M02 - JavaScript Fundamentals

•••

Debugging

Debugging

What is debugging?

- Debugging is the process of finding and fixing errors within a script

- All modern browsers and most other environments support debugging tools
  - A special UI in developer tools that makes debugging much easier
  - It also allows to trace the code step by step to see what exactly is going on

Debugging

What is debugging?

- Poor-man debugging (use of console.log)
  - To output something to console from our code, there's console.log function.
  - For instance, this outputs values from 0 to 4 to console:

```
// open console to see
for (let i = 0; i < 5; i++) {
    console.log("value,", i);
}
```

- Regular users don't see that output, it is in the console
- To see it, either open the Console panel of developer tools or press Esc while in another panel: that opens the console at the bottom

Debugging

What is debugging?

- In order to have a more debugger controlled environment:
  - to understand the flow execution of the code
  - to inspect the evolution of variables during code life cycle
  - to add breakpoints and conditional breakpoints to pause the code in suspicious parts

- We should use debugging facilities of the development environment
  - in the browser (e.g. Chrome DevTools)
  - in the editor (e.g. Visual Studio Code debugger)

Debugging

1. Debugging in Chrome
2. Debugging in Visual Studio Code

Debugging

1. Debugging in Chrome

Debugging

1. Debugging in Chrome

- In order to explain how to debug web apps in Chrome, start by creating:
  - index.html
  - index.js

```html
<body>
  <script src="index.js"></script>
</body>
```

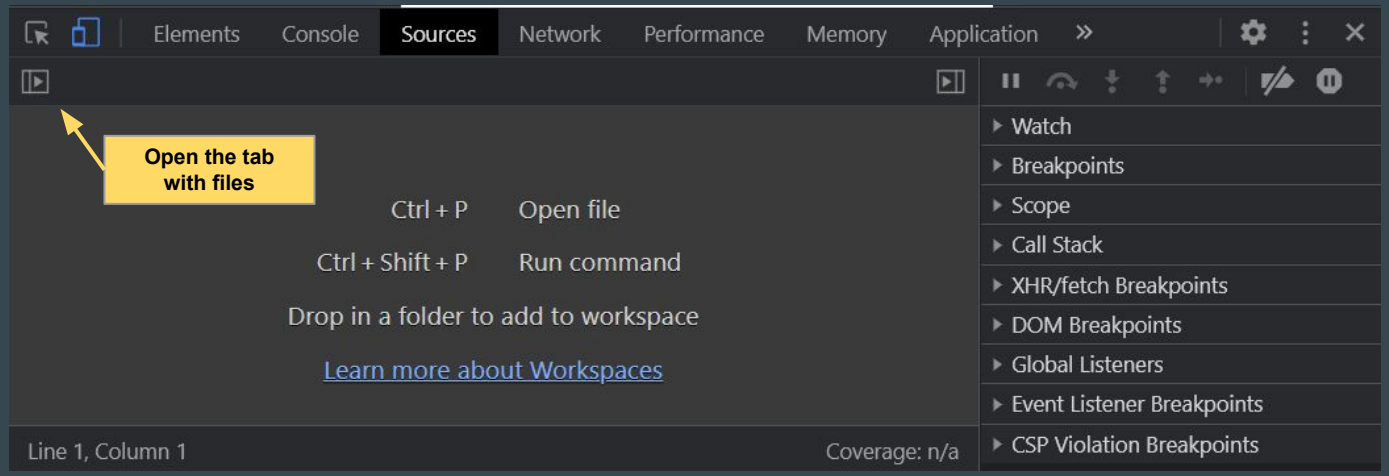
```javascript
function hello(name) {
    let phrase = `Hello, ${name}!`
    say(phrase)
}

function say(phrase) {
    alert(`** ${phrase} **`)
}
```

- Open the index.html in Chrome

Debugging

1. Debugging in Chrome

- Open DevTools (F12) and its Sources tab
- Open the tab with files



Debugging

1. Debugging in Chrome

- Select the index.js file
- Three panels: File navigator, Code editor e JS debugging

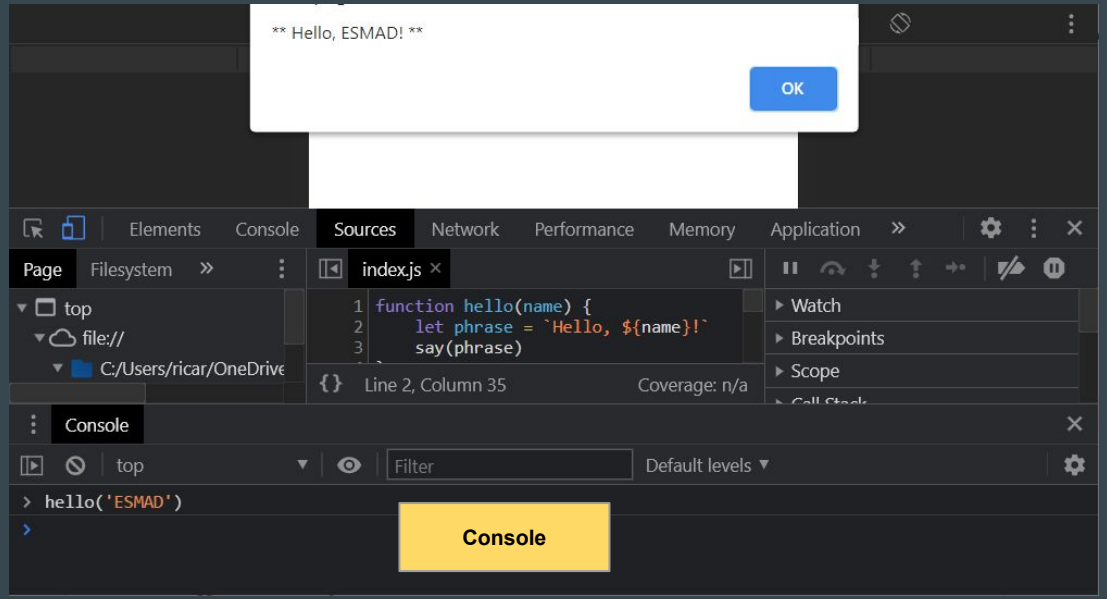
Debugging

1. Debugging in Chrome

- To execute code press ESC to show the Console

- Write: hello(‘ESMAD’)

- The app will show an alert box as expected



Debugging

1. Debugging in Chrome

Breakpoints

- Point of code where the debugger will automatically pause the JS execution

- While the code is paused, we can examine current variables, execute commands in the console, etc.

- In other words, we can debug it!

index.js

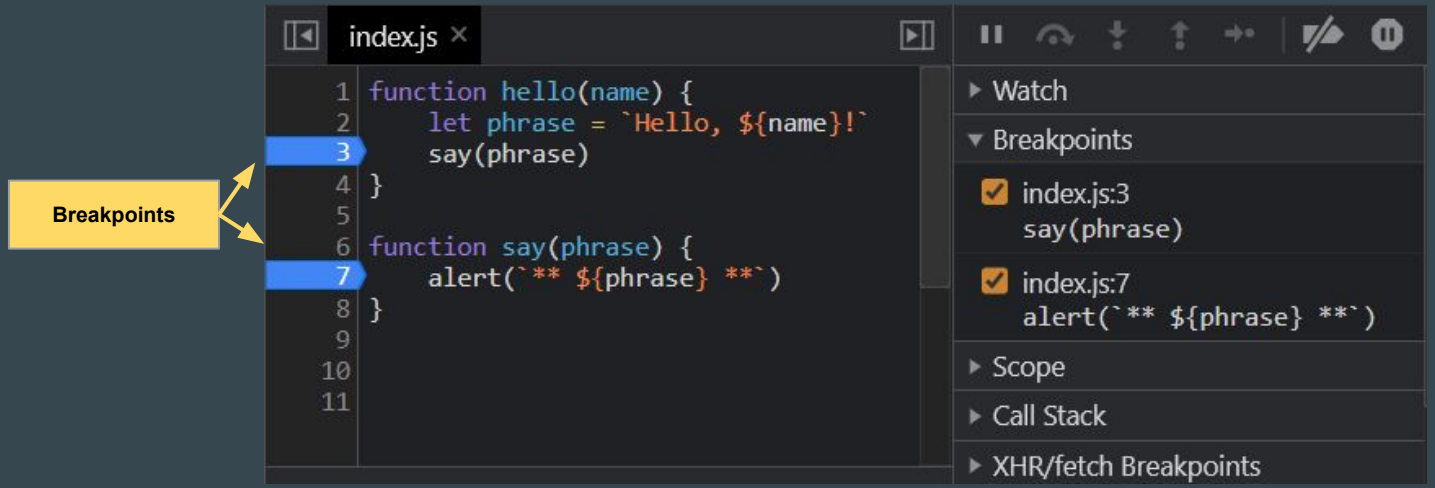
```
1  function hello(name) {
2      let phrase = `Hello, ${name}!`
3      say(phrase)
4  }
5
6  function say(phrase) {
7      alert(`** ${phrase} **`)
8  }
9
10
11
```

Debugging

1. Debugging in Chrome

Breakpoints

- To define a breakpoint in a code line, click in its line number

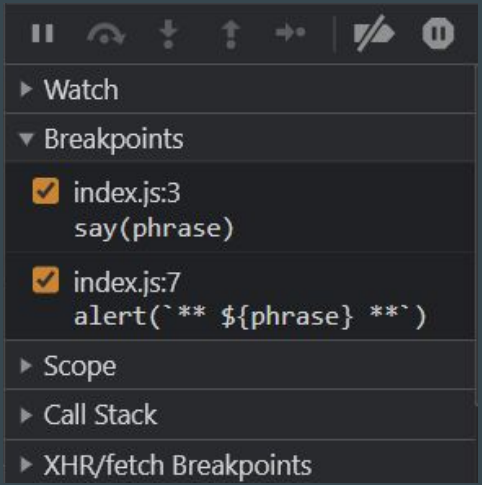


Debugging

1. Debugging in Chrome

Breakpoints

- A list of breakpoints can be seen in the right panel (breakpoints section)
- Useful for when we have several breakpoints
- Main features:
  - Quickly jump to the breakpoint in the code (by clicking on it in the right panel).
  - Temporarily disable the breakpoint by unchecking it
  - Remove the breakpoint by right-clicking and selecting Remove

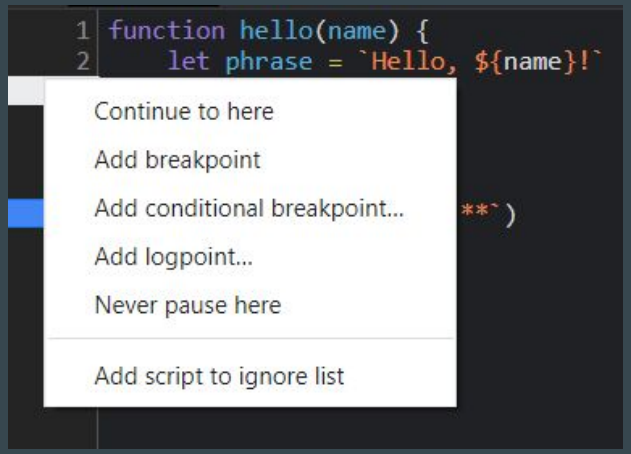


Debugging

1. Debugging in Chrome

Breakpoints

- Right click on line number allows:
  - Add Conditional breakpoints
    - it only triggers when the given expression is truthy
    - Handy when we need stop only for a variable value
  - Add Logpoints
    - to insert personalized logs
  - Continue to here
    - when we want to move multiple steps forward to the line, but we're too lazy to set a breakpoint.



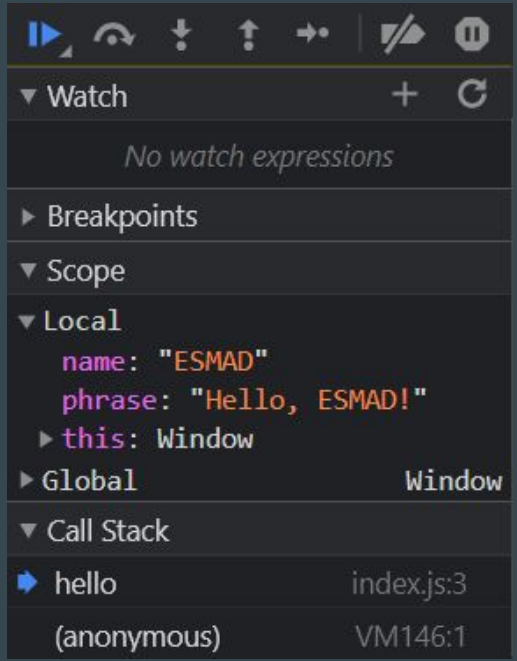
Debugging

1. Debugging in Chrome

Initiate debugging

- Define breakpoints in line 3 and 7
- Call from the console: hello('ESMAD')
- The execution pauses at the 3th line:

```
1 function hello(name) {   name = "ESMAD"
2     let phrase = `Hello, ${name}!`   phrase = "Hello, ESMAD!"
3     say(phrase)
4 }
5
6 function say(phrase) {
7     alert(`** ${phrase} **`)
8 }
```

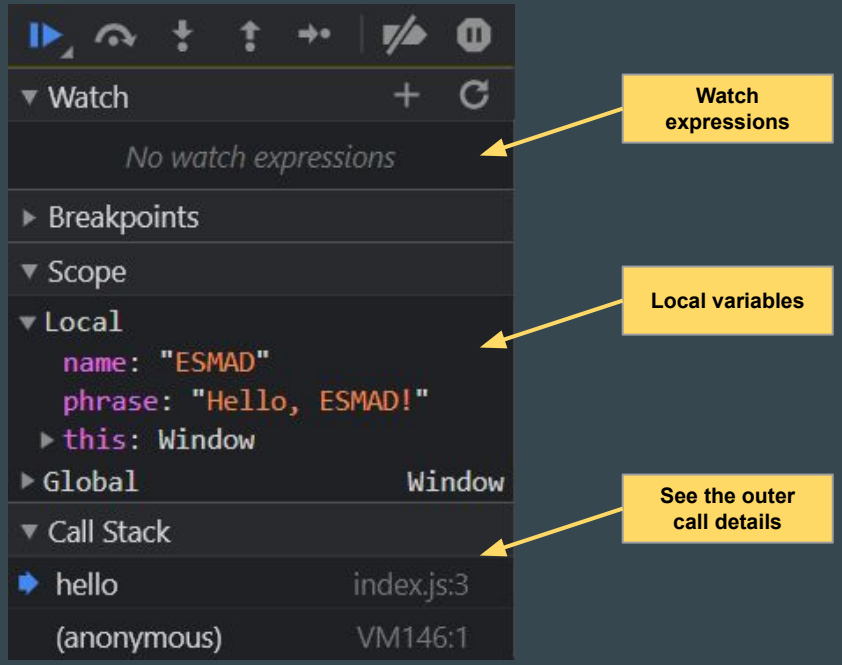


Debugging

1. Debugging in Chrome

Initiate debugging

- Sections in the right panel
  - Watch
  - Scope
  - Call stack

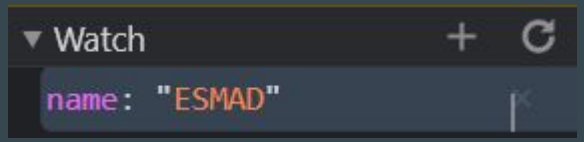


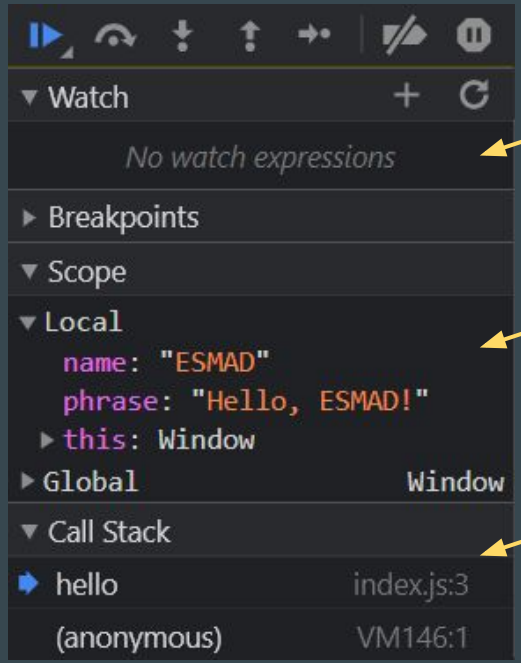
Debugging

1. Debugging in Chrome

Initiate debugging

- Watch
  - shows current values for any expressions
  - You can click the plus + and input an expression
  - The debugger will show its value at any moment, automatically recalculating it in the process of execution





Watch expressions

Local variables

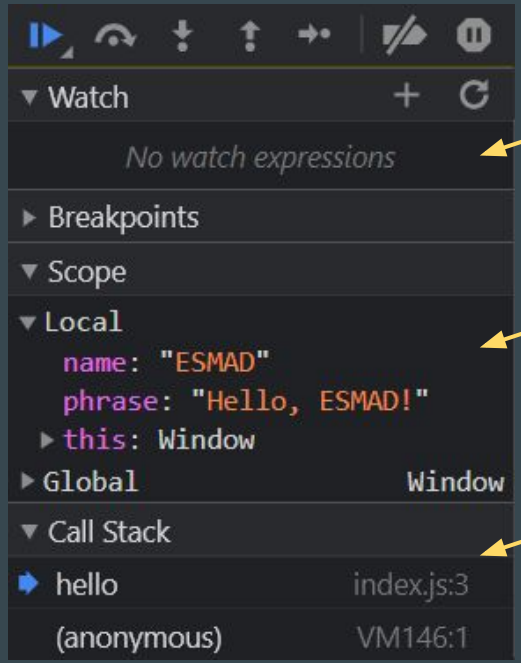
See the outer call details

Debugging

1. Debugging in Chrome

Initiate debugging

- Scope
  - Current values
  - Local shows local function variables
  - Global has global variables (out of any functions)
  - There's also this keyword there that we didn't study yet, but we'll do that soon



Watch expressions

Local variables

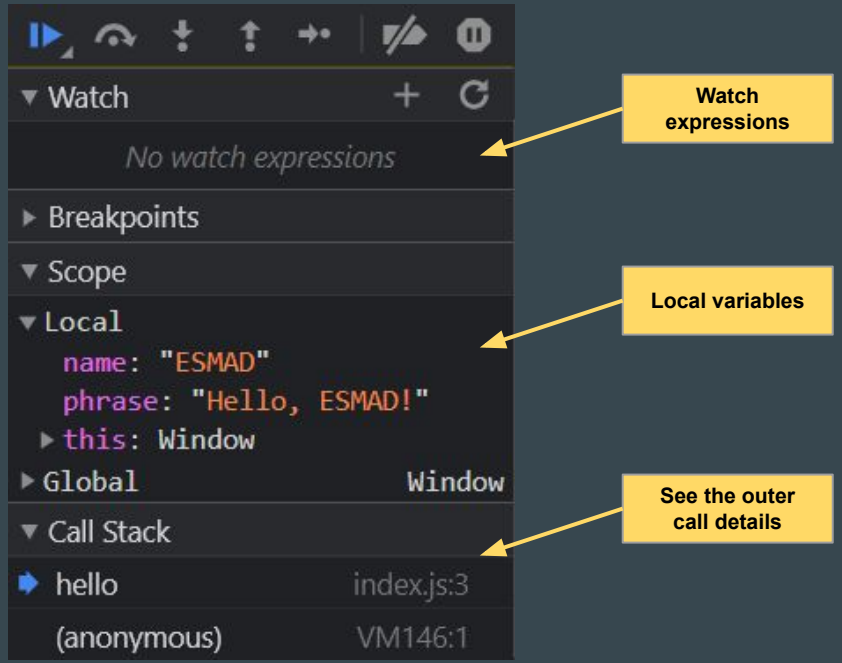
See the outer call details

Debugging

1. Debugging in Chrome

Initiate debugging

- Call stack
  - shows the nested calls chain
  - At the current moment the debugger is inside hello() call, called by index.js
    (no function there, so it's called "anonymous")
  - If you click on a stack item (e.g. "anonymous"), the debugger jumps to the corresponding code, and all its variables can be examined as well

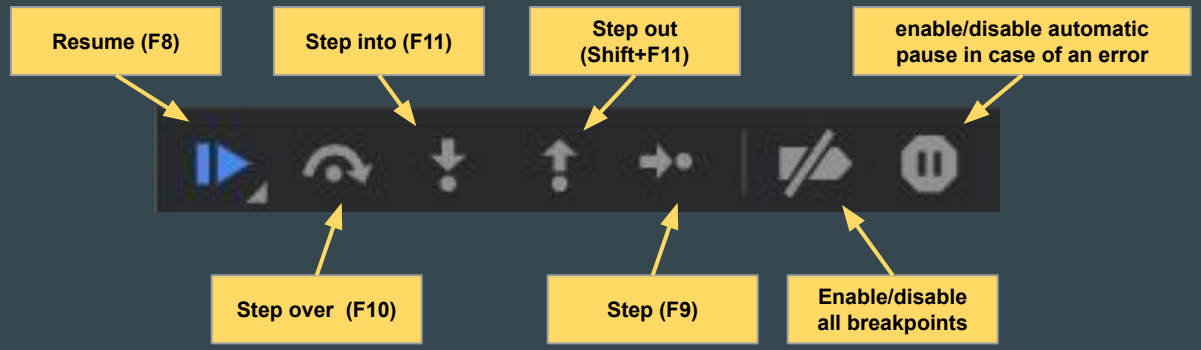


Debugging

1. Debugging in Chrome

Tracing the execution

- Now it's time to trace the script.
- There are buttons for it at the top of the right panel

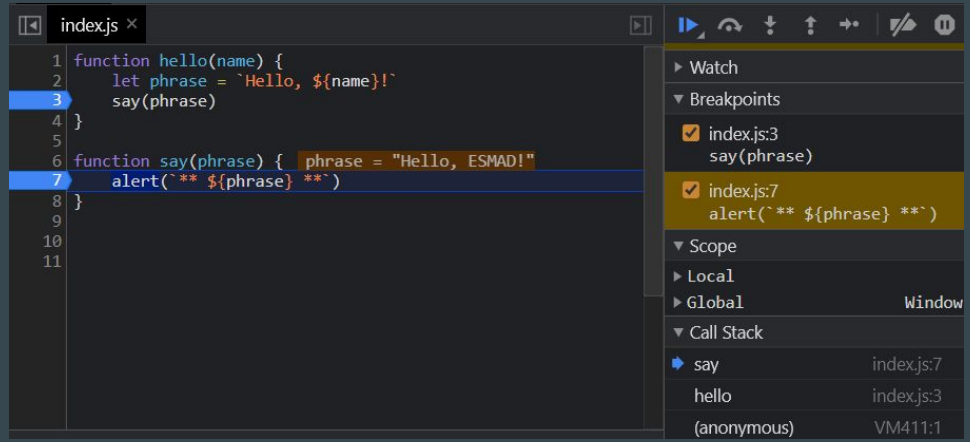


Debugging

1. Debugging in Chrome

Tracing the execution

- Resume (F8)
  - Continue the execution until the next breakpoint and paused there
  - If there are no additional breakpoints, then the execution just continues and the debugger loses control

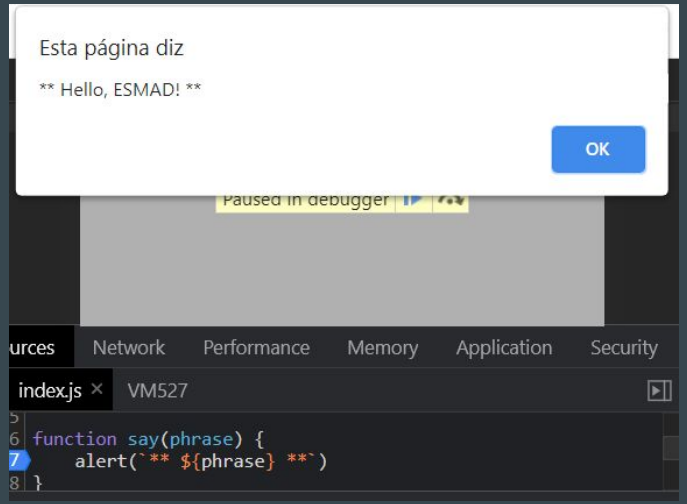


Debugging

1. Debugging in Chrome

Tracing the execution

- Step (F9)
  - Run the next command
  - If we click it now, alert will be shown
  - Clicking this again and again will step through all script statements one by one



Debugging

1. Debugging in Chrome

Tracing the execution

- Step over (F10)
  - Run the next command, but don't go into a function
  - If the next statement is a function call (not a built-in, like alert), it executes the nested function call invisibly, skipping the function internals
  - The execution is then paused after that function
  - That's good if we're not interested to see what happens inside the function call

Debugging

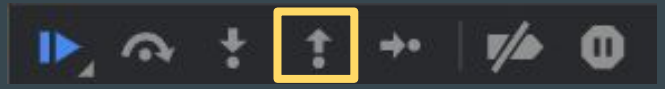
1. Debugging in Chrome

Tracing the execution

- Step into (F11)
  - That's similar to "Step", but behaves differently in case of asynchronous function calls
  - If you're only starting to learn JavaScript, then you can ignore the difference, as we don't have asynchronous calls yet.

Debugging

1. Debugging in Chrome



Tracing the execution

- Step out (Shift + F11)
  - Continue the execution till the end of the current function
  - Continue the execution and stop it at the very last line of the current function
  - That's handy when we accidentally entered a nested call using , but it does not interest us, and we want to continue to its end as soon as possible

Debugging

1. Debugging in Chrome

Tracing the execution

- Enable/Disable all breakpoints
  - That button does not move the execution. Just a mass on/off for breakpoints

- Enable/disable automatic pause in case of an error
  - When enabled, and the developer tools is open, a script error automatically pauses the execution
  - Then we can analyze variables to see what went wrong
  - So if our script dies with an error, we can open debugger, enable this option and reload the page to see where it dies and what's the context at that moment

Debugging

1. Debugging in Chrome

The debugger command

- We can also pause the code by using the debugger command in it, like this:

```
function hello(name) {
    let phrase = `Hello, ${name}!`
    debugger;  // <-- the debugger stops here
    say(phrase)
}
```

- That's very convenient when we are in a code editor and don't want to switch to the browser and look up the script in developer tools to set the breakpoint!

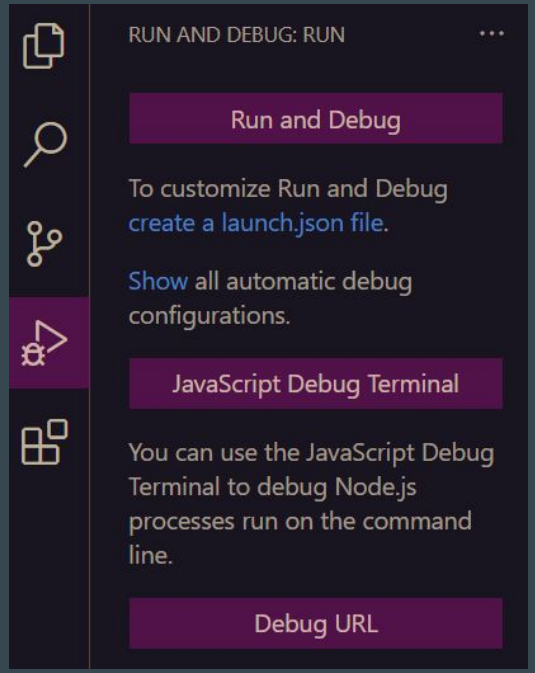
Debugging

2. Debugging in Visual Studio Code

Debugging

2. Debugging in Visual Studio Code

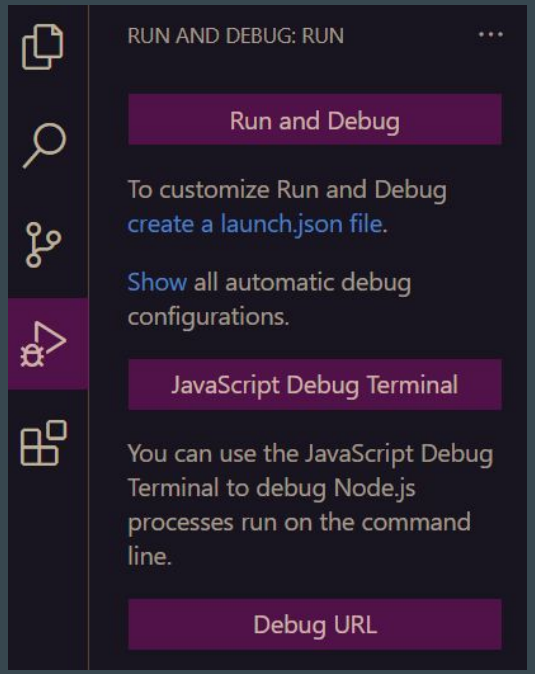
- One of the key features of VSC is its debugging support
- VSC's built-in debugger helps accelerate your edit, compile and debug loop.
- In the Activity Bar, select Run and Debug (Ctrl+Shift+D)
- If running and debugging is not configured (no launch.json created), VSC shows the Run start view



Debugging

2. Debugging in Visual Studio Code

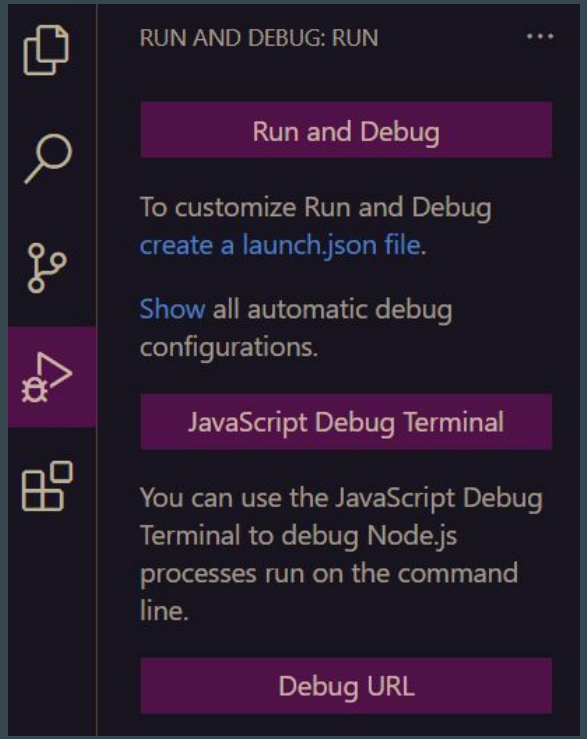
- Run and Debug
  - run or debug a simple app in VSC (F5) and VSC will try to run your currently active file
- Create a launch.json file
  - for most debugging scenarios, creating a launch configuration file is beneficial because it allows you to configure and save debugging setup details
- Javascript Debug Terminal
  - to debug Node.js processes on the command line
- Debug URL
  - to debug a app through a URL given



Debugging

2. Debugging in Visual Studio Code

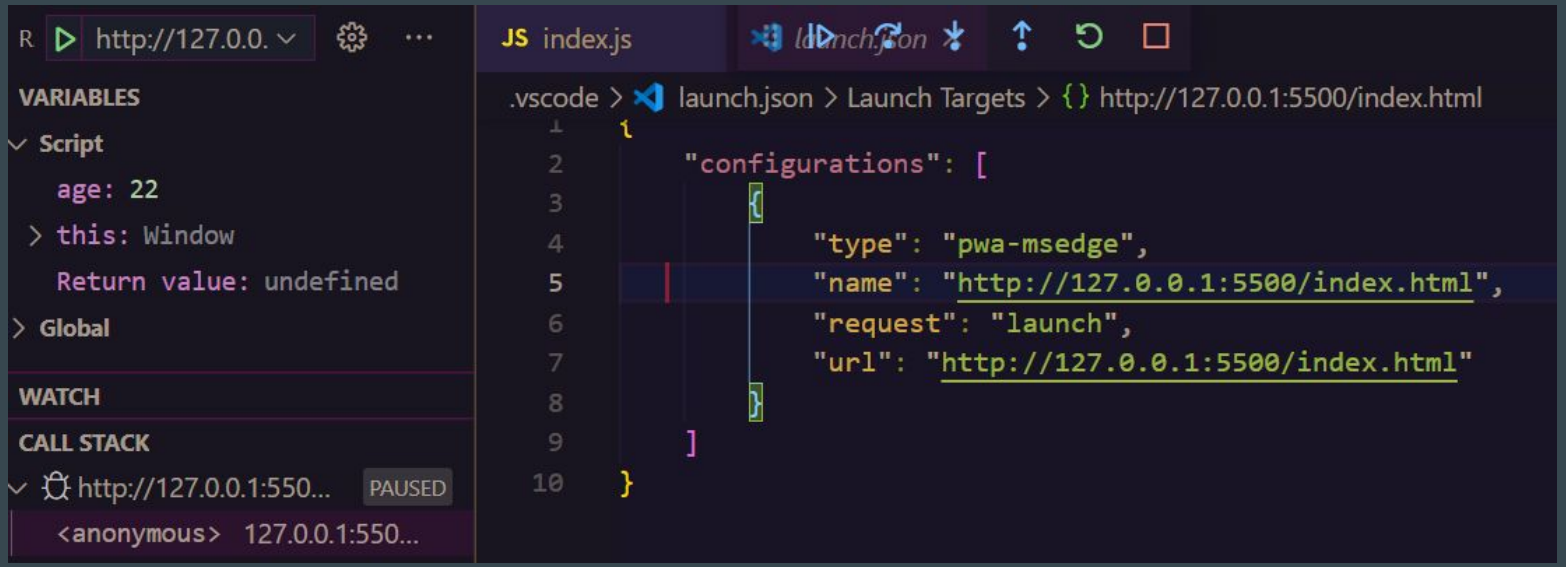
- Debug URL
  - Run index.html through Live Server
  - Copy URL: http://127.0.0.1:5500/index.html
  - Click Debug URL button and paste the URL
  - The config file launch.json is automatically updated!



Debugging

2. Debugging in Visual Studio Code

- Configuration file (.vscode/launch.json)



Debugging

2. Debugging in Visual Studio Code

- You can add more config instances

Chrome config

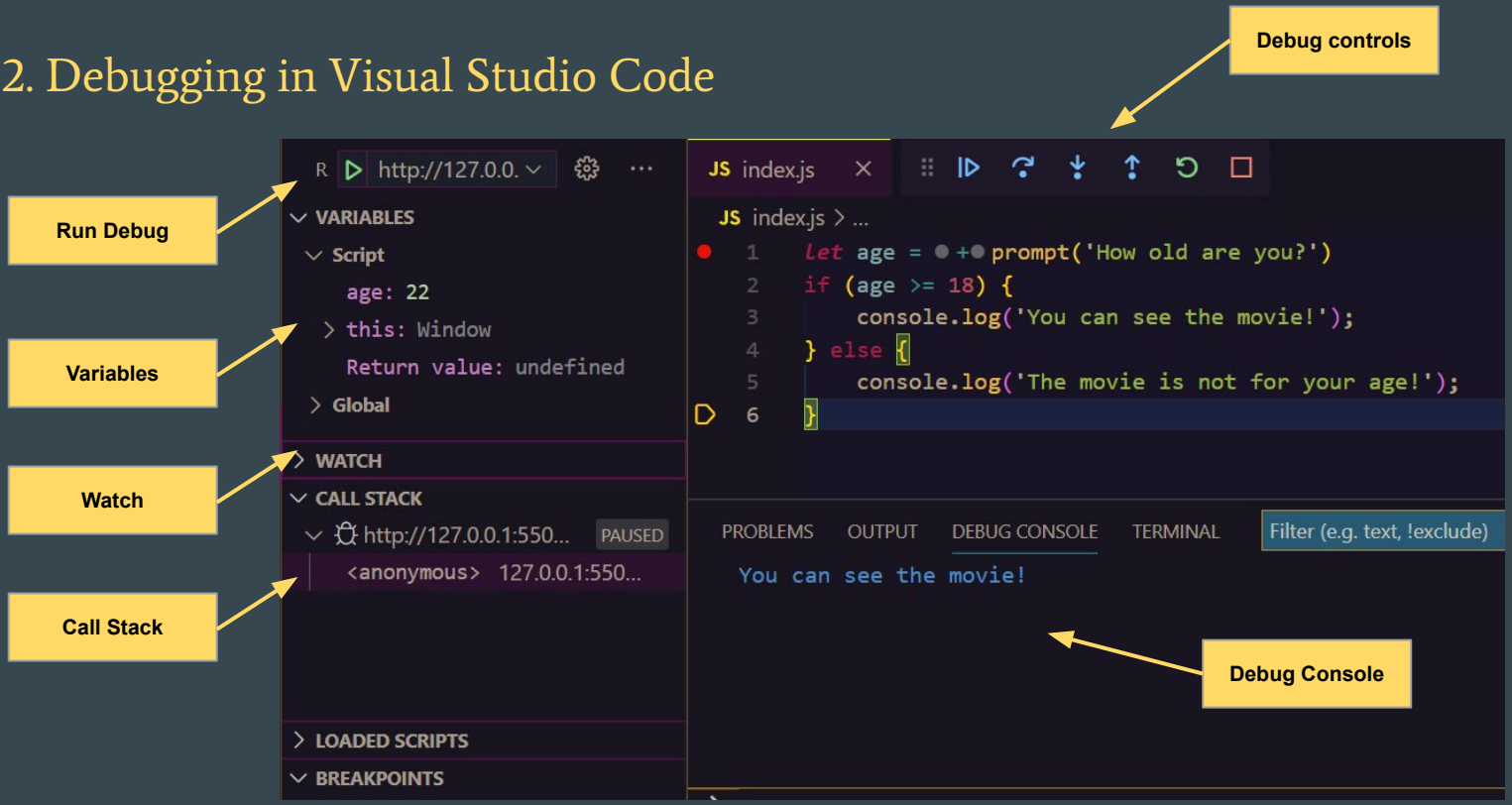
```
"configurations": [
    {
        "name": "Launch Chrome",
        "request": "launch",
        "type": "pwa-chrome",
        "url": "http://localhost:8080",
        "webRoot": "${workspaceFolder}"
    },
    {
        "type": "pwa-msedge",
        "name": "http://127.0.0.1:5500/index.html",
        "request": "launch",
        "url": "http://127.0.0.1:5500/index.html"
    }
]
```

Add config

Add Configuration...

Debugging

2. Debugging in Visual Studio Code



Debugging

2. Debugging in Visual Studio Code