UNIT-II:

Node.JS Basics: Primitive Types, Object Literal, Functions, Buffer, Access Global Scope.

Node.JS Modules: Module, Module Types: Core Modules, Local Modules, Third Party Modules, Module Exports. Using Modules in a Node.js File, Using the Built in HTTP, URL, Query String Module, Creating a Custom Module.

Node Package Manager: NPM, Installing Packages Locally, Adding dependency in package.json, Installing packages globally, Updating packages.

PRAMOD NAIK

Variable: (Name given to the memory location):

A **variable** is a **named location in memory** that stores data that can be modified during program execution. A variable is used to represent data that the program can manipulate.

Data Types:

Data Types define the **type of data** a variable can hold. They are crucial for memory management, as they specify the **amount of memory to be allocated** and the operations that can be performed on the data.

Data types are broadly classified into **primitive** (**basic**) and **non-primitive** (**derived or user-defined**) data types based on their complexity and usage.

PRAMOD NAIK

Primitive Types:

Primitive data types are the **built-in** data types provided by JavaScript. They represent **single values** and are **not mutable (Immutable)**. These types are immutable, meaning their values **cannot be changed directly**.

JavaScript supports the following primitive data types:

1. Number
2. BigInt
3. String
4. Boolean
5. Undefined
6. Null
7. Symbol

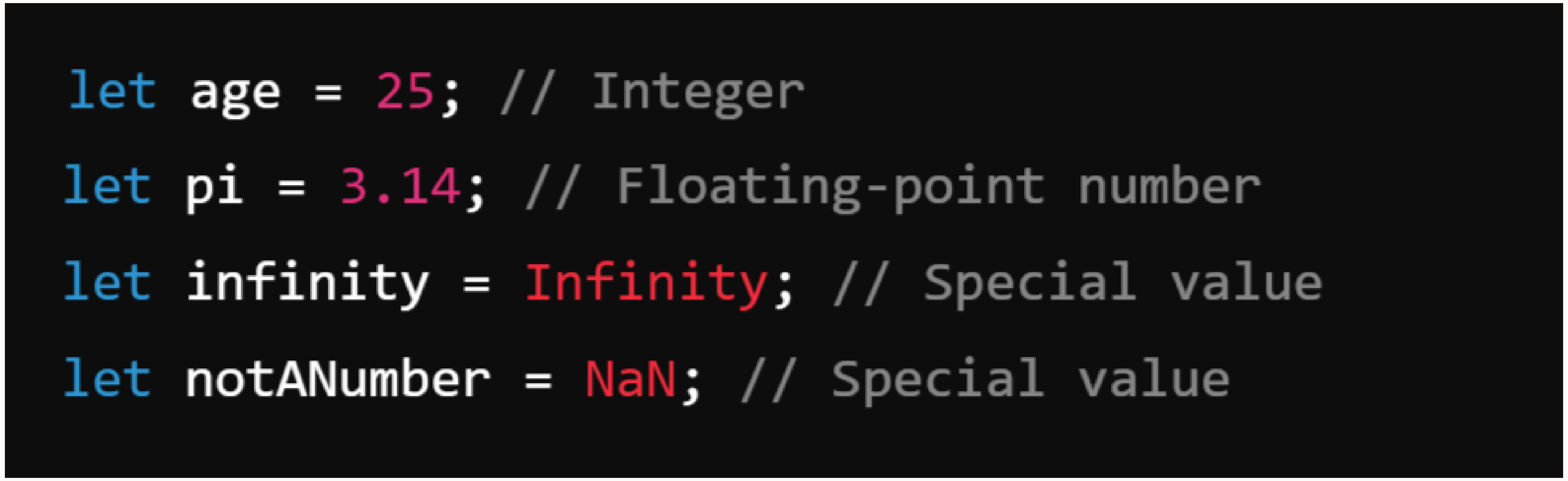
Note:

When we say **primitive types are immutable**, it means their **values cannot be altered** once created. If you perform an operation on a primitive type, a new value is created, and the original value remains unchanged.

PRAMOD NAIK

1. **Number:** Represents both **integer** and **floating-point** numbers.

Example:



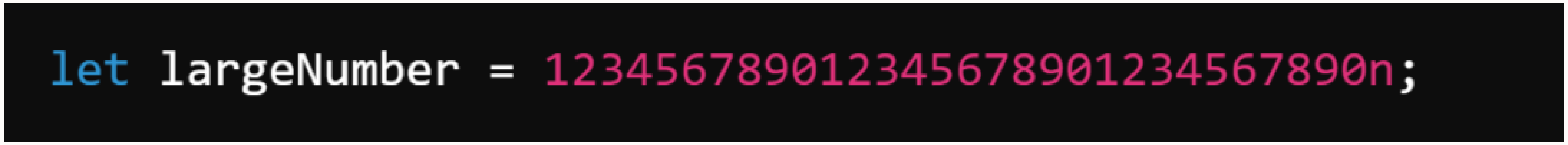
```javascript
let age = 25; // Integer
let pi = 3.14; // Floating-point number
let infinity = Infinity; // Special value
let notANumber = NaN; // Special value
```

PRAMOD NAIK

2. BigInt:

- Used to represent integers that are larger than **Number.MAX_SAFE_INTEGER (2^53 - 1)**.
- BigInts are created by appending **n** to the end of an integer.

Examples:



```
let largeNumber = 123456789012345678901234567890n;
```

PRAMOD NAIK

3. String:

- Represents a sequence of characters (text).
- Strings are immutable.

Example-1:

```javascript
let name = "Alice";
let greeting = 'Hello, World!';
```

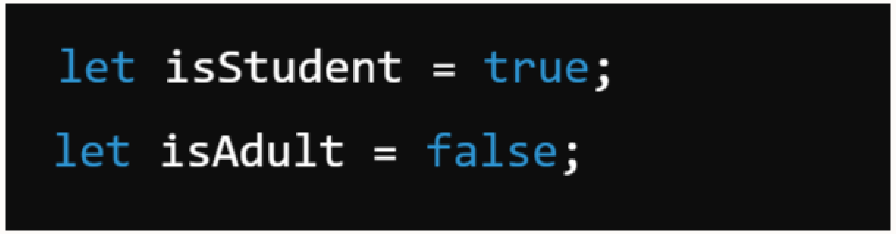
Example-2:

```javascript
let greeting = "Hello";
// Trying to change a character directly
greeting[0] = "h";
console.log(greeting); // Output: "Hello" (unchanged)
```

PRAMOD NAIK

4. Boolean: Represents logical values: true or false.

Examples:

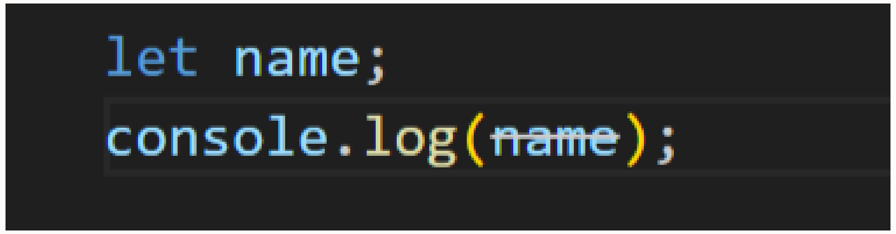


```
let isStudent = true;
let isAdult = false;
```

5. Undefined

- Indicates that a variable has been **declared** but has **not been assigned a value**.

Example:

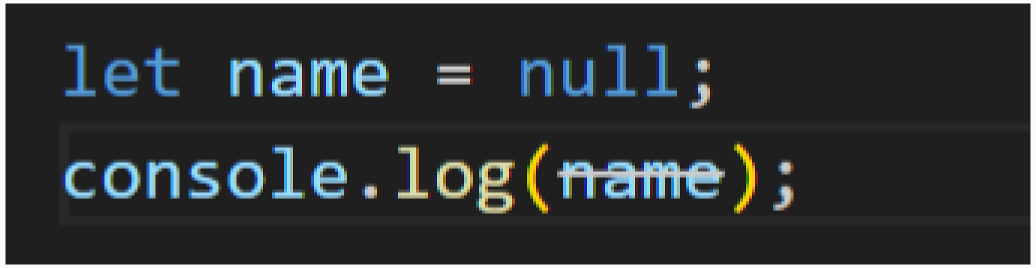


```
let name;
console.log(~~name~~);
```

6. Null:

- Represents the **intentional absence** of any value.
- This data type can hold only one possible value that is **null**.

Example:



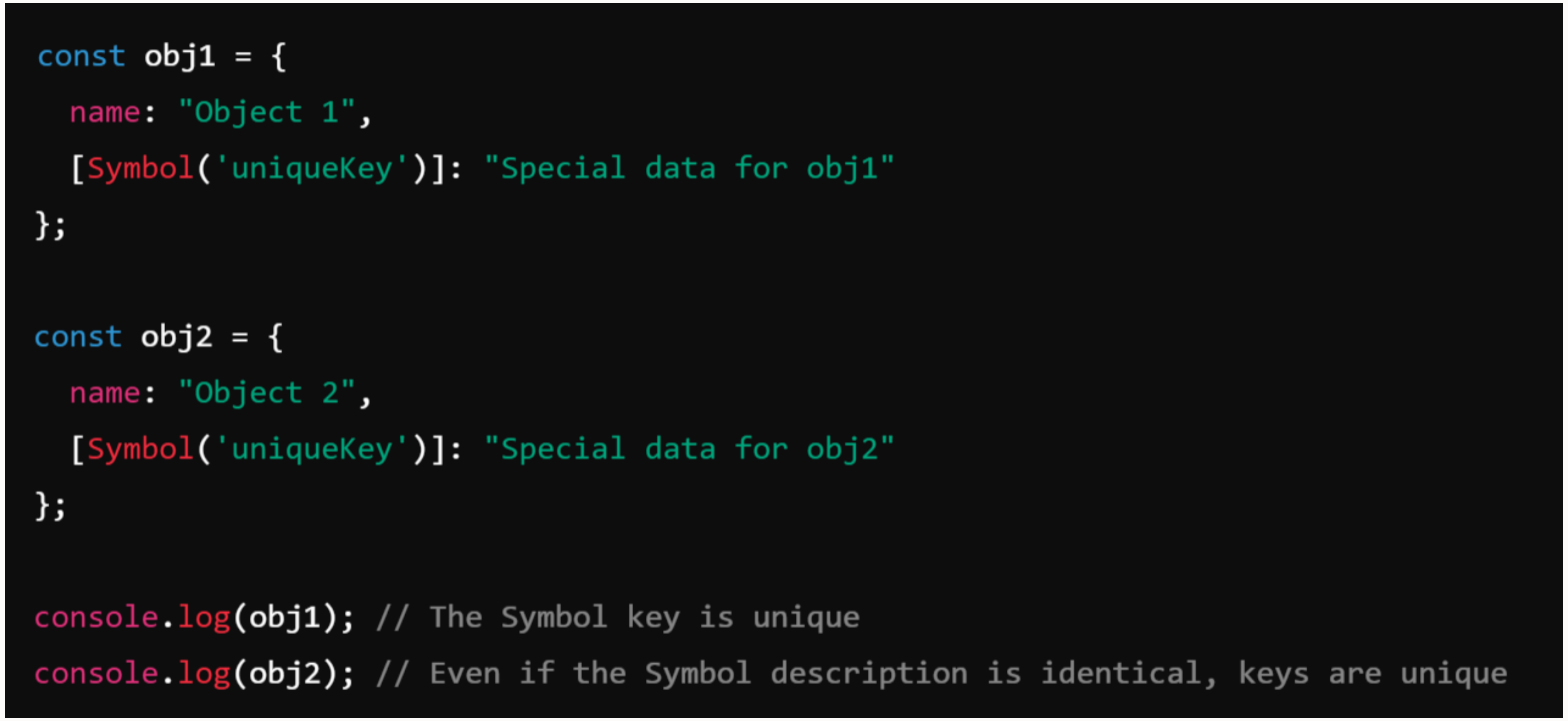
```
let name = null;
console.log(name);
```

PRAMOD NAIK

7. Symbol:

- Introduced in ES6, represents a **unique** and **immutable value**.
- Symbol data type **is used to create objects** which will always be unique. these objects can be created using Symbol constructor.
- Used primarily as unique object property keys.

Example: Here, even though both objects use Symbol('uniqueKey') as a key, the Symbol ensures that the **property keys** are different and don't conflict.



```
const obj1 = {
  name: "Object 1",
  [Symbol('uniqueKey')]: "Special data for obj1"
};

const obj2 = {
  name: "Object 2",
  [Symbol('uniqueKey')]: "Special data for obj2"
};

console.log(obj1); // The Symbol key is unique
console.log(obj2); // Even if the Symbol description is identical, keys are unique
```

PRAMOD NAIK

Non-primitive Data Types

Non-primitive data types, also known as **reference types**, are **objects** and **derived data types**. They can store collections of **values** or **more complex entities**. They differ from primitive types in that they are **mutable** and **referenced by memory addresses**, not stored directly by value.

Types of Non-Primitive Data Types

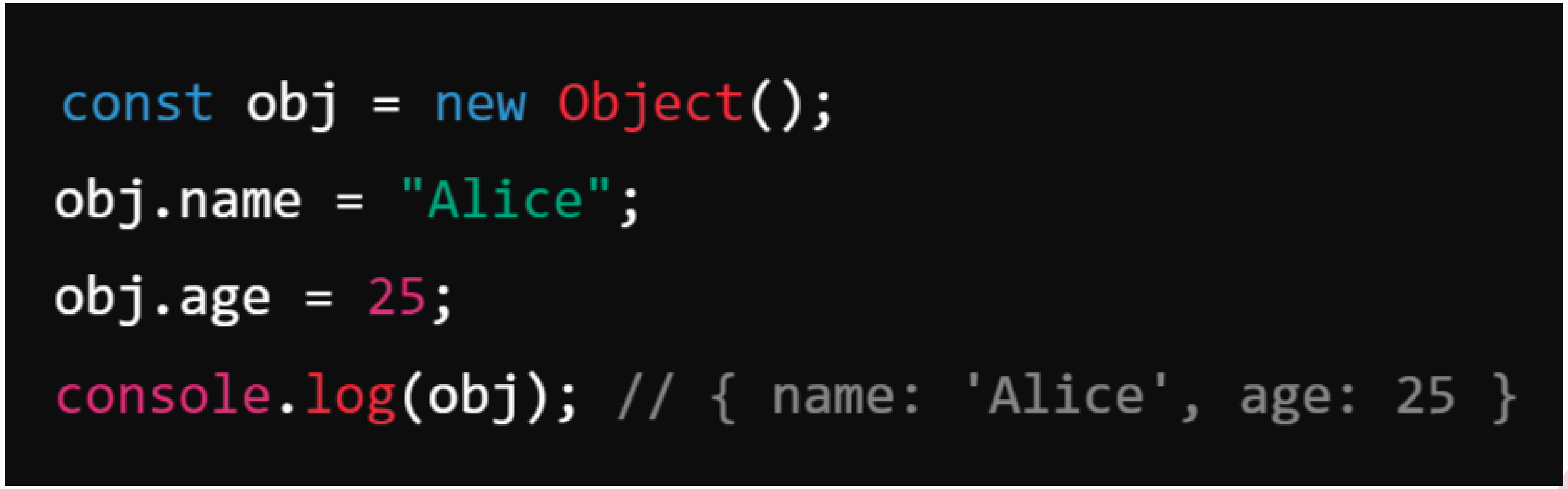
1. **Object**
2. **Array**
3. **Function**

Object or **Object Literals:**

- An object in JavaScript is a **data structure** used to store collections of **key-value pairs.**
- Objects can be created in various ways, such as using **constructors**, the **Object class**, or **Object.create**.

Examples:

1. **Using the Object constructor:**



```
const obj = new Object();
obj.name = "Alice";
obj.age = 25;
console.log(obj); // { name: 'Alice', age: 25 }
```

PRAMOD NAIK

2. Using Object.create:

```javascript
const prototypeObj = { greeting: "Hello" };
const obj = Object.create(prototypeObj);
obj.name = "Alice";
console.log(obj); // { name: 'Alice' }
console.log(obj.greeting); // "Hello" (inherited from prototype)
```

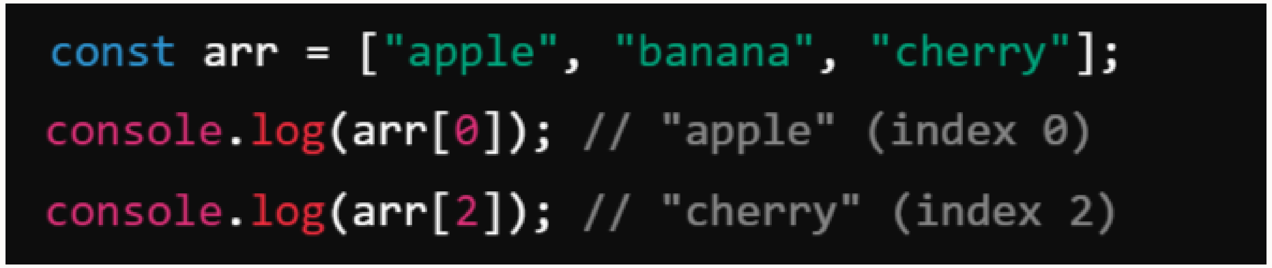
3. Object Literal:

```javascript
const obj = {
  name: "Alice",
  age: 25
};
console.log(obj); // { name: 'Alice', age: 25 }
```

2. Array in JavaScript

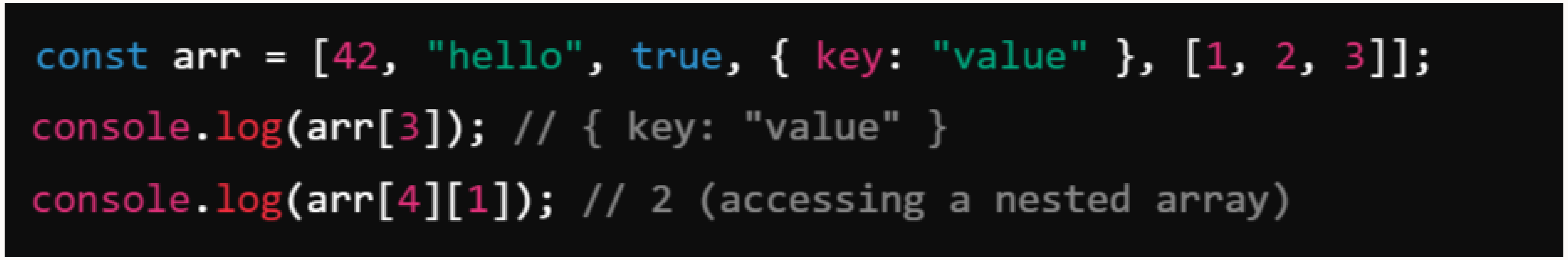
- An **array** in JavaScript is a **special object** used to store **multiple values in a single variable**. Arrays are **ordered collections** of elements, where each element is **indexed** and can be accessed using its index number. JavaScript arrays are flexible, allowing you to store a **mix of data types**.

Example-1:



```
const arr = ["apple", "banana", "cherry"];
console.log(arr[0]); // "apple" (index 0)
console.log(arr[2]); // "cherry" (index 2)
```

Example-2:



```
const arr = [42, "hello", true, { key: "value" }, [1, 2, 3]];
console.log(arr[3]); // { key: "value" }
console.log(arr[4][1]); // 2 (accessing a nested array)
```

Functions:

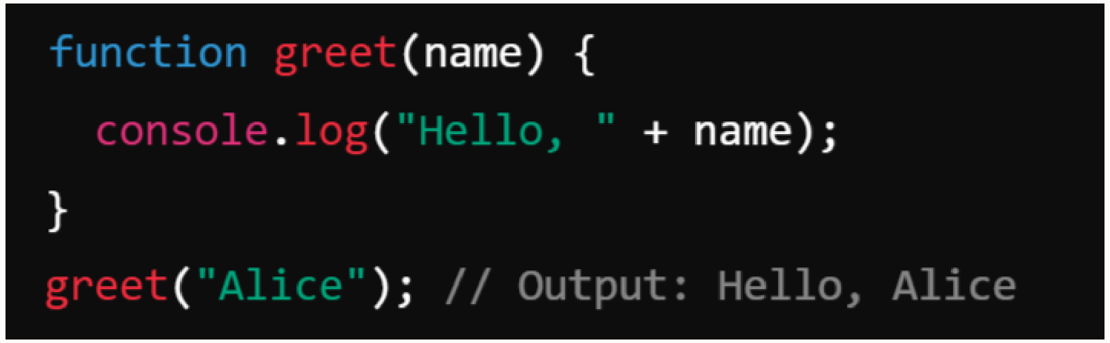
Functions in JavaScript (and by extension, in Node.js) are **blocks of reusable code** designed to **perform** a **specific task**. They are fundamental in both client-side JavaScript (browsers) and server-side JavaScript (Node.js). Functions allow you to structure your code efficiently, manage logic, and create reusable components.

Definition: A Function is a **self-contained programming segment** which is used to perform **specific** and **well-defined task**.

Defining Functions in JavaScript / Node.js:

1. Using function Keyword:

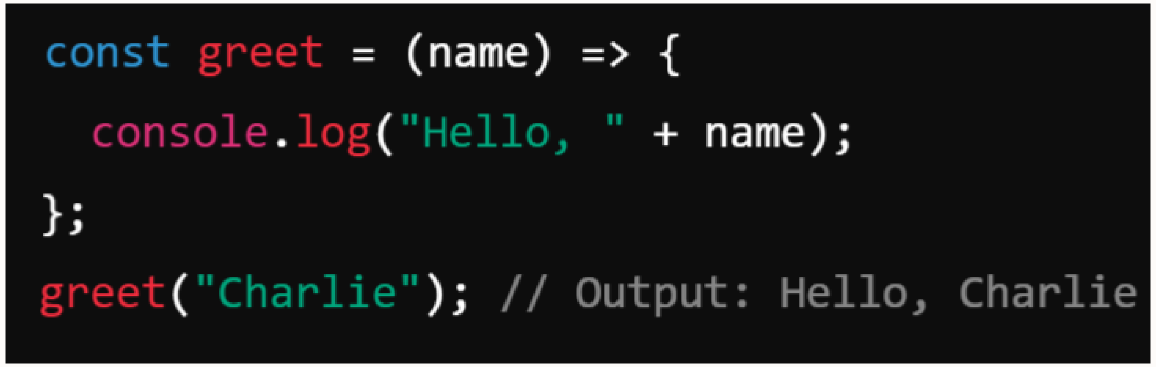
Functions in Node.js are similar to functions in JavaScript and can be defined using the **function** keyword:



```javascript
function greet(name) {
  console.log("Hello, " + name);
}
greet("Alice"); // Output: Hello, Alice
```

2. Arrow Functions (ES6):

Arrow functions provide a **shorter syntax** and do not have their own this context, which can be useful in certain cases (e.g., with callbacks).



```javascript
const greet = (name) => {
  console.log("Hello, " + name);
};
greet("Charlie"); // Output: Hello, Charlie
```

PRAMOD NAIK

3. Function with Default Parameters (ES6):

- Functions can have default values for parameters if no value is passed.

```
function greet(name = "Guest") {
  console.log("Hello, " + name);
}
greet();        // Output: Hello, Guest
greet("Alice"); // Output: Hello, Alice
```

4. Function with Rest Parameters (ES6):

The **...** syntax allows you to collect all remaining arguments into an array.

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}
console.log(sum(1, 2, 3)); // Output: 6
```

Buffer Class:

In Node.js, a Buffer is a **global class** that is specifically designed for handling **binary data**. It provides a way to work with **raw binary data directly**, which is **not possible with JavaScript strings** because strings are meant for **textual data** and use Unicode encoding.

Buffers are particularly important in Node.js because it is **used for server-side operations** that often involve dealing with **streams of binary data**, such as **file reading**, **HTTP requests** and **responses**, and **TCP streams**.

Note:

Buffers in Node.js are widely used to **handle raw binary data efficiently**. They are crucial for scenarios involving data streams, file handling, and network communication, as **JavaScript traditionally lacks a mechanism to deal with binary data directly**.

PRAMOD NAIK

Key Characteristics of Buffers:

1. Raw Binary Data:

A buffer is a **sequence of bytes**, and each byte represents a value between **0** and **255**.

2. Fixed Size:

Once a buffer is created, **its size cannot be changed**.

3. Global in Node.js:

The Buffer class is available **globally** in Node.js and **does not need to be imported**.

4. Efficient Memory Usage:

Buffers allow **efficient handling of binary data** without the need to convert it to string or other data types.

PRAMOD NAIK

Creating a Buffer:

1. Allocating a Buffer:

- Create a buffer of a specified size. The **default representation of a Buffer** in Node.js uses **hexadecimal format**, where each byte (8 bits) is represented as a two-digit hexadecimal number. So, it shows like <Buffer 00 00 00 00 00 00 00 00 00 00>

```javascript
const buf = Buffer.alloc(10); // A buffer with 10 bytes, initialized to 0
console.log(buf); // <Buffer 00 00 00 00 00 00 00 00 00 00>
```

2. From a String:

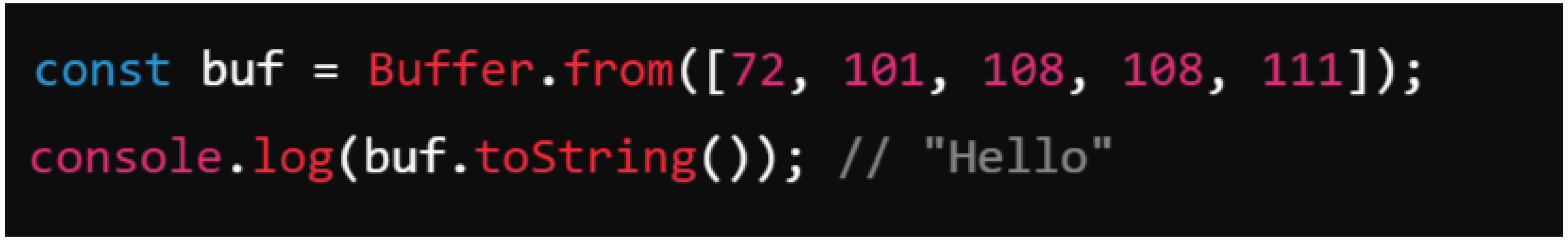
- Buffers are used to handle binary data, and in this case, the **string is converted into its binary representation** (a sequence of bytes).

```javascript
const buf = Buffer.from("Hello, Node.js!");
console.log(buf); // <Buffer 48 65 6c 6c 6f 2c 20 4e 6f 64 65 2e 6a 73 21>
console.log(buf.toString()); // "Hello, Node.js!"
```

PRAMOD NAIK

3. From an Array:

- Create a buffer from an array of bytes. The toString() method will convert the ASCII into it’s string representation.



```
const buf = Buffer.from([72, 101, 108, 108, 111]);
console.log(buf.toString()); // "Hello"
```

Note:

Buffers are commonly used in Node.js to **work with binary data**, such as **reading from** or **writing to files**, or **working with network protocols**.

PRAMOD NAIK

Access Global Scope:

In JavaScript, **variables declared outside** of any **function** or **block** have global scope.

The **global scope** is the **outermost scope** in any JavaScript environment. It contains **variables**, **functions**, and **objects** that are accessible **throughout the entire application** without explicitly importing or defining them in every module.

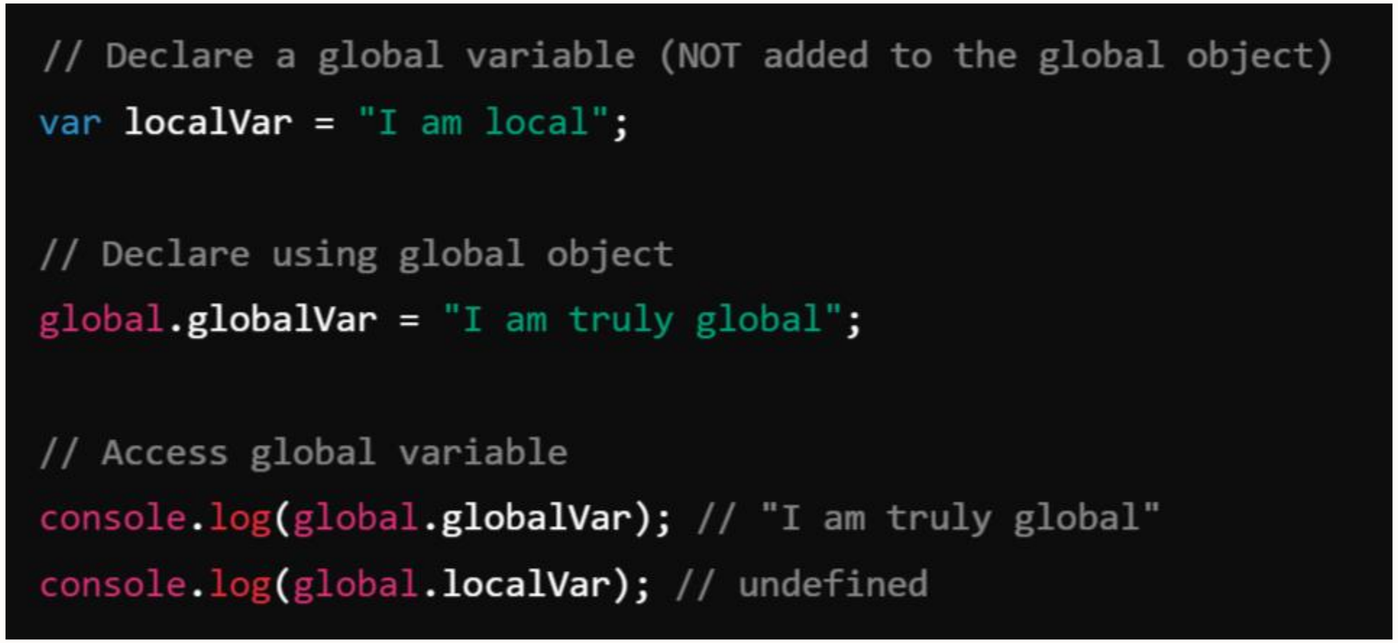
In Node.js, the global scope is represented by the **global object.** However, variables declared in Node.js modules do not automatically become global properties.

In Node.js, Global Scope refers to the **scope that is accessible from anywhere** in **the application**, regardless of where the code is executed. In the global scope, certain **objects**, **variables**, and **functions** are available without needing to **explicitly require** or **import** them.

PRAMOD NAIK

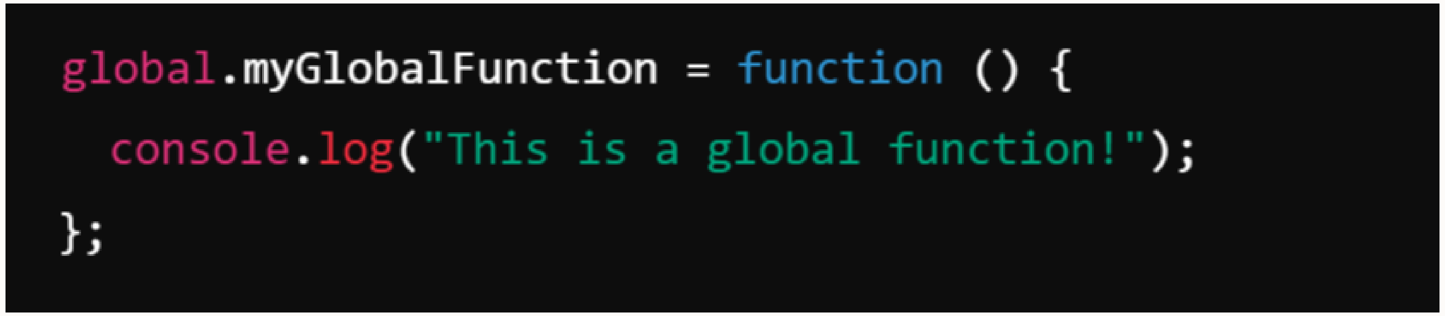
Example : Global Variable and Function:

File Name: test_global.js



```
// Declare a global variable (NOT added to the global object)
var localVar = "I am local";

// Declare using global object
global.globalVar = "I am truly global";

// Access global variable
console.log(global.globalVar); // "I am truly global"
console.log(global.localVar); // undefined
```

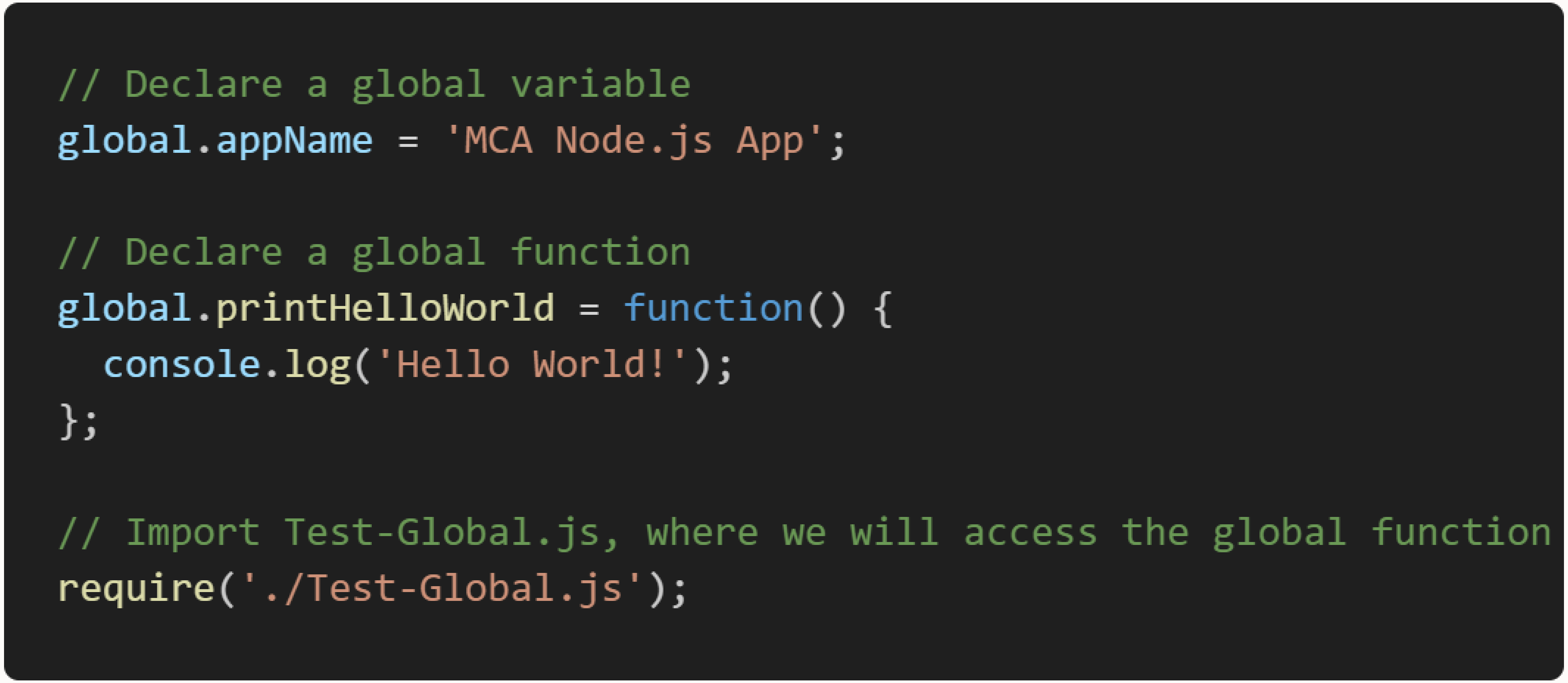
Global Function:



```
global.myGlobalFunction = function () {
  console.log("This is a global function!");
};
```

PRAMOD NAIK

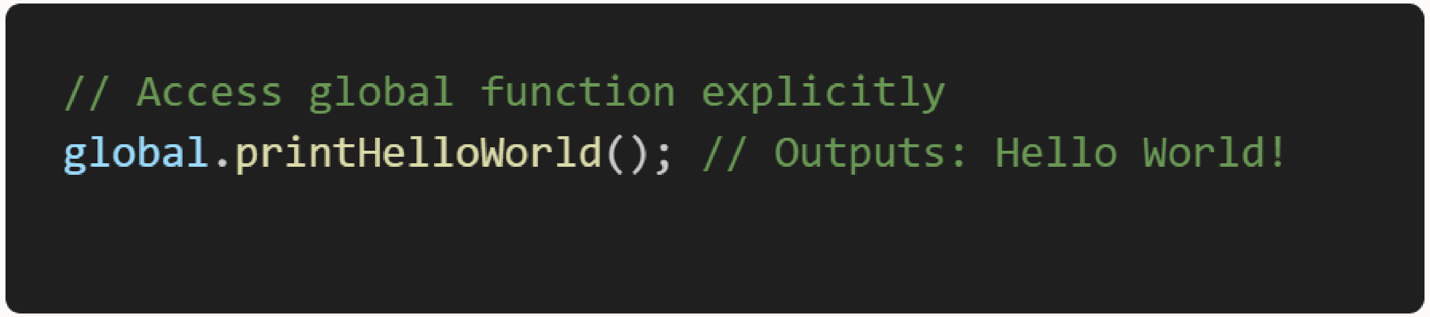
Example-2: app.js



```javascript
// Declare a global variable
global.appName = 'MCA Node.js App';

// Declare a global function
global.printHelloWorld = function() {
  console.log('Hello World!');
};

// Import Test-Global.js, where we will access the global function
require('./Test-Global.js');
```

PRAMOD NAIK

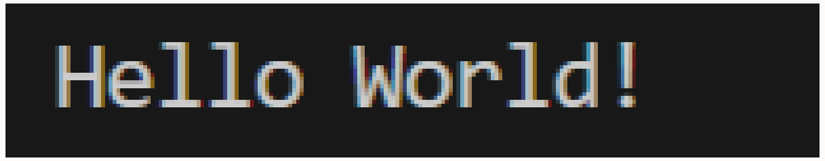
Example-2: Test-Global.js



```
// Access global function explicitly
global.printHelloWorld(); // Outputs: Hello World!
```

RUN: node app.js

Output:



```
Hello World!
```

PRAMOD NAIK

What does global.printHelloWorld mean?

In Node.js, the global object is a global context that is shared across all files executed within the **same process**. This means that when you define a function or variable on **global**, it becomes accessible globally across your entire Node.js application as long as the **files are executed within the same process**.

But each file/module has its own **isolated scope**. So, when you define printHelloWorld on global, it becomes available globally across all files in the same process, but only if you've already **run** or **required** the file where you defined it.

PRAMOD NAIK

- **Key Characteristics of Global Scope in Node.js:**

1. **Global Object:** In the global scope, Node.js provides a **global object** (similar to **window** in browsers) that **holds global variables** and **functions**.
2. **Global Variables:** Any **variable declared outside a function** (i.e., at the top level) in a Node.js module will be in the **module scope**, but not in the global scope. However, you can explicitly create global variables using the **global object**.
3. **Global Functions:** Certain built-in functions like **setTimeout**(), **setInterval**(), and **console.log**() are accessible globally **without needing to import them**.
4. **Module Scope:** Each file/module in Node.js **has its own scope**. Even though you can create global variables using the global object, typically, it's recommended to avoid polluting the global namespace to prevent conflicts.

Node.js Modules

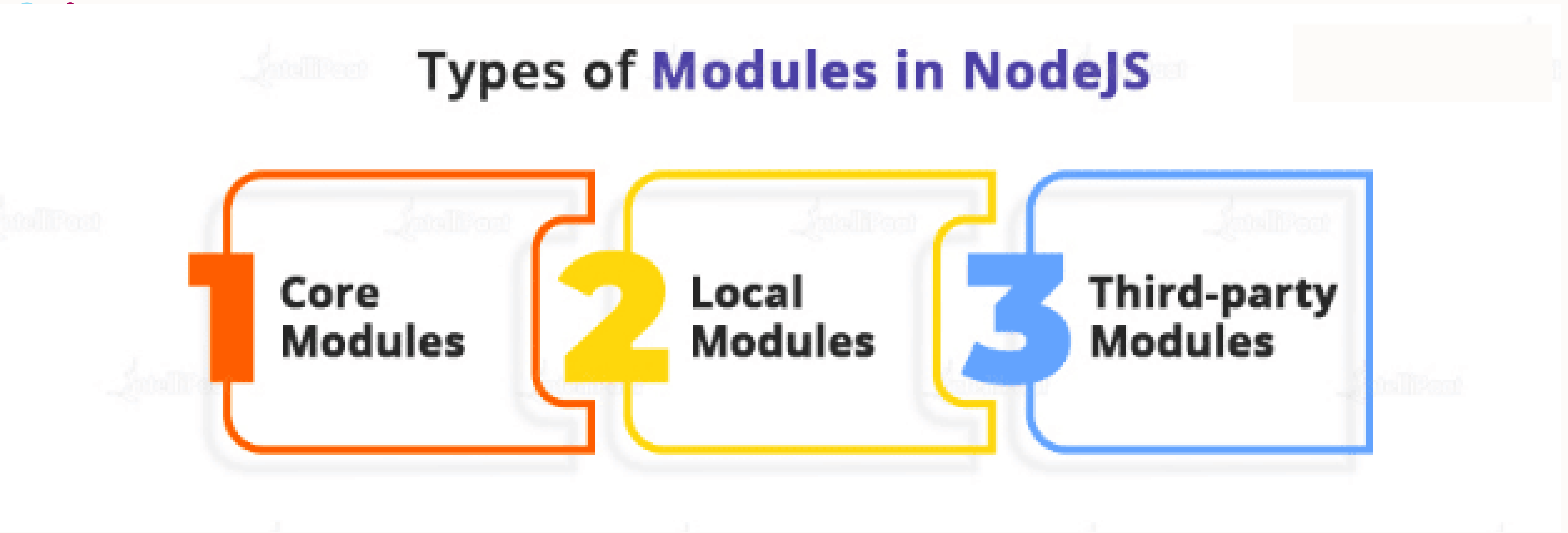
PRAMOD NAIK

Module:

In **Node.js**, a **module** is a **reusable piece of code** that can be **exported** from one file and **imported** into another file. Modules are a fundamental aspect of Node.js that allow developers to **organize code** into **smaller**, **maintainable**, and **reusable chunks**.

- A module in Node.js is a reusable block of code that encapsulates **related functionality**.
- Each module in Node.js has its own scope, so the **variables** and **functions** defined in a module are **private to that module by default**.
- A module is a **single JavaScript file** or a **collection of related files** that expose specific functionality through **exports** or **module.exports**.
- In Node.js, **every JavaScript file is treated as a module**.
- **Example:** A file containing utility functions (**math.js**) or configurations.

PRAMOD NAIK



Types of Modules in NodeJS

1. Core Modules
2. Local Modules
3. Third-party Modules

Types of Modules in Node.js:

1. Core Modules: These are modules that are **included with Node.js** and provide basic functionality. Examples include fs (file system), http (HTTP server), os (operating system), etc.

2. Local Modules: These are modules that **you create** in your Node.js application. You can create a local module by defining a JavaScript file and using **module.exports** to export functions or objects from that file.

3. Third-party Modules: These are modules created by **third-party developers** and are **not included with Node.js**. You can install third-party modules using **npm** (Node Package Manager) and then use **require()** to include them in your application.

PRAMOD NAIK

Core Modules :

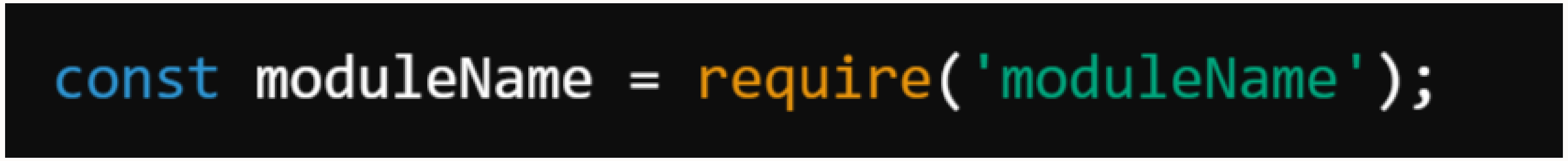
In Node.js, **Core Modules** are the **built-in modules** that come **pre-installed with Node.js**. These modules provide **essential functionalities** for building applications and **eliminate the need for additional installation**. They are designed to perform various tasks such as **file handling**, **HTTP operations**, **cryptography**, **stream handling**, and more.

Key Features of Core Modules

1. **No Installation Needed**: Core modules are **part of the Node.js runtime** and can be used directly without installing any additional packages.
2. **High Performance**: These modules are written in **C++** for better performance and are optimized for Node.js.
3. **Global Availability**: Core modules can be imported and used in any Node.js application.
4. **Common Core Modules:** fs(File System), http, url, path, os

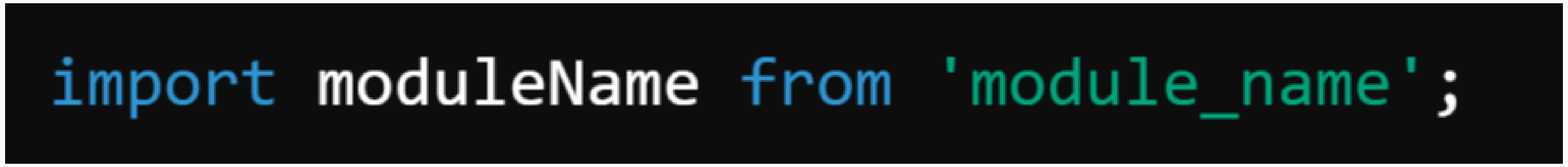
Using Core Modules:

To use a core module in Node.js, you import it using the **require() function** or **import** Keyword:



```
const moduleName = require('moduleName');
```

OR

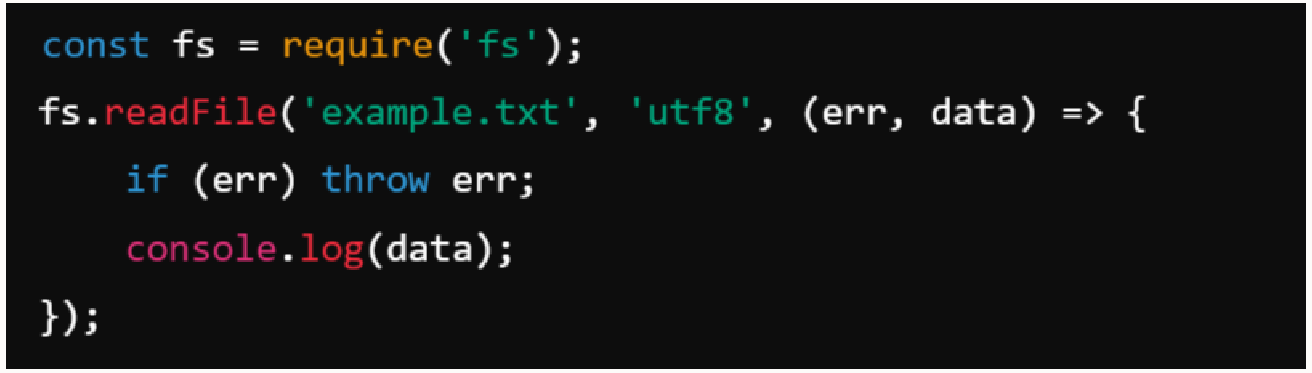


```
import moduleName from 'module_name';
```

PRAMOD NAIK

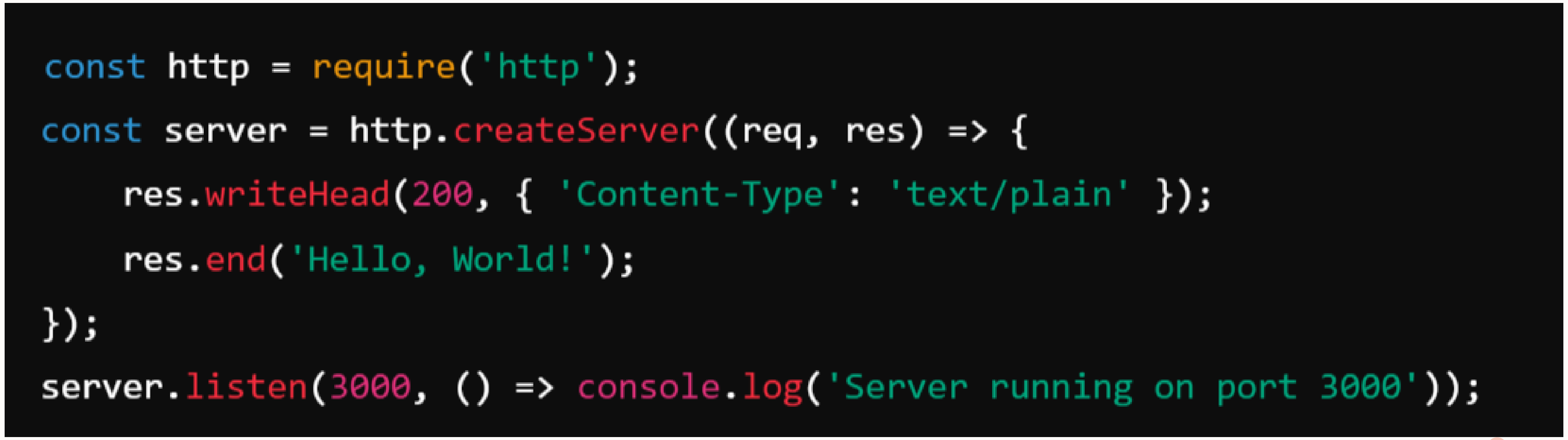
Example:

1. fs (File System): Used for **handling file operations** such as **reading**, **writing**, and **deleting** files.



```
const fs = require('fs');
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
});
```

2. http: Used to create and manage **HTTP servers** and handle **HTTP requests** and **responses**.



```
const http = require('http');
const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello, World!');
});
server.listen(3000, () => console.log('Server running on port 3000'));
```

PRAMOD NAIK

- **2. Local Modules:**

  Local Modules in Node.js are **custom modules created by developers** to **encapsulate specific functionality** within their applications. Unlike **core modules** (built into Node.js) or **third-party modules** (installed via npm), **local modules are part of your project**, often created to organize and reuse code.

  Characteristics of Local Modules:

  - **Custom-built:** Designed to address specific requirements of the application.
  - **Reusable:** Can be **imported** and **used** across **different parts of the project**.
  - **File-based:** Each local module **corresponds to a JavaScript file** or a **set of files** in your project

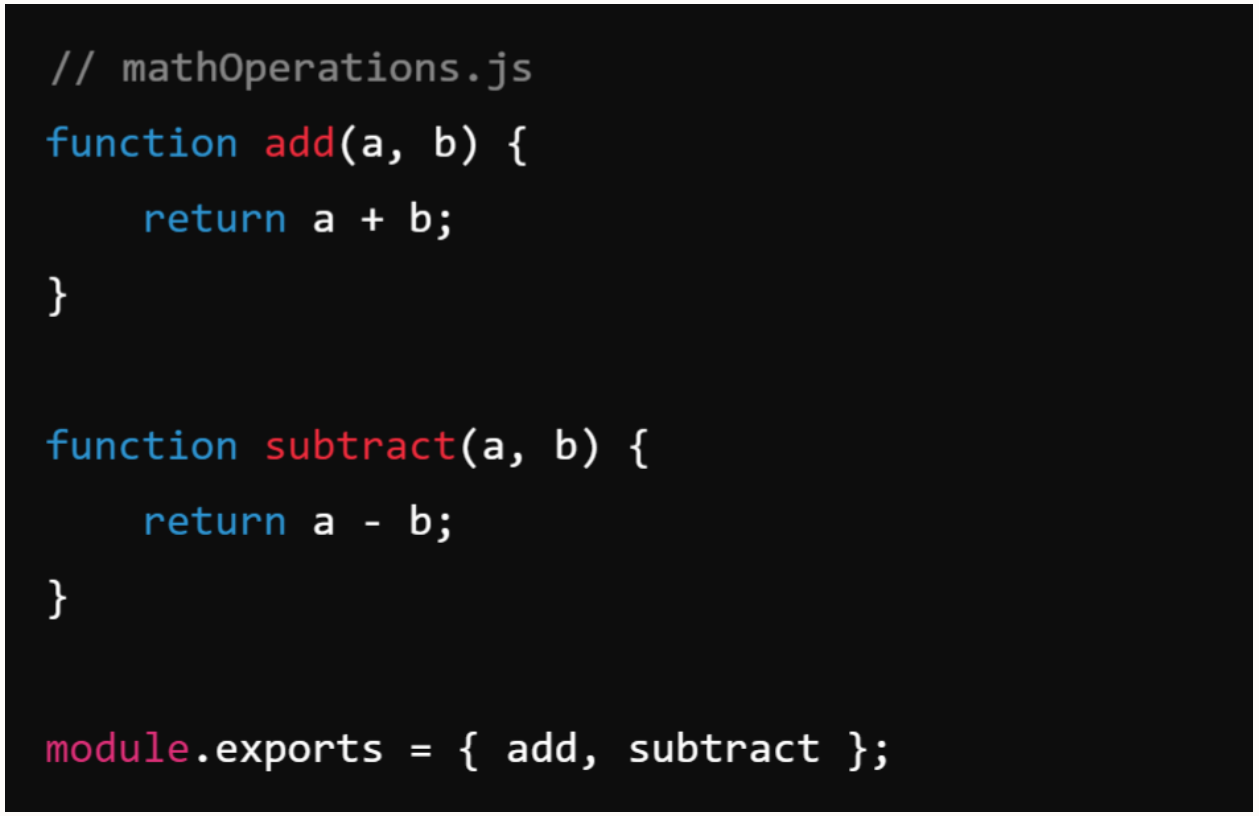
PRAMOD NAIK

Creating a Local Module:

- A local module is **just a JavaScript file** where you define your functionality and **export** it using **module.exports** or **export**.

Example: Creating and Using a Local Module

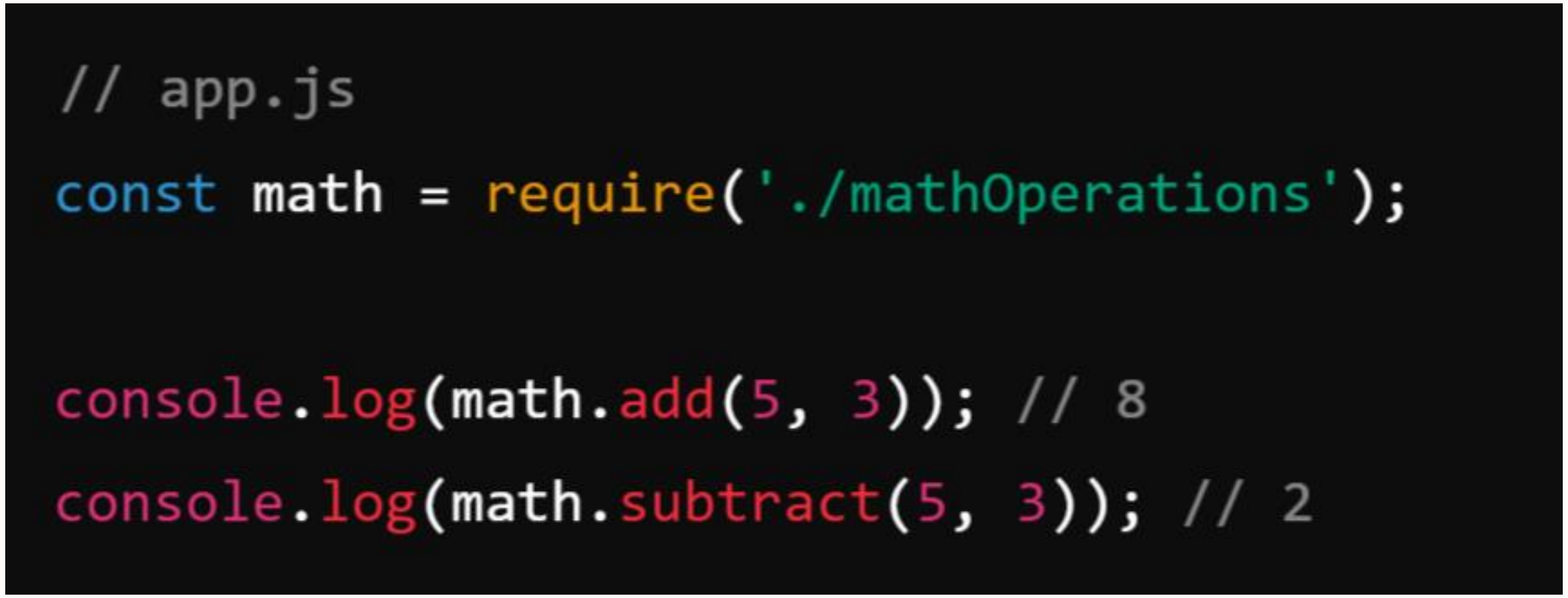
1. Create a File for the Module Let's create a module named **mathOperations.js**:



```
// mathOperations.js
function add(a, b) {
    return a + b;
}

function subtract(a, b) {
    return a - b;
}

module.exports = { add, subtract };
```

PRAMOD NAIK

2. **Import** and **Use** the Module Import the module into another file using **require():**



```
// app.js
const math = require('./mathOperations');

console.log(math.add(5, 3)); // 8
console.log(math.subtract(5, 3)); // 2
```

PRAMOD NAIK

3. Third-party Modules:

Third-party modules in Node.js are **external packages created by the developer community** and **published to** the **npm (Node Package Manager) repository**. These modules extend the functionality of Node.js, allowing developers to use pre-built solutions **instead of writing everything from scratch**.

Characteristics of Third-party Modules

1. **Community-driven**: Developed and maintained by the **Node.js community** or **organizations**.
2. **Installable via npm**: Available in the **npm registry** and installed using the npm command-line tool.
3. **Reusable**: Can be used across multiple projects.
4. **Versioned**: Versions are maintained to ensure **stability** and **compatibility**.
5. **Examples of Popular Third-party Modules: Express, Mongoose, Axios,**

PRAMOD NAIK

How to Use Third-party Modules:

1. Install the Module: Use **npm install** to add the module to your project:

```
npm install <module-name>
```

2. Require the Module: Import the module in your JavaScript file using **require():**

```
const moduleName = require('module-name');
```

OR

```
import moduleName from 'module_name';
```

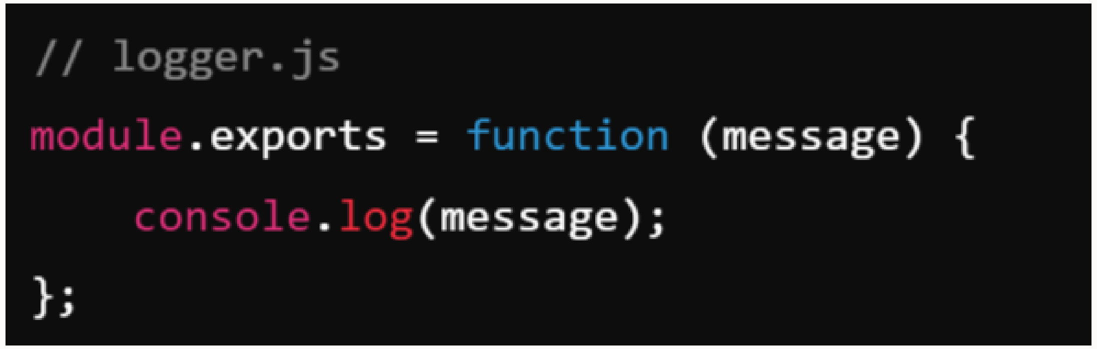
3. Use the Module: Call the **functions** or use the classes provided by the module in your application.

PRAMOD NAIK

Diff ways of Importing and Exporting Node.js Modules:

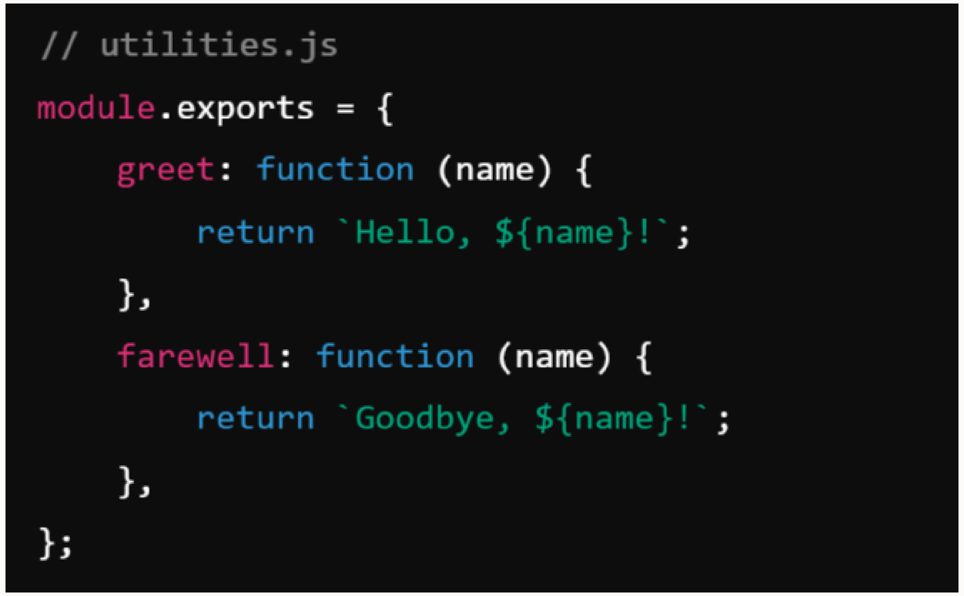
1. Standard Way (CommonJS Syntax): This is the **default** module system in Node.js before ES modules were supported.

- **Single Export:**



```
// logger.js
module.exports = function (message) {
    console.log(message);
};
```

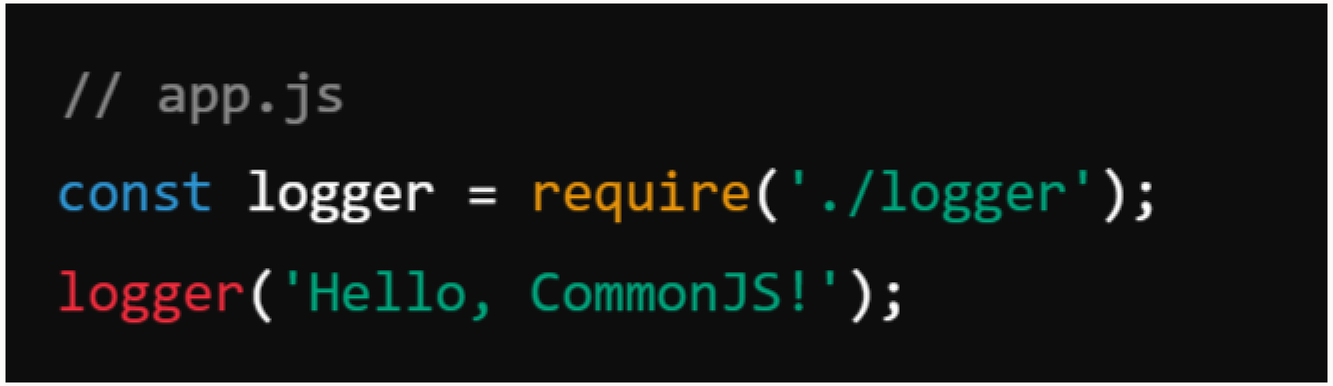
- **Multiple Exports:**



```
// utilities.js
module.exports = {
    greet: function (name) {
        return `Hello, ${name}!`;
    },
    farewell: function (name) {
        return `Goodbye, ${name}!`;
    },
};
```

PRAMOD NAIK

- **Importing:**

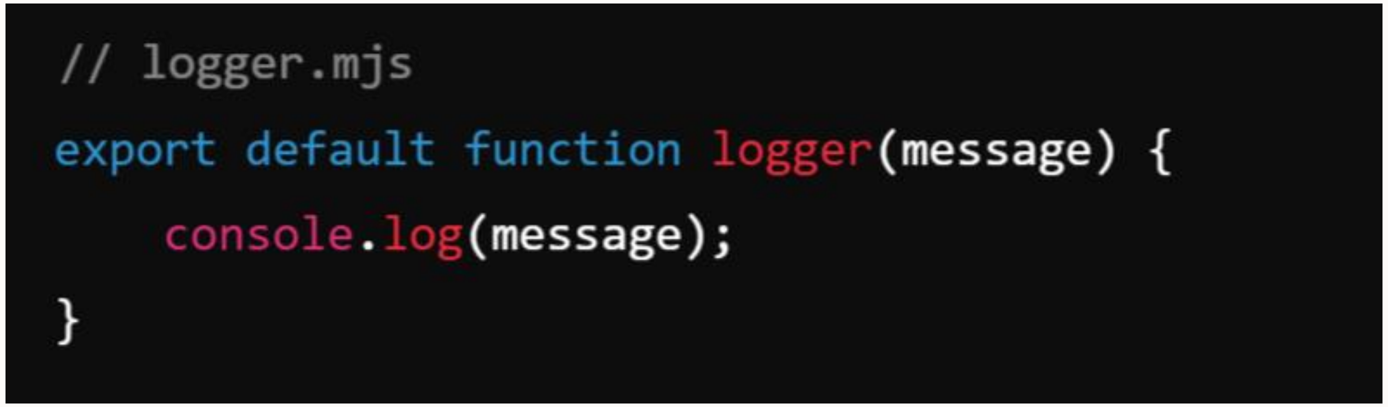


```
// app.js
const logger = require('./logger');
logger('Hello, CommonJS!');
```

PRAMOD NAIK

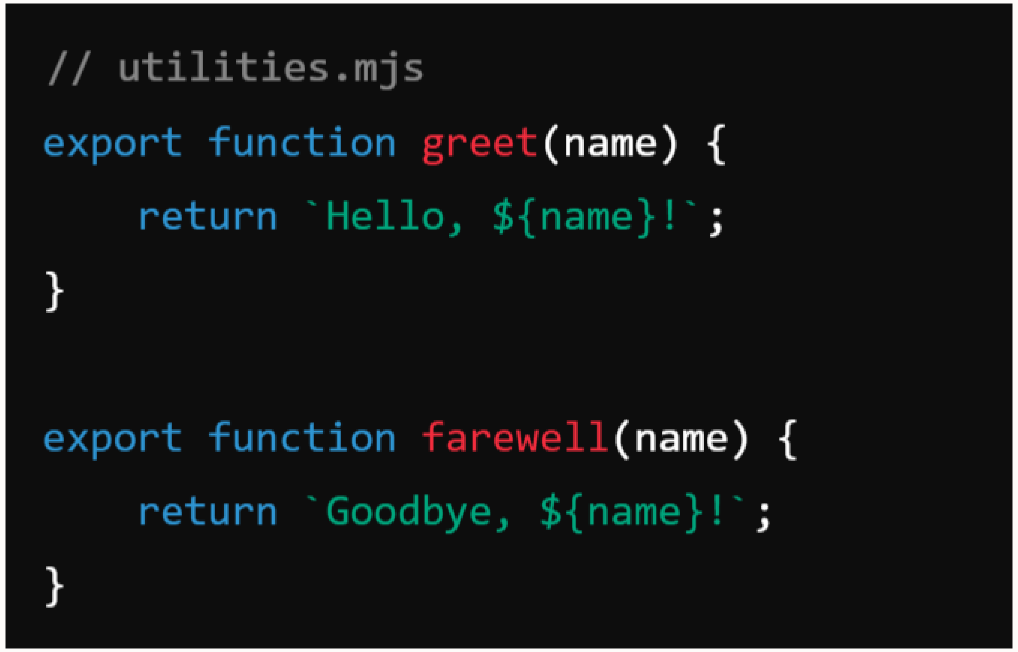
2. ESM Syntax: ECMA stands for **European Computer Manufacturers Association** (now officially called **ECMA** International). JavaScript is standardized under ECMA.

- **Single Export:**



```javascript
// logger.mjs
export default function logger(message) {
    console.log(message);
}
```

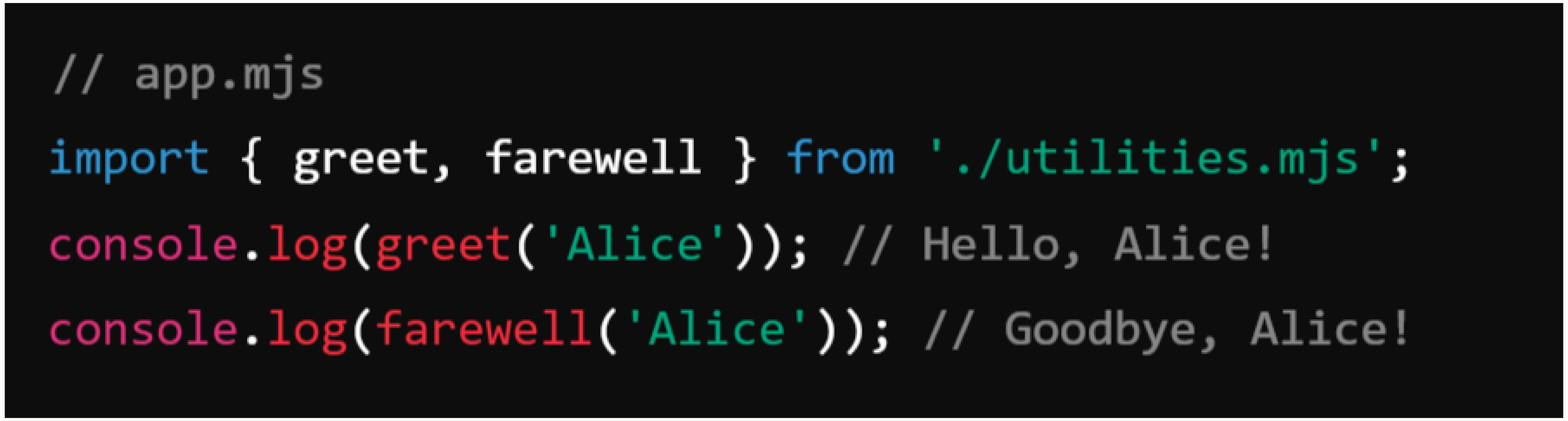
- **Multiple Exports:**



```javascript
// utilities.mjs
export function greet(name) {
    return `Hello, ${name}!`;
}

export function farewell(name) {
    return `Goodbye, ${name}!`;
}
```

PRAMOD NAIK

- **Importing:**



```
// app.mjs
import { greet, farewell } from './utilities.mjs';
console.log(greet('Alice')); // Hello, Alice!
console.log(farewell('Alice')); // Goodbye, Alice!
```

PRAMOD NAIK

Important Built-in Modules:

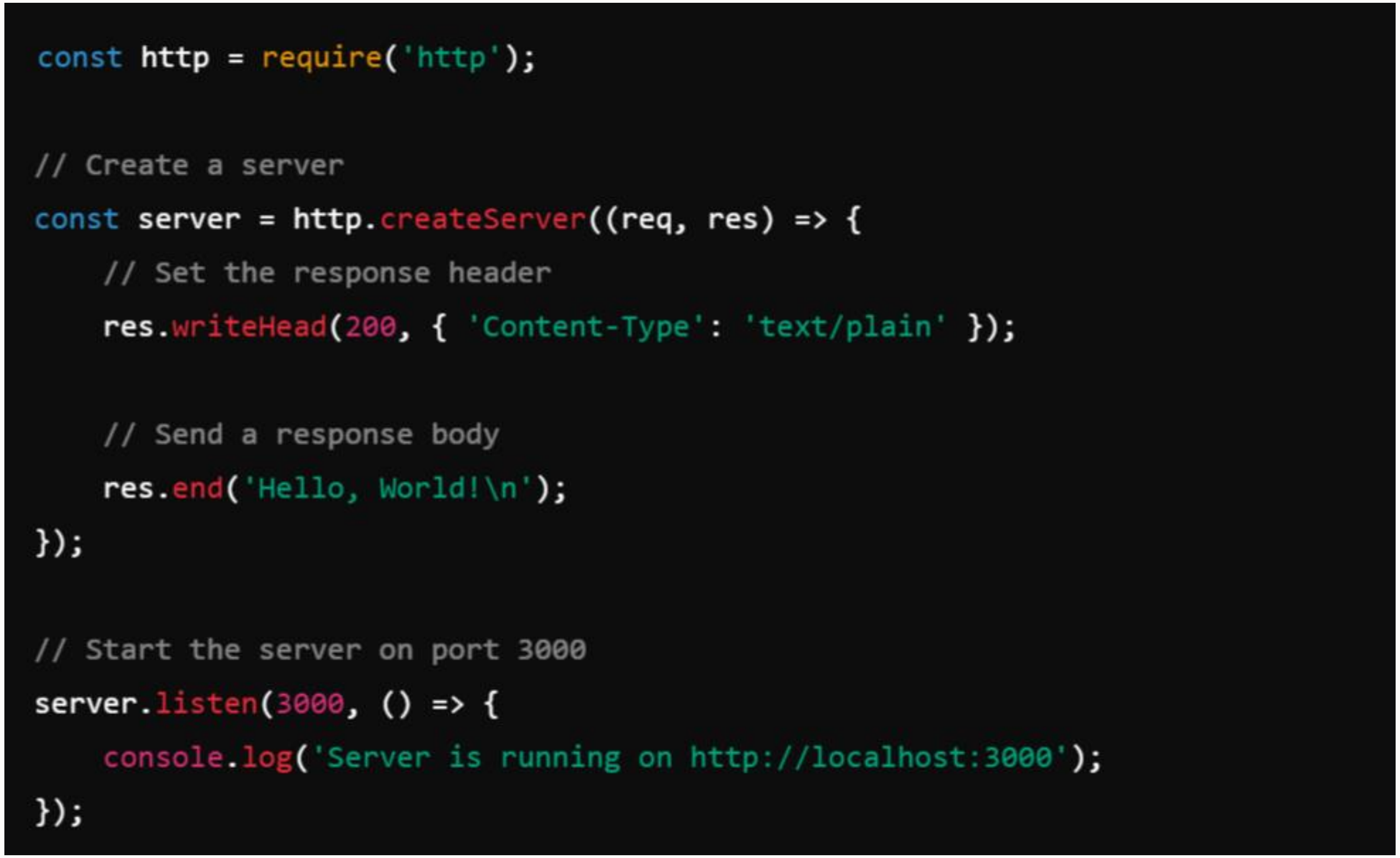
1. http
2. URL
3. Query String

PRAMOD NAIK

- **HTTP:**

The built-in **HTTP module** in Node.js allows you to **create** and **manage** **HTTP servers** and **clients**. This module provides functionality to handle **HTTP requests** and **responses**, enabling you to **build web servers** without requiring third-party libraries.

- **Creating an HTTP Server:** An **HTTP server listens for requests** from **clients** and **sends back responses**.



```
const http = require('http');

// Create a server
const server = http.createServer((req, res) => {
    // Set the response header
    res.writeHead(200, { 'Content-Type': 'text/plain' });

    // Send a response body
    res.end('Hello, World!\n');
});

// Start the server on port 3000
server.listen(3000, () => {
    console.log('Server is running on http://localhost:3000');
});
```

PRAMOD NAIK

How It Works:

1. http.createServer() creates an **HTTP server** that executes the provided **callback function** for every incoming request.

2. The **callback function** receives two parameters:

- **req**: The request object containing **data about the client request.**
- **res**: The response object used to **send data back to the client**.

3. res.writeHead() sets the response **headers**.

4. res.end() sends the **response body** and **closes the connection.**

5. server.listen(3000, () => {}): Starts an **HTTP server** and **listens** for **incoming requests on a specific port 3000.**

URL:

- The URL module in Node.js is a **built-in module** that provides **utilities** for **parsing**, **constructing**, and **working** with **URLs** (**Uniform Resource Locators**). It is particularly useful for extracting **query parameters**, **hostname**, **path**, and **other components** of a URL.

Importing the URL Module:

To use the URL module, you first **import** it **using** **require()** or **import**:

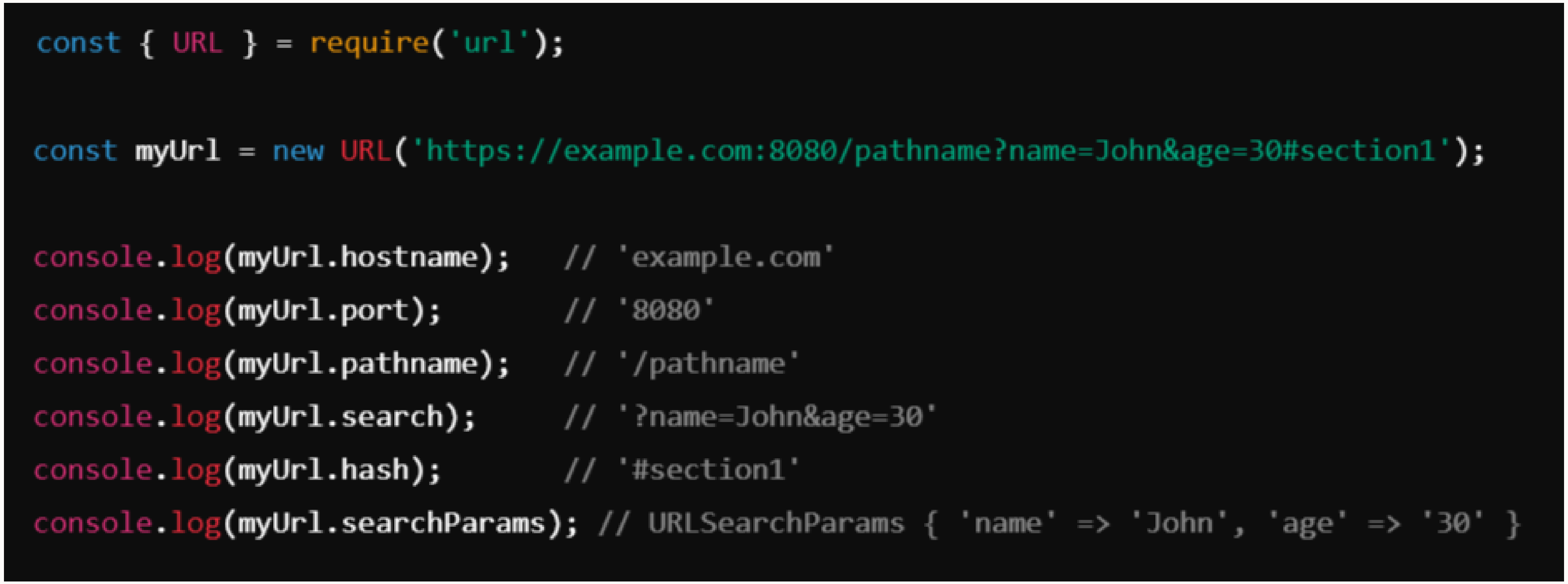
```
const url = require('url');
```

OR

```
import { URL } from 'url';
```

Key Features of the URL Module:

1. URL Parsing: You can **parse** a URL into its components using the **URL class**.



```
const { URL } = require('url');

const myUrl = new URL('https://example.com:8080/pathname?name=John&age=30#section1');

console.log(myUrl.hostname);    // 'example.com'
console.log(myUrl.port);        // '8080'
console.log(myUrl.pathname);    // '/pathname'
console.log(myUrl.search);      // '?name=John&age=30'
console.log(myUrl.hash);        // '#section1'
console.log(myUrl.searchParams); // URLSearchParams { 'name' => 'John', 'age' => '30' }
```

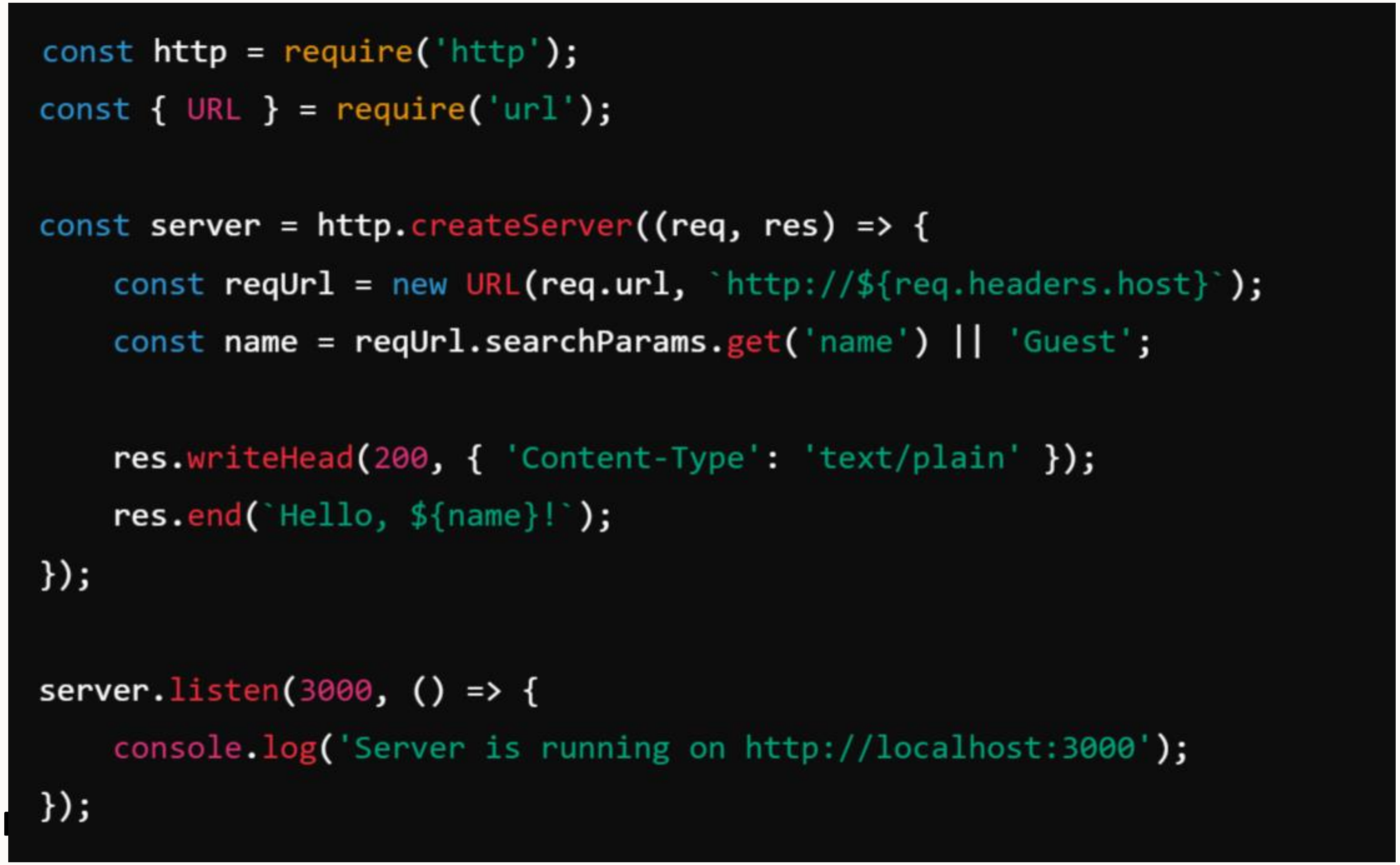
2. URL Components: The URL class provides properties to access **different parts** of a URL:

Property	Description	Example Value
href	The full URL as a string.	https://example.com:8080/pathname?name=John&age=30#section1
protocol	The protocol used (e.g., http, https, ftp).	https:
hostname	The domain name or IP address (without port).	example.com
port	The port number.	8080
pathname	The path of the URL (after the domain and port).	/pathname
search	The query string (including ?).	?name=John&age=30
hash	The fragment identifier (including #).	#section1
searchParams	Query parameters as a URLSearchParams object.	URLSearchParams { 'name' => 'John' }
origin	The protocol, hostname, and port combined.	https://example.com:8080

PRAMOD NAIK

Example: Parsing URL in a Server

- You can use the URL module in a Node.js HTTP server to extract information from incoming requests.



```
const http = require('http');
const { URL } = require('url');

const server = http.createServer((req, res) => {
    const reqUrl = new URL(req.url, `http://${req.headers.host}`);
    const name = reqUrl.searchParams.get('name') || 'Guest';

    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(`Hello, ${name}!`);
});

server.listen(3000, () => {
    console.log('Server is running on http://localhost:3000');
});
```

Here:

- **req.url:** Contains the endpoint from the client like / or /login.
- **Req.headers.host:** Contans host name like: http://localhost
- **new URL(req.url, base):** Combines the request path (req.url) with the host (req.headers.host) to form a **complete URL** object. For Example: http://localhost/admin

- **What is URL:**
  - A URL (**Uniform Resource Locator**) is a **reference** or **address** to a **resource on the internet**.
  - It specifies the **location of the resource** and the **mechanism to retrieve it**.
  - URLs are a **subset of URIs** (Uniform Resource Identifiers) but **are more specific** because they include the **means to locate the resource**.

PRAMOD NAIK

Use Cases of the URL Module

1. **Routing:** Determine the **path** and **query parameters** of incoming HTTP requests.
2. **URL Validation: Validate** and **manipulate URLs** in an application.
3. **API Calls: Construct** and **parse** URLs when interacting with APIs.
4. **Query Parameter Manipulation: Add**, **remove**, or **update query parameters** dynamically.

The URL module is an **essential tool** for any Node.js developer working with **web applications** or **APIs**!

PRAMOD NAIK

1. What is res.writeHead(200, { 'Content-Type': 'text/plain' })

- The res.writeHead() method is used in Node.js to **set the HTTP status code** and **response headers** for the server's response. It informs the client about the **status of the response** and provides **metadata** (headers) about the response content.

2. What is { 'Content-Type': 'text/plain' } (Headers):

This is an **object** representing **HTTP headers**, which provide **additional information about the response from the Server to Client.**

3. What is Content-Type?

The Content-Type **header tells the client what type of content it should expect** in the **response**. This helps the client **interpret the response correctly**.

PRAMOD NAIK

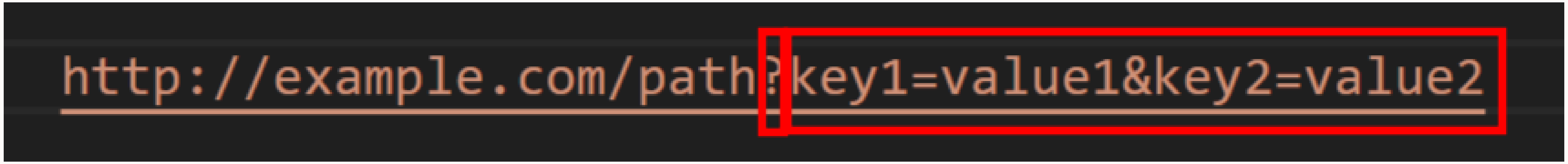
Examples of Content-Type Values

Content-Type	Description
text/plain	The content is plain text with no formatting.
text/html	The content is HTML and should be rendered as a web page.
application/json	The content is JSON data.
image/png	The content is a PNG image.
application/xml	The content is XML data.
text/css	The content is CSS stylesheets.
text/javascript	The content is JavaScript code.

PRAMOD NAIK

What is Query Strings in URL:

- A query string is a **part of a URL** that **contains additional data** in the form of **key-value pairs**. It is used to **pass information from the client** (e.g., a web browser) **to the server** as part of an HTTP request.
- The query string appears after the question mark (**?**) in the URL and consists of one or more key-value pairs, separated by an ampersand (**&**).
- **Structure of a Query String:** The basic structure of a URL with a query string is:



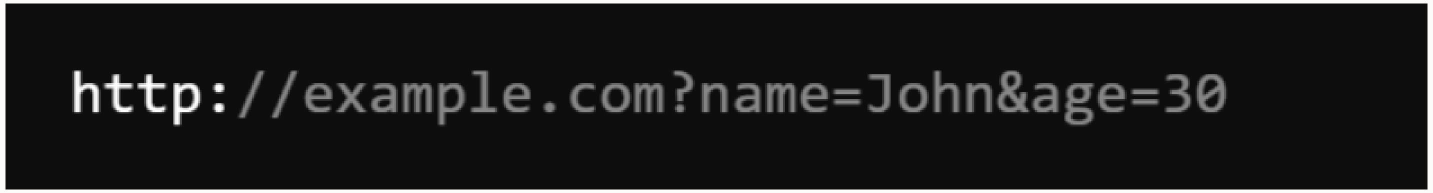
```
http://example.com/path?key1=value1&key2=value2
```

PRAMOD NAIK

Query String Module:

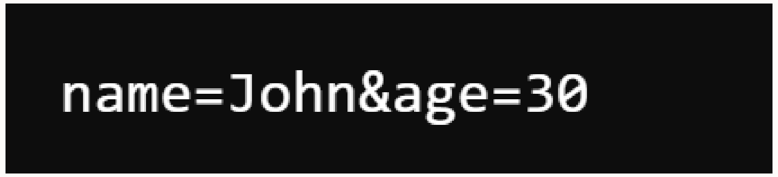
- The Query String module in Node.js is **used to work with query strings**, which are the **parts of a URL** that come after the **?**. These strings typically contain **key-value pairs** that pass parameters between the client and server.

For example, in the URL:



```
http://example.com?name=John&age=30
```

The **query string** is:



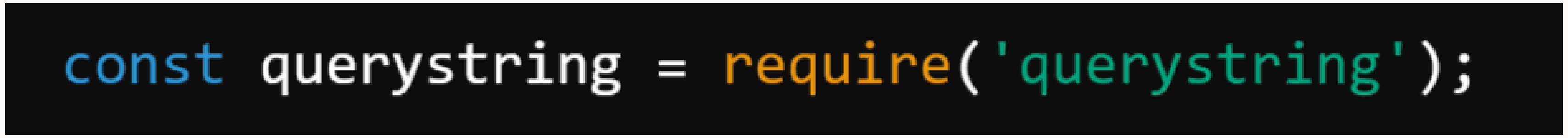
```
name=John&age=30
```

The Query String module provides **methods** for **parsing** and **stringifying** these query strings.

PRAMOD NAIK

Importing the Query String Module

To use the Query String module, you must import it using:

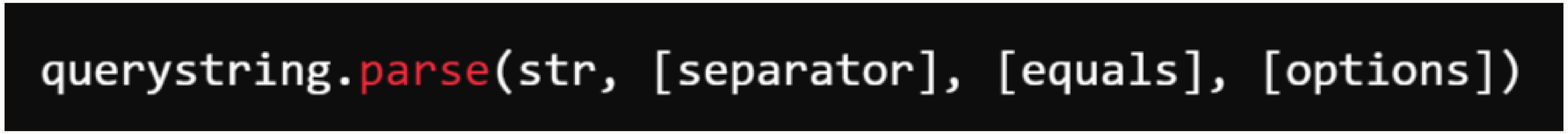


```
const querystring = require('querystring');
```

Key Methods:

1. querystring.parse(): This method parses a query string into a JavaScript object.

Syntax:

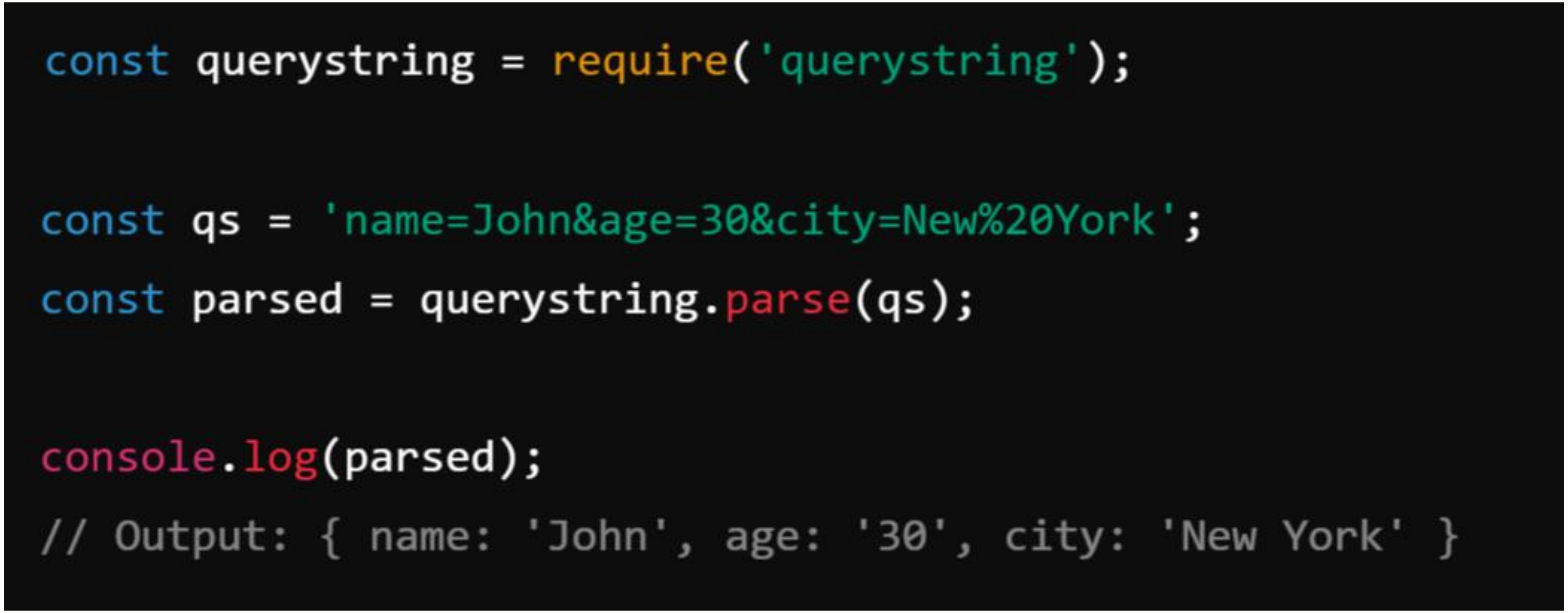


```
querystring.parse(str, [separator], [equals], [options])
```

- **str:** The query string to parse.
- **separator (optional):** The character used to separate **key-value pairs** (**& by default**).
- **equals (optional):** The character used to separate **keys and values (= by default**).
- **options (optional):** Options to **customize parsing behavior**.

PRAMOD NAIK

Example:

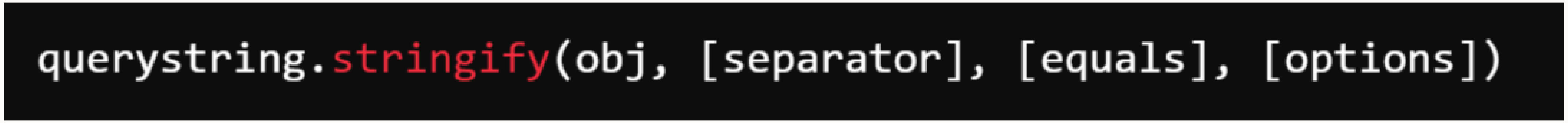


```
const querystring = require('querystring');

const qs = 'name=John&age=30&city=New%20York';
const parsed = querystring.parse(qs);

console.log(parsed);
// Output: { name: 'John', age: '30', city: 'New York' }
```

PRAMOD NAIK

- **2. querystring.stringify(): ** This method **converts a JavaScript object** into a **query string**.

Syntax:

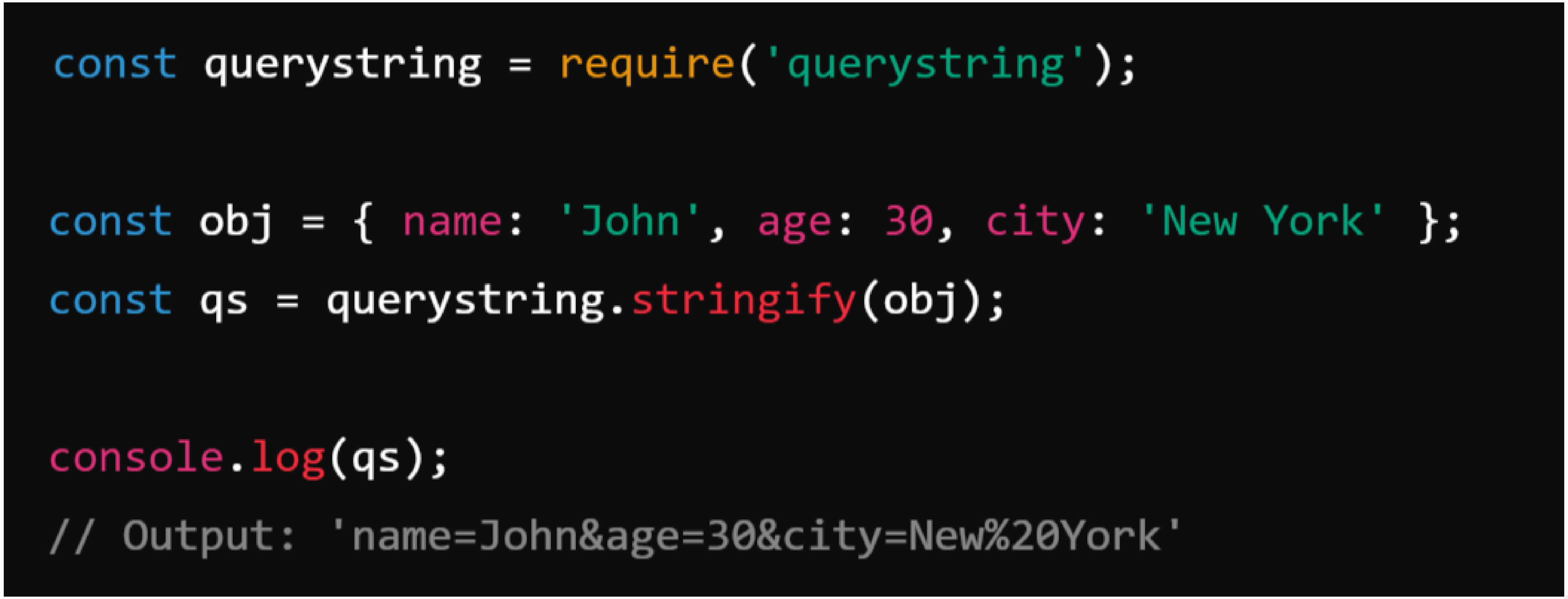


```
querystring.stringify(obj, [separator], [equals], [options])
```

- **obj:** The object to stringify.
- **separator (optional):** The character used to separate **key-value pairs** **(& by default).**
- **equals (optional):** The character used to separate **keys and values** **(= by default).**
- **options (optional):** Options to **customize stringification behavior**.

PRAMOD NAIK

Example:



```
const querystring = require('querystring');

const obj = { name: 'John', age: 30, city: 'New York' };
const qs = querystring.stringify(obj);

console.log(qs);
// Output: 'name=John&age=30&city=New%20York'
```

PRAMOD NAIK

3. querystring.encode(): An **alias** for querystring.**stringify**().

4. querystring.decode(): An **alias** for querystring.**parse**().

Diff Between Query String and **Query Parameters** :

Query String and **Query Parameters** are **closely related concepts** in the context of URLs, but they are **not exactly the same**.

1. Query String

The query string is the **entire part of a URL that comes after the ?.** It is a string containing **key-value pairs separated by &,** and each key is separated from its value by **=**.

Example: In the URL:

```
https://example.com/search?name=John&age=30&city=NewYork
```

The query string is:

```
name=John&age=30&city=NewYork
```

PRAMOD NAIK

2. Query Parameters:

The query parameters are the **individual key-value pairs within the query string**. They represent the **actual data** being passed in the URL.

Example: Using the same URL:

```
https://example.com/search?name=John&age=30&city=NewYork
```

The query parameters are:

- name=John
- age=30
- city=NewYork

PRAMOD NAIK

What is URL Encoding?

URL encoding, also known as **percent-encoding**, is a mechanism used to encode information in a URL so that it **conforms to the standards of the Uniform Resource Locator (URL).** It ensures that the URL remains valid and **interpretable by browsers and servers.**

URLs can only include certain characters from the **ASCII** character set. URL encoding converts **characters that are not allowed in URLs or have special meanings** into a format that can be **safely transmitted over the internet**.

PRAMOD NAIK

How URL Encoding Works:

URL encoding **replaces unsafe** or **reserved characters** in a string with a **%** followed by **two hexadecimal digits** representing the character's ASCII value.

- **Reserved Characters:**

Reserved characters have special purposes in URLs. For example:

? separates the URL path from the query string.

& separates query parameters.

To use these characters in other contexts, they must be **encoded**.

- **Unreserved Characters:**

Unreserved characters **do not need encoding**. These include:

  - **Letters:** a-z, A-Z
  - **Digits:** 0-9
  - **Symbols**: -, _, ., ~

Encoding Formats:

Character	URL Encoded Value
Space	%20
&	%26
/	%2F
?	%3F
=	%3D

PRAMOD NAIK

Example:

Original URL:

```
https://example.com/search?name=John Doe&age=30
```

Encoded URL:

```
https://example.com/search?name=John%20Doe&age=30
```

Here:

- **Space** in "John Doe" → **%20**

PRAMOD NAIK

Creating a Custom Module in Node.js:

In Node.js, **custom modules** are **user-defined JavaScript files** that encapsulate functionality and can be **imported** into other parts of your application. This modular approach helps keep code **organized**, **reusable**, and **maintainable**.

Steps to Create a Custom Module

1. **Create a JavaScript File:**

A custom module is **simply a .js file** containing the code you want to reuse.

2. Export the Functionality:

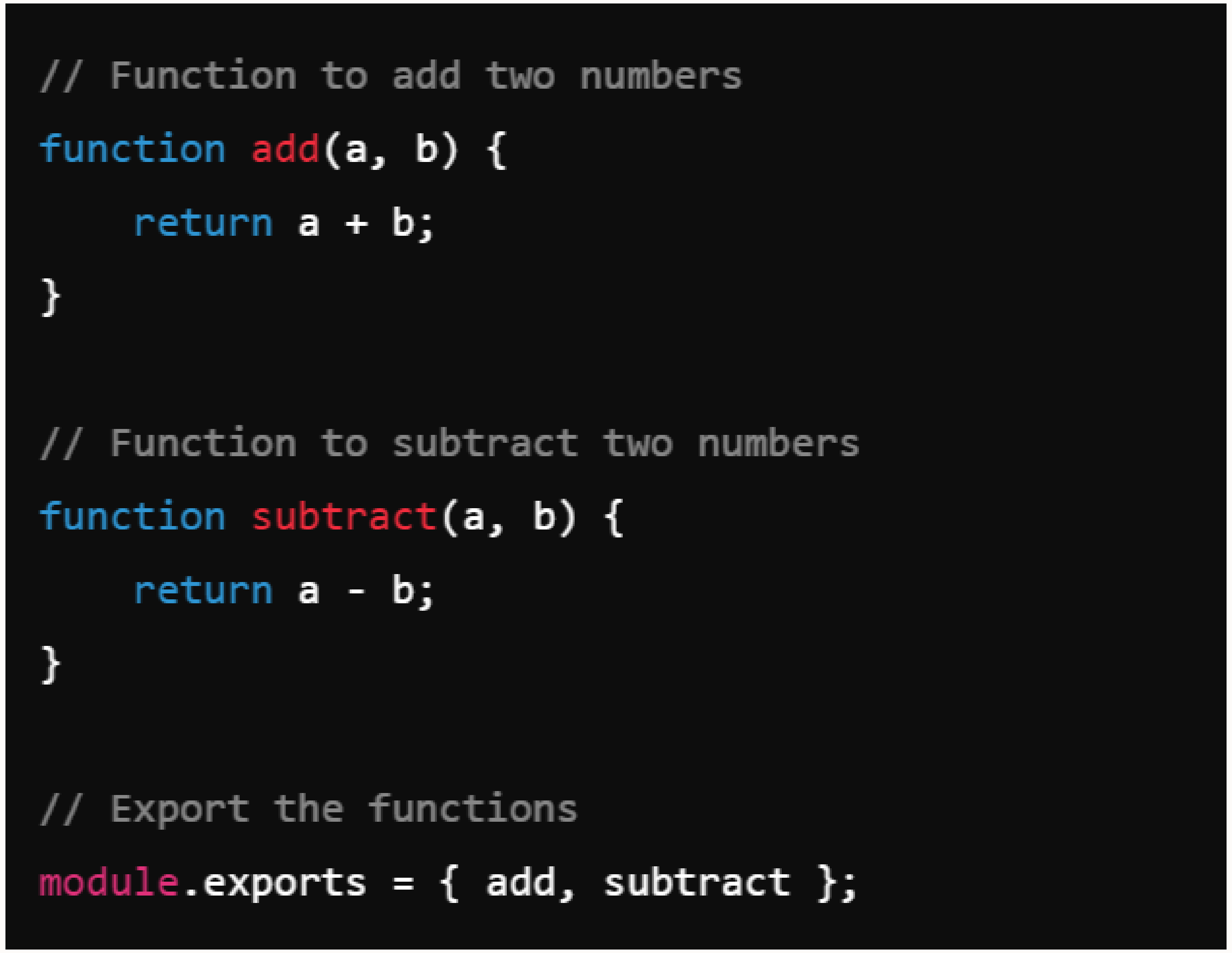
Use **module.exports** or **ES Module export** syntax to define what parts of the module should be accessible to other files.

3. Import and Use the Module:

Use **require()** for CommonJS modules or **import** for ES Modules to include the custom module in another file.

Example: Creating and Using a Custom Module

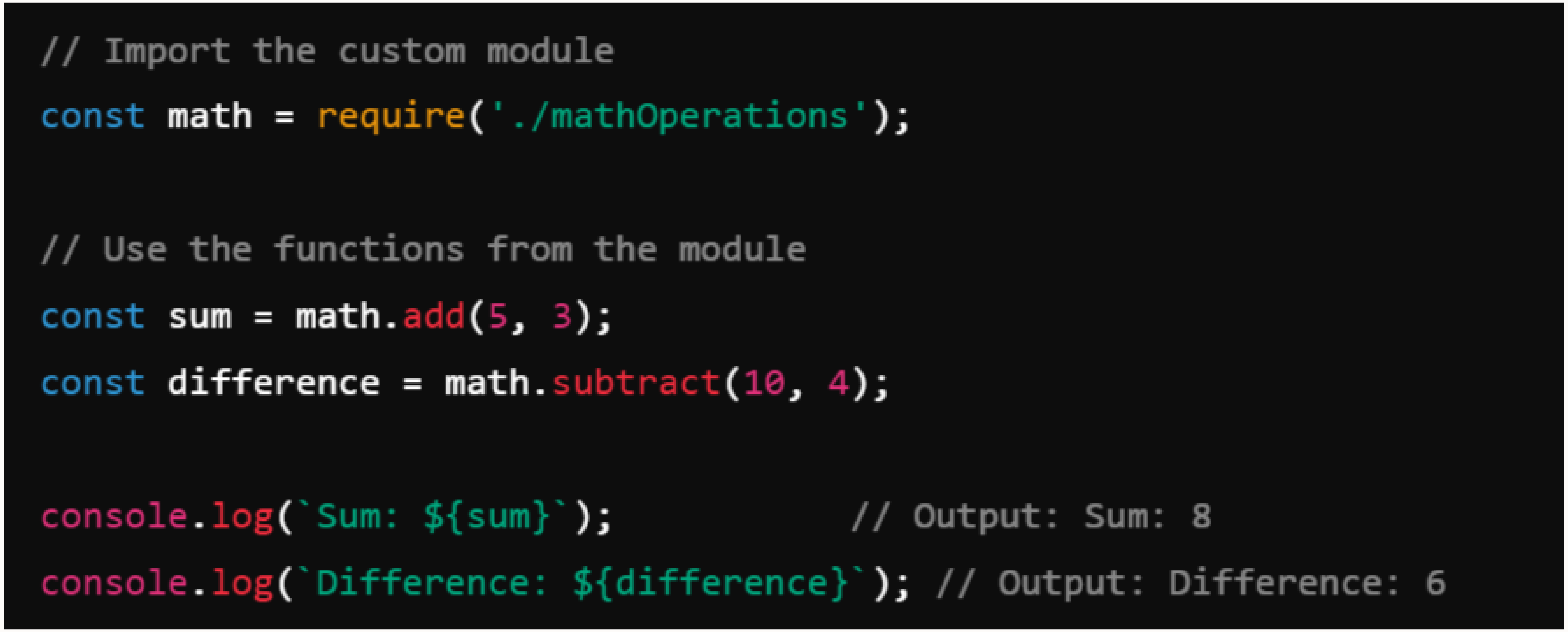
1. **Create the Module File:** Create a file called **mathOperations.js**:



```javascript
// Function to add two numbers
function add(a, b) {
    return a + b;
}

// Function to subtract two numbers
function subtract(a, b) {
    return a - b;
}

// Export the functions
module.exports = { add, subtract };
```

PRAMOD NAIK

2. Use the Module in Another File: Create another file, e.g., **app.js**:



```javascript
// Import the custom module
const math = require('./mathOperations');

// Use the functions from the module
const sum = math.add(5, 3);
const difference = math.subtract(10, 4);

console.log(`Sum: ${sum}`);           // Output: Sum: 8
console.log(`Difference: ${difference}`); // Output: Difference: 6
```

PRAMOD NAIK

Node Package Manager

PRAMOD NAIK

What is Node Package Manager (npm)?

Node Package Manager (**npm**) is a **command-line tool** and a **package manager** for the **JavaScript runtime Node.js**. It simplifies the process of **sharing**, **managing**, and **using reusable code** modules in your Node.js projects.

NPM is a **crucial part of Node.js** that helps manage **dependencies** and **packages** for JavaScript projects. It is **used to install**, **update**, and **manage libraries** and **tools that you can use in your Node.js applications.**

PRAMOD NAIK

Key Features of npm:

1. **Package Management:** npm allows you to **install packages** (or **modules**) from the **npm registry.** These packages can be libraries or tools that provide functionality like **authentication**, **database connections**, **utilities**, and more.
2. **Version Control:** It helps in **versioning packages**, ensuring that the **required version** of a package is installed and used in your project. You can specify **version ranges** in your package.json file, allowing for flexibility and control over the updates to your dependencies.
3. **Local and Global Installation**
4. **Managing Project Dependencies:** The **package.json** file defines the **dependencies of your project.** When you run npm install, npm installs all the required packages listed in the **dependencies** and **devDependencies** sections of this file.
5. **Scripts:** npm allows you to **define scripts** in your **package.json** file to **automate common tasks**, such as **starting** a development server, **running** tests, or **building** your project. You can run these scripts using **npm run <script-name>.**
6. **npm Registry:** npm uses a **public registry** to **store** and **share packages**. You can publish your own packages to the registry or use the ones shared by others.
7. **Package Locking:** npm generates a **package-lock.json** file **when installing dependencies**, which **locks** the specific version of each package installed. This ensures that your project has consistent versions of dependencies across different environments.

PRAMOD NAIK

Installing Packages Locally:

Installing packages locally in Node.js using npm means **adding dependencies** to your project in the **node_modules directory**, which is located in your **project's root folder.**

These packages are then **available only within that specific project** and can be used to add functionality, such as libraries or frameworks, that your project needs to run.

How Local Installation Works:

1. **node_modules Directory:** When you install a package locally, it is placed inside the **node_modules directory** in your project folder. This is the **default location** where npm installs dependencies.
2. **package.json File:** npm also **updates** the **package.json** file to include the package in the dependencies or devDependencies section, depending on whether it is needed for **development** or **production**.
3. **Project-Specific:** The installed packages are **only accessible in the project directory**, and they **won't affect other projects on your machine**.

PRAMOD NAIK

How to Install Packages Locally:

Step 1: Initialize Your Project (If Not Already Done)

If you haven't already initialized your project, you can create a package.json file by running:

```
npm init
```

OR

```
npm init -y
```

This will create a **package.json** file in your project directory.

Step 2: Install a Package Locally: To install a package locally, simply run:

```
npm install <package-name>
```

This command:

- **Downloads the latest stable version of the package** and its dependencies.
- Creates a **node_modules** directory if it doesn't exist.
- Adds **package** to your **package.json** file under the **dependencies section**.

PRAMOD NAIK

Installing a Specific Version Locally:

If you want to install a **specific version** of a package, you can specify the version number:

```
npm install <package-name>@<version-number>
```

Example:

```
npm install axios@4.17.1
```

This command will install version 4.17.1 of the **axios package** and update your **package.json** (if the package is listed as a dependency) and package-lock.json accordingly.

If you want to install this version as a **devDependency**, you can use the --save-dev flag like this:

```
npm install express@4.17.1 --save-dev
```

PRAMOD NAIK

Installing packages globally:

Installing packages globally in Node.js means **making the package available system-wide**, **outside the context of any specific project**. This allows you to use the package’s command-line tools and utilities from **anywhere on your machine**, **regardless of which project you are working on**.

How Global Installation Works:

1. **System-wide Installation:** When you install a package globally, **npm places it in a system-wide directory**, **making it available globally on your system**. This allows you to access the installed package's commands from anywhere in your terminal.
2. **Global Bin Folder:** The executable files of global packages are typically placed in a **system-wide bin directory** (e.g., /usr/local/bin/ on macOS/Linux or **C:\Users\<user>\AppData\Roaming\npm\bin\** on Windows).
3. **Global Package Path:** The global packages are installed in a specific folder (**outside your project directory**), making them accessible **to all Node.js projects on your system.**

PRAMOD NAIK

How to Install Packages Globally:

- **Command:** To install a package globally, you need to use the **-g** or **--global flag** with the **npm install command**:

```
npm install -g <package-name>
```

For example, to install **typescript globally**:

```
npm install -g typescript
```

This command will:

- **Download** and **install** the specified package.
- Place it in the **global directory** (e.g., /usr/local/lib/node_modules on Linux/macOS or C:\Users\<user>\AppData\Roaming\npm\node_modules on Windows).
- Make the **command-line executable** (e.g., tsc for TypeScript) available globally via the terminal.

PRAMOD NAIK

How to Use a Globally Installed Package:

Once installed globally, you can use the **package's command-line tools** from **anywhere** in your terminal.

For example, after installing **TypeScript globally**, you can use the **tsc command** to **compile TypeScript files:**

```
tsc myfile.ts
```

This works even if you're **not in the directory** where **TypeScript was installed** because it's **globally available**.

Installing a Specific Version Globally: You can also install a specific version of a package globally:

```
npm install -g <package-name>@<version>
```

PRAMOD NAIK

Adding dependency in package.json:

In Node.js projects, the **package.json file** plays a **crucial role** in **managing the project's dependencies** (both runtime and development dependencies). When you add a dependency to a Node.js project, you can do so **manually by editing the package.json file**, or more commonly, by **using npm commands** to **automatically update this file.**

What is a Dependency?

A dependency is a **package that your project requires to function (i.e Work Properly)**. These packages can be **libraries**, **tools**, or **frameworks that your project depends on** at runtime or during development.

Runtime dependencies: These are packages required for your **application to run.**

Development dependencies: These are packages needed **only during development** (e.g., for testing, bundling, or linting).

PRAMOD NAIK

- **How Dependencies Are Managed in package.json:**

1. **dependencies:** Lists packages required to run your application in **production**.
2. **devDependencies:** Lists packages required **only during development**.
3. **peerDependencies:** Lists packages that **your project is compatible with**, but are expected to be installed by the **consumer of your package**.
4. **optionalDependencies:** Lists packages that are **not essential** but can be installed if needed.

Adding Dependencies Using npm:

1. Install a Dependency and Add It Automatically:

You can add dependencies to your **package.json** file by using **npm commands**. These commands will install the package and update the package.json file accordingly.

- To add a regular dependency (runtime dependency):

```
npm install <package-name>
```

Example:

```
npm install express
```

This command:

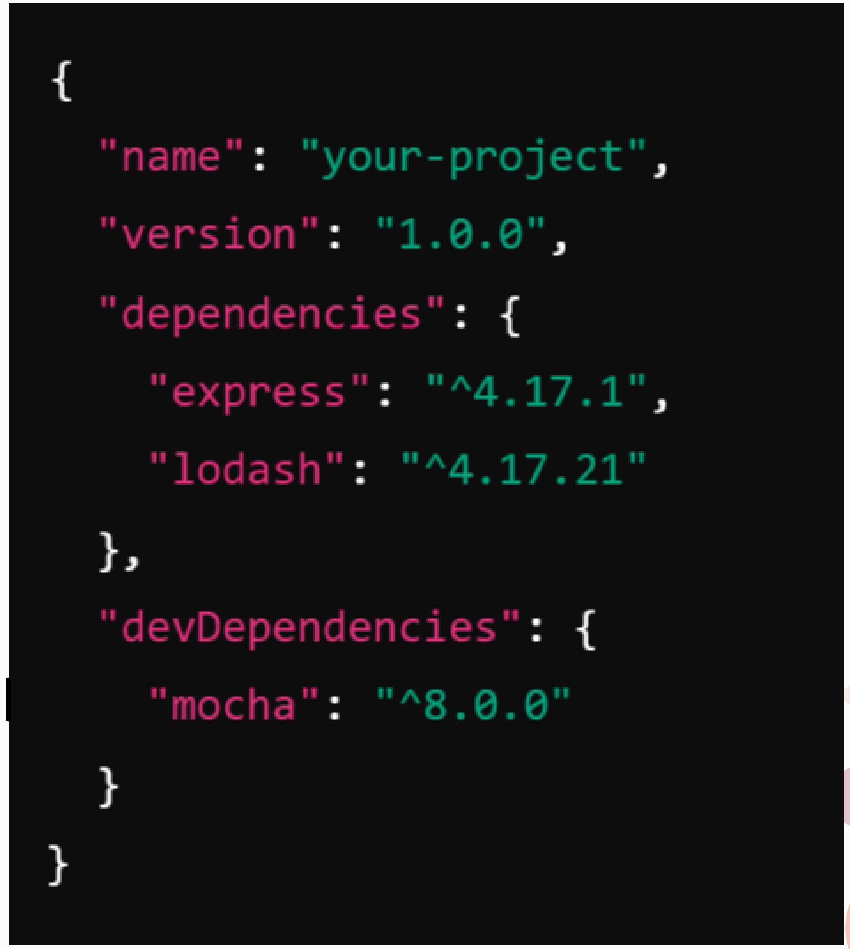
- Installs the express package in the **node_modules** folder.
- Adds express to the **dependencies** section of the **package.json**.

PRAMOD NAIK

2. Manually Adding Dependencies to package.json:

You can also **manually edit the package.json** file **to add dependencies**. This method is **less common**, but useful in specific scenarios, such as when you want to lock down **certain versions of packages** or **manage dependencies manually**.

Example of adding dependencies manually:



```
{
  "name": "your-project",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.17.1",
    "lodash": "^4.17.21"
  },
  "devDependencies": {
    "mocha": "^8.0.0"
  }
}
```

In this case:

- **express** and **lodash** are runtime dependencies (under the **dependencies** section).
- **mocha** is a **development dependency** (under the **devDependencies** section)

Note: Once after adding the any type of dependencies **manually Run** the **npm install** to install all those newly added dependencies.

PRAMOD NAIK

3. Install All Dependencies:

After adding dependencies to package.json, you can install them all at once by running:

```
npm install
```

This command **installs all the dependencies listed** in **dependencies** and **devDependencies**, as well as any other required packages for the project.

Updating packages:

In Node.js projects, **keeping dependencies up-to-date** is essential for ensuring **security**, **stability**, and **access to the latest features** and **bug fixes**. You can update packages in your project using npm commands. Here's how to update packages in your Node.js project.

1. Updating a Single Package:

To update a **specific package** to the **latest version** (within the version range specified in package.json), you can use the following command:

```
npm update <package-name>
```

This command:

- Will check if the version of the <package-name> in your node_modules folder is outdated (according to the version specified in package.json).
- If the version is outdated, **npm will update it to the latest version** that satisfies the **version range defined** in package.json.

PRAMOD NAIK

2. Updating All Packages:

If you want to update **all the dependencies listed in your package.json** file to their latest versions within the allowed version range, you can simply run:

```
npm update
```

This will:

- **Check all the dependencies** listed in **dependencies** and **devDependencies**.
- Update them to the latest versions that match the version ranges specified in the package.json.

However, this will not update dependencies to a version outside the specified range (e.g., ^1.2.3 will not update to 2.0.0).

PRAMOD NAIK

3. Checking for Available Updates:

To see **which packages are outdated** (i.e., have newer versions available), you can run:

```
npm outdated
```

This command will provide a **list of packages that have newer versions available**, showing:

- The **current version installed**.
- The **wanted version** (the latest version that fits the version range in package.json).
- The **latest version available** (the most recent release regardless of your package.json version range).

PRAMOD NAIK

Answer the following 7 Mark Questions:

1. Explain the concept of primitive types in Node.js. Provide examples of primitive types and their uses in Node.js.
2. What is an object literal in Node.js? How can you create and use object literals in your Node.js applications?
3. Describe the role of functions in Node.js. How are functions defined and called in Node.js, and what are their advantages?
4. What is a buffer in Node.js? How are buffers used to handle binary data in Node.js applications?
5. How can you access the global scope in Node.js? What precautions should be taken when working with the global scope in Node.js applications?
6. What is the purpose of the Node Package Manager (npm) in Node.js development? How can you use npm to manage dependencies in your Node.js projects?
7. How can you adda dependency to your project's package.json file using npm? Explain the benefits of adding dependencies to the package.json file.
8. Explain the difference between installing packages locally and globally using npm. Whenwould you use each approach in your Node.js projects?

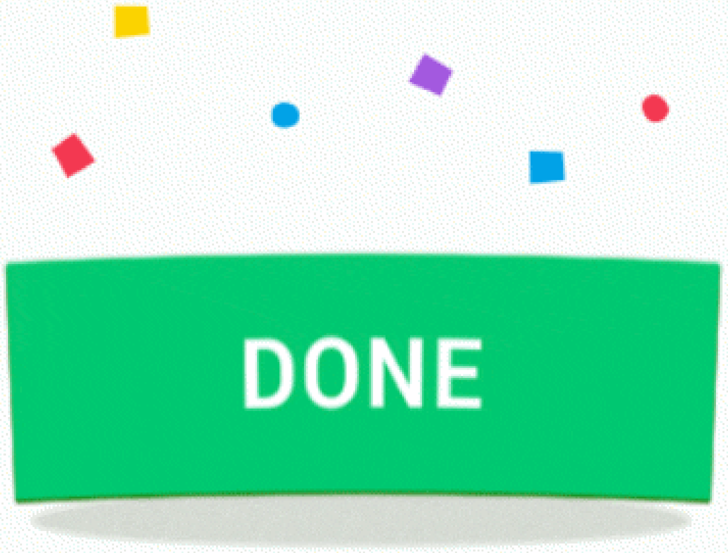
PRAMOD NAIK

Answer the Following 10 marks:

9. Discuss the concept of modules in Node.js. Explain the different types of modules (core modules, local modules, third-party modules) and how they are used in Node.js applications. Provide examples to illustrate each type of module.

10. Explain the process of using modules in a Node.js file. How can you import and use modules in your Node.js applications? Provide examples to demonstrate the import and use of modules.

UNIT-II: Introduction to Node.js



PRAMOD NAIK