

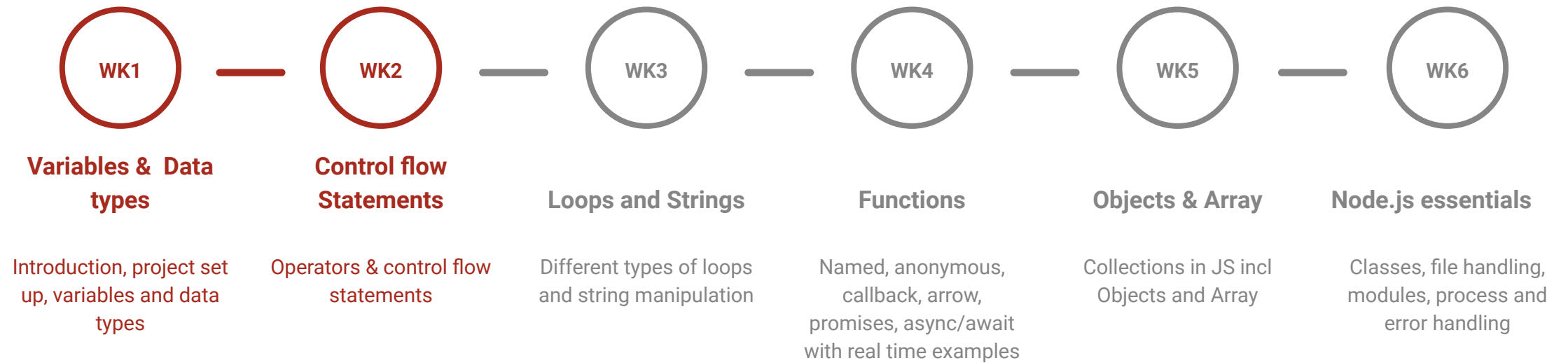
# JavaScript & Node.js Essentials for Test Automation

Cope Automation

What will you learn?

# Course Content - 6 weeks

---



# JavaScript Overview

# JavaScript Overview

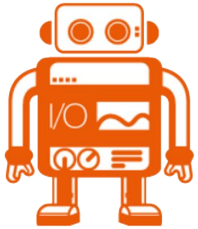
---

- JavaScript(JS) is a **cross-platform, dynamic** scripting language used to make **webpages** interactive
- Node.js is an open-source, cross-platform, back-end **JavaScript runtime environment** that runs on the V8 engine and executes JavaScript code **outside a web browser**.
- It was invented by **Brendan Eich** (co-founder of the Mozilla project)
- JavaScript has a **prototype-based object model** instead of the more common class-based object model
- JavaScript is standardized at **ECMA** International (European Computer Manufacturers Association)
- Celebrating its **25th** years
- JavaScript is **case-sensitive** (**Apple** is not same as **apple**)

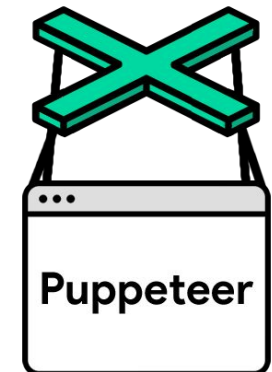
# Why Learn JavaScript?

# 1. Variety of Testing tools and frameworks

---

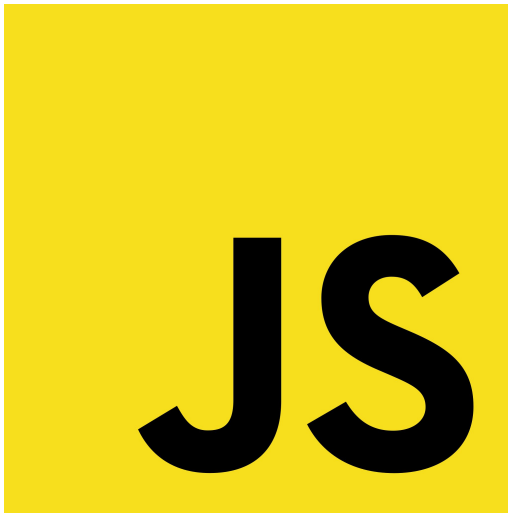


**cypress.io**

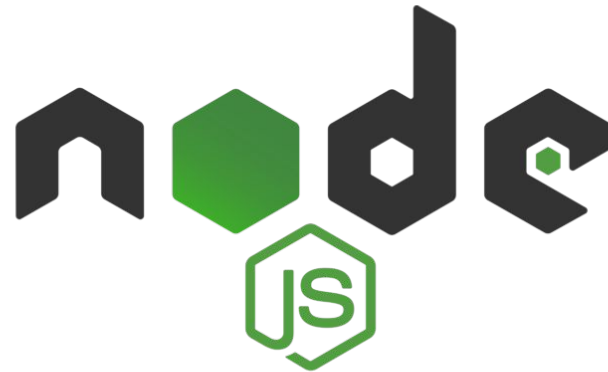


## 2. From Web development language to full stack programming language

---

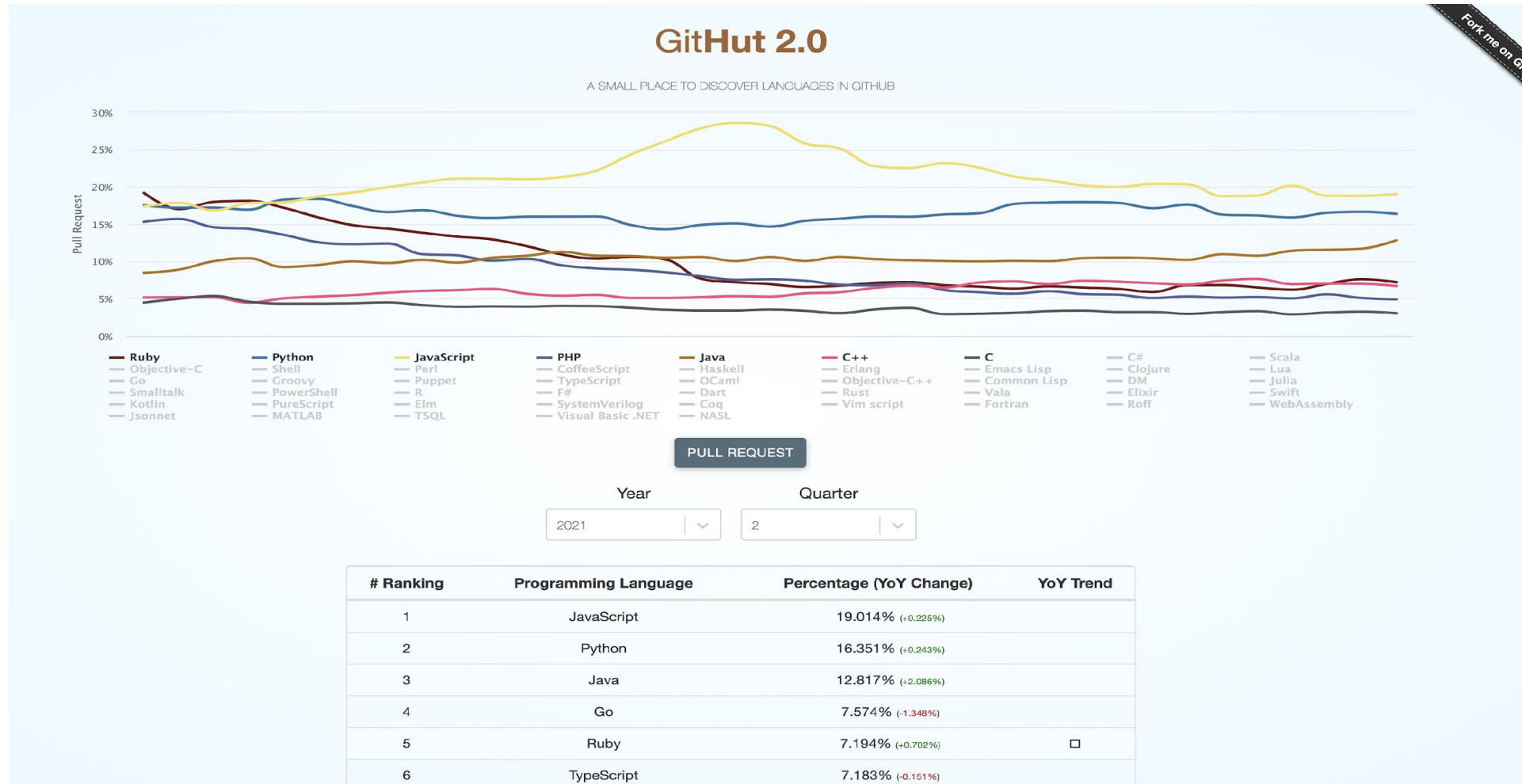


+





### 3. Leading modern programming language



Setup node project

# First node project

---

1. Create folder under `Documents/workspace/{project_name}`
2. Initialize a node project
  - a. `npm init [-y]`
  - b. Check if `package.json` file is created
3. Run a Hello World program

Comments

# Comments

---

1. What is a comment?
  - a. Comment provides a way to explain the code (e.g as a header for a function)
  - b. Comments are ignored by the interpreter
2. How many types of comments we have in JS?
  - a. Single line
  - b. Multi-line
  - c. Hashbank
3. What's its syntax?
  - a. `// <comment>`
  - b. `/* <comment(s)> */`
  - c. `#!`

# Variables

# Variable Definition and Syntax

---

1. What is a variable?
  - a. Variable is an **identifier/name** that **stores value/data**
2. What's its syntax?

```
{ var | let | const } {variableName} [= value]
```

3. How do you read the above syntax?
  - a. Declare a variable and initialize it with a value

# Variable Rules

---

1. Variable name should start with a letter, or underscore ( ), or dollar sign (\$)
2. System reserved keywords can't be used e.g. if, else, throw, break
3. Variable names are case-sensitive
4. Can't have same name for other types (e.g. function name, object name)
5. Multiple variables can be declared in single line (separated by comma)
6. When you just declare a variable, the data type will be undefined



# var, let and const

---

| # | Statement    | Definition   | Comments  |
|---|--------------|--|---|
| 1 | <b>var</b>   | Declares a variable, optionally initializing it to a value                     | This was the only var declaration statement before ES6. In ES6 (ECMAScript2015), let and const are introduced |
| 2 | <b>let</b>   | Declares a block-scoped, local variable, optionally initializing it to a value | * Preferred wherever possible   |
| 3 | <b>const</b> | Declares a block-scoped, read-only named constant.                             |   |

## Rules:

1. **const** variable must be initialized with a value
2. **const** variable can not be re-assigned
3. Generally, **const** variables will be capitalized

# Variable Scoping

---

- I. Why do you need to use with `let` or `const` over `var`?
  - a. Because `let` and `const` support block scoping

# Data types

# Data types

---

1. How many data types available in JavaScript?
  - a. A total of **8** types where **7** are primitives
2. What are the 8 data types?
  - a. string (e.g. "Hello")
  - b. number (e.g. 5, 10.5, 20)
  - c. boolean (true/false)
  - d. undefined
  - e. null
  - f. bigint
  - g. symbol (data type whose instances are unique and immutable)
  - h. Object type (Object, Array, Date..etc)

# Data types literals and its notations

---

1. string
  - a. Single quote, double quotes, template literals (`${}`)
2. number
  - a. Integer or floating points
3. boolean (true/false)
4. Object
  - a. `{}`
5. Array
  - a. `[]`
6. Regexp
  - a. `//`
7. Undefined and null

# typeof operator and its use cases

---

1. What's use of typeof Operator?
  - a. To understand a data type of JS components
2. What's its syntax?
  - a. `typeof <any_valid_JS_type>`
  - b. `typeof(any_valid_JS_type)`

# Different forms of data

---

1. Literals
2. Variable (Assigned to a variable)
3. Expression (Evaluates to, mostly using operators or return statement)

# Truthy, falsy and nullish values

---

## 1. What are the falsy values?

- a. false
- b. undefined
- c. null
- d. 0
- e. NaN
- f. "" (empty string)

## 2. Boolean and non-boolean context of nullish value

a.

| Type      | Boolean context (e.g. if condition) | Numeric context |
|-----------|-------------------------------------|-----------------|
| undefined | false                               | NaN             |
| null      | false                               | 0               |



# Data types conversions

---

- I. The following inbuilt functions can be used to convert data types
  - a. `parseInt()`
  - b. `parseFloat()`
  - c. unary plus operator
  - d. `toString()` method

# Data type - Summary points

---

1. A total of 8 data types where 7 primitives and 1 object type
2. `typeof` operator can be used to check a data type of variable/expression
3. The `typeof` operator returns a string value starting with lowercase
4. A data be represented in three forms: *Literals, variables and expression*
5. Following 6 values will be considered falsy in a boolean context `false`, `undefined`, `null`, `0`, `NaN` and empty string
6. `parseInt()`, `parseFloat()`, `.toString()` method can be used to convert datatype
7. Be familiar with these notations: `{}`, `[]`, `/abc/`, `""`
8. All primitive data types are immutable
9. Why do you need to stick with `let` or `const`?
  - a. Because `let` and `const` support block scoping
10. Except for 'undefined' and 'null', every other 5 primitives have wrapper object types (starts with uppercase)

# Operators

# Operators - Intro and types

---

- I. Key things to note when using an operator
  - a. Operator: Understand the notation and syntax
  - b. Operands
  - c. Returned value

| # | Operator type | Examples         |
|---|---------------|------------------|
| 1 | Unary         | typeof "Hello"   |
| 2 | Binary        | $x + y$          |
| 3 | Ternary       | isTrue ? Yes: No |

# How many ? & What are the most used ones?

---

| #  | Operator type | Operators                      | Examples                         |
|----|---------------|--------------------------------|----------------------------------|
| 1  | Assignment    | =, +=, -=,                     | let count = 1                    |
| 2  | Arithmetic    | +, -, *, **, /, %, ++, --      | for (let i = 1; i <= 5, i++) { } |
| 3  | Comparison    | ==, ===, !=, !==, >, <, <=, >= |                                  |
| 4  | Logical       | &&,   , !                      |                                  |
| 5  | String        | +                              |                                  |
| 6  | Ternary       | ?                              |                                  |
| 7  | Unary         | typeof, delete, void           |                                  |
| 8  | Relational    | in, instanceof                 | Returns a boolean                |
| 9  | Comma         |                                |                                  |
| 10 | Bitwise       |                                |                                  |

Strict mode

# Strict mode

---

1. What's strict mode?
  - a. Strict mode prevents semantics(syntax related) error in JavaScript
2. What's the use case?
  - a. Prevents any undeclared global variable
  - b. Use of implements, interface, let, package, private, protected, public, static, and yield as identifiers
3. How to turn on strict mode?
  - a. “use strict”

# Conditional Statements (If...else & switch case)



# How do you read these?

---

[ Example 1 ]


```
If (status = "PASS") {}
```

[ Example 2 ]

```
If (status === "PASS") {}
```

# Syntax

---



```
if (condition) {  
    statement_1;  
} else {  
    statement_2;  
}
```

Evaluates to true

```
if (condition_1) {  
    statement_1;  
} else if (condition_2) {  
    statement_2;  
} else if (condition_n) {  
    statement_n;  
} else {  
    statement_last;  
}
```

## Falsy values

The following values evaluate to `false` (also known as [Falsy](#) values):

- `false`
- `undefined`
- `null`
- `0`
- `NaN`
- the empty string ( `"` )

All other values—including all objects—evaluate to `true` when passed to a conditional statement.

# Switch case [break, default]

---

## 1. What does switch case do?

- a. It tries to match given expression with case labels and executes the program

## 2. What's the syntax?

- a. A `switch` statement looks like this:

```
switch (expression) {  
    case label_1:  
        statements_1  
        [break;]  
    case label_2:  
        statements_2  
        [break;]  
    ...  
    default:  
        statements_def  
        [break;]  
}
```

# Switch case

---

1. When to use switch case over if..else?
  - a. When the inputs/cases are known (useful for a predefined set of cases)
2. What is the difference between if..else and switch case
  - a. If..else requires expression to evaluate to true/false but not for switch...case
3. Rules:
  - a. Remember to include `break` statement at each case block
  - b. `default` statement will be executed if no previous cases are matched
  - c. `default` is optional and generally written at the end of all cases
  - d. Remember these four statements: `switch`, `case`, `break` and `default`

# Loops

# Loops Introduction

---

1. What are loops used for?
  - a. To iterate over a collection (more than one element)
2. What are the basic use cases?
  - a. To find a SPECIFIC element/member from a collection
  - b. Do something with ALL elements
3. General rules
  - a. Loop should end - check condition
  - b. If you initiate a variable inside a loop, it reset for every iteration

# Different types of loops

---

[ (Standard) for loop ]

[ forEach ]

[ while ]

[ do while ]

[ for in ]

[ for of ]

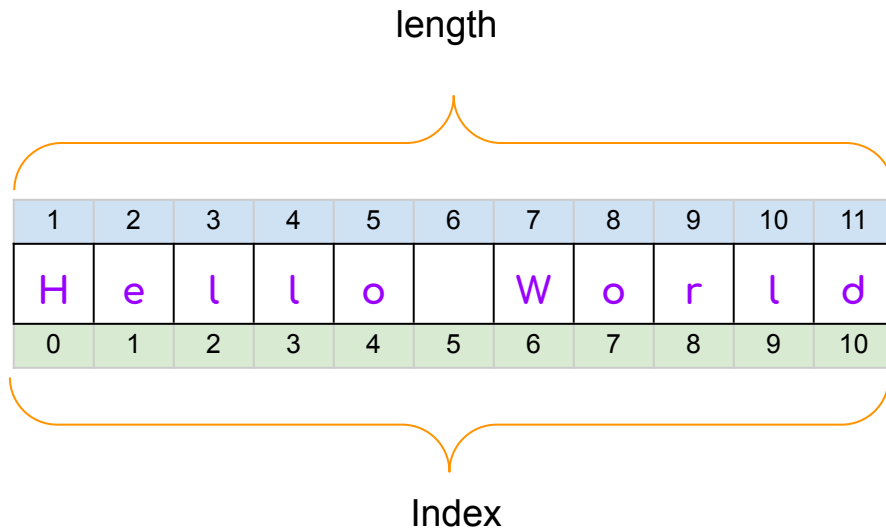
# String Manipulation



# String - Basics

---

1. What is a string?
  - a. It is a sequence of characters (text value)
2. What are the core characteristics of a string?



## General rules:

1. Index starts from 0; each char can be accessed by knowing its index
2. `length` returns the total number of characters

# Use of backslash

# Use of backslash (\) in string

---

- I. What is backslash is called?
  - a. Escape character
2. What are the rules?
  - a. When the interpreter hits the backslash in string, it looks for the next character
    - i. If next char is a letter and has a special meaning(e.g. tab, new line), then it does the special thing
    - ii. If the next char is a letter and does not mean anything, the \ is skipped from the string
    - iii. If the next char is a special char( ' , " , \ ), then the special char will be included in the string

# What are those special chars and its meaning?

---

| Character | Meaning                    |
|-----------|----------------------------|
| \b        | Backspace                  |
| \f        | Form feed                  |
| \n        | New line                   |
| \r        | Carriage return            |
| \t        | Tab                        |
| \v        | Vertical tab               |
| \'        | Apostrophe or single quote |
| \"        | Double quote               |
| \\        | Backslash character        |

# Functions

# Function - Introduction

---

## 1. What is a function?

- a. A set of statements to perform a task or calculate a value - similar to a procedure but function takes input and returns an output

## 2. What are the two main stages of a function?

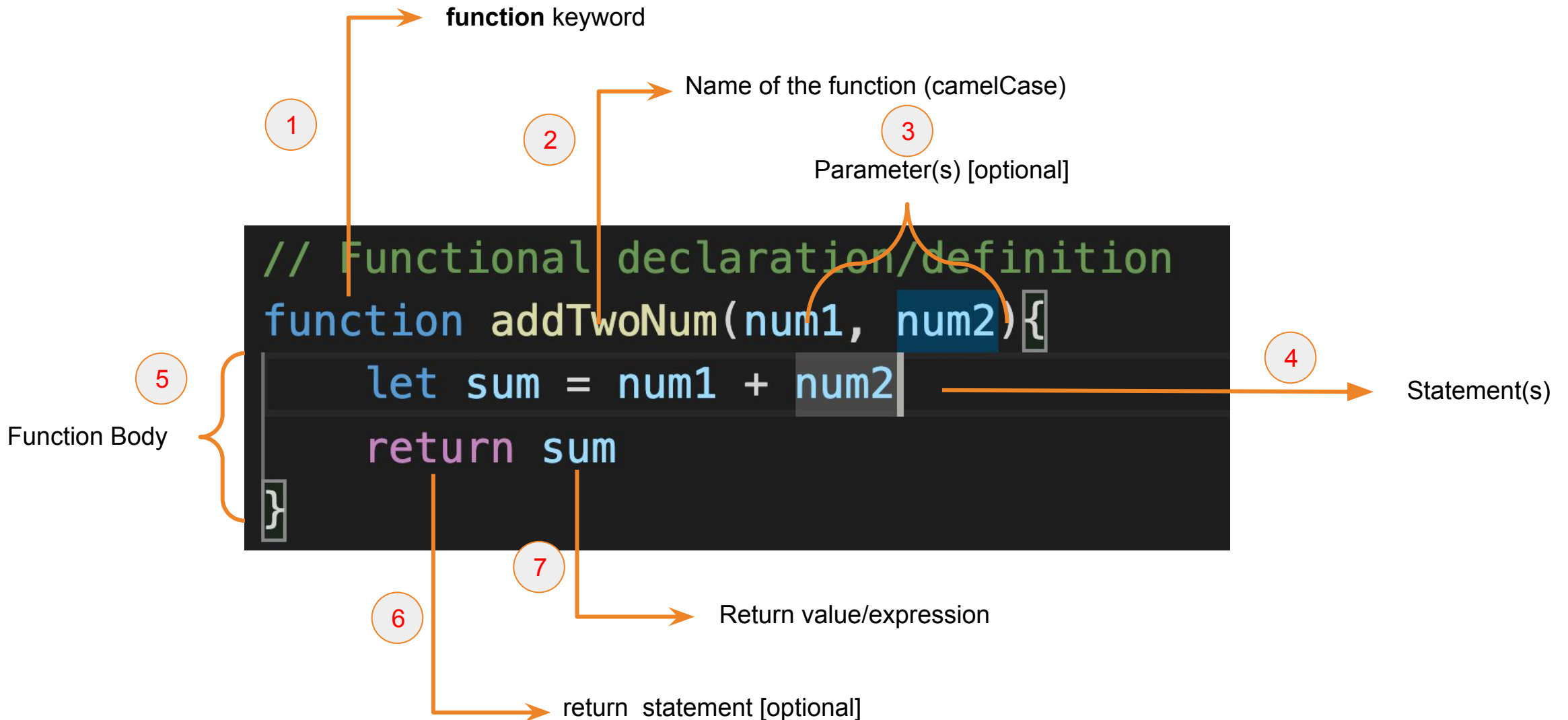
- a. Declare/define
- b. Calling

## 3. What are the two main types of functions?

- a. Named function (aka function declaration/definition)
- b. Anonymous function (aka function expression)
  - i. Callback function, arrow function

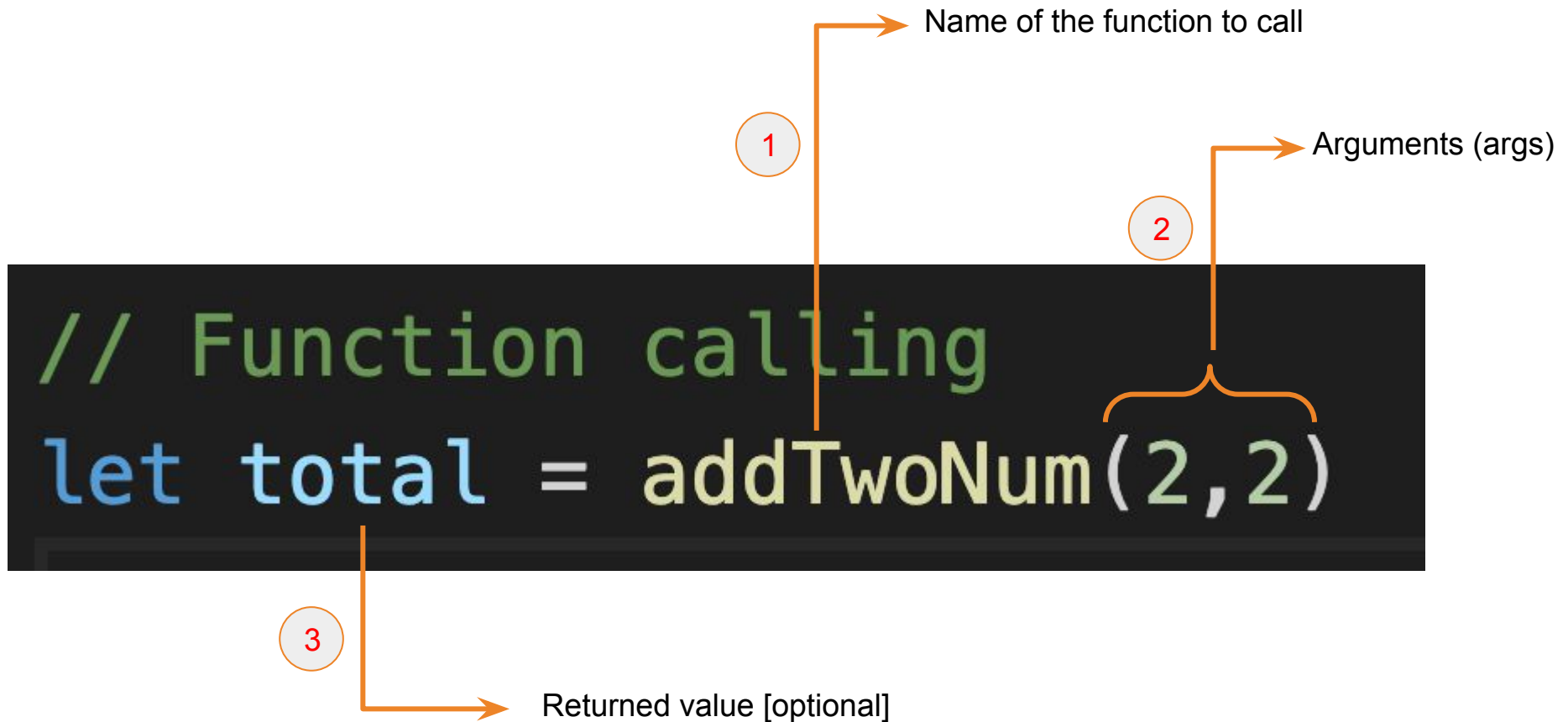
# Function definition - Basic Syntax

---



# Function calling - Basic Syntax

---





# General notes on function...

---

## 1. Parameter Vs Argument?

- a. When you define a function the required values are called as parameter(s) (aka params) and when the function is called with actual values, then the actual values are called as argument(s) (aka args)

## 2. Remember these points

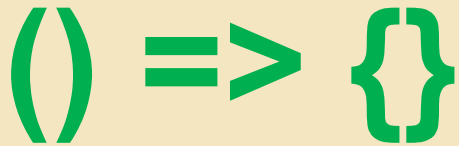
- a. You can provide any type of data as a function arg (no checking is done)
- b. You can provide more or less args when calling a function - no check is performed on # number of parameter(s) Vs Number of args received

[Fat] Arrow( $\Rightarrow$ ) Functions

# Arrow function

---

1. What's [fat] arrow function?
  - a. It's a shorter version of function expression
2. What's the real time use case?
  - a. Used mostly in callback context
3. What's its basic syntax?



The diagram illustrates the basic syntax transformation of an arrow function. It shows a green opening parenthesis '(', followed by a green equals sign '=' and a green greater-than sign '>' combined as '=>', and finally a green closing curly brace '}'.

# Arrow function rules

---

## I. What are the rules?

- a. No “function” keyword
- b. `=>` is must and left and right of arrow is context-driven
- c. Arrow function can be called when it's assigned it to a variable
- d. No binding of *this* keyword
- e. Rules around syntax

| # | Notations          | Should present             | Can skip                       |
|---|--------------------|----------------------------|--------------------------------|
| 1 | ()                 | If zero or more param      | If single param                |
| 2 | <code>=&gt;</code> | Always present             |                                |
| 3 | { }                | More than one statement    | If single statement/expression |
| 4 | return             | If more than one statement | If just single statement       |

# Callback functions

# Recap...

---

1. What's node.js used for?
  - a. To run JavaScript outside of browser. It is a run-time environment for JS
2. What are two functional purposes of JS?
  - a. Client side JS - for building website (HTML + CSS + JS)
  - b. Server side scripting - Node.js
3. What will be our use case - building web or using the language for testing?
  - a. Testing
4. What confirms whether a project is a Node.js project?
  - a. The present of package.json file in the root
5. Can we execute tests like the way, the browser was loading ?
  - a. No, that's where we need to know how to handle async operations and make it work in Sync way

# Basic understanding of Node.js

---



1. It is an asynchronous(eventual completion of a task) platform
2. Single threaded
3. Non-blocking I/O (runs in parallel)

# General understanding of Node.js

---

## 1. Synchronous Vs Asynchronous

- a. Synchronous executes one after the other(line-by-line)
- b. Asynchronous runs them parallel

## 2. What's an example of a async actions

- a. Getting an API response
- b. DB operations
- c. I/O file operations



## Now what is the callback function ?

---

*“A function passed as an argument to another function is called as callback function”*

*In other words, A function within a function as an argument*

Two functions here:

- Higher order function/main function
- Callback function (function expression)

# Callback functions - Summary

---

1. Main function executes the callback function after the eventual completion of a task/at some point in time and optionally it can **ATTACH** some processed value
2. The main purpose of callback function would be to **RECEIVE** some **PROCESSED** value [as a result of an async action like API, DB, Files IO]
3. [Mostly] From automation point of view, we will be the consumer of the callback functions
4. The advance coding style of callback function are: Promises or Async/Await
5. How do you know whether a function accepts a callback function as its argument?
  - a. By referring to the docs/function signature
6. While most of the callback functions are “Async” in nature, but not all of them

# Promises


# Why Promises?

---

*Callback Hell*

*Pyramid of Doom*

*Christmas Tree*



```
4 loadLink(win, REMOTE_SRC+ '/assets/css/style.css', function() {
5   loadLink(win, REMOTE_SRC+ '/lib/async.js', function() {
6     loadLink(win, REMOTE_SRC+ '/lib/easyXDM.js', function() {
7       loadLink(win, REMOTE_SRC+ '/lib/json2.js', function() {
8         loadLink(win, REMOTE_SRC+ '/lib/underscore.min.js', function() {
9           loadLink(win, REMOTE_SRC+ '/lib/backbone.min.js', function() {
10            loadLink(win, REMOTE_SRC+ '/dev/base_dev.js', function() {
11              loadLink(win, REMOTE_SRC+ '/assets/js/deps.js', function() {
12                loadLink(win, REMOTE_SRC+ '/src/' + win.loader_path + '/loader.js', function() {
13                  async.eachSeries(SRIPTS, function(src, callback) {
14                    loadScript(win, BASE_URL+src, callback);
15                  });
16                });
17              });
18            });
19          });
20        });
21      });
22    });
23  });
24 }
```

# Promises - Intro

---

## 1. Why promises?

- a. To overcome challenges of **nested callbacks** (aka – callback hell and Pyramid of doom)

## 2. What is a promise?

- a. It's an **enhancement** to callback functions for executing and handling **ASYNC** operations
- b. Promise is an object that represents **eventual completion** (SUCCESS or FAILURE) of an Async action

## 3. How to check if a given object is a promise object?

- a. Use **instanceof** operator to see if a function returns a promise object

# Promises - Syntax and Rules

---

## 1. What's the syntax?

`new Promise( (resolve, reject) => { } )`

## 2. What are the states a promise can be in?

- i. Pending - Initial state before being resolved or rejected
  - ii. Fulfilled
  - iii. Rejected
- } Settled

## 3. How to consume a promise?

- a. `.then()` -> To get resolved value (success)
- b. `.catch()` -> To get rejected value (failure)

## 4. Can promise be chained?

- a. Both `.then()` and `.catch()` returns a promise object again, therefore chaining happens

Async/Await

# Async Function Coding Style

---

The coding style of Async function looks like this

**Callback – old fashioned**

||

**Promises – Modern**

||

**Async/Await – Concise**



# Async/Await - Intro

---

## 1. Why Async/Await?

- a. To write synchronous way of coding
- b. An async function allows you to **handle asynchronous code** in a manner that appears synchronous.

## 2. What's the syntax?

```
async function getData() {  
    // code to get data  
}
```

## 3. What are the rules?

- a. async functions uses promises under the hood
- b. async is function; await is an operator
- c. **await** : await can be used within an async function and **will wait until a promise settles** (either resolved or rejected) before executing the designated code

# Objects

# Objects - Overview

---

## 1. What's an object in JS?

- a. An object is a collection of properties (key: value)
- b. If the property is a function, then it is called as '**method**'
- c. One of the data types
- d. Unordered named collections (key: value) pair

## 2. How many ways Object can be created?

- a. Literals/Object initializer, constructor function, Object.Create(<prototype>)
- b. Classes are other way of creating object from ES5

## 3. General rules

- a. Object is mutable (Any changes to the original Object will be reflected elsewhere)
- b. The literals form of an object is denoted by **{}**
- c. **for...in** loop can be iterate over an object
- d. All objects in JavaScript inherit from at least one other object. The object being inherited from is known as the **prototype**

## 4. What's the use case in Test Automation?

- a. To store test data (key: value) - mainly for config (e.g env, url), storing creds (user, pass)

# Array

# Recap

---

1. What are the methods that returned an array in the past?
  - a. `String.split()` => String array
  - b. `Object.keys()` => String array
2. What is the literal form of an array?
  - a. `[]`
3. What is the `typeof` operator return value for an array?
  - a. `object`
4. Is array mutable?
  - a. Yes

# Array - Overview

---

## 1. What's an Array in JS?

- a. It's an ordered collection [by index value]

## 2. General rules/Understanding

- a. At the implementation level, JavaScript's arrays actually store their elements as standard **object** properties [with key as index, value as actual value]
- b. Standard for, forEach(), for...of loop can be iterate over Array
- c. The element of an array can be **any type**

## 3. What's the use case in Test Automation?

- a. To store a collection of data

# Classes

# Classes - Context

---

1. What is the alternative way of creating an Object?
  - a. Classes are other way of creating object from ES5
2. Have you heard about this phrase - “Page Object Model”?
  - a. One of the design patterns in Test Automation, where each web page will be considered as an Object (fields and actions)
3. What is the use cases of classes in Test Automation?
  - a. Mainly when adopting to “Page Object Model” design pattern



# Classes - Overview

---

## 1. What is a class in JS?

- a. Classes are a **template** for creating objects. [recipe for a dish]
- b. As we know, an object can have **properties and methods**

## 2. General rules

- a. By convention, class name will start with uppercase
- b. A class can have fields/variables, constructor function, and methods
- c. A class can extend another class for inheriting properties and methods
- d. A class can be getters and setters for its variables
- e. A constructor can use the super keyword to call the constructor of the super class

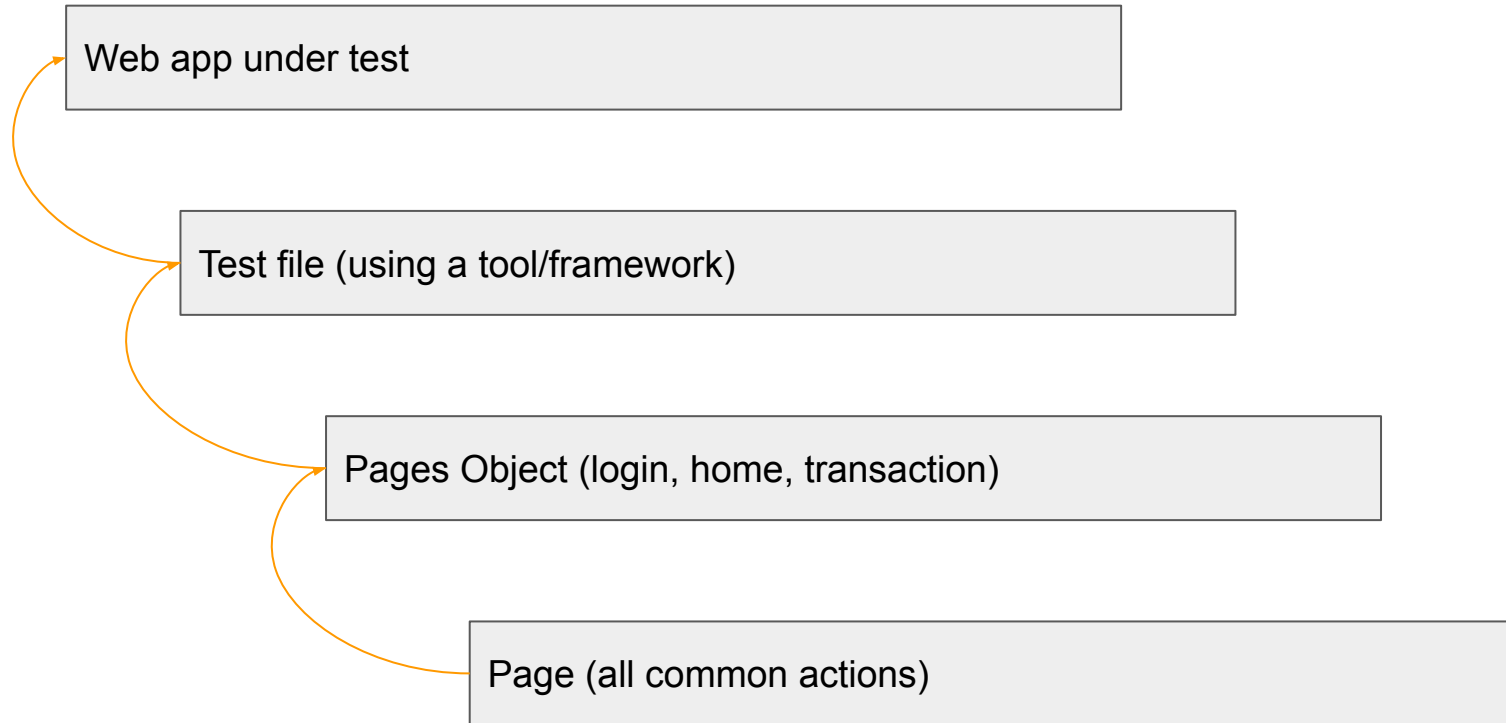
# Class - Common Syntax

---

```
wk1 > class > demo.js > ...  
1  "use strict";  
2  1  
3  class Demo {  
4    // Variables  
5    var1;  
6    var2; 2  
7  
8    // Constructor function 3  
9    constructor() {}  
10  
11    // Getter 4  
12    get getSomething() {  
13      | return "something";  
14    }  
15  
16    // Method 5  
17    doSomething() {  
18      | // do it here  
19    }  
20  } 6  
21  
22  // Object creation  
23  let demo = new Demo();  
24  demo.doSomething();  
25
```

# Page Object Model - Context Diagram

---



# Getter Property Rules

---

## I. Rules

- a. A getter property should return something
- b. A getter should not expect an argument

# Node.js Essentials

# Recap - Node.js

---



1. Node.js is a **runtime environment** that executes JS code outside a web browser
2. It is asynchronous, Single threaded and Non-blocking I/O (runs in parallel)
3. It is bundled with couple of packages that are used for **Network I/O, File Operations and processes**, reference: <https://nodejs.org/api/>

# Modules

# Modules - Overview

---

1. What is a module?
  - a. A file containing JS code
  - b. It's a way in which JavaScript code can be shared (exported and imported) for reusability
2. How many module system node.js supports?
  - a. CommonJS modules (**default**)
  - b. ECMAScript modules (ESM)
3. How to enable ECMAScript module ?
  - a. Add `"type" : "module"` in package.json
4. What statements are used for both module systems?
  - a. CommonJS - `module.exports` and `require`
  - b. ESM - `export` and `import`



# Node.js - fs module

# fs module - overview

---

1. What is fs module?
  - a. A native Node.js module system used to interact with files
2. Do we need to use npm install for the these native modules?
  - a. No, we can directly require/import it and without installing
3. What are the modes fs module interacts with filesystems?
  - a. Sync
  - b. Async
  - c. Streams
4. What's the typical use case in automation?
  - a. Mainly to deal with JSON files with
    - i. read,
    - ii. convert into JS object,
    - iii. traverse and update,
    - iv. convert back to JSON and then write

# Error Handling

# Error Handling (try...catch...[finally])

---

## 1. What is error handling all about?

- a. You can use `throw` statement to throw error - Note: This STOPS the node execution
- b. And can handle Error using `try...catch...[finally]` statements

## 2. How many ways an error can be created?

- a. Using `Error` constructor -> function call
- b. Using `new Error` -> Object creation

## 3. What are the two main sources of error?

- a. Your own code
- b. External packages

## 4. What's the use case in Test Automation?

- a. Manage and control the execution flow - instead of just failing tests, this includes,
  - i. Retry
  - ii. Log an error msg and move on...
  - iii. Fail with custom error msg instead of a default error

# throw syntax

---


```
throw expression;
```

```
throw 'Error2';    // String type  
throw 42;          // Number type  
throw true;        // Boolean type  
throw {toString: function() { return "I'm an object!"; } };
```

## try...catch...[finally] syntax

---

```
1
2  try {
3      // Code that might cause error
4      } catch (error) {
5          // Code to run when error occurs
6      } finally {
7          // Code that always executes
8      }
9
10
11
12
```



The diagram illustrates the flow of error handling in a try-catch-finally block. An orange arrow points from the 'catch' block back to the 'try' block, labeled 'catchID'.

# Error Handling - Rules

---

## I. Rules

- a. **catchID** is when the catch block is entered and lasts only for the duration of the catch block
- b. **Finally block:** Executed after try...catch IRRESPECTIVE of whether the exception was thrown or not .This is mostly used to free up resources (e.g. closing db connection, files..etc)