it public).
- For the modularity to work properly, every Node application needs to have at least one module called main file. This is like the starting point of the application.

1.6. Standards used to write Node modules: commonJS and ES6

- **CommonJS standard to handle Node modules**: Around the time Node was initially launched, in 2009, JavaScript did not have a standard way of handling modularity, and Node.js filled this gap with the CommonJS module format. CommonJS is an agreed-upon standard for how core modules should be structured. It was the standard for Node.js modules before the ES standard. Due to that, there are many Node.js libraries/ modules written with CommonJS. This means that if we want to write a single piece of code that is going to be plugged into Node, we will have to follow the modular specification set by the CommonJS group. It is this standardization that led to the existence of the amazing NPM ecosystem. Every module on NPM follows this standard:
  - **Creating a Node module using CommonJS**: Steps to creating a module:
    - Create a js file. That's it, you just created a module. CommonJS follows a file-based system: One file is one module. One package includes at least one file.

      Example: abebe.js.
      - To help us manage the modules we write, there is this object called module. Console this object (module) and

see what it includes:

console.log(module). It has:

- id (Every module has a unique id)
- exports (What is made available public)
- filename

- Write your code
- Declare your variables and functions
- Export your module (only the things you want to be public). This is just adding it to the export object inside the module object.

- **Node modules use wrapper functions to avoid name collision**: Variables/functions in any Node module will remain private because each module is wrapped in a function (IFEE function) by Node.js before executing the code. **syntax of this function wrapper function**:

```
(function(exports, require, module, __filename, __dirname) {
  // entire module code here
});
```

- The entire code written inside a Node module is private to the module unless explicitly stated (exported) otherwise. Even if you define a global variable in a module using var, let, or const keywords, the variables are scoped locally to the module rather than being scoped globally

- **The five Node module parameters (exports, require, module, __filename, __dirname)**: These parameters are available inside each module in Node and are also local to the module as they are wrapped by a function. The parameters provide valuable information related to a module.

- **__filename**: console.log(__filename) returns the name of your module
- **The "module" object in Node:** It refers to the object that represents your current module/JavaScript file. To check that, you can console module.exports and see your module printed in an object form.
    - **Example**: **// index.js**

```
function eat() {
 console.log("Hana ate");
  }
module.exports.e = eat; //  exporting eat() function
```

    - If you console.log(module.exports), it will return your index.js module in an object form like this:

```
console.log(module.exports); //{ e: [Function: eat] }
```

- **"exports" parameter in Node/CommonJS:** It is a key value of the "module" object. If you console module.exports, you will see an empty object printed. The "module.exports" makes our module available for other modules to import it by allowing these two things:
    - **Explicit exports**: Expose the variables and functions that you want other modules to use. This is just adding what you want to expose to the export object or module.export
        - **Format**: export.varName
        - **Example**: export.almaz

- **Explicit imports**: Only include the things you want to use to build your module. Here, you will use require ():

```
Format:
var VariableName = require(pathToModule);
Example:
const abebe = require('./abebe.js').
Example of importing and exporting modules:
// kebede.js
function someFunction() {
console.log(AAvar3);
}
  module.exports.fromKebe = someFunction; // exporting someFunction from Kebede.js
// app.js
 const kebe = require("./kebede.js"); // importing someFunction from kebede.js
```

- **The "require() function" to load a Node module**: The require() function is a built-in function in Node built to include modules that exist in different files. We need the require function to load a module. The way the require() function works is that it first reads the module/JS file, executes the file, and then proceeds to return the exports object. This function takes one argument, the path of the module you want to load. **Note:** The relative path principle applies here. **Example**: var something = require("./some.js").

- **Look at the following example**:

```
// index.js
function eat() {
console.log("Hana ate");
}
module.exports.e = eat; //  exporting eat() function
 console.log(module.exports);  // returns  {  e:  [Function:
eat] }
// main.js
const eatFunction=require("./index.js"); //importing
index.js
eatFunction.e(); // prints “Hana ate”
eatFunction.e(); // prints “Hana ate”
```

  - **Drawbacks of the CommonJS standard**: Because CommonJS is not adopted by the JavaScript standard (ECMAScript), browsers do not understand it. That means, if you write a code following the CommonJS standard, and you want it to be executed on the browser, you need something to translate your code to the standard JS format. These helpers are called Module Bundlers.
    - **Module Bundlers**: They take all the JavaScript files written following the commonJS pattern and intelligently convert them into a single JavaScript file. This will avoid a naming collision. One of the most used module bundlers is called Webpack
- **ES6 standard to handle Node modules**: Later in 2015, JavaScript recognized the need to support a standard way of building modular systems and produced its module standard called "ES6 Module Standard".

Now, the ES module format is the official standard format to package JavaScript code for reuse, and most modern web browsers natively support the modules.

- Just like CommonJS, ES6 Module is also a file-based standard. Meaning that one file is for one module.
- With ES6, objects, functions, classes, or variables are made available to the outside world with the use of the import and export keywords instead of the require() function in CommonJS.
- **Note**: By default, any Node.js module is treated as a CommonJS module. If we want our module to be treated as ES6 module, we have to make sure to set the file type as **"type": "module**" in the package.json or use the **.mjs extension** for our file. Otherwise, our module will be treated as a CommonJS module, and anything we write following the ES6 standard will not be understood.
- **Two ways to export/import in ES6**: In both cases, please know that you should use the keyword "export" when exporting and "import" when importing.
    - **Regular export**: You can export members one by one. What's not exported will not be available directly outside the module.
        - **Example**: **// myPractice.js**

```
function eat() {
  console.log("Hana ate");
}
function dance() {
  console.log("kebede danced");
}
export { dance, eat }; // exporting both functions
```

- **Regular import**: **Format**: import {varName} from 'pathToModule'.
    - **Example**: // **app.js**

```
// importing dance and eat functions from myPractice.js
import { eat, dance } from "./myPractice.js";
eat(); // returns “Hana ate”
dance(); // returns “kebede danced”
```

- **The default export/ import**: You just add the keyword default when you want to make that export/import the default.

  Note: Default export lets you import a module by giving it a name of your choice.

    - **Default export format**: export default varToExport/function name.

      Example: // **myPractice.js**

```
function eat() {
console.log("Hana ate");
}
function dance() {
console.log("kebede danced");
}
```

    - **Default import**: You can import the default export by giving it a name of your choice. **Format**: export varToExport/function name
        - **Example**: // **myPractice.js**

```
// see  default export being imported under a new name
```

```
import newName, { eat, dance } from "./myPractice.js";
eat();
dance();
newName (); // using the import name to call speak()
```

- **How to convert ES6 module formats to CommonJS**: You might sometimes come across scenarios where you must convert a module that is written following the ES6 module to work in a system that follows the CommonJS standard. **Example**: React
  - **Babel**: Babel is one of the most used JavaScript plugins that convert ES6 to CommonJS. The main purpose of Bable is to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript
  - Refer to these websites if you want to know more about Babel
    - https://www.npmjs.com/package/@babel/plugin-transform-modules-commonjs
    - https://babeljs.io/docs/en/babel-plugin-transform-modules-commonjs

1.7. Managing Node modules using NPM: core and contributed modules

- **The three types of Node modules**: Core, local, and third-party/contributed modules
  - **Core modules**: These are built-in modules and come automatically when we install Node.js. Some of the popular third-party modules are http, fs, path, and os
    - These modules can be loaded into the program by using the require function. **Syntax**: var module = require('module_name');

        - **For more core Node modules, please visit this website:**
            - https://www.w3schools.com/nodejs/ref_modules.asp
    - **Third-party/contributed modules**: These are modules that are available online using the Node Package Manager (NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are mongoose, express, angular, and react
    - **Local Modules**: Unlike core/built-in and external modules, local modules are the modules you create locally in your Node.js application
- **The most used core Node modules**: Before we list out the most used core modules, let us try to guess which ones are going to be the most used ones. We can do so by listing out the common things a server-side programming language does in addition to forming the logic behind your application. This means that we find out the common interactions you want to make with the server computer
    - **fs**: It is used to handle file and directory/folder system. Examples:
        - Creating a folder
        - Accessing and opening a file
        - Editing a file
        - Copying a file
        - Removing a file or a directory
    - **os**: It provides information about your computer's operating system. Examples:
        - Getting the name of the host computer
        - Getting the right network information of the computer
    - **path:** It includes methods to deal with file paths. Examples:
        - Identify the path of a specific file or folder
        - Identify the path to the root directory
        - Identify the extension of a file

- **http**: It creates an HTTP server in Node.js. Examples:
    - Ability to receive and handle HTTP requests, which involves understanding the request methods and the status code
    - Managing connections
- **Events:** It is used to own and trigger events
    - Creating, firing, and listening for your own events
    - Attach and detach one or more event listeners to the named event
    - Managing asynchronous requests using event loop

- **The most useful contributed Node modules/packages**:
    - **Package**: It is just a collection of modules that are working to achieve a specific goal
    - **Examples of most common contributed Node modules**: React, Express, JSHint, Angular
        - We will discuss React and Express in detail later
    - **For a list of useful contributed modules, here is a link**: https://github.com/aravindnc/A-to-Z-List-of-Useful-Node.js-Modules