WILEY

ENABLING DISCOVERY | POWERING EDUCATION | SHAPING WORKFORCES

NodeJS Fundamentals

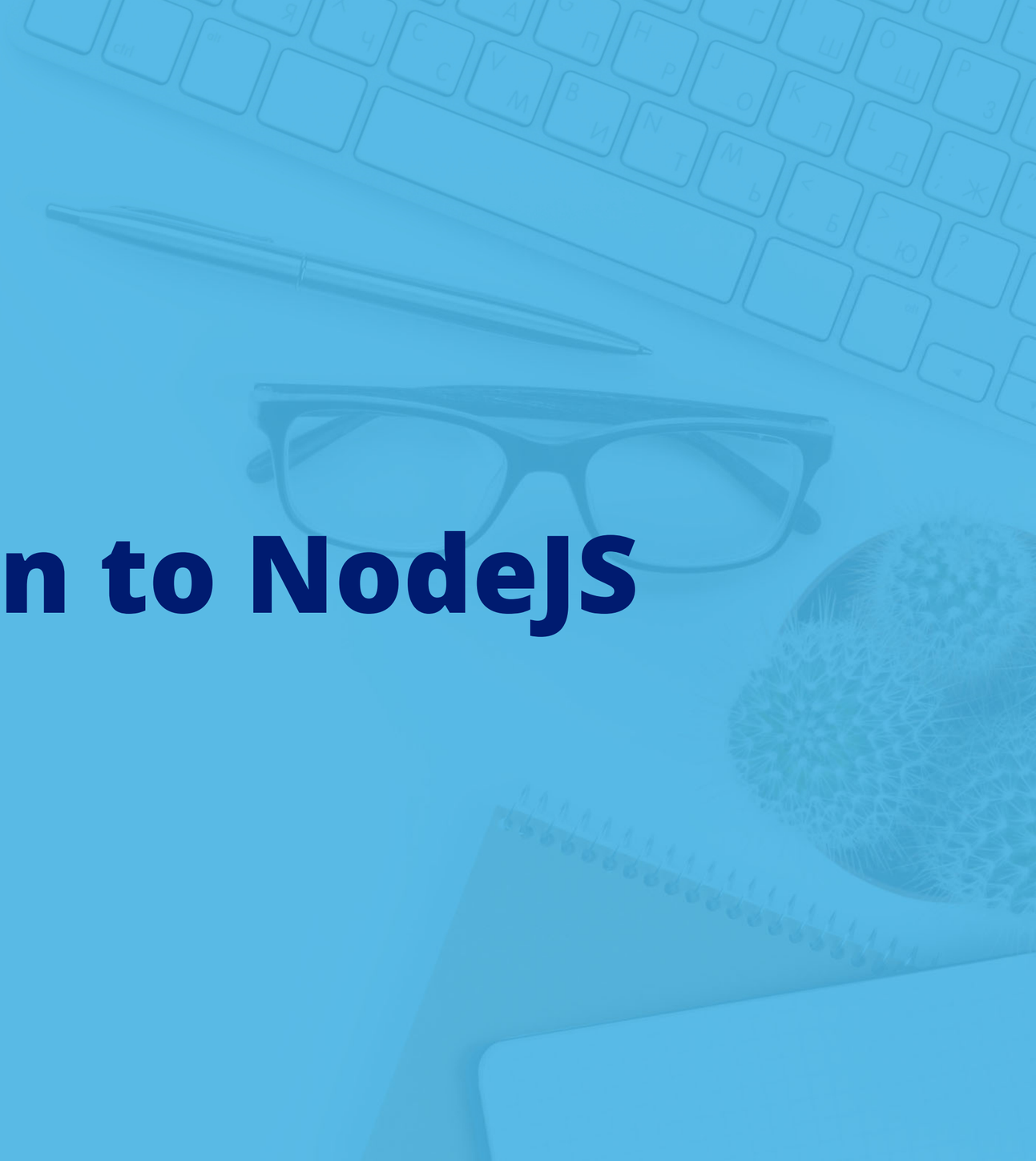
Objectives

- Overview and Building Block of NodeJS
- Installing and Developing NodeJS Environment
- Node.js vs. others Server-Side Frameworks
- Node.js processing via event loop

Objectives

- Synchronous vs. Asynchronous Approach
- NodeJS Modules and its Module Types
- Node Package Manager
- Publishing your own module

Introduction to NodeJS

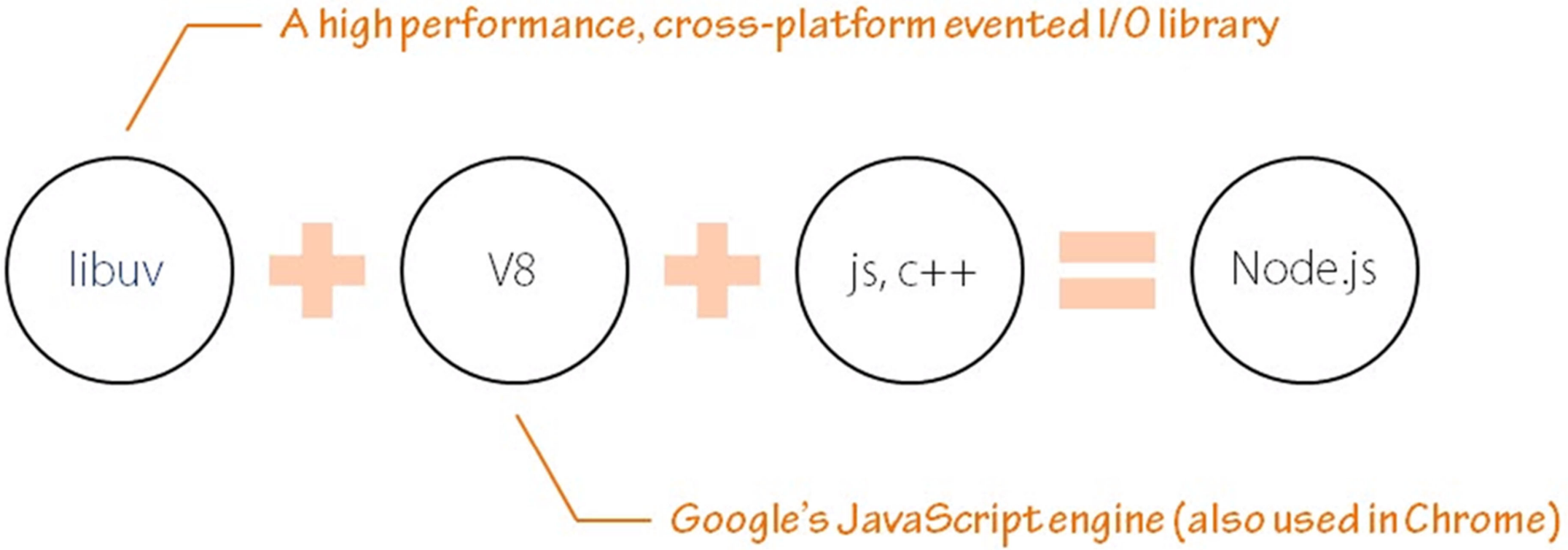


Introduction to NodeJS

- "Node.js" is a server-side "JavaScript" platform.
- It was first introduced to the community by its creator, Ryan Dahl on May 27, 2009.
- In 2011, windows version was released.
- Its presentation was received with a standing ovation.
- Since that time, "Node" has continued to evolve, with contributions from the community as well as the project's primary sponsor, cloud computing Company Joyent.

NodeJS Building Block

- At a high level, "Node" is comprised of three building blocks.



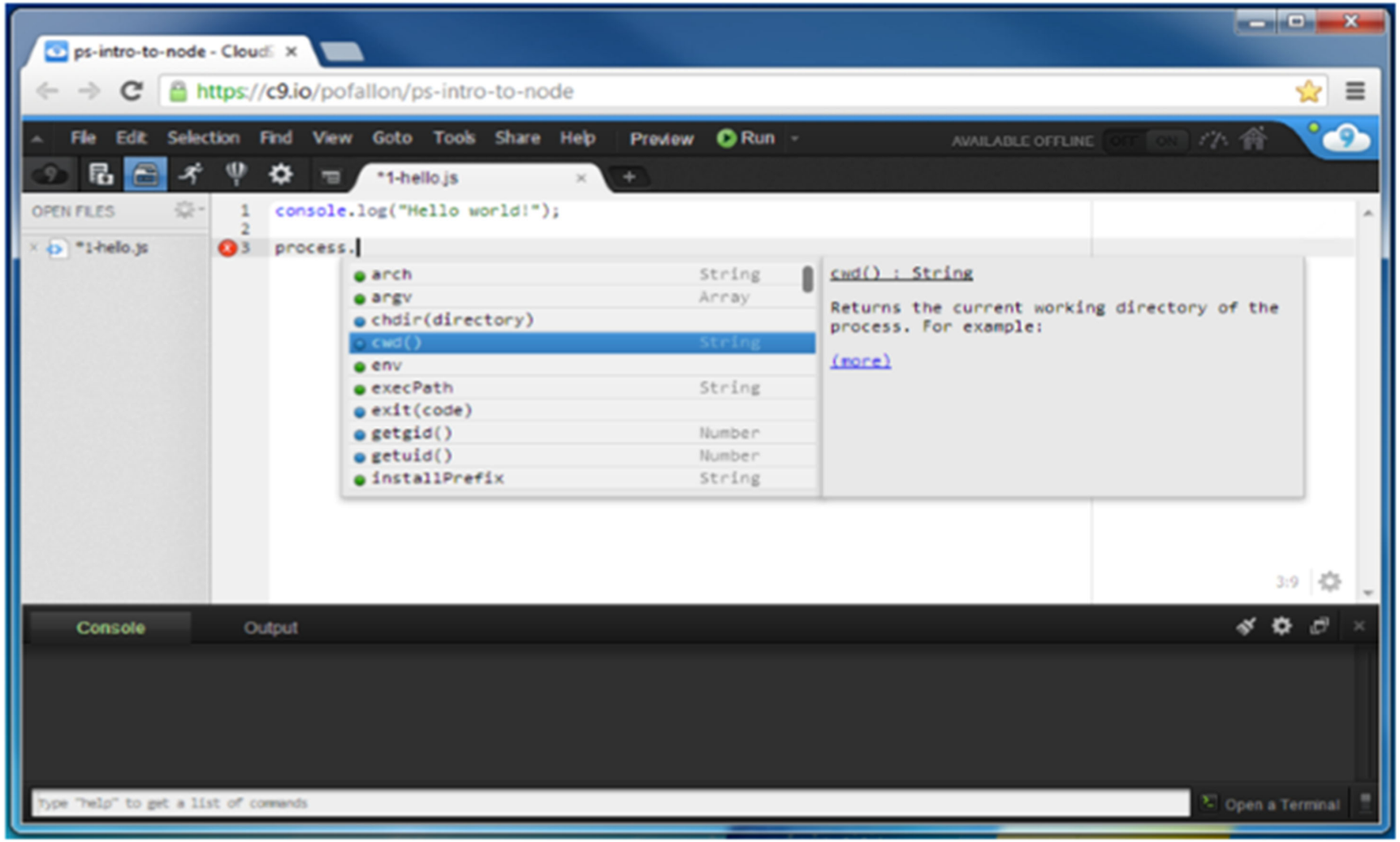
NodeJS Installation

http://nodejs.org/download/

- Installers available for Windows & Mac OS X
- Binaries available for Windows, Mac, Linux and SunOS
- Also available via many Linux package managers
- Source code also available

Developing for Node with Cloud9 IDE

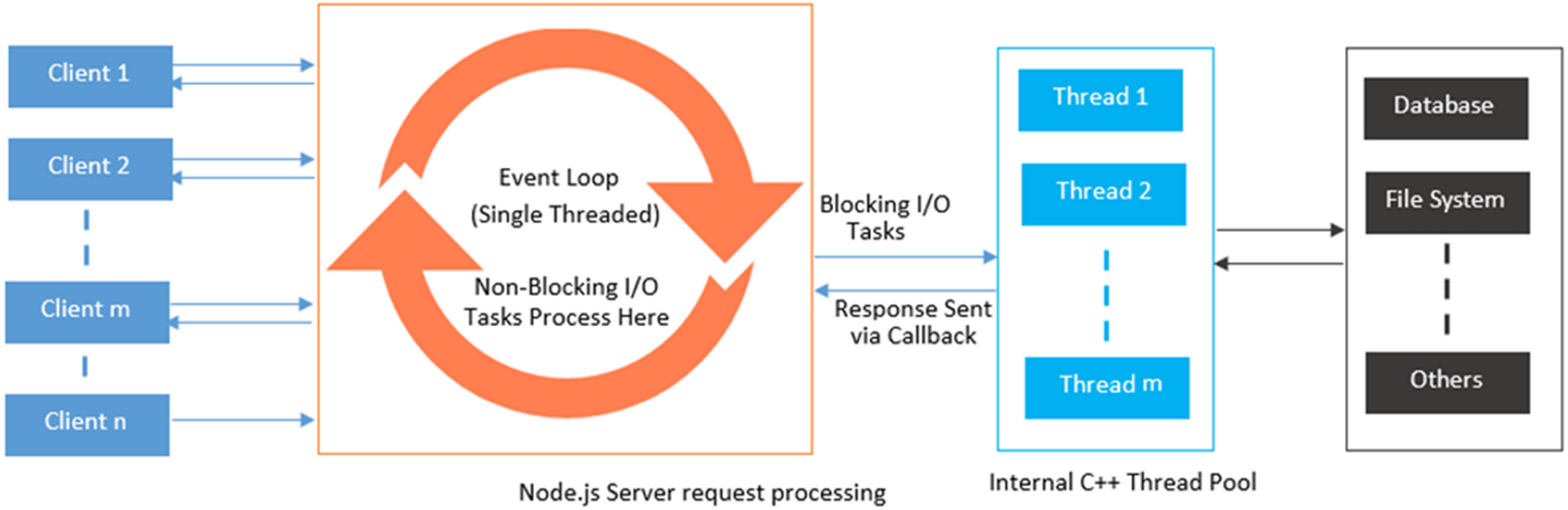
- We can develop Node.js applications using Cloud9 IDE via http://c9.io



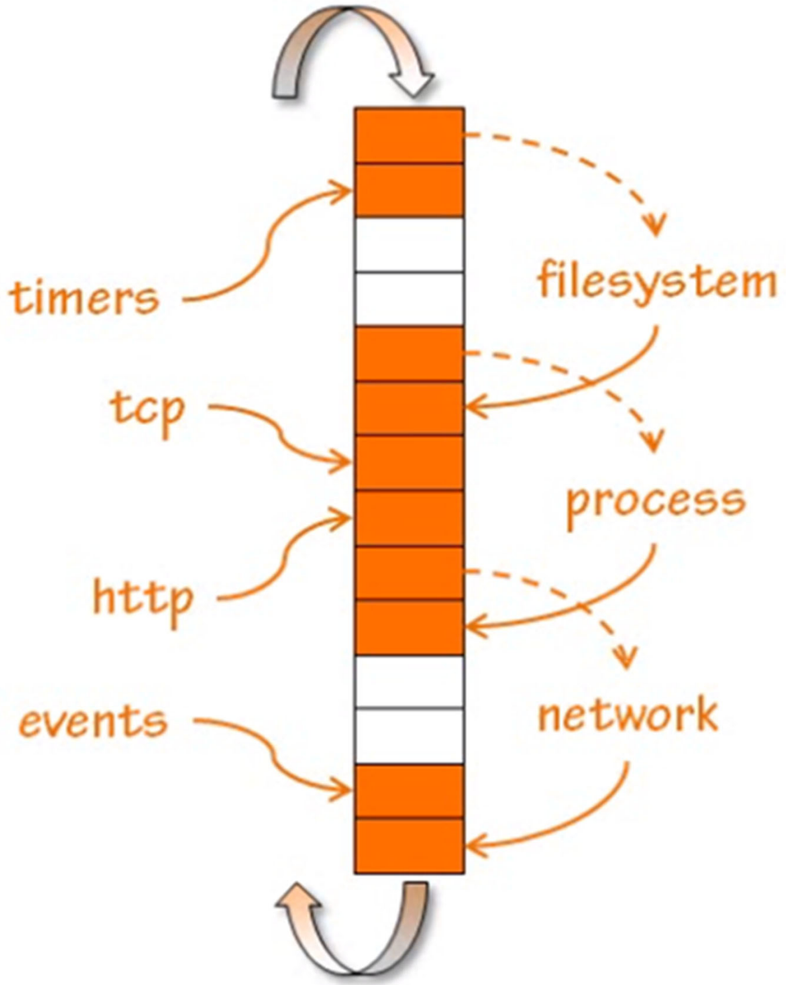
NodeJS vs. other Server-Side Frameworks

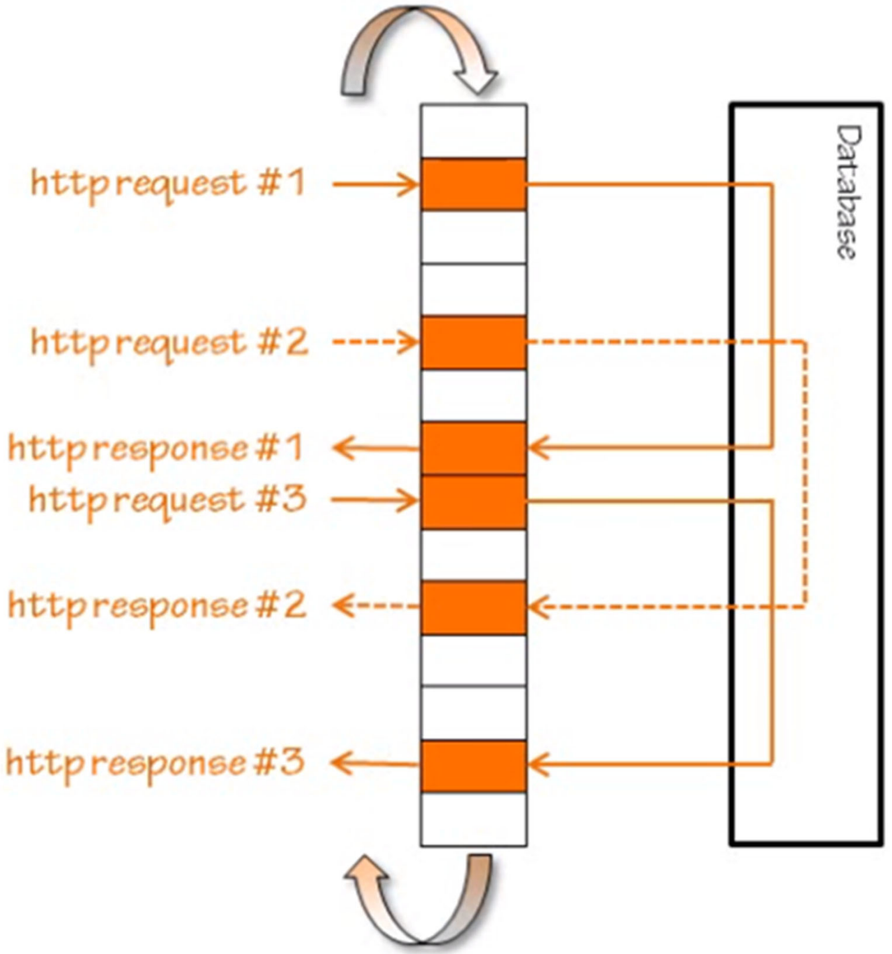
- Node.js is different from existing server-side frameworks.
- NodeJS is based on asynchronous events via JavaScript callback functionality
- It uses the JavaScript as a programming language.
- Moreover, everything inside Node.js runs in single thread.
- While existing server-side framework like ASP.NET, JSP etc. are based on multiple threads web servers.
- In multiple threads system, there is a limit of maximum number of threads, beyond which the throughput decreases.

NodeJS Processing

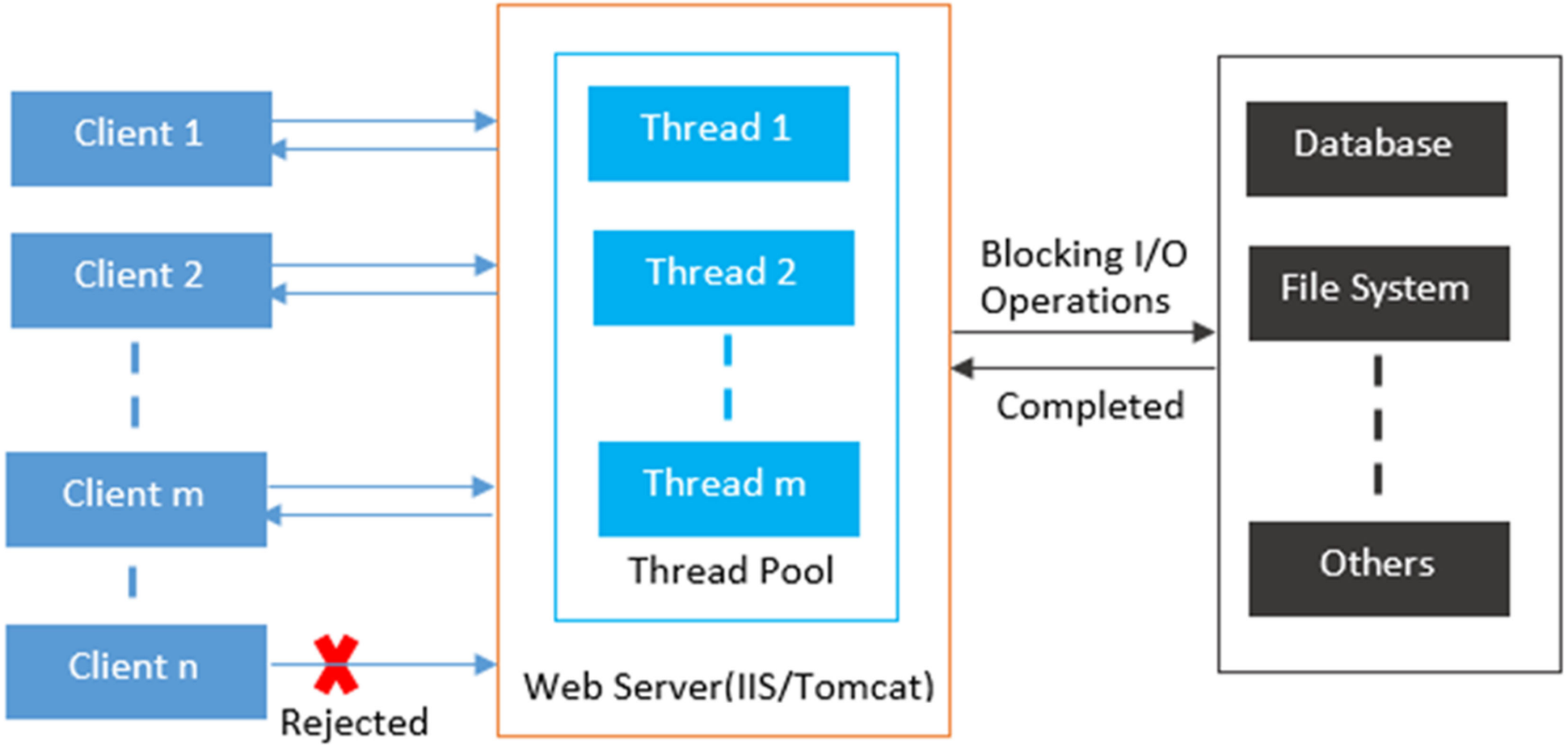


NodeJS Event Loop





Server-side Framework Processing



Multi-Threaded Web Server request processing

TYPICAL BLOCKING OR SYNCHRONOUS APPROACH

A typical approach

```
var conn = getDbConnection(connectionString);
var stmt = conn.createStatement();
var results = stmt.executeQuery(sqlQuery);
for (var i=0; i<results.length; i++) {
  // print results[i];
}
```

Non-Blocking or Asynchronous Approach

An asynchronous, “non-blocking” approach

```
getDbConnection(connectionString, function(err, conn) {
  conn.createStatement(function(err, stmt) {
    var results = stmt.executeQuery(sqlQuery);
    results.on('row', function(result) {
      // print result
    });
  });
});
```

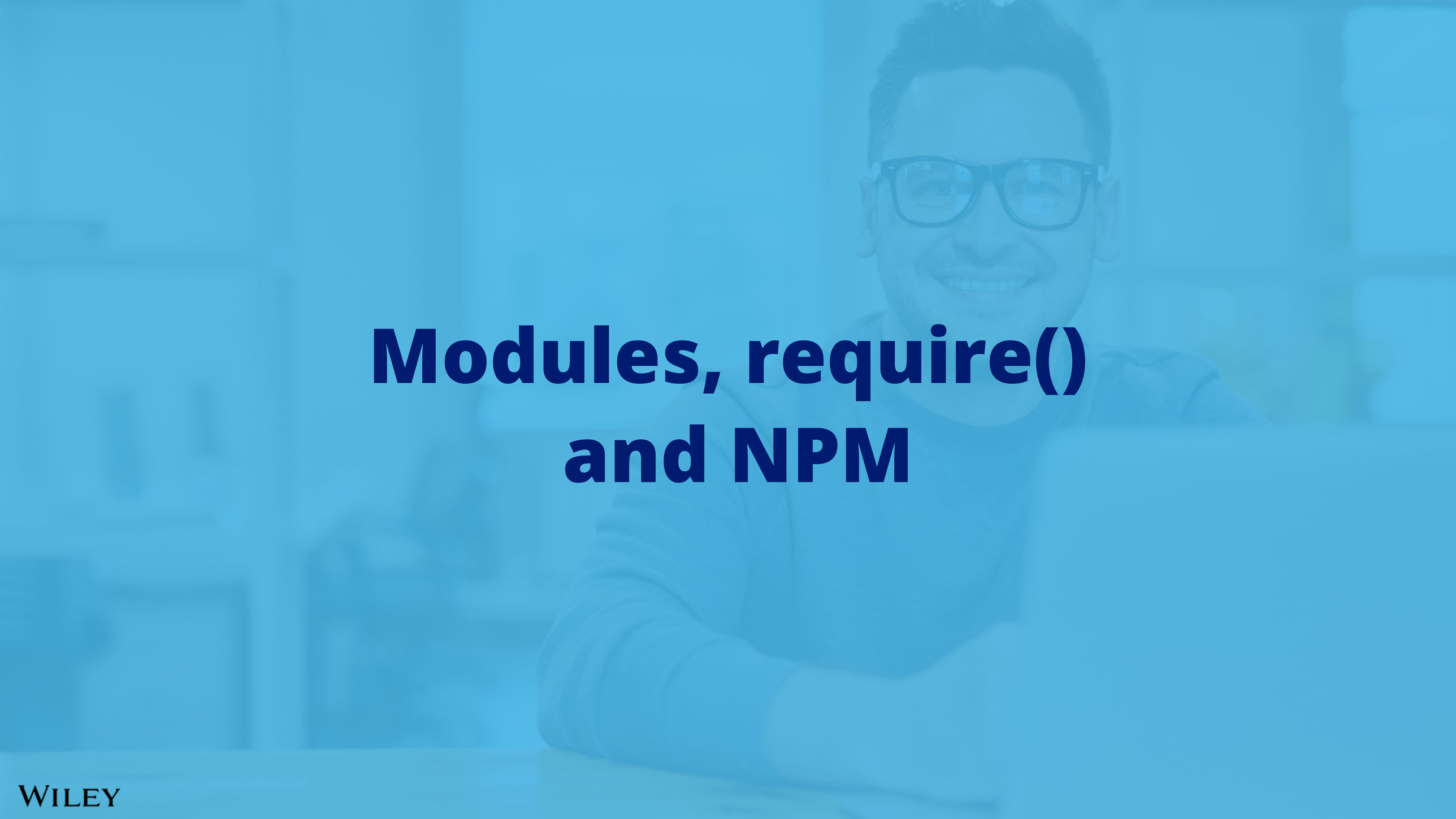
Writing Async Code with Callbacks

② Error is first parameter to callback function

```
var handleResults = function(error, results) {
  // if error is undefined…
  // do something with the results
}

getStuff(inputParam, handleResults);
```

① Callback is last parameter in async function call



Modules, require() and NPM

NodeJS Modules

- Module in NodeJS is a simple or complex functionality organized in single or multiple JavaScript files.
- NodeJS modules can be reused throughout the Node.js application.
- Each module in Node.js has its own context.
- So, it cannot interfere with other modules or pollute global scope.
- Each module can be placed in a separate .js file under a separate folder.
- Node.js implements CommonJS modules standard.

CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

Using Modules in your Application

- To use Node.js core or NPM modules, you first need to import it using require() function.
- The require() function will return an object, function, property or any other JavaScript type.

```
var foo = require('foo');
var Bar = require('bar');
var justOne = require('largeModule').justOne;

var f = 2 + foo.alpha;          <-- Modules can export variables
var b = foo.beta() * 3;         <-- ... including functions

var bar = new Bar();            <-- Modules may export objects

console.log(justOne());
```

NodeJS Module Types

NodeJS includes three types of modules

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

Core Modules

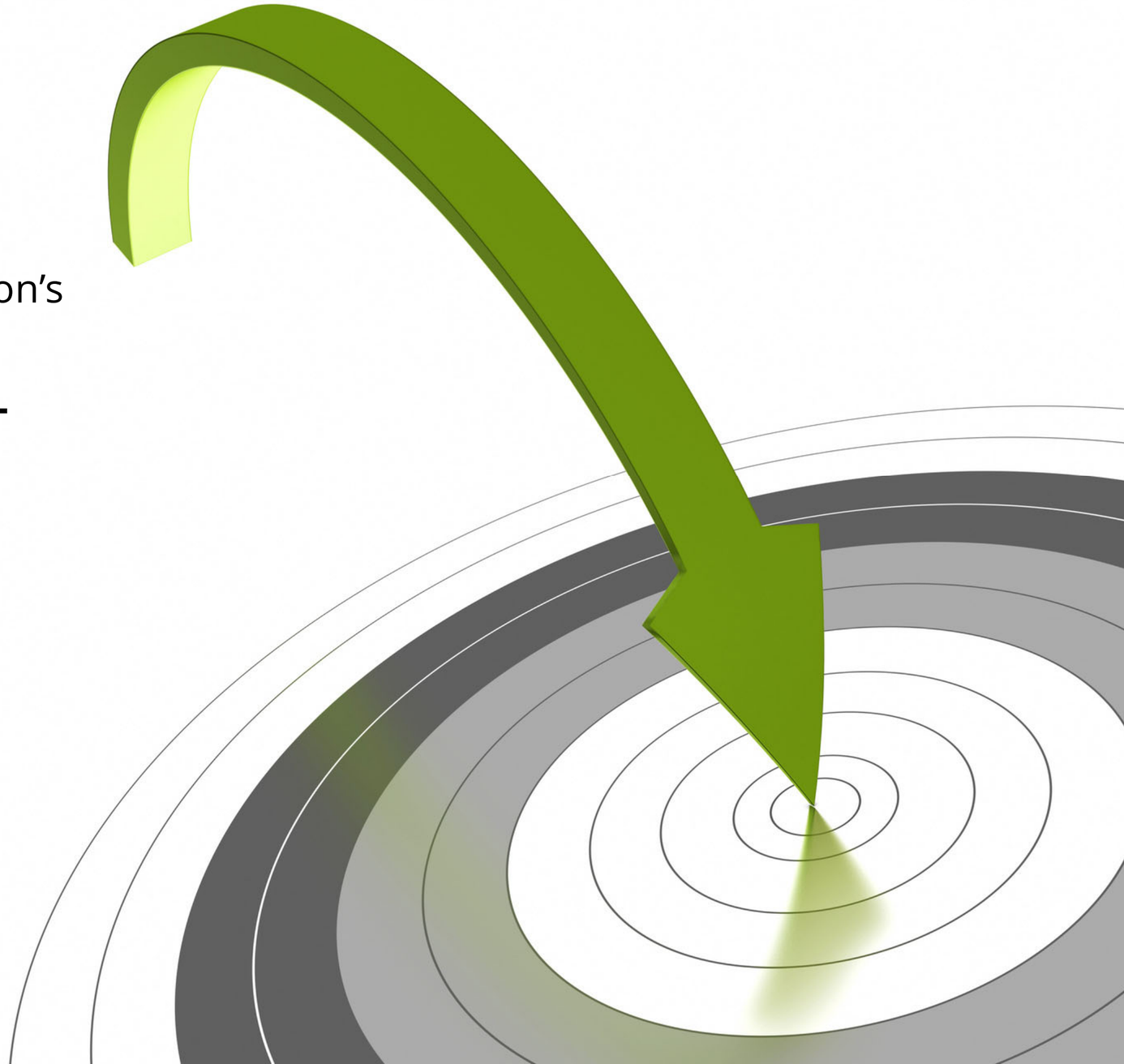
- Come pre-packaged with Node
- Are require() with a simple string identifier

  var fs = require('fs');
- The following table lists some of the important core modules in Node.js.

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

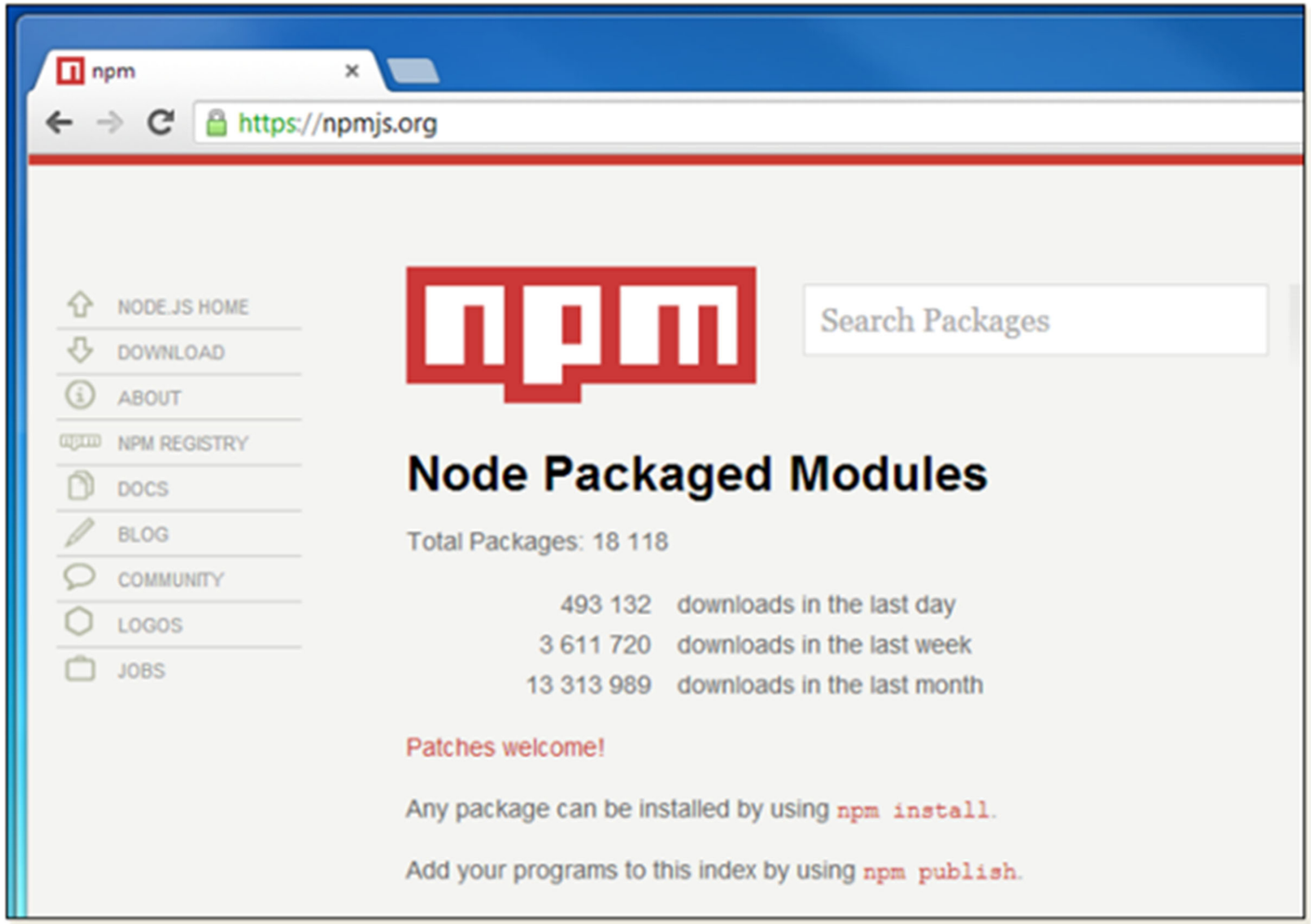
Local Modules

- Each .js file is its own module
- A great way to modularize your application's code
- **Each file is require()'d with file system-like semantics:**
  - –var data = require('./data');
  - – var foo = require('./other/foo');
  - –var bar = require('../lib/bar');
- **Single variable require() still valid:**
  - –var justOne = require('./data').justOne;



Third-Party Modules

Third Party Modules via Node Package Manager (NPM) registry.



THIRD-PARTY MODULES

- Third Party Modules via Node Package Manager (NPM) registry
- Installed via “npm install module_name” into “node_modules” folder
- Are require()’d via simple string identifiers, similar to built-ins
- –var request = require(‘request’);
- Can require() individual files from within a module, but be careful!
- –var BlobResult = require(‘azure/lib/services/blob/models/blobresult’);
- Some modules provide command line utilities as well
- Install these modules with “npm install –g module_name”
- –Examples include, express, mocha, azure-cli

Node Package Manager (NPM)

- Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages. It is also an online repository for open-source Node.js packages.
- The node community around the world creates useful modules and publishes them as packages.
- Official website: https://www.npmjs.com
- NPM is included with Node.js installation.

It has now become a popular package manager for other open-source JavaScript frameworks like AngularJS, jQuery, Gulp, Bower etc.

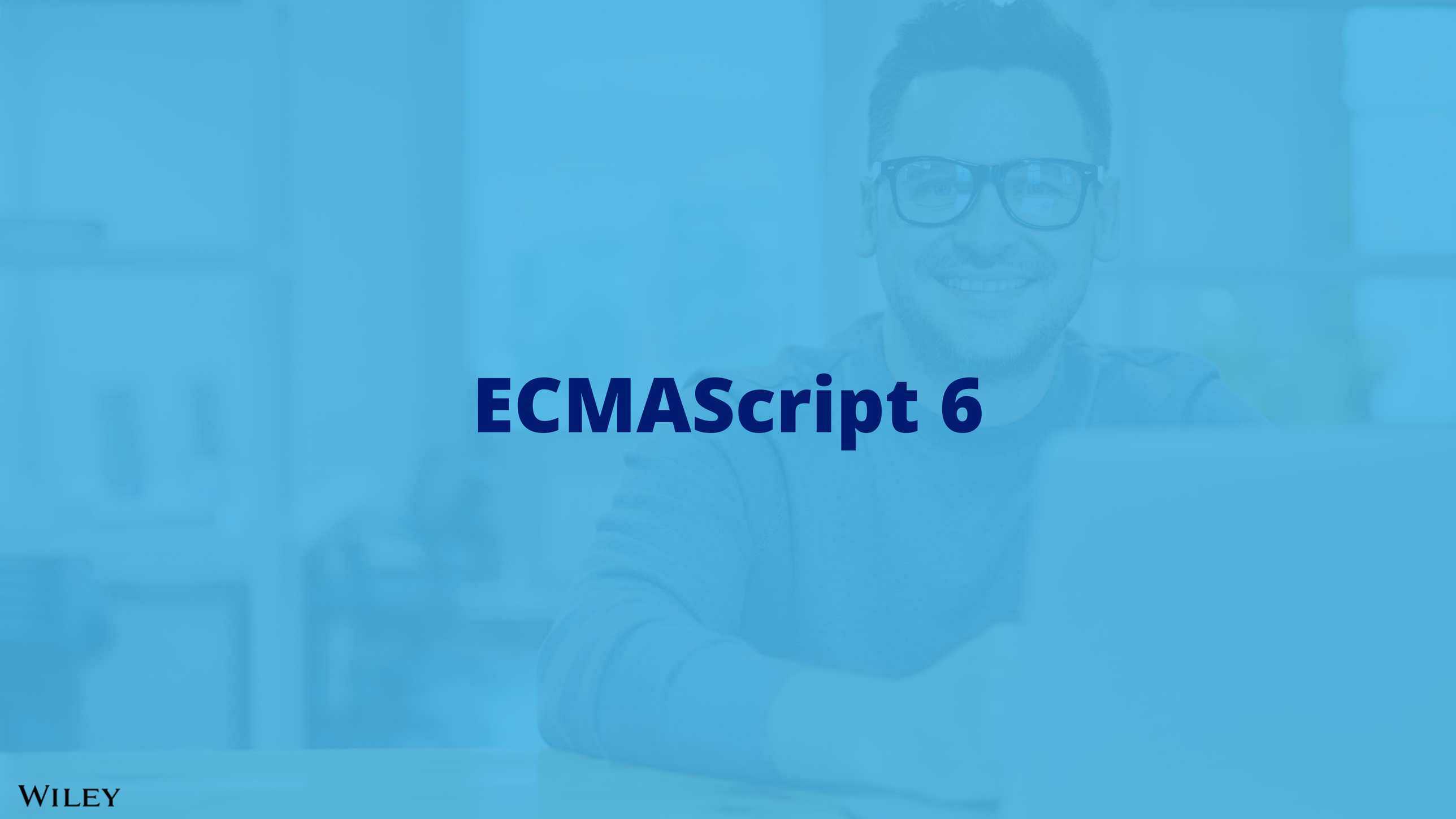
Introducing package.json

The package.json file is core to the Node.jsThe package.json is used as what equates to a manifest about applications, modules, packages, and more - it's a tool to that's used to make modern development streamlined, modular, and efficient

```
{
  “name” : “coolstuff”,    // required
  “version” : “0.0.1”,     // required
  “author” : “Paul O’Fallon”,
  “description” : “A cool module!”,
  “keywords” : [“cool”, “awesome”],
  “repository”: {
    “type” : “git”,
    “url” : “https://github.com/pofallon/coolstuff.git”
  },
  “dependencies” : {
    “underscore” : “1.4.x”,
    “request” : “>=2.1.0”,
  },
  “main” : “lib/cool.js”
}
```

PUBLISHING YOUR OWN MODULE

- package.json (located in your project root)
- “npm publish .” (from within project root)
- “npm install module_name” (from an empty directory) – verify it!



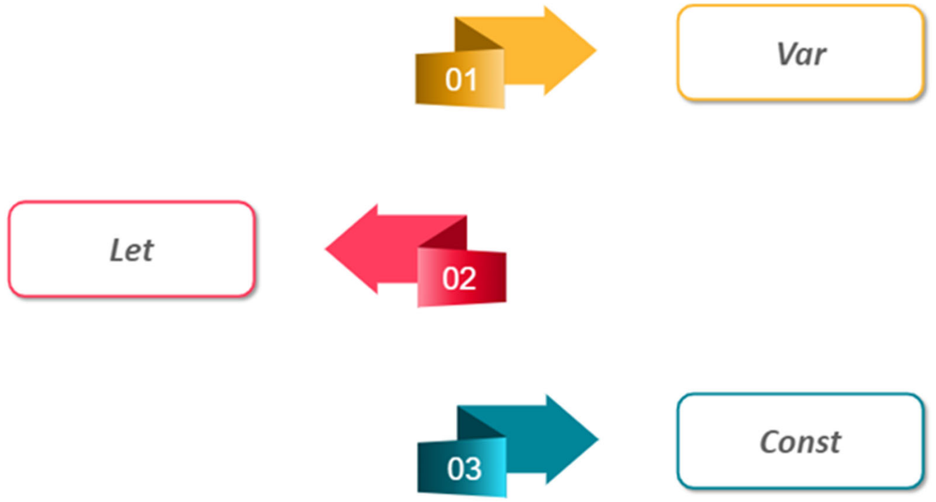
ECMAScript 6

WILEY

ECMASCRIPT VARIABLES

Variables are the defined spaces in memory, to store values in program.

JavaScript supports **dynamic typing.** This means that a JavaScript Variable can hold value of any data type. The majorly used EcmaScript variable types are:



EcmaScript Variables: Var

ES5 makes use of var keyword and it was only way to declare the variables JavaScript, but it had certain issues like:

It has a function scope that is if you want to restrict the access of variables within the functions it should be defined inside the function, otherwise it will belong to the global scope.

Here variables can be declared more than one time and their values can be re-assigned anytime in the same project.

This leads to wrap the code in a function whenever we need to create a new scope.

EcmaScript Variables: VAR (Example)

Input Screen:

```
var x = 10
function test() {
var x = 100
console.log("value of x as per defined function "+x)
}
console.log("value of x before declaration of function "+x)
test()
```

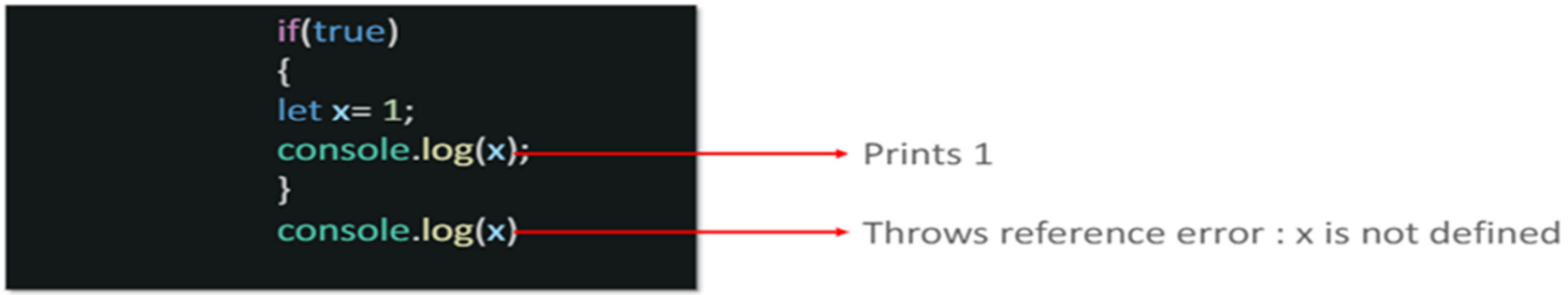
Declaring variable globally

Declaring variable locally

EcmaScript Variables: Let

- ES6 introduced variables let and const which provide Block Scope
- Block Scope means any variables declared within curly braces {} like:
- functions, if-statement, or loops remain available only in this scope.
- With let, you can re-assign the value of the variable any time but you cannot declare the variable more than one time.

Example:



EcmaScript Variable: Let

Difference between var and let.

```
var x = 10;
var x = 20;
console.log(x);
```

```
PS C:\Users\archana\Desktop\Node.js\Node.js-Demo\firstapp> node ES.js
20
```

```
let x = 10;
let x = 20;
console.log(x);
```

```
PS C:\Users\archana\Desktop\Node.js\Node.js-Demo\firstapp> node ES.js
C:\Users\archana\Desktop\Node.js\Node.js-Demo\firstapp\ES.js:2
let x = 20;
    ^

SyntaxError: Identifier 'x' has already been declared
```

Hence, any variable declared using the let keyword is assigned the block space.

EcmaScript Variables: Const

The value assigned to a constant variable is **immutable.**

Value must be assign to a const variable at the time of declaration and it later can not be reassign or re-declared within the same program. In case of declaring a new array using const variable, you can change the value of an element within the array but u cannot change the value of the entire array.

Example:

```
const a =0
const a = 1
const b = [1,2]
b.push(3);
b[3] = 4
console.log(a,b)
```

Assigns value ‘0’to constant a

TypeError: Assignment to constant variable

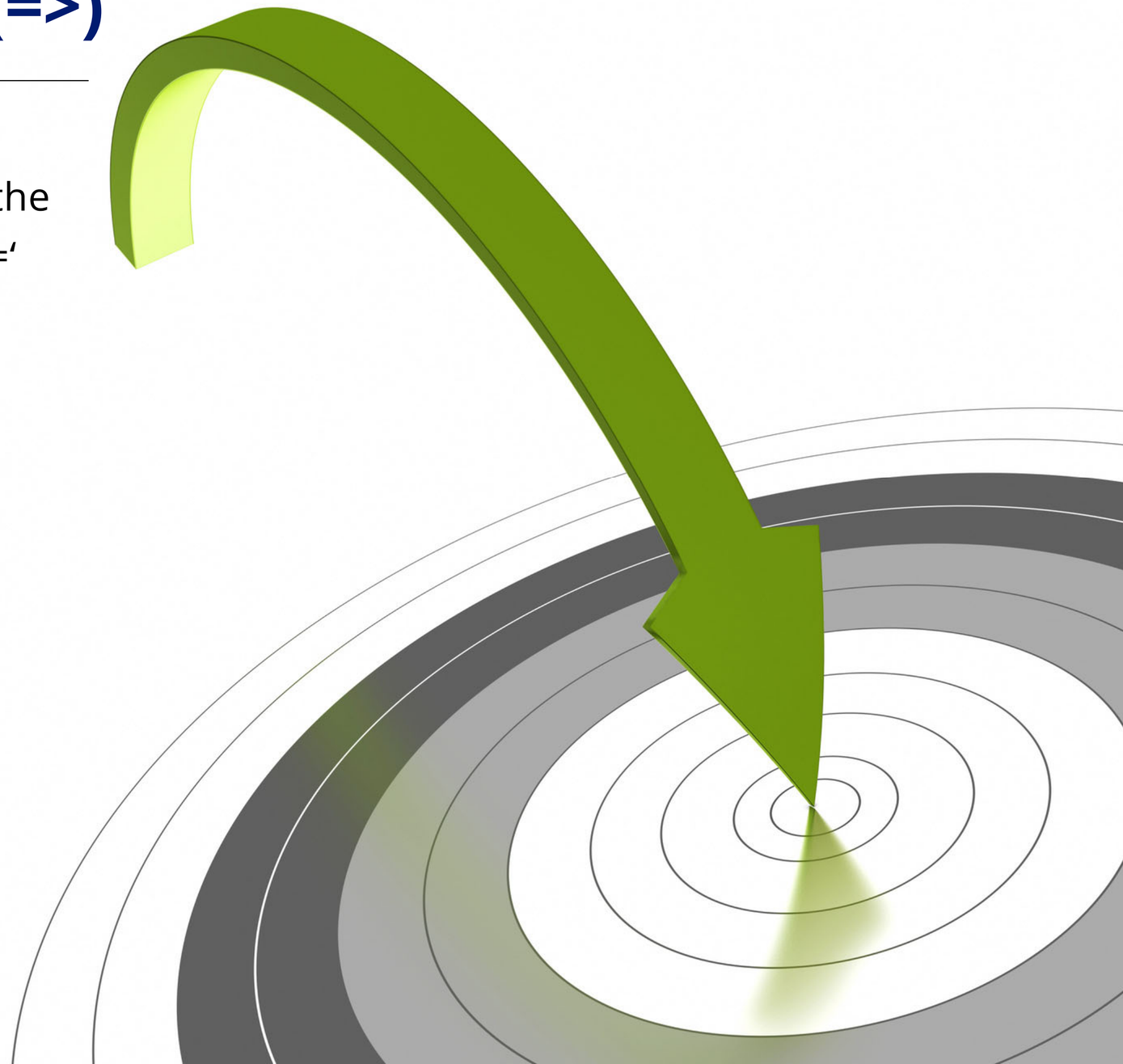
Sets b = [1,2,3]

Sets b = [1,2,3,4]

ES6 Vs ES5: Arrow Function (=>)

- **Arrow functions** are the functions where you don’t have to use curly brackets, neither type the **function** keyword. They utilize a new token, ‘.=‘

  Syntax: (parameters)=>{statements}



ES6 Vs ES5: Arrow Function (=>)

Arrow function can be majorly used in following cases:

Return Number function

Return Array function

Return Number function with parameter

Return Object function

ES6 Vs ES5: Module Exports and Imports

- This Syntax is useful when we export a module by default and we have to import that whole module in our source file.
- So, ES6 also provides us with an ability to export and import multiple variables from a single module.
- So, in your module file you will export your module:

```
export const a = 1;
export const b = 2;
```

- And import them all together:

```
Import {a,b} from'./testmodule';
```

Export module

```
var testModule = { a: 1, b: 2 };
```

ES5: module.exports = testModule;

ES6: export default testModule;

Import module

ES5:

```
var testModule = require(./testModule)
```

ES6:

```
import testModule from './testModule';
```

Thank You