**Global Object:**

* The object accessible in every file.
* Some examples of global objects are console, setTimeout(), clearTimeout(), setInterval(), clearInterval().
* In browser we have **“window”** object that represent global scope.
* So all variable/functions defined globally, we can access them through **“window”** object. ex. window.console.log(“Hello World”) i.e. Same as console.log(“Hello World”)
* If we define any variable/function in file then this variable/function is added to window object and hence we can access this variable/function through window object. ex. define a variable **var message = “OK”;**  we **can** access message variable like this => **“window.message”**
* But in node we have global object called **“global”**. Hence we can access console object like this => **global.console.log(“Hello World”)**
* But unlike browser, the variables defined in file are not added to the “global” object. ex. var message = “OK”; we **can’t** access message variable like this => **“global.message”.** Hence we cannot access variable/function defined in a file outside of that file.
* And this is because of the **node’s modular system**

**Module:**

Module is a block of code that is encapsulated and not available outside the module/file.

Node Automatically wraps the code in each file with an IIFE(Immediately-invoked Function Expression )to create scope. So, variables and functions defined in one file are only scoped to that file and not visible to other files unless explicitly exported (**module.exports** is used to export the variables and functions)

In node.js every file is module

"Module" is a variable which represents the curent module



**Module Wrapper Function:**

Node Automatically wraps the code in each file with an IIFE(Immediately-invoked Function Expression )to create scope. The functions has signature like this

**(function (exports, require, module, \_\_filename, \_\_dirname) {**

**});**

Where,

***exports*** is a reference to module.exports .i.e it is shortcut.

hence module.exports.xyz = variable/function is same as exports.xyz = variable/function

***module***  is a variable which represents the curent module

***\_\_filename*** : Stores the complete path to current file with file name

***\_\_dirname*** : Stores the directory in which current file is stored

**Path Module:**

The path module provides utilities for working with file and directory paths. It can be accessed using:

const path = require('path');

If we want to work with paths then it is easier to work with “path module” as opposed to working with strings.

const path = require("path");

const pathObj = path.parse("C:/Desktop/Akash.txt")

console.log(typeof(pathObj))

console.log(pathObj);



**OS Module:** The os module provides operating system-related utility methods and properties. It can be accessed using:

const os = require('os');

Accessing OS parameters is possible because of node.

const os = require('os')

var totalMemory = os.totalmem();

var freeMemory = os.freemem();

console.log("Total Memory: " + totalMemory);

//console.log("Free Memory: " + freeMemory)

console.log(`Free Memory: ${freeMemory}`);



**File System Module:**

The fs module enables interacting with the file system in a way modeled on standard POSIX functions.

To use this module:

const fs = require('fs');

All file system operations have synchronous, callback, and promise-based forms.

**fs module has both asynchronous and synchronous methods but in real world we should use asynchronous because they are non-blocking methods.**

Synchronous example:

const fs = require('fs')

const files = fs.readdirSync('./') // ./ represents current directorry

console.log(files)



Asynchronous example:

All asynchronous function accept a callback function as their last parameter

The callback function has 2 attributes **error** and **output.** Each time one of the parameter has value and other is NULL.

const fs = require('fs')

fs.readdir('./', function(err, output){

    if(err) // if err is not null display error

        console.log("Error", err)

    else // if output is not null display ouput

        console.log("Output=> ", output)

})



**Event Module:**

* Event is a signal that indicate something is happened in our application
* When even is created our program should listen to the even and give the appropriate response to it

Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") emit named events that cause Function objects ("listeners") to be called.

For instance: a [net.Server](https://nodejs.org/dist/latest-v14.x/docs/api/net.html#net_class_net_server) object emits an event each time a peer connects to it; a [fs.ReadStream](https://nodejs.org/dist/latest-v14.x/docs/api/fs.html#fs_class_fs_readstream) emits an event when the file is opened; a [stream](https://nodejs.org/dist/latest-v14.x/docs/api/stream.html) emits an event whenever data is available to be read.

All objects that emit events are instances of the EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object. Typically, event names are camel-cased strings but any valid JavaScript property key can be used.

When the EventEmitter object emits an event, all of the functions attached to that specific event are called *synchronously*. Any values returned by the called listeners are *ignored* and discarded.

The following example shows a simple EventEmitter instance with a single listener. The eventEmitter.on() method is used to register listeners, while the eventEmitter.emit() method is used to trigger the event.

const EventEmitter = require('events')

const emitter = new EventEmitter()

emitter.on('event', function(){

    console.log("Event listener called")

})

emitter.emit('event')

* Event module returns the class.
* Hence we have to instantiate that class in order to use its methods
* We must register event listener before event is called. This can be done using **addListener() or on()**

**Event Arguments:** To sendarguments with events.

The argument **“e”** in callback function of addListener is storing the arguments of the event.

const EventEmitter = require('events')

const emitter = new EventEmitter();

emitter.addListener("Logging", function(e){

    console.log(`Username: ${e.name}\nPassword: ${e.password}`)

})

emitter.emit("Logging", {name:"User", password: "UserPassword"})

**Extending the EventListener class:**

**App.js file**

const EventEmitter = require('events')

const Logger = require('./module.js')

const logger = new Logger.log();

logger.on('Logging', function(arg){

    console.log("Event Listener is on: ", arg);

})

logger.hello("World")

**module.js file**

const EventEmitter = require('events')

class Logger extends EventEmitter{

    name = "Akash";

    hello(name) {

        console.log("Hello " + name)

        this.emit("Logging", {Username: "Node.js", Password: "PasswordJS"})

    }

}

module.exports.log = Logger;

**HTTP Module:**

We can use http module to create servers. This can respond to the requests on port.

const http = require('http')

const server = http.createServer(function(req, res) {

    if(req.url === '/'){

        res.write("hello world");

        res.end();

    }

    if(req.url === '/api/courses'){

        res.write(JSON.stringify([1, 2, 3]));

        res.end();

    }

});

server.listen(5000);

console.log('Listening on port 3000 ')

But in real world we don’t use the http module to build backend services for applications. Because as we add more routes if(req.url === '/'), if(req.url === ‘api/courses’) .

Then our application becomes complex because we add the routes in linear way inside callback function. So instead we use framework called **“Express”** which gives our application a clean structure to handle various routes. Internally the **Express framework is built on top of http module in node**.