Installation of NodeJS

What is Nodejs?

- **Node.js®** is a **JavaScript runtime** built on **Chrome's V8 JavaScript engine**.
- It is created by **Ryan Dahl**
- It is maintained by **Joyent**.
- Current version is 14.13.0 version.

Chrome's V8 JavaScript Engine:

- V8 is the name of the JavaScript engine that powers Google Chrome.

Browser name	Java Script Engine name
Firefox	SpiderMonkey
Safari	JavaScriptCore (also called Nitro)
Edge	Chakra

npm:

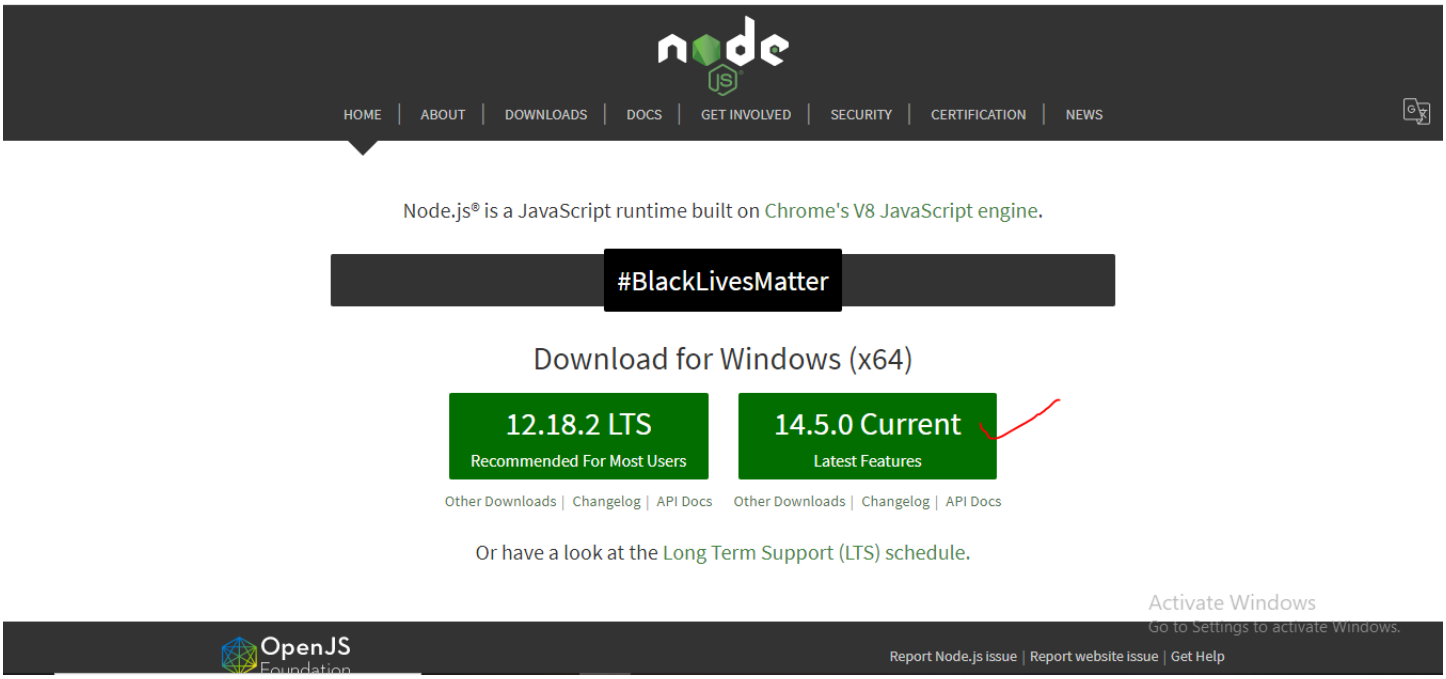
- npm is the world's largest Software **Registry**.
- This registry contains over 800,000 code packages.
- Open-source developers use npm to share software.
- Many organizations also use npm to manage private development.
- It is installed automatically with nodejs.

Syntax:

npm install <package> options

Steps to Download and Installation of NodeJS:

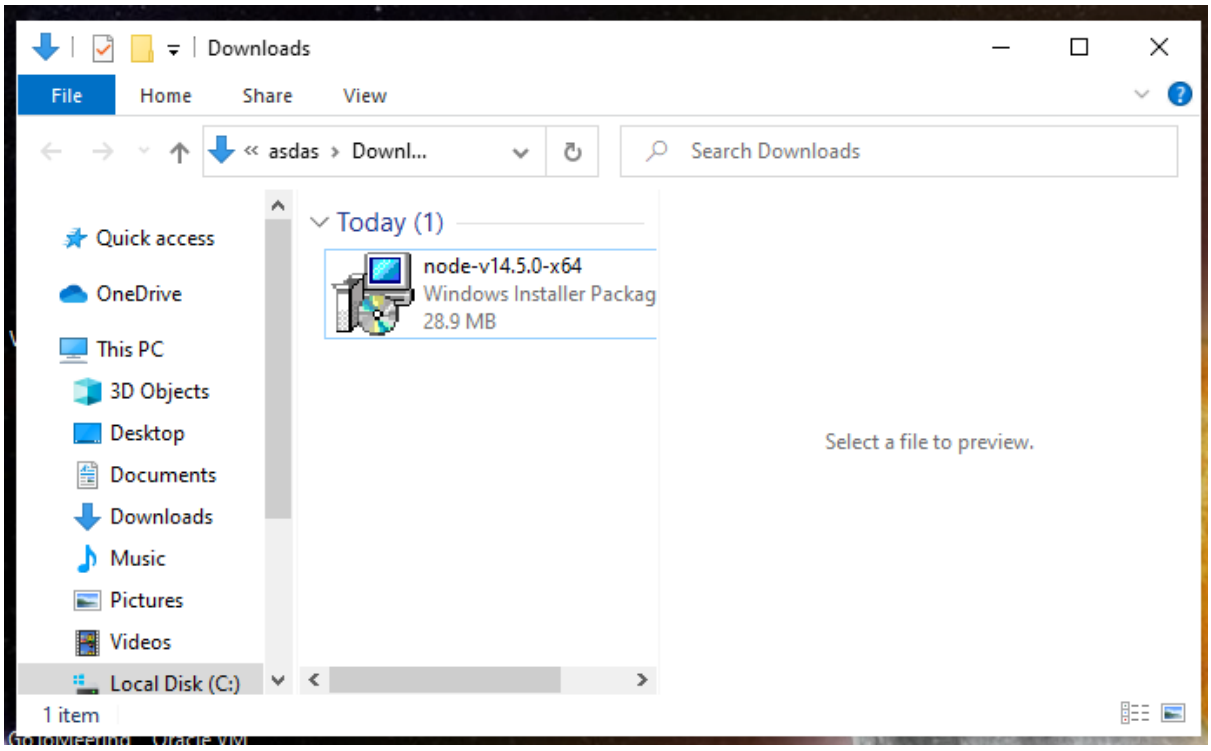
Go to https://nodejs.org



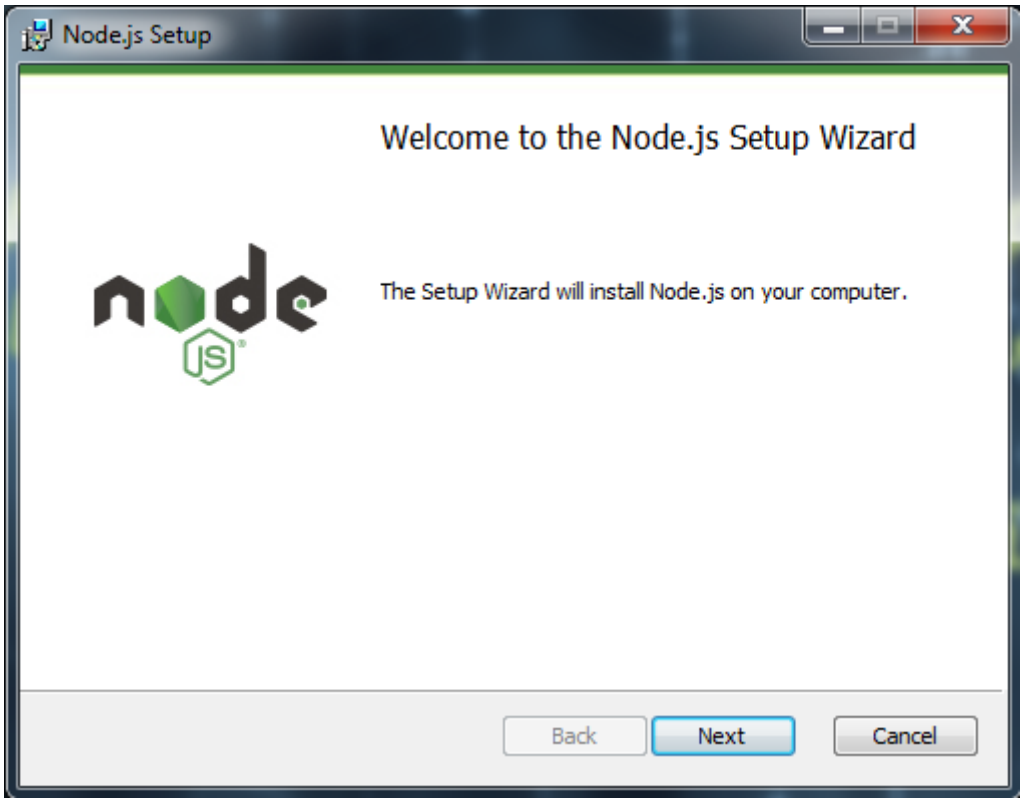
Click on 14.5.0 version and download it.

Note: The version number may be varied. Choose the latest version.

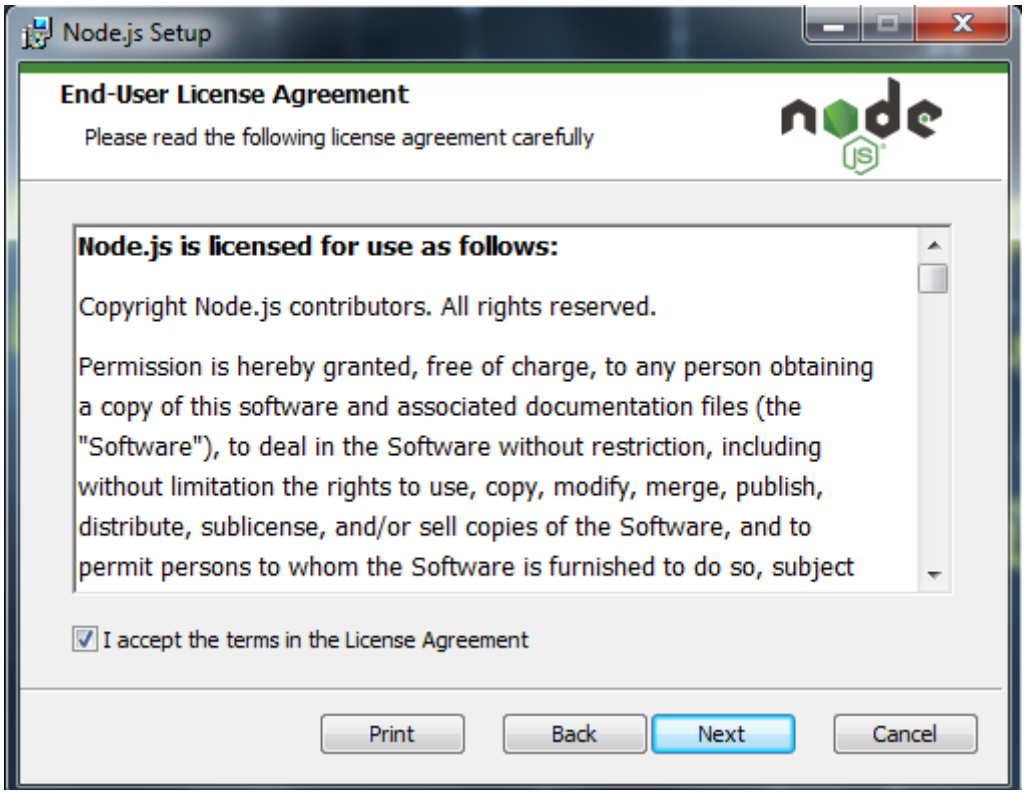
Go to **Downloads** folder and double click on "**node-v14.5.0-x64.msi".**



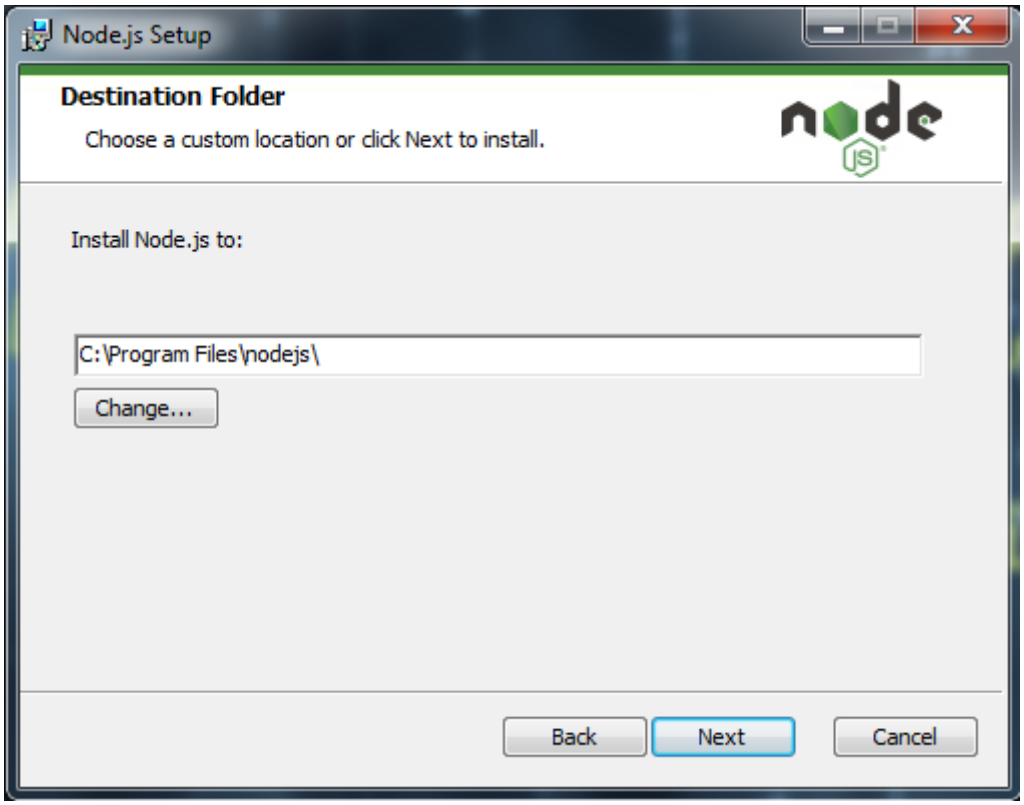
Click Next



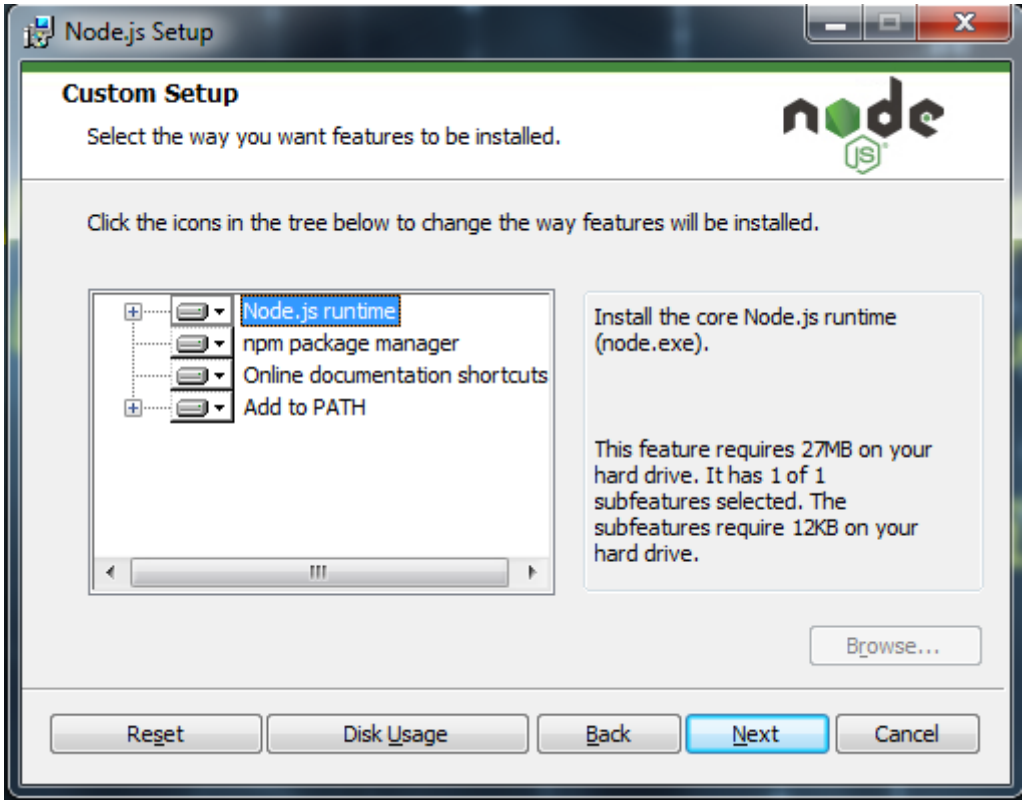
Accept license and Click on Next.



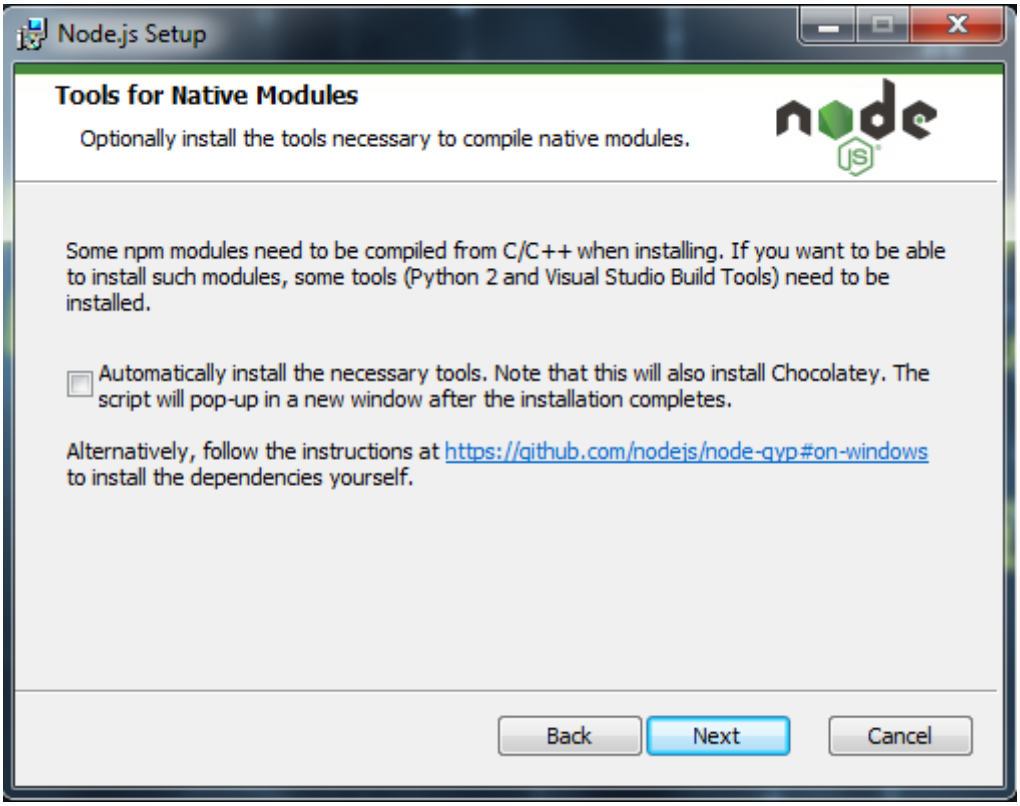
Check the checkbox “I accept the terms in the License Agreement” and click on “Next”.



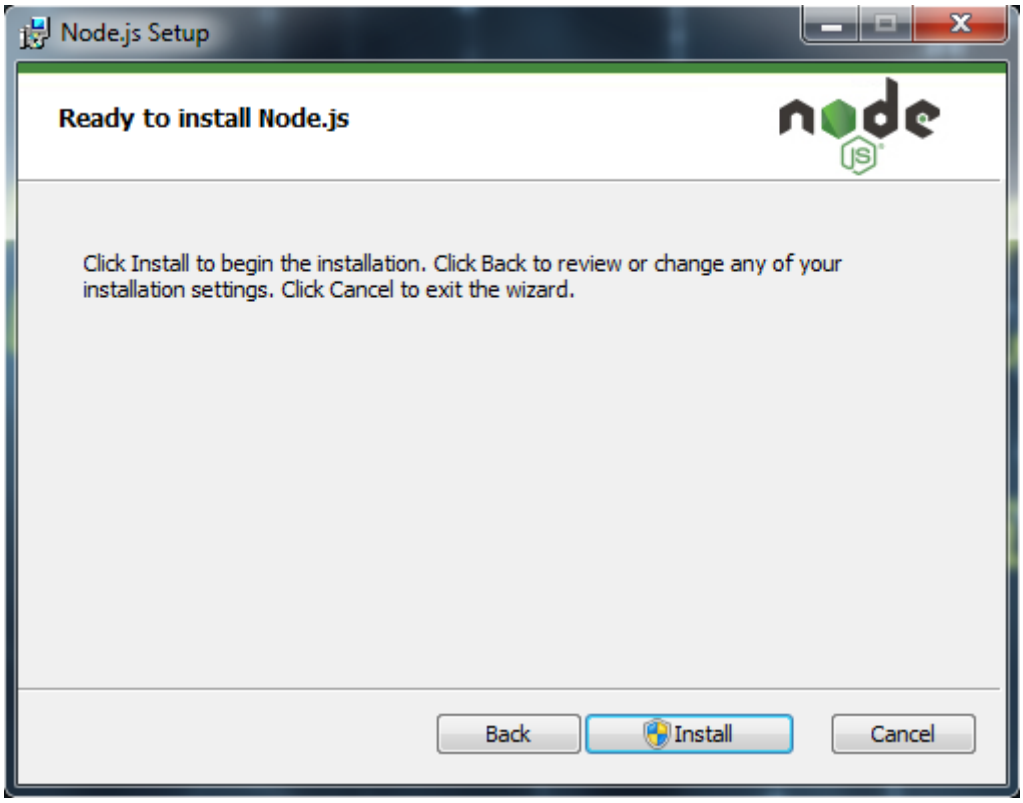
Click on “Next”



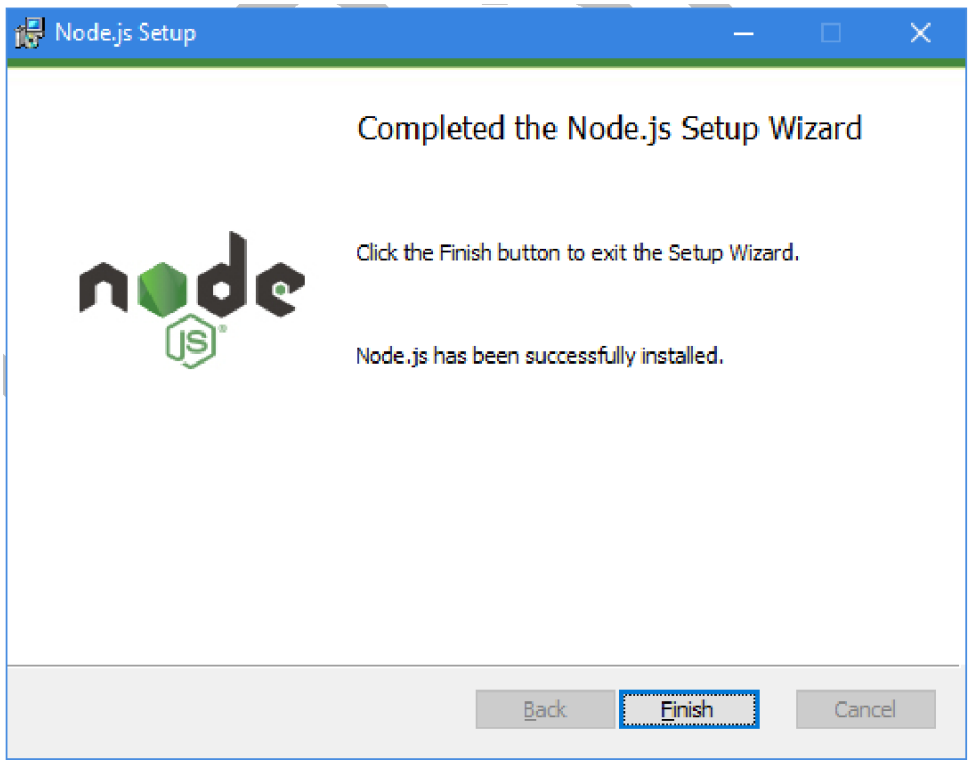
Click on “Next”



Click on Install.


Node.js Setup
Ready to install Node.js
Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.
Back
Install
Cancel

After installation is completed, click on "**Finish**".



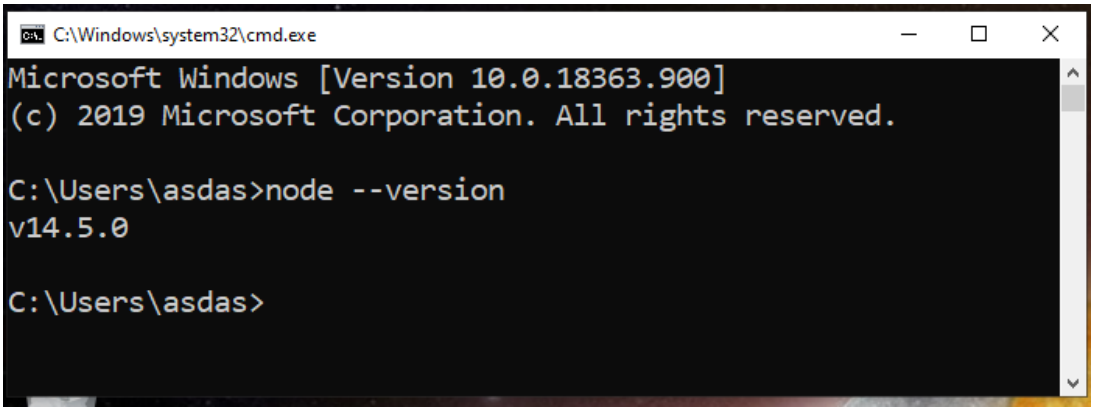
Check the Node version.

Open "**Command Prompt**" and type follow command.

>node –v

or

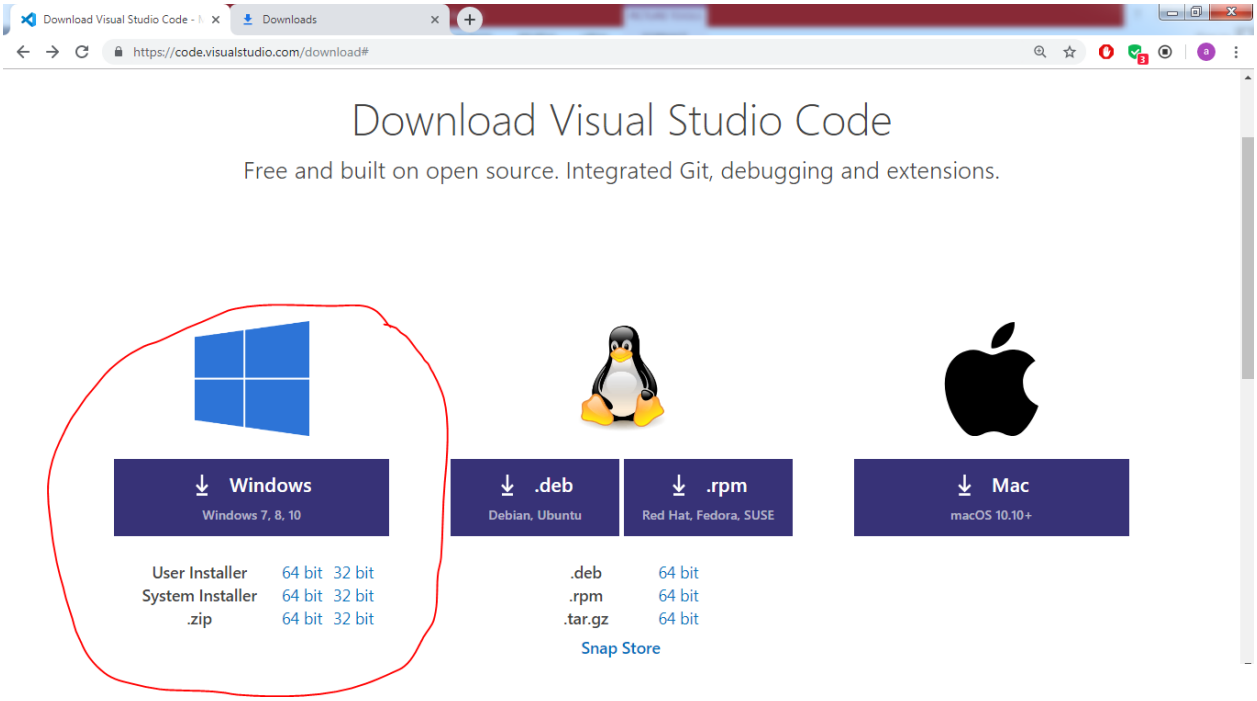
>node --version



Installation Visual Studio Code

- "Visual Studio Code" is the recommended editor for working with UI technologies like Angular, React, Vue, etc.

Go to https://code.visualstudio.com

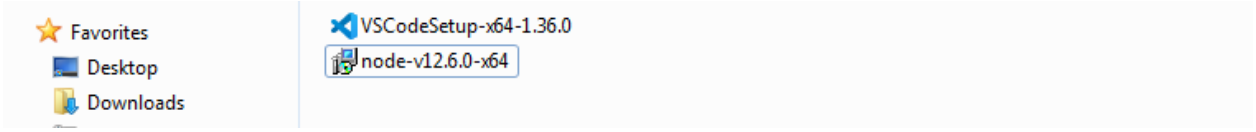


Click on "Download for Windows".

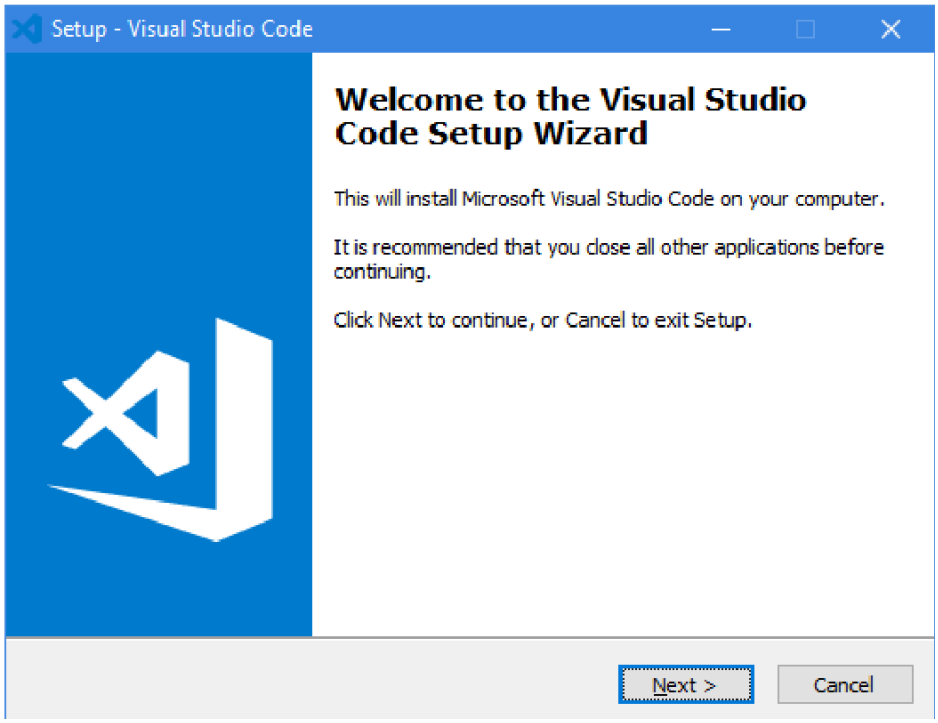
Note: The version number may be different at your practice time.

Go to "Downloads" folder; you can find "**VSCodeSetup-x64-1.36.0**" file.

Double click on "**VSCodeSetup-x64-1.36.0.exe**" file.

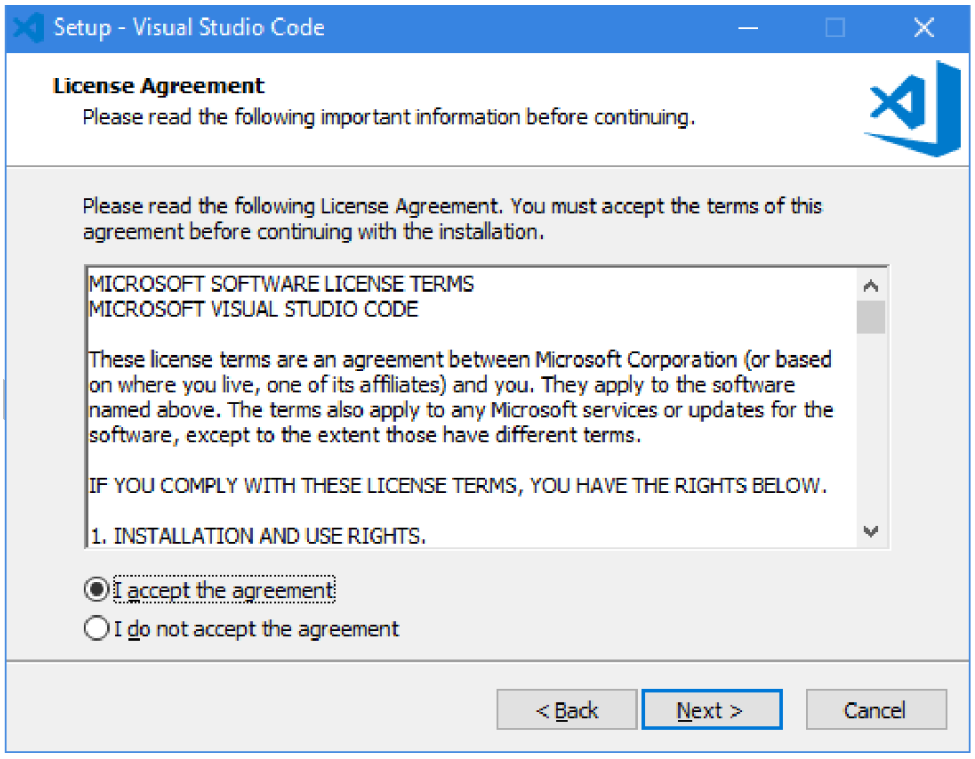


Click on “Next”.

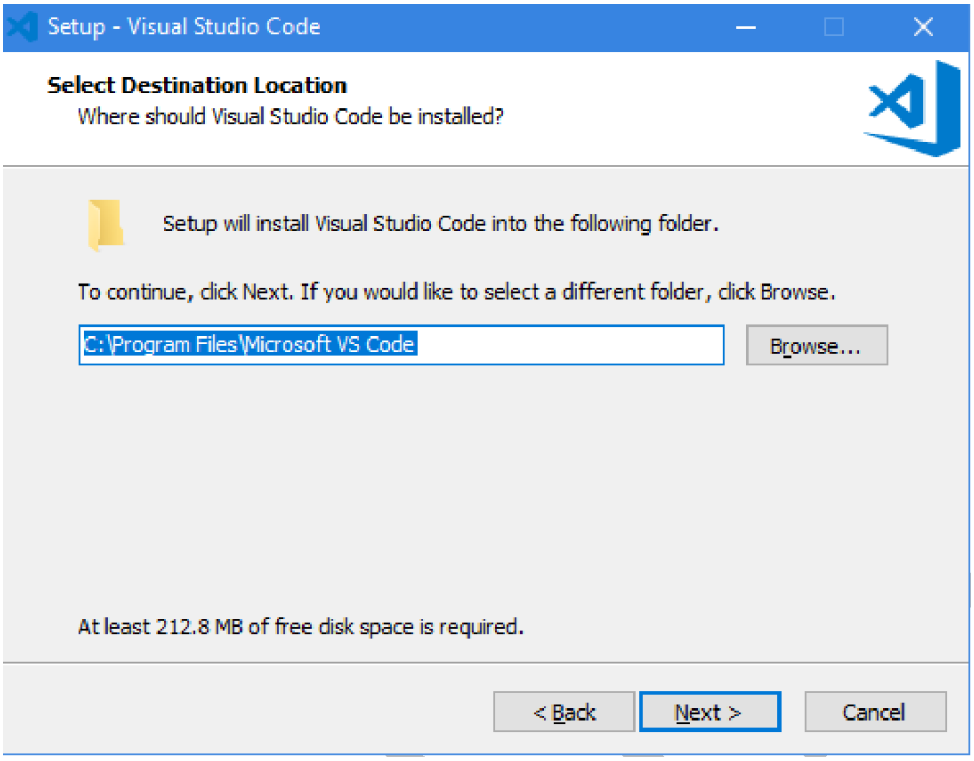


Click on “I accept the agreement”.

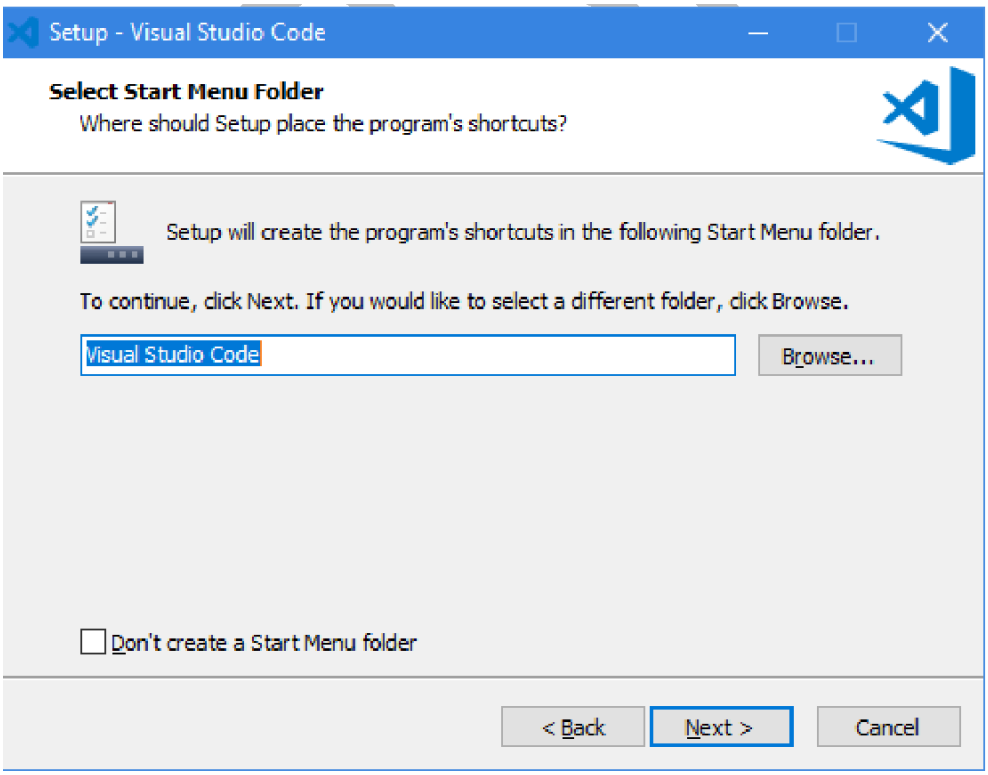
Click on “Next”.



Click on “Next”.

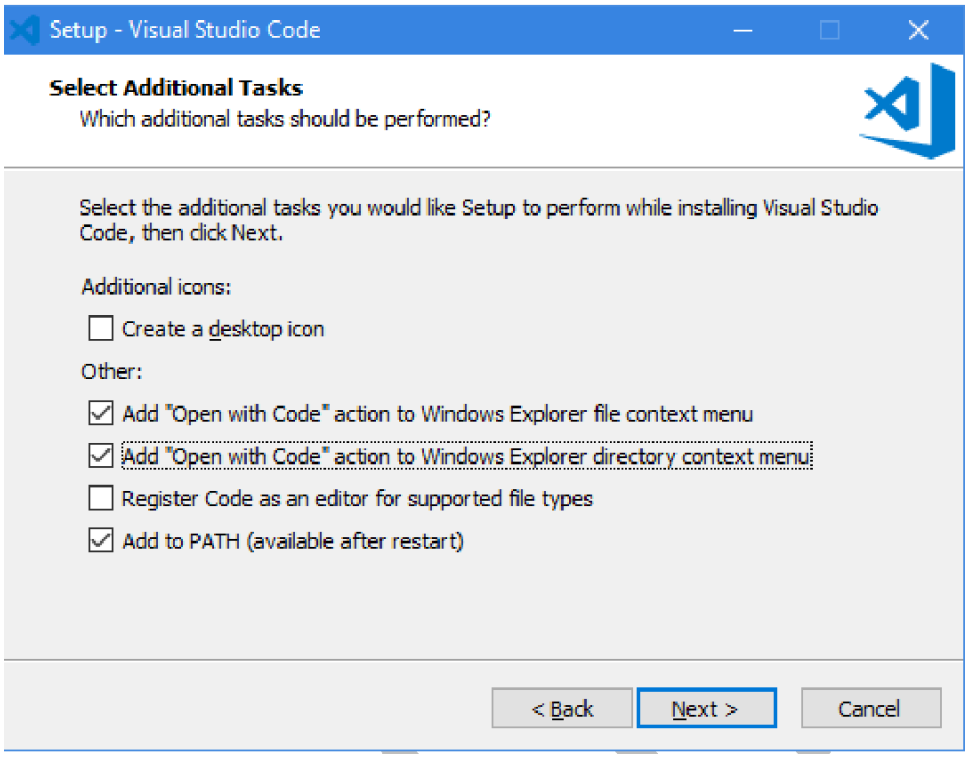


Click on “Next”.

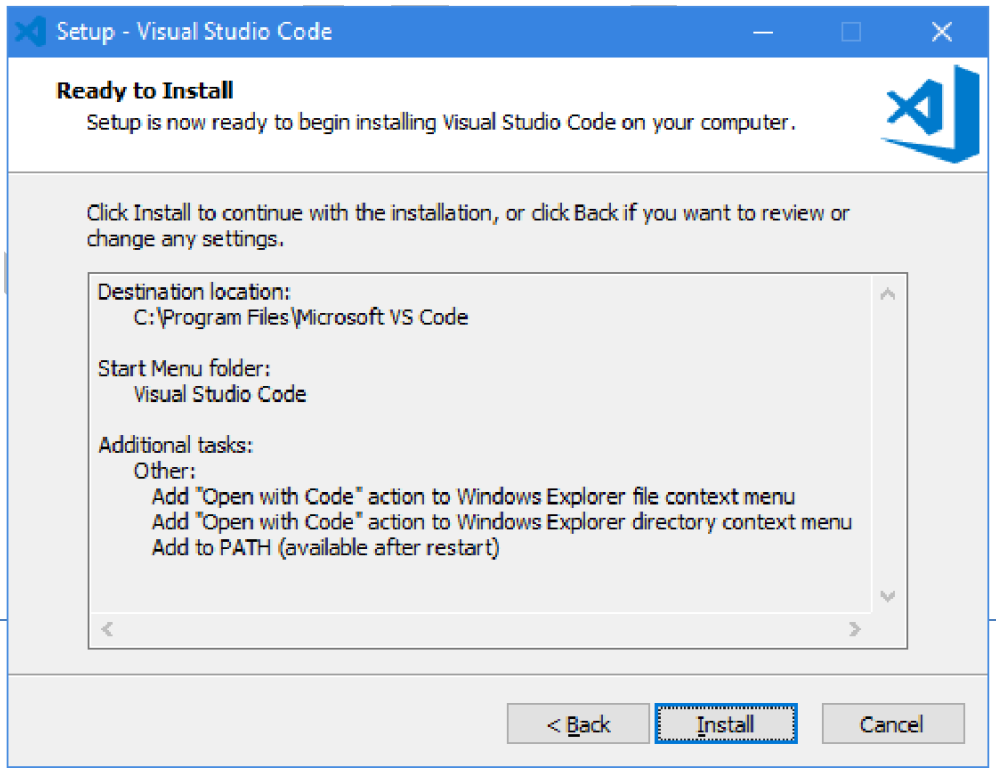


Check the checkbox "**Add Open with Code action to Windows Explorer file context menu**".
Check the checkbox "**Add Open with Code action to Windows Explorer directory context menu**".
Check the checkbox "**Add to PATH (available after restart)**

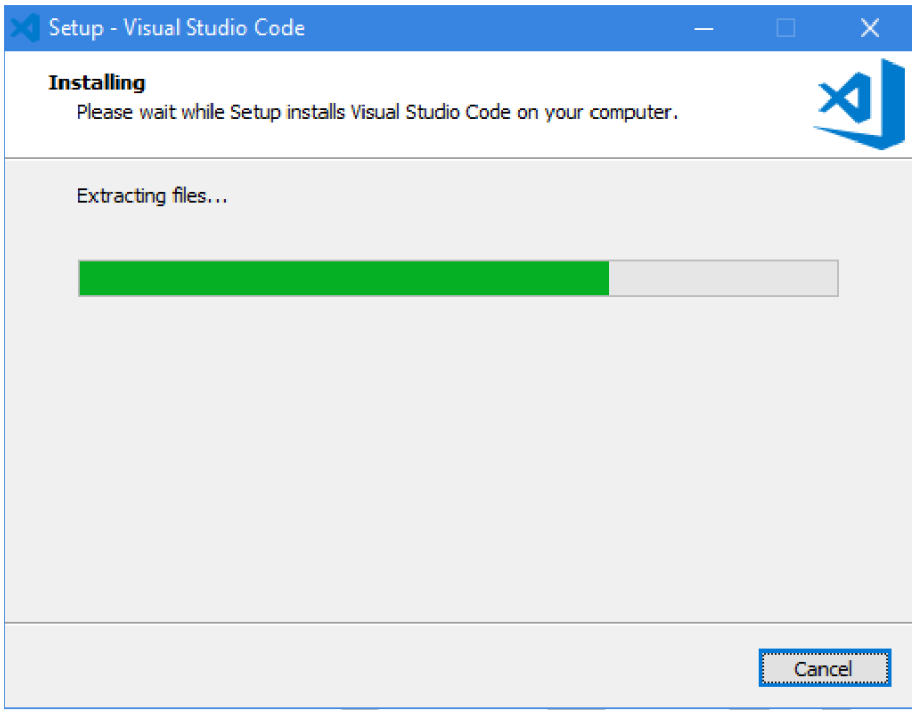
Click on "Next".



Click on "Install".



Installation is going on….



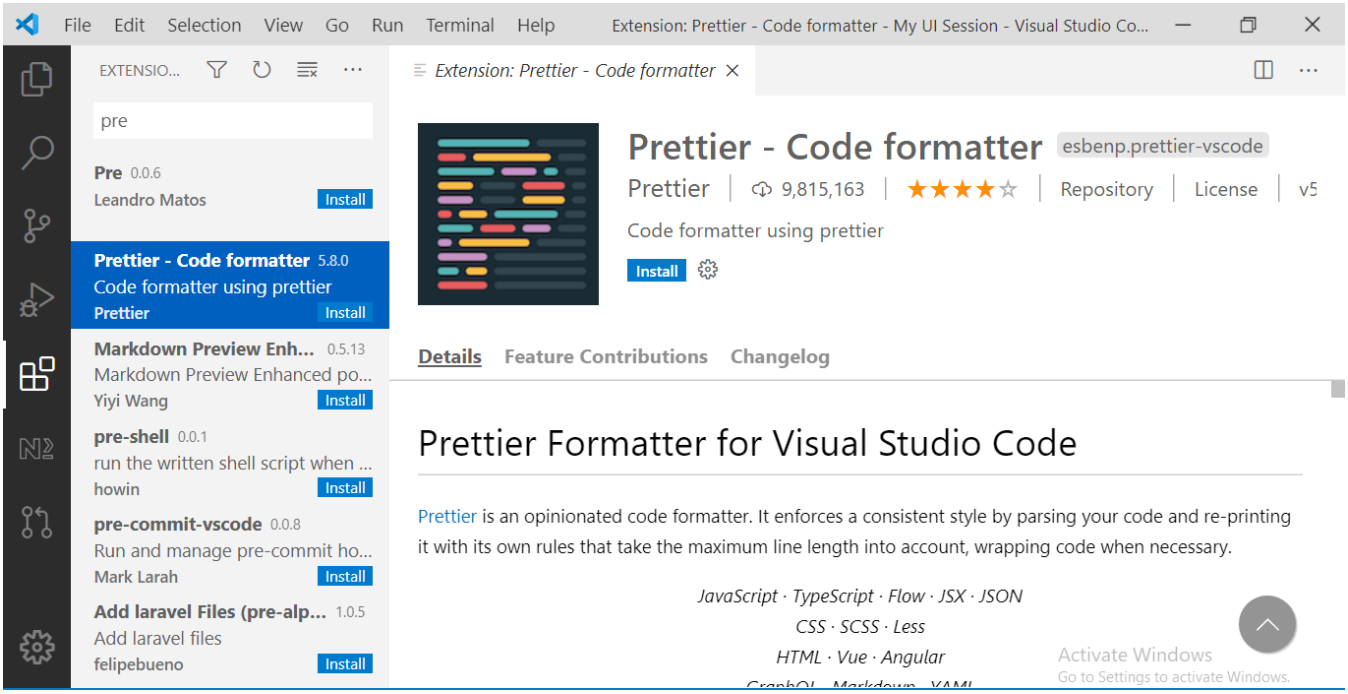
Once installation completed.

Click on "**Finish**".

Open the VS Code:

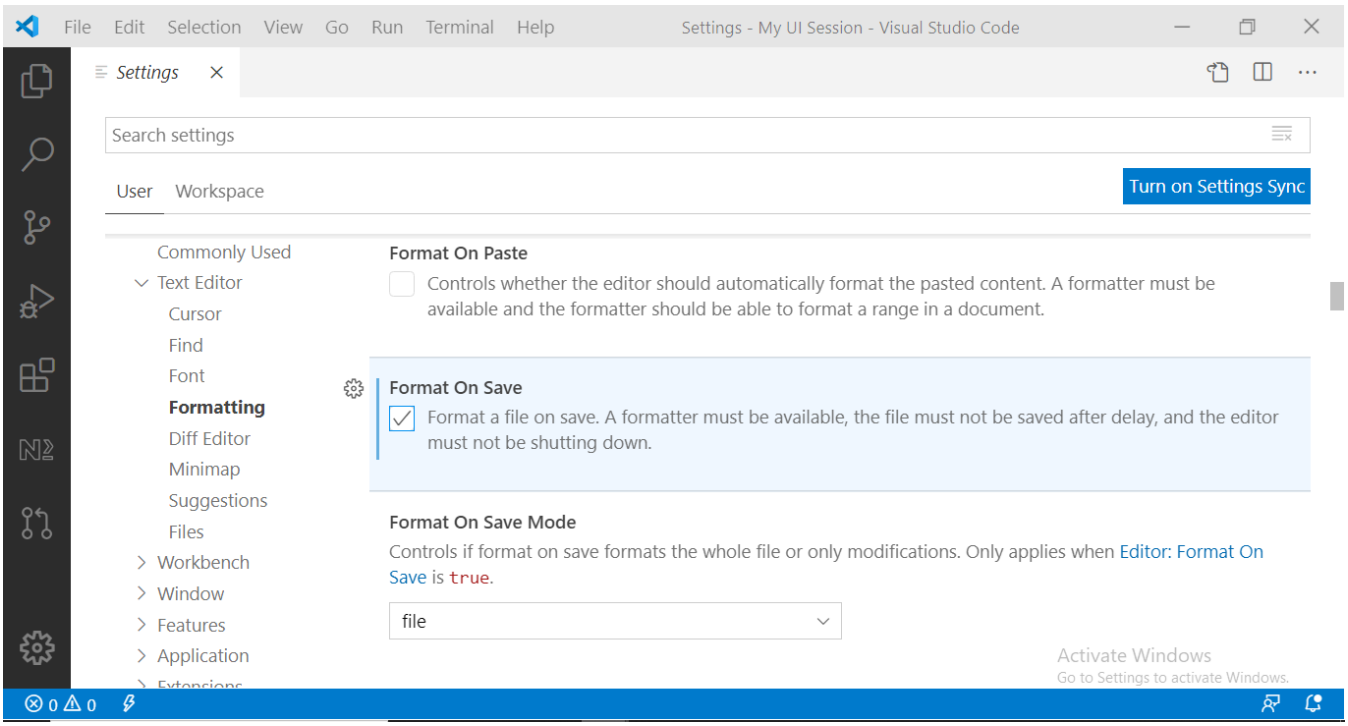
Start => select VS Code.

Install **Prettier-Code formatter** in VS Code and reload VS Code again to get effected.



Enable Format on Save.

File -> Preferences -> Settings ->



Note: If Prettier is not formatting

View -> Command Palette…

Search for **Format Document With…**

Select **Configure Default Formatter and**

Select **Prettier**

Java Script Brush Up

Working with Variables

- Variable is a named memory location in RAM, to store a value at run time.

Syntax:

let variableName = value;

Example:

let a = 100;

Data Types:

- "Data type" specifies what type of value that can be stored in a variable.
- Java Script supports dynamic typing.

Static Typing:	Dynamic Typing:
➢ The data type of the variable is fixed at the time of declaration. The type of a variable can't be changed throughout the program, then it is said to be "**Static Typing**". Eg: C, C++, Java, and C#.NET	➢ The data type of the variable is not fixed while declaration, and the data type will be taken by the runtime engine automatically at the time of program execution, and possible to assign any type of value to the variable, then it is said to be "**Dynamic typing**". Eg: JavaScript, Python, etc.

Eg:
test.js

```
let x = 10;
console.log(x);

x = "Jones";
console.log(x);

x = 5500.00;
console.log(x);

x = true;
console.log(x);
```

```
let employeeId = 10;
let employeeName = "Sai";
let married = true;
let employeeAddress = "Hyd, Tel, India";
let employeeSal = 5500.00

console.log(employeeId);
console.log(employeeName);
console.log(married);
console.log(employeeAddress);
console.log(employeeSal)
```

var vs let vs const

Can you predict the output?

```
var x = 10;

{
    var y = 20;
}

console.log(x);
console.log(y)
```

- In many languages like C# and Java, flower brackets { } create scope, but not JavaScript before ES 6.
- In the above example, 'y' will considered as a global variable though it is defined inside braces.
- From **ES6** onwards we declare the variable at **Block Level Scope** using **let** keyword.

```
let x = 10;

{
    let y = 20;
}

console.log(x);
console.log(y)
```

```
Output:
ReferenceError: y is not defined
```

- Both **let** and **const** are used to specify block level scope.
- Where **const** is used to specify block level scope and read-only variable.

```
let x = 10;

{
    let y = 20;
    y = 200;
    console.log(y)

    const PI = 3.14;
    PI = 3.19;
    console.log(PI);
```

```
}

console.log(x);
```

```
Output:
TypeError: Assignment to constant variable.
```

Working with Functions

- A function is a block of reusable code.
- A function allows us to write code once and use it as many times as we want just by calling the function.

Types of Functions:

Defining a function using function declaration

```
function add(x, y) {
    return x + y;
}

let result = add(10, 20);
console.log(result)
```

Anonymous Function Expression:

- We are creating a function without a name and assigning it to variable **add**.
- We use the name of the variable to invoke the function.

```
let add = function(x, y) { return x + y; };
let result = add(10, 20);
console.log(result)
```

Self-invoking function expression:

Syntax: (Function Definition)(Parameter Supply);

```
(function(x, y) {
    console.log(x + y)
})(10, 10);
```

Function with default-initialized parameters:

- ES6 supports default-initialized parameters that set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place.

Eg:

```
function greet(name = "Guest") {
    console.log("Welcome..." + name);
}

greet();
greet(undefined);
greet("Jones");
```

```
Output:
Welcome...Guest
Welcome...Guest
Welcome...Jones
```

Function with Rest Parameters:

- Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will take.

  The rest parameter (...) allows a function to treat an indefinite number of arguments as an array.

  In ES6, you can work with the arguments directly using the **arguments variable** that is visible inside every function body.

```
function getNames(...names) {
    console.log(names.join(" "));
}

getNames("Jones");
getNames("Anna", "Alice");
getNames("Wills", "Smith", "Jones");
```

```
Output:
Jones
Anna Alice
Wills Smith Jones
```

Arrow Functions (=>)

- Arrow functions are used for writing **function expressions** in short form.
- By using Arrow Functions we create concise code.

Syntax:

(Parameter list) => Function Logic.

Eg:

```
//Zero argument function
let wish = () => console.log("Hello...");
wish();

//One  argument function
let squareNum = num => console.log(num * num);
squareNum(10);

//2 Argument function
let sum = (num1, num2) => console.log(num1 + num2);
sum(100, 200);
```

```
Output:
Hello...
100
300
```

Template literals or string literals:

- It allows embedded expressions.
- You can use multi-line strings and string interpolation features with them.
- Template literals are enclosed by the **back tick** (` `) character instead of double or single quotes.
- Template literals can contain placeholders. These are indicated by the dollar sign and curly braces **${expression}.**

Eg:

```
let employeeName = "Sai";
let employeeContact = 5555222288;

console.log(employeeName + " contant number is " + employeeContact);
console.log(`${employeeName} contact number is ${employeeContact}`)
```

```
Output:
Sai contant number is 5555222288
Sai contact number is 5555222288
```

OOPS Concepts

What is Object?

- Object is the primary concept in OOP (Object Oriented Programming).
- "Object" represents a physical item that represents a **person or a thing**.
- Object is a collection of **properties** (details) and **methods** (manipulations).

  For example,

  Student, Employee, Product, etc
- We can create any no. of objects inside the program.

What is Property?

- Properties are the details about the object.
- Properties are used to store a value.

  For example,

  studentname="Scott" is a property in the "student object".
- Properties are stored inside the object.
- The value of property can be changed any no. of times.

What is Method?

- Methods are the operations / tasks of the object, which manipulates / calculates the data and do some process.
- Methods are specified inside the object.

What is Class?

- "Class" represents a model of the object, which defines list of properties and methods of the object.

  Eg:

  "Student" class represents structure (list of properties and methods) of every student object.

- We can create any no. of objects based on a class, using "**new**" keyword.
- All the objects of a class, shares same set of properties & methods of the class.
- We can store the reference of the object in "**reference variable**"; using which you can access the object.

```
class ClassName{
    properties
    methods
    constructor
}
```

Note:

- A JavaScript class is not an object.
- It is a template for JavaScript objects.

Eg:

```
class Student {
    studentId;
    studentName;
    studentMarks;

    getResult() {
        if (this.studentMarks >= 35) {
            return "Pass";
        } else {
            return "Fail";
        }
    }
}
let s1 = new Student();
s1.studentId = 101;
s1.studentName = "Sai";
s1.studentMarks = 90;

console.log(s1);
console.log(s1.studentId);
console.log(s1.studentName);
console.log(s1.studentMarks);
console.log(s1.getResult());
```

```
Student { studentId: 101, studentName: 'Sai', studentMarks: 90 }
```

```
101
Sai
90
Pass
```

Working with Constructors:

- Constructor is a special function which is a part of the class which is used to **initialize properties** of the class.
- Constructor will be called automatically when we create a new object for the class.
- Constructor will be called each time when new object is created.
- Constructor's name should be always "**constructor**" only.

Syntax of Constructor

```
constructor( parameter1, parameter2, … ){
    code here
}
```

Constructor - Example

```
class Student {
    studentId;
    studentName;
    studentMarks;

    constructor(studentId, studentName, studentMarks) {
        this.studentId = studentId;
        this.studentName = studentName;
        this.studentMarks = studentMarks;
    }
    getResult() {
        if (this.studentMarks >= 35) {
            return "Pass";
        } else {
            return "Fail";
        }
    }
}
let s1 = new Student(101, "Sai", 90);
console.log(s1);
console.log(s1.getResult());
```

```
let s2 = new Student(102, "Ram", 34);
console.log(s2);
console.log(s2.getResult());
```

```
Student { studentId: 101, studentName: 'Sai', studentMarks: 90 }
Pass
Student { studentId: 102, studentName: 'Ram', studentMarks: 34 }
Fail
```

Instance Properties vs Static Properties:

Instance Properties	Static Properties
➢ Belongs to object.	➢ Belongs to class.
➢ Instance Properties will get memory allocated for each and every object.	➢ Static Properties memory will get allocated only once at class level.
➢ Can be accessed only with **object reference variable.**	➢ Can be accessed with **class name** where it is declared

```
class Student {
  studentName = "Anna";
  static college = "Abc College";

}

let s = new Student();
console.log(s.studentName);
console.log(Student.college);
```

Instance Methods vs Static Methods

Instance Method	Static Method
➢ Instance method belongs to the object.	➢ A static method belongs to the class rather than object of a class.
➢ Instance method can be invoked using only **reference variable**.	➢ A static method can be invoked without creating instance of class.
➢ Instance methods are used for specific to the object only.	➢ Static Methods are used to create utility methods.

Example:

```
class Student {
    firstName;
    lastName;

    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    getFullName() {
        console.log(this.firstName + " " + this.lastName);
    }

}
let s1 = new Student("Adiseshu", "Dasari");
s1.getFullName();
```

```
class Calc {
    static add(x, y) {
        console.log(x + y);
    }
    static mul(x, y) {
        console.log(x * y);
    }
}
Calc.add(10, 5);
Calc.mul(10, 5);
```

Techniques to reuse the members of a class

- We can reuse the members of one class inside another class by using the following 2 ways.
  1. **By Composition (Has-A Relationship)**
  2. **By Inheritance (IS-A Relationship)**

Composition (Has-A Relationship):

- By using Class Name or by creating object we can access members of one class inside another class is nothing but composition (Has-A Relationship).

  Eg:

  Customer Has Address

  Car Has Engine
- The main advantage of Has-A Relationship is **Code Reusability**.

Example:

```
class Address {
    city;
    state;
    country;

    constructor(city, state, country) {
        this.city = city;
        this.state = state;
        this.country = country;
    }
}
class Customer {
    customerId;
    customerName;
    customerAddress;

    constructor(customerId, customerName, customerAddress) {
        this.customerId = customerId;
        this.customerName = customerName;
        this.customerAddress = customerAddress;
    }
}

let customerAddress = new Address('Hyderabad', 'Telangana', 'India');
let customer = new Customer(101, 'Jones', customerAddress);
console.log(customer);
```

```
Customer {
  customerId: 101,
  customerName: 'Jones',
  customerAddress: Address { city: 'Hyderabad',
state: 'Telangana', country: 'India' }
}
```

Inheritance:

- Inheritance is a concept of extending the parent class, by creating the child class.

- In this process, all the members (Variables and Methods) of parent class will be inherited (derived) into the child class.

- The main advantage of inheritance is **Code Reusability** and we can extend existing functionality with some more extra functionality.

- The "**extends**" keyword is used to create inheritance.

  For example,
  "Student" class extends "Person class".
  "Car" class extends "Vehicle" class.
  "SmartMobile" class extends "Mobile" class

Note:

1. When parent class has a constructor, the child class's constructor must call the parent class's constructor explicitly. Otherwise Error.

2. The "**super**" keyword represents parent class. It can be used to call the constructor or method of parent class.

Example:

```
class Person {
    name;
    age;

    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
    getDetails() {
        return this.name + " " + this.age;
    }
}
class Student extends Person {
    studentId;
    marks;

    constructor(name, age, studentId, marks) {
        super(name, age);
        this.studentId = studentId;
        this.marks = marks;
    }
    getDetails() {
        return this.name + " " + this.age + " " + this.studentId + " " + this.marks;
    }
}
let p = new Person("Sai", 25);
console.log(p);
console.log(p.getDetails());

let s = new Student("Ram", 30, 101, 35);
console.log(s);
console.log(s.getDetails());
```

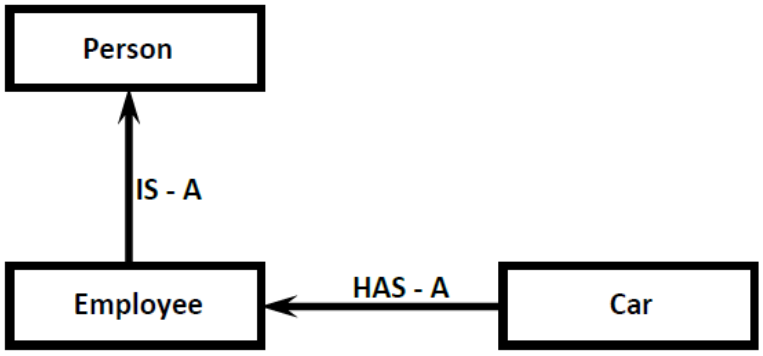
```
Person { name: 'Sai', age: 25 }
Sai 25
Student { name: 'Ram', age: 30, studentId: 101, marks: 35 }
Ram 30 101 35
```

IS-A vs HAS-A Relationship:

If we want to **extend** existing functionality with some more extra functionality then we should go for **IS-A Relationship**	If we don't want to extend and just we have to use existing functionality then we should go for **HAS-A Relationship**
It is also known as **Inhertance**.	It is also known as **Composition**

Eg: Employee class extends Person class Functionality

But Employee class just uses Car functionality but not extending



Overriding:

- Whatever the parent has by default available to the child.
- If the child is not satisfied with parent class implementation then child is allowed to redefine its implementation in its own way.
- This process is called **Overriding**.

Eg:

```
class Mobile {
    call() {
        console.log("Audio Calling");
    }
}
class SmartMobile extends Mobile {
    call() {
        console.log("Audio+Video Calling");
    }
}

let m = new Mobile();
m.call();

let sm = new SmartMobile();
sm.call();
```

```
Audio Calling
Audio+Video Calling
```

Working with Arrays:

- JS supports arrays concept to represent a group of homogeneous elements.

  Eg:

Creating Array:

Eg:

```
//traditional Syntax
let myArray1 = [90, 20, 30, 70, 50];
console.log(myArray1);

//Array() Syntax
let myArray2 = new Array();
myArray2 = [90, 20, 30, 70, 50];
console.log(myArray2)
```

Array Traversal using for of loop:

```
let myArray = [90, 20, 30, 70, 50];

console.log("Tradition For loop");
for (let x = 0; x < myArray.length; x++) {
    console.log(myArray[x])
}

console.log("Using For in loop")
for (let y in myArray) {
    console.log(myArray[y])
}

console.log("Using For of loop");
for (let a of myArray) {
    console.log(a)
}
```

Array Traversal Using forEach():

- The forEach() method executes a provided function once for each array element.

  Syntax:

```
arr.forEach(callback(currentValue [, index [, array]])[, thisArg])
```

Eg:

Using Named Function syntax	Using Arrow Function syntax

`let myArray = [90, 20, 30, 70, 50];`
 	
`function myFunction(value) {`
`  console.log(value);`
`}`
 	
`myArray.forEach(myFunction);`
	`let myArray = [90, 20, 30, 70, 50];`
	
	`myArray.forEach(value => console.log(value));`

Important Functions of Arrays:

map():

- The map() method **creates a new array** populated with the results of calling a provided function on every element in the calling array.

  Syntax:

```
let new_array = arr.map(function callback( currentValue[, index[, array]]) {
   // return element for new_array
}
```

Eg: Multiply each array value by 2

Using Named Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.map(myFunction);

function myFunction(value) {
    return value * 2;
}
console.log(result);
```

Using Arrow Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.map((value) => value * 2);
console.log(result);
```

filter():

- The filter() method creates a new array with all elements that pass the test implemented by the provided function.

  Syntax:

```
let newArray = arr.filter(callback(element[, index, [array]])[, thisArg])
```

Eg: generate an array with elements which are greater than 50

Using Named Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.filter(fun);

function fun(value) {
    return value > 50;
}

console.log(result);
```

Using Arrow Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.filter(value => value > 50);

console.log(result);
```

reduce():

- The reduce() method executes a reducer function (that you provide) on each element of the array, resulting in a **single output** value.

  Syntax:

```
arr.reduce(callback( accumulator, currentValue[, index[, array]] )[, initialValue])
```

Eg: Find the sum of elements of the given array.

Using Named Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let sumOfArrayElement = myArray.reduce(myFunction);

function myFunction(total, value) {
    return total + value;
}

console.log(sumOfArrayElement);
```

Using Arrow Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let sumOfArrayElement = myArray.reduce((total, value) => total + value);

console.log(sumOfArrayElement);
```

every():

- The every() method tests whether all elements in the array pass the test implemented by the provided function.
- It returns a Boolean value.

Syntax:

```
arr.every(callback(element[, index[, array]])[, thisArg])
```

Eg: Find whether all array elements are greater than or equal to 20 or not.

Using Named Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.every(fun);

function fun(value) {
    return value >= 20;
}

if (result)
    console.log("All elements are greater than equal to 20");
else
    console.log("All elements are not greater than equal to 20");
```

Using Arrow Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.every(value => value >= 20);

if (result)
    console.log("All elements are greater than equal to 20");
else
    console.log("All elements are not greater than equal to 20");
```

some():

- The some() method tests whether at least one element in the array passes the test implemented by the provided function.
- It returns a Boolean value.

Syntax:

Eg: Check is there element which is divisible by 30.

Using Named Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.some(fun);

function fun(value) {
    return value % 30 === 0;
}

console.log(result);
```

Using Arrow Function syntax:

```
let myArray = [90, 20, 30, 70, 50];

let result = myArray.some(value => value % 30 === 0);

console.log(result);
```

Some more functions on Arrays:

```
let myArray = [90, 20, 30, 70, 50];

console.log(myArray);
myArray.push(100);
console.log(myArray);
myArray.pop();
console.log(myArray);
myArray.sort()
console.log(myArray);
```

Working with Modules:

- It is not recommended or not possible to write entire code (variables, classes, interface, etc) in a single file.
- In large scale applications, it is recommended to write each class or functions in a separate file.
- To **separate code concerns** and **responsibilities** we need to go for **Modules** concept.
- Module is a simple .js file which contain at top-level "**import"** or "**export**" keywords
- We can export the code (variables, functions, classes, etc) from one module to other modules by using "**export**" keyword.
- We can use the exported the code (variables, functions, classes, etc) from one module to other modules by using "**import**" keyword.

Note:

1. We can't import the code (variables, functions, classes, etc) that are not exported from the module.
2. Module names are case insensitive.

Loading Modules:

1. Current folder, we use "**./module1.js**"
2. Sub folder in Current folder, we use **"./sub_folder_name/module1.js**"
3. Parent folder, we use "**../module1.js**"

Steps for development of modules:
Step 1: Export a class of mymodule.js

Syntax 1:
mymodule.js

```
export class Class1{}
export class Class2{}
export class Class3{}
```

Syntax 2:
mymodule.js

```
class Class1{}
class Class2{}
class Class3{}

export {Class1, Class2, Class3}
```

Step 2: Import the exported code in module.ts

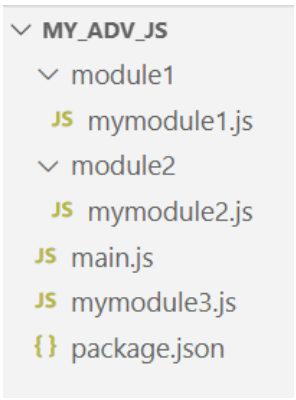
Importing single class.

```
import { Class1 } from "./mymodule.js";
```

Importing multiple classes.

```
import { Class1,Class2,Class3 } from "./mymodule.js ";
```

Eg:



package.json

```
{
    "type": "module"
}
```

Note:

If we don't to add package.json file with above property then we will get

SyntaxError: Cannot use import statement outside a module

module1\mymodule1.js

```
const PI = 3.14;

function greet() {
  console.log("Hello...");
}
export { PI, greet };
```

module2\mymodule2.js

```
class Calc {
  static doubleIt(num) {
    console.log(num + num);
  }
  static tripleIt(num) {
    console.log(num + num + num);
  }
}
export { Calc };
```

mymodule3.js

```
class Product {
  productId = 0;
  productName = null;
  productCost = 0.0;

  constructor(productId, productName, productCost) {
    this.productId = productId;
    this.productName = productName;
    this.productCost = productCost;
  }
}

export { Product };
```

main.js

```
import { PI, greet } from "./module1/mymodule1.js";
import { Calc } from "./module2/mymodule2.js";
import { Product } from "./mymodule3.js";

console.log(PI);

greet();

Calc.doubleIt(10);
Calc.tripleIt(10);

let p = new Product(101, "Laptop", 55000);
console.log(p);
```

Output:

```
my_adv_js>node main
3.14
Hello...
20
30
Product { productId: 101, productName: 'Laptop', productCost: 55000 }
```