# CMSC335

## Web Application Development with JavaScript


# NodeJS

Department of Computer Science

University of MD, College Park

**Slides material developed by Ilchul Yoon, Nelson Padua-Perez**

# Node.js

- Node.js
  - Asynchronous event driven JavaScript runtime
  - Designed for scalable network applications
  - Can be used to developed full web applications
  - Reference: https://nodejs.org/en/about/
- Relies on JavaScript V8 Engine
  - Written in C++
  - V8 incorporates just-in-time(JIT) compiler
    - » Compiles JavaScript to machine code rather than interpreting it
- Why use it? Performance reasons

# Installation

- For installation: https://nodejs.org/
- **REPL**
  - Read-Eval-Print-Loop (REPL) is a command-line tool for processing **Node. js** expressions
- Using VS Code, open the folder with examples and then a terminal window
- To start REPL type of the terminal "node“
- Let’s define a variable and printed

  let x = 10;

  console.log(x)
- To exit REPL “.exit”
- For help “.help”
- Node Version is displayed when starting node
- You can write JS in the REPL

# Important (run npm i in folder with examples)

- We are removing the **node_modules** folder from the lecture examples we are posting
  - This folder has the “libraries” (modules) the examples rely on
- Before you run any code examples, execute **npm i** (**not** npm init)
- npm i (or npm install) will install in the **node_modules** folder any necessary modules (based on the file package.json)

# WebServer Example

- **Example:** webServer.js
  - Open a terminal in VS Code so we can run **Node**
  - To run the example
    - » Type on the command line node webserver.js
    - » Start **live server** and use the URL displayed after the server has been started
  - module - is a library
  - **require** statement imports the module
  - The **http** module is one of Node's core modules
  - To stop the program, use **CTRL-C** on the terminal

# WebServer Example

- About modules
  - To add modules, type **npm install <module name>** at the command line
  - **Example:** Installing Connect Middleware (CM) module
    - » C:\tempExample>**npm install cm** (or **npm i cm**)
  - Files **package.json** and **package-lock.json** will be created and directory **node_modules**
  - **package.json** - contains metadata about a project, such as a name, version, and dependencies
  - **package-lock.json** - holds information on the dependencies or packages installed for a project, including version numbers
    - » Ensures that each installation results remain identical and reproducible

# WebServer Example

- When you type the name of a program that **Node** cannot find, you will get an error similar to the following (in the example we are trying to execute **webServer.js**, but wrote **webServe.js**)
- It considers the program a module

```
PS Y:\ClassRelated\cmsc335Spring2024\lectures\Week10\NodeJSCode> node .\webServe.js
node:internal/modules/cjs/loader:1146
  throw err;
  ^

Error: Cannot find module 'Y:\ClassRelated\cmsc335Spring2024\lectures\Week10\NodeJSCode\webServe.js'
    at Module._resolveFilename (node:internal/modules/cjs/loader:1143:15)
    at Module._load (node:internal/modules/cjs/loader:984:27)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:135:12)
    at node:internal/main/run_main_module:28:49 {
  code: 'MODULE_NOT_FOUND',
  requireStack: []
}

Node.js v20.12.0
```

# Creating a Project in Node

- A Node project has a file called **package.json** providing information such as project's name, author, version, and dependencies (which modules your project relies on)
- You can create this file yourself, or you can rely on **npm init**
- **Example**: Let's create a project
  - Create a folder named example
  - In the folder, execute **npm init**
  - Let's examine the package.json file

# Asynchronous Programming

- **fs** module - File System module
- Node supports both synchronous and asynchronous versions of most File System Functions
- **Synchronous Programming**
  - Code that performs one task after another, waiting for one to complete before starting another
  - **Example:** readFileSync.js
- **Asynchronous Programming**
  - We don't wait for the code to finish
  - Callback will take care of processing once an event has been triggered
  - **Example:** readFileAsync.js

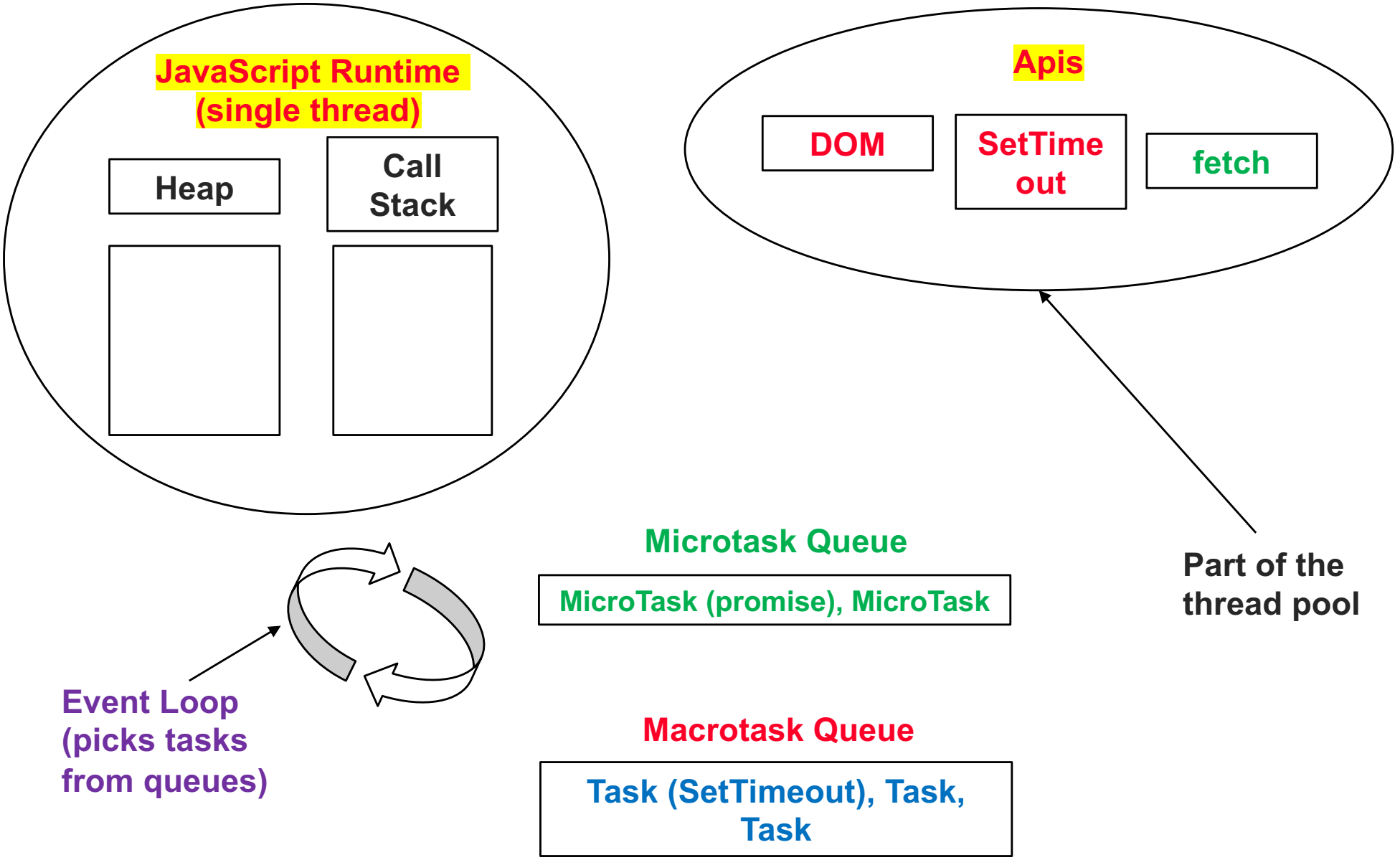
# Global Objects

- **global** - similar to the browser global object (**window**)
  - Provides access to all globally available **Node** objects and functions
  - **Example:** In node REPL, execute **console.log(global)**
- **process**
  - Provides information about the **Node** environment and the runtime environment
  - Standard input/output occurs through **process**
  - **process.stdin** - stream for stdin
  - **process.stdout** - stream for stdout
  - **process.stderr** - stream for stderr
  - **Example:** webServerControl.js

# Global Objects

- **Example:** translator.js
  - Translates from English to Spanish
  - Relies on command-line arguments
  - Using process.stdout.write() as console.log automatically adds a newline
  - To run the example
    - » node translator.js Spanish
- **Example:** imageServer.js
  - URL to try: http://localhost:5000/?imageName=umcp
  - URL to try: http://localhost:5000/?imageName=terps
    - » This one is invalid as there is no **terps** image

# Processing in JavaScript (Event Loop)



# Event Loop

- JavaScript is single-threaded and has a **Call Stack** that supports the execution of code line by line
- Although single-threaded, the JavaScript environment is supported by Web APIs (e.g., DOM manipulation, setTimeOut, fetch, Geolocation, etc.), each of which can be visualized as a separate thread
- Web APIs place tasks in two data structures: **MacroTask** queue and the **Microtask** queue
  - **Example:** setTimeOut places tasks in the **MacroTask** queue and fetch in the **Microtask** queue
- **Event Loop** - Responsible for scheduling/managing **tasks** in the queues. The **Event Loop** decides which one gets executed next in the **Call Stack**

# Event Loop

- **Event Loop**
  - When a new iteration of the event loop begins, a macrotask is selected from the Macrotask queue
    - » Macrotasks added after the iteration begins will not run until the next iteration
  - Each time a macrotask exits and the **Call Stack** is empty, each microtask in the Microtask queue will be executed. The execution of microtasks continues until the queue is empty (even if new ones arrive). That is, microtasks can enqueue new microtasks, which will be executed before the next macrotask begins and before the end of the current event loop iteration
  - **References:**
    - » Web Stories (https://www.youtube.com/@web_stories) 's video: https://www.youtube.com/watch?v=3Jma9VYvSz8
    - » https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide/In_depth

# Event Loop

- Event Loop Video: https://www.youtube.com/watch?v=8aGhZQkoFbQ
  by Philip Roberts (starting at timestamp 2:14)
- Animation Tool: https://goo.gl/iJRGvT

# About setTimeout

- When setTimeout is called with a value of 0, it does not mean the code will be executed immediately.  It means the API will place the callback immediately in the **Macrotask** queue

# Events

- **For timers**
  - setTimeout() - executes callback after delay time (milliseconds)
  - setInterval() - callback is executed at periodic intervals
  - clearInterval() - clears timer
  - **Example:** timer.js

# References

- Learning Node, 2nd Edition
  - By: Shelley Powers
  - SBN-13: 978-1-4919-4312-0