**6. Create a module that exports a function and import it into another file.**

**JavaScript Importing and Exporting Modules**

JavaScript Modules are basically libraries which are included in the given program. They are used for connecting two JavaScript programs together to call the functions written in one program without writing the body of the functions itself in another program.

**Importing a library:** It means include a library in a program so that use the function is defined in that library. For this, use ‘require’ function in which pass the library name with its relative path.

**Example:** Suppose a library is made in the same folder with file name library.js, then include the file by using require function:

const lib = require('./library')

which will return a reference to that library. Now if there is an area function defined in the library, then use it as lib.area().

**Exporting a library:** There is a special object in JavaScript called module.exports. When some program include or import this module (program), this object will be exposed. Therefore, all those functions that need to be exposed or need to be available so that it can use in some other file, defined in module.exports.

**Example:**In this example, we will write two different programs and then see how to use functions defined in the library (Module) in a given program. We will define two simple functions in the library for calculating and printing area and perimeter of a rectangle when provided with length and breadth. Then export the functions so that other programs can import them if needed and can use them.

**Exporting Module Example:**library.js

* Javascript

|  |
| --- |
| let area = **function** (length, breadth) {          let a = length \* breadth;          console.log('Area of the rectangle is ' + a + ' square unit');      }        let perimeter = **function** (length, breadth) {          let p = 2 \* (length + breadth);          console.log('Perimeter of the rectangle is ' + p + ' unit');      }        module.exports = {          area,          perimeter      } |

**Importing Module Example**

For importing any module, use a function called ‘require’ which takes in the module name, and if it is a user-defined module then its relative path as an argument and returns its reference.

The *script.js* contains the above JavaScript module (library.js).

**Script.js**

* Javascript

|  |
| --- |
| const lib = require('./library');      // " ./ " is used if both the files are in the same folder.      let length = 10;      let breadth = 5;        lib.area(length, breadth);      lib.perimeter(length, breadth); |

**Output:**

Area of the rectangle is 50 square unit

Perimeter of the rectangle is 30 unit

**Note:** To run the script first make both files in the same folder and then run script.js using NodeJs interpreter in the terminal.

**7. Convert a JavaScript object to JSON and back using JSON.stringify() and JSON.parse().**

**How To Include a JavaScript File in Another JavaScript File?**

The**import**and**export**syntax available in ES6 (ECMAScript 2015) module is used to include a JavaScript file in another JavaScript file. This approach is the most modern and recommended way to share code between JavaScript files.

It allows you to break your code into smaller modules and then import the needed functions, classes, or variables from one file into another.

**Using ES6 Modules (import and export)**

With [ES6,](https://www.geeksforgeeks.org/introduction-to-es6/) you can export code from one file and import it into another. This method is the simplest way to include a [JavaScript](https://www.geeksforgeeks.org/javascript/) file in another.

**Example:**In the **math.mjs** file, export the functions or variables:

*// math.mjs*

**export** **function** add(n1, n2) {

**return** n1 + n2;

}

**export** **const** pi = 3.14159;

In the **main.mjs** file, import the exported functions or variables

*// main.mjs*

**import** { add, pi } **from** './math.mjs';

console.log(add(2, 3));

console.log(pi);

**Output**

5

3.14159

In this example, the add function and the pi constant are exported from the**math.mjs**file and imported into the **main.mjs** file using the import statement.

*Note: If you are going to use ES6 modules in a node js environment, remember to name your files with .mjs extension or set “type”: “module” in the package.json file.*

**Table of Content**

* [Using require() (CommonJS)](https://www.geeksforgeeks.org/how-to-include-a-javascript-file-in-another-javascript-file/#using-require-commonjs)
* [Using script Tags for Browser-based JavaScript](https://www.geeksforgeeks.org/how-to-include-a-javascript-file-in-another-javascript-file/#using-script-tags-for-browserbased-javascript)

**Using require() Method (CommonJS)**

In[Node.js](https://www.geeksforgeeks.org/nodejs/) or environments that support CommonJS, you can use the require() function to include one JavaScript file in another. This approach is commonly used in Node.js.

**Example:**

*// math.js*

**function** add(n1, n2) {

**return** n1 + n2;

}

**const** pi = 3.14159;

module.exports = { add, pi };

In the main.js file, use require() to import the module:

*// main.js*

**const** { add, pi } = require('./math.js');

console.log(add(2, 3));

console.log(pi);

**Output**

5

3.14159

Here, module.exports is used to export the add function and pi constant, and require() is used to import them in another file.

**Using script Tags for Browser-based JavaScript**

If you are working with JavaScript in a browser environment, you can include multiple JavaScript files by using[<script> tags](https://www.geeksforgeeks.org/html-script-tag/) in your HTML file. You can load external JavaScript files this way.

*<!-- index.html -->*

<**head**>

<**script** src="math.js"></**script**>

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

</**head**>

<**body**>

<**h1**>JavaScript File Inclusion</**h1**>

</**body**>

In this example, both math.js and main.js will be loaded in the specified order.

**Which Approach Is Better in Different Cases?**

* **ES6 Modules (import/export):** This is the preferred approach for modern JavaScript applications. It is well-suited for both client-side and server-side JavaScript and works well with tools like Webpack.
* **require() (CommonJS):**CommonJS is primarily used in Node.js. If you are working in a Node.js environment, using require() is still common, but ES6 modules are now supported as well.
* **Using <script> Tags:**This is suitable for simple projects or when you need to load JavaScript files directly in an HTML file. However, for complex projects, module bundlers like [Webpack](https://www.geeksforgeeks.org/introduction-to-webpack-module-bundler/) or [ES6 modules](https://www.geeksforgeeks.org/es6-modules/) are recommended.

**8. Define a class Animal with properties and a method, and create instances.**

**How to Create & Use Classes in JavaScript?**

In [JavaScript](https://www.geeksforgeeks.org/javascript/), you can create and use classes to define blueprints for creating objects with similar properties and behaviors.

Classes provide a way to implement object-oriented programming (OOP) concepts such as encapsulation, inheritance, and polymorphism.

**Example:**Here, we define a class called Animal, representing animals with a name and a sound. It has a method makeSound() to display the animal's name and the sound it makes. Two instances of Animal class, cat and dog, are created with specific names and sounds ('Cat' and 'Meow' for cat, 'Dog' and 'Woof' for dog). Using these instances, the makeSound() method is called for each animal, displaying their names and respective sounds.

*// Class Declaration*

**class** Animal {

**constructor**(name, sound) {

**this**.name = name;

**this**.sound = sound;

}

makeSound() {

console.log(`**${this**.name**}** says **${this**.sound**}**`);

}

}

*// Creating Instances*

**const** cat = **new** Animal('Cat', 'Meow');

**const** dog = **new** Animal('Dog', 'Woof');

*// Using Instances*

cat.makeSound(); *// Output: Cat says Meow*

dog.makeSound(); *// Output: Dog says Woof*

**Output**

Cat says Meow

Dog says Woof

**9. Write a promise that resolves after a delay and convert it into an async function using async/await.**

**How to delay a loop in JavaScript using async/await with Promise?**

In JavaScript, you can delay a loop by using async/await with Promise. By wrapping a setTimeout() inside a Promise and using await, you can pause execution at each iteration, creating delays between loop iterations for asynchronous tasks without blocking the main thread.

**What is async and await?**

[async and await](https://www.geeksforgeeks.org/async-await-function-in-javascript/) in JavaScript are used for handling asynchronous operations. async declares a function as asynchronous, while await pauses the function’s execution until a [Promise](https://www.geeksforgeeks.org/javascript-promise/)is resolved, enabling cleaner, non-blocking code.

**Syntax:**

*async* function delay() {

return new Promise(resolve => {resolve()})

}

[**JavaScript await**](https://www.geeksforgeeks.org/async-await-function-in-javascript/)makes a function wait for a Promise: await is mainly used while calling a function.

**Syntax:**

await delay();

**Approach:**A Promise in JavaScript pauses code execution until it resolves, returning control to the calling method once completed. The waitforme function delays code for a specified duration in milliseconds, allowing controlled pauses during asynchronous execution.

**Example:**This example shows the use of the above-explained approach.

**function** waitforme(millisec) {

**return** **new** Promise(resolve => {

setTimeout(() => { resolve('') }, millisec);

})

}

**async** **function** printy() {

**for** (**let** i = 0; i < 10; ++i) {

**await** waitforme(1000);

console.log(i);

}

console.log("Loop execution finished!)");

}

printy();

**Output:**

0

1

2

3

4

5

6

7

8

9

Loop execution finished!)

**Note:**We can change the value of the parameter of waitforme function while calling the function to increase/decrease the delay in the code.

**10. Use VSCode to set breakpoints and inspect variables in a JavaScript program.**

**Debug a JavaScript or TypeScript app in Visual Studio**

1. [Configure debugging](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#configure-debugging)
2. [Debug server-side script](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#debug-server-side-script)
3. [Debug client-side script](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#debug-client-side-script)
4. [Generate source maps for debugging](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#generate_source_maps)

You can debug JavaScript and TypeScript code using Visual Studio. You can hit breakpoints, attach the debugger, inspect variables, view the call stack, and use other debugging features.

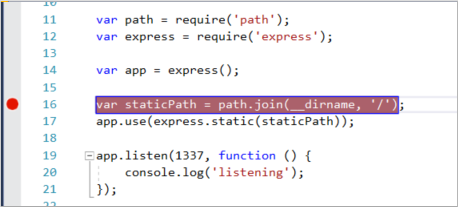
**Configure debugging**

For *.esproj* projects in Visual Studio 2022, Visual Studio Code uses a *launch.json* file to configure and customize the debugger. *launch.json* is a debugger configuration file.

Visual Studio attaches the debugger only to user code. For *.esproj* projects, you can configure user code (also called *Just My Code* settings) in Visual Studio using the skipFiles setting in *launch.json*. This works the same as the *launch.json* settings in VS Code. For more information about *skipFiles* and other debugger configuration options, see [Skipping Uninteresting Code](https://code.visualstudio.com/docs/nodejs/nodejs-debugging#_skipping-uninteresting-code) and [Launch configuration attributes](https://code.visualstudio.com/docs/nodejs/nodejs-debugging#_launch-configuration-attributes).

**Debug server-side script**

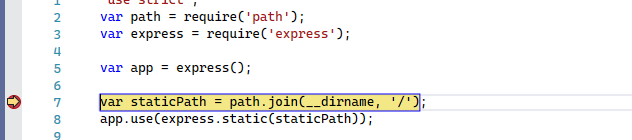
1. With your project open in Visual Studio, open a server-side JavaScript file (such as *server.js*), click in the gutter to set a breakpoint:



Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code, so you can look at the values of variables or the behavior of memory, or whether or not a branch of code is getting run.

1. To run your app, press **F5** (**Debug** > **Start Debugging**).

The debugger pauses at the breakpoint you set (IDE highlights the statement in the yellow background). Now, you can inspect your app state by hovering over variables currently in scope, using debugger windows like the **Locals** and **Watch** windows.



1. Press **F5** to continue the app.
2. If you want to use the Chrome Developer Tools, press **F12** in the Chrome browser. Using these tools, you can examine the DOM or interact with the app using the JavaScript Console.

**Debug client-side script**

Visual Studio provides client-side debugging support only for Chrome and Microsoft Edge. In some scenarios, the debugger automatically hits breakpoints in JavaScript and TypeScript code and embedded scripts on HTML files.

* For debugging client-side script in ASP.NET apps, choose **Tools** > **Options** > **Debugging**, and then select **Enable JavaScript Debugging for ASP.NET (Chrome, Edge, and IE)**.

If you prefer to use Chrome Developer Tools or F12 Tools for Microsoft Edge to debug client-side script, you should disable this setting.

* For Node.js applications and other JavaScript projects, follow the steps described in this article.

**Prepare your app for debugging**

If your source is minified or created by a transpiler like TypeScript or Babel, use [source maps](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#generate_source_maps) for the best debugging experience. You can even attach the debugger to a running client-side script without the source maps. However, you may only be able to set and hit breakpoints in the minified or transpiled file, not in the source file. For example, in a Vue.js app, the minified script gets passed as a string to an eval statement, and there's no way to step through this code effectively using the Visual Studio debugger unless you use source maps. For complex debugging scenarios, you may want to use Chrome Developer Tools or F12 Tools for Microsoft Edge instead.

**Prepare the browser for debugging**

For this scenario, use either Microsoft Edge or Chrome.

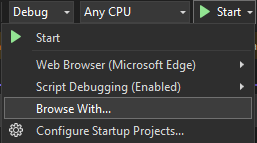
1. Close all windows for the target browser, either Microsoft Edge or Chrome instances.

Other browser instances can prevent the browser from opening with debugging enabled. (Browser extensions may be running and intercept full debug mode, so you may need to open Task Manager to find and close unexpected instances of Chrome or Edge.)

For best results, shut down all instances of Chrome, even if you're working with Microsoft Edge. Both the browsers use the same chromium code base.

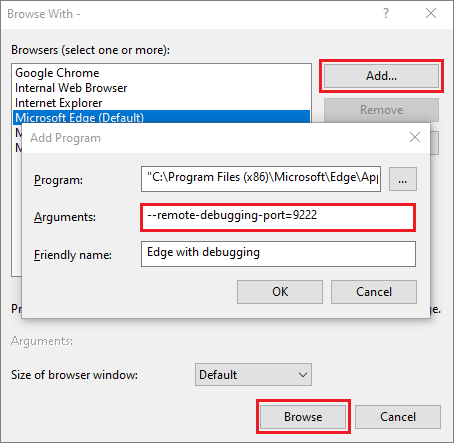
1. Start your browser with debugging enabled.

Starting in Visual Studio 2019, you can set the --remote-debugging-port=9222 flag at browser launch by selecting **Browse With...** > from the **Debug** toolbar.



If you don't see the **Browse With...** command in the **Debug** toolbar, select a different browser, and then retry.

From the Browse With dialog box, choose **Add**, and then set the flag in the **Arguments** field. Use a different friendly name for the browser, like **Edge Debug Mode** or **Chrome Debug Mode**. For details, see the [Release Notes](https://learn.microsoft.com/en-us/visualstudio/releases/2019/release-notes-v16.2).



Select **Browse** to start your app with the browser in debug mode.

Alternatively, open the **Run** command from the Windows **Start** button (right-click and choose **Run**), and enter the following command:

msedge --remote-debugging-port=9222

or,

chrome.exe --remote-debugging-port=9222

This starts your browser with debugging enabled.

The app isn't yet running, so you get an empty browser page. (If you start the browser using the Run command, you need to paste in the correct URL for your app instance.)

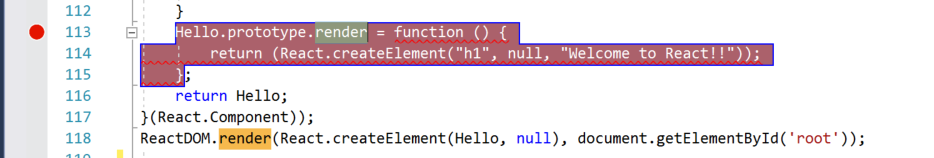
**Attach the debugger to client-side script**

To attach the debugger from Visual Studio and hit breakpoints in the client-side code, it needs help with identifying the correct process. Here's one way to enable it.

1. Make sure your app is running in the browser in debug mode, as described in the preceding section.

If you created a browser configuration with a friendly name, choose that as your debug target, and then press **Ctrl**+**F5** (**Debug** > **Start Without Debugging**) to run the app in the browser.

1. Switch to Visual Studio and then set a breakpoint in your source code, which might be a JavaScript file, TypeScript file, or a JSX file. (Set the breakpoint in a line of code that allows breakpoints, such as a return statement or a var declaration.)



To find the specific code in a transpiled file, use **Ctrl**+**F** (**Edit** > **Find and Replace** > **Quick Find**).

For client-side code, to hit a breakpoint in a TypeScript file, *.vue*, or *JSX* file typically requires the use of [source maps](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#generate_source_maps). A source map must be configured correctly to support debugging in Visual Studio.

1. Choose **Debug** > **Attach to Process**.

**Tip**

Starting in Visual Studio 2017, after you attach to the process the first time by following these steps, you can quickly reattach to the same process by choosing **Debug** > **Reattach to Process**.

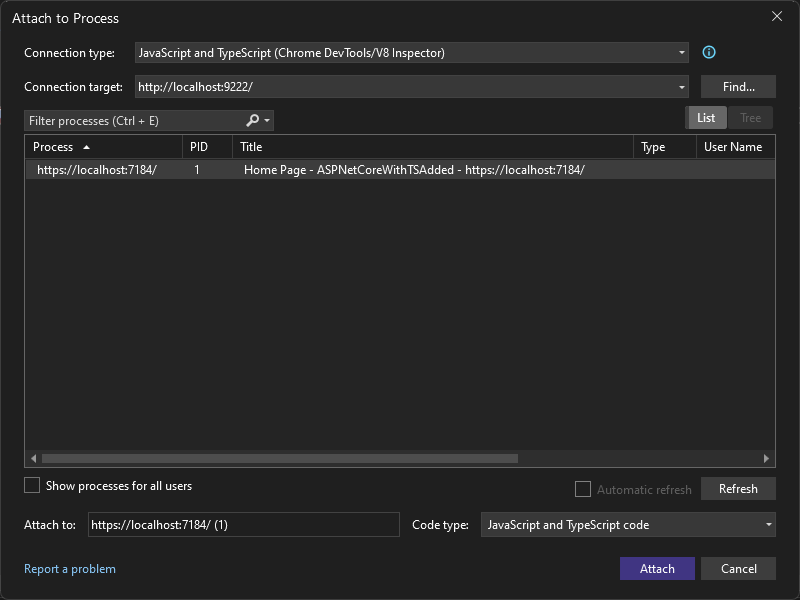
1. In the **Attach to Process** dialog, select **JavaScript and TypeScript (Chrome Dev Tools/V8 Inspector)** as the **Connection Type**.

The debugger target, such as http://localhost:9222, should appear in the **Connection Target** field.

1. In the list of browser instances, select the browser process with the correct host port (https://localhost:7184/ in this example), and select **Attach**.

The port (for example, 7184) may also appear in the **Title** field to help you select the correct browser instance.

The following example shows how this looks for the Microsoft Edge browser.



**Tip**

If the debugger does not attach and you see the message "Failed to launch debug adapter" or "Unable to attach to the process. An operation is not legal in the current state.", use the Windows Task Manager to close all instances of the target browser before starting the browser in debugging mode. Browser extensions may be running and preventing full debug mode.

1. The code with the breakpoint may have already been executed, refresh your browser page. If necessary, take action to cause the code with the breakpoint to execute.

While paused in the debugger, you can examine your app state by hovering over variables and using debugger windows. You can advance the debugger by stepping through code (**F5**, **F10**, and **F11**). For more information on basic debugging features, see [First look at the debugger](https://learn.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour?view=vs-2022).

You may hit the breakpoint in either a transpiled .js file or source file, depending on your app type, which steps you followed previously, and other factors such as your browser state. Either way, you can step through code and examine variables.

* + If you need to break into code in a TypeScript, JSX, or .vue source file and are unable to do it, make sure that your environment is set up correctly, as described in the [Troubleshooting](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#troubleshooting_source_maps) section.
  + If you need to break into code in a transpiled JavaScript file (for example, *app-bundle.js*) and are unable to do it, remove the source map file, *filename.js.map*.

**Troubleshooting breakpoints and source maps**

If you need to break into code in a TypeScript or JSX source file and are unable to do it, use **Attach to Process** as described in the previous section to attach the debugger. Make sure that your environment is set up correctly:

* Close all browser instances, including Chrome extensions (using the Task Manager), so that you can run the browser in debug mode.
* Make sure you [start the browser in debug mode](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#prepare_the_browser_for_debugging).
* Make sure that your source map file includes the correct relative path to your source file and that it doesn't include unsupported prefixes such as *webpack:///*, which prevents the Visual Studio debugger from locating a source file. For example, a reference like *webpack:///.app.tsx* might be corrected to *./app.tsx*. You can do this manually in the source map file (which is helpful for testing) or through a custom build configuration. For more information, see [Generate source maps for debugging](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#generate_source_maps).

Alternatively, if you need to break into code in a source file (for example, *app.tsx*) and are unable to do it, try using the debugger; statement in the source file, or set breakpoints in the Chrome Developer Tools (or F12 Tools for Microsoft Edge) instead.

**Generate source maps for debugging**

Visual Studio has the capability to use and generate source maps on JavaScript source files. This is often required if your source is minified or created by a transpiler like TypeScript or Babel. The options available depend on the project type.

* A TypeScript project in Visual Studio generates source maps for you by default. For more information, see [Configure source maps using a tsconfig.json file](https://learn.microsoft.com/en-us/visualstudio/javascript/debug-nodejs?view=vs-2022#configure_source_maps).
* In a JavaScript project, you can generate source maps using a bundler like webpack and a compiler like the TypeScript compiler (or Babel), which you can add to your project. For the TypeScript compiler, you must also add a tsconfig.json file and set the sourceMap compiler option. For an example that shows how to do this using a basic webpack configuration, see [Create a Node.js app with React](https://learn.microsoft.com/en-us/visualstudio/javascript/tutorial-nodejs-with-react-and-jsx?view=vs-2022).