CONTENTS

[JAVASCRIPT 3](#_Toc72692939)

[PRIMITIVE DATATYPES IN JAVASCRIPT 3](#_Toc72692940)

[DOM EVENTS IN JS 3](#_Toc72692941)

[INCLUDING SCRIPTS 5](#_Toc72692942)

[ARRAYS 6](#_Toc72692943)

[JAVASCRIPT EXECUTION 7](#_Toc72692944)

[OOPS IN JAVASCRIPT 10](#_Toc72692945)

[ENCAPSULATION 10](#_Toc72692946)

[ABSTRACTION 10](#_Toc72692947)

[INHERITANCE 11](#_Toc72692948)

[POLYMORPHISM 11](#_Toc72692949)

[CREATING JAVASCRIPT OBJECTS 11](#_Toc72692950)

[CONSTUCTOR PROPERTY OF AN OBJECT 12](#_Toc72692951)

[OBJECTS 13](#_Toc72692952)

[FUNCTIONS 15](#_Toc72692953)

[UNDERSTANDING THIS 20](#_Toc72692954)

[PROTOTYPE OBJECTS 20](#_Toc72692955)

[ES6 CONCEPTS 23](#_Toc72692956)

[VARIABLES 23](#_Toc72692957)

[STRING CONCATENATION – TEMPLATE LITERALS 24](#_Toc72692958)

[OBJECT LITERALS 24](#_Toc72692959)

[ARROW FUNCTIONS 24](#_Toc72692960)

[DESTRUCTURING 24](#_Toc72692961)

[DEFAULT PARAMETERS 25](#_Toc72692962)

[SPREAD OPERATOR 25](#_Toc72692963)

[REST PARAMETER 26](#_Toc72692964)

[FOR OF LOOP 27](#_Toc72692965)

[MAPS 27](#_Toc72692966)

[WEAKMAP 27](#_Toc72692967)

[SETS 27](#_Toc72692968)

[WEAKSET 28](#_Toc72692969)

[ASYNCHRONOUS JAVASCRIPT 28](#_Toc72692970)

[PROMISES 28](#_Toc72692971)

[PROMISES CHAINING 30](#_Toc72692972)

[ASYNC AND AWAIT 31](#_Toc72692973)

[CLASSES IN JAVASCRIPT 32](#_Toc72692974)

[SYNTAX 32](#_Toc72692975)

[INHERITANCE 33](#_Toc72692976)

[POLYMORPHISM 34](#_Toc72692977)

[HTML 5 WEB WORKER 35](#_Toc72692978)

[DEDICATED WEB WORKERS 35](#_Toc72692979)

[SHARED WEB WORKERS 37](#_Toc72692980)

[TYPESCRIPT 38](#_Toc72692981)

[TYPESCRIPT DEVELOPMENT ENVIRONMENT SET UP 38](#_Toc72692982)

[TYPESCRIPT TYPES 39](#_Toc72692983)

[TS FUNCTIONS 40](#_Toc72692984)

[IMPLICIT TYPING (TYPE INFERENCE) 40](#_Toc72692985)

[TYPESCRIPT CLASSES AND FUNCTION 40](#_Toc72692986)

[INHERITANCE 41](#_Toc72692987)

[GETTERS AND SETTERS IN CLASSES 42](#_Toc72692988)

[STATIC PROPERTIES AND METHOD 42](#_Toc72692989)

[ABSTRACT CLASSES 42](#_Toc72692990)

[SINGLETON AND PRIVATE CONSTRUCTORS 42](#_Toc72692991)

[INTERFACES 43](#_Toc72692992)

[MODULES IN TYPESCRIPT 43](#_Toc72692993)

[TYPESCRIPT COMPILER ARGUMENTS 44](#_Toc72692994)

[CREATING A TYPESCRIPT PROJECT 46](#_Toc72692995)

[CREATE A NODE PROJECT 46](#_Toc72692996)

[WEB PACK 48](#_Toc72692997)

[MODULES IN JAVASCRIPT 48](#_Toc72692998)

[MODULE FORMATS/SYSTEM 49](#_Toc72692999)

[IMPLEMENTING COMMON JS MODULE FORMATS 49](#_Toc72693000)

[INSTALLING AND CONFIGURING WEBPACK 50](#_Toc72693001)

[INSTALLING AND CONFIGURING WEBPACK 50](#_Toc72693002)

[MODULE LOADERS IN WEBPACK 51](#_Toc72693003)

[ES6 MODULES SYSTEM 52](#_Toc72693004)

[CSS FILES IN WEBPACK 54](#_Toc72693005)

[CODE SPLITING USING WEBPACK 56](#_Toc72693006)

# JAVASCRIPT

### PRIMITIVE DATATYPES IN JAVASCRIPT

1. Number
2. String
3. Boolean
4. undefined
5. null

* JavaScript has dynamic typing- The datatypes are automatically assigned to the variable when the variable is assigned to it.
* The variable should not **start** with a number or special character except the ‘\_ ‘and ‘$’.
* The variable should not be reserved JS keyword.
* We can use a variable without explicitly declaring it.

|  |  |
| --- | --- |
| If the variable is used without any explicit declaration, they end up having global scope | If the variable is used with an explicit declaration, they end up having local scope |

#### STRICT MODE

* This prevents the use of undeclared variables.
* Restrict the use of reserved keywords as a variable name.
* Enabling strict mode will enable thowing of error rather than failing silently - when an exception occurs

#### TYPE COERSION

|  |  |
| --- | --- |
| var a=10, b=”10”;  a==b 🡪TRUE  a===b 🡪FALSE | “==” 🡪 in this comparison JavaScript do the automatic type conversion to compare the variables so it returns “TRUE”. This automatic type conversion is called type coersion  “===” 🡪 This compares the value and its type too. |
| **COMMON EXAMPLES OF TYPE COERSION (TYPE COERSION TO BOOLEAN)** | |
| For integers , any non-zero value is considered to be **TRUE**  var a=10; b =0;  var resultA = (a)? true:false;  var resultB = (b)? true:false;  console.log("a is "+resultA); 🡪 a is true  console.log("b is "+resultB); 🡪 b is false | For string , any non-zero length string is TRUE  var a='Hello'; b ='';  var resultA = (a)? true:false;  var resultB = (b)? true:false;  console.log("a is "+resultA); 🡪 a is true  console.log("b is "+resultB); ); 🡪 b is false |
| undefined && null  All “**undefined**” and “**null**” value is consider to be FALSE | var a; b =null;  var resultA = (a)? true:false;  var resultB = (b)? true:false;  console.log("a is "+resultA); 🡪 a is true  console.log("b is "+resultB); 🡪 b is false |

### DOM EVENTS IN JS

1. Event is some notable action for which script can respond.
2. An event handler is a code associated with a part of document and particular event

#### EVENT BINDING

|  |  |  |
| --- | --- | --- |
| **HTML** | **EVENT HANDLERS** | |
| <button id="clk">Click</button | function display(){ console.log("Display Me!"); } | function sayHello(){ console.log("Hello!");} |

##### EVENT BINDING USING ON

|  |  |
| --- | --- |
| var btnId = document.getElementById("clk"); | btnId onclick= display; |

##### EVENT BINDING USING ADD EVENT LISTENER

|  |  |
| --- | --- |
| **USING addEventListener():** Binds the handler with an event  var btnId = document.getElementById("clk");  btnId.**addEventListener**("click",display,false); | addEventListener() can bind multiple handlers with the same object .  btnId.addEventListener("click",display,false);  btnId.addEventListener("click",sayHello,false); |

##### EVENT UNBINDING USING ADD EVENT LISTENER

|  |  |
| --- | --- |
| **removeEventListener() : U**nbinds the event from the handler | btnId.**removeEventListener**("click",sayHello,false); |

##### EVENT PROPAGATION

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The event propagation happens in 2 ways in the XHTML DOM   1. Event Capturing 🡪In capturing phase the event propagate happens from top to button in the DOM tree 2. Event Bubbling 🡪 In bubbling phase the event propagate happens from top to button in the DOM tree   **IMPLEMENTING EVENT CAPTURING AND BUBBLING**  We can implement event capturing and bubbling using  ***element*.addEventListener(*event, function, useCapture*);**   1. The first parameter is the type of the event 2. The second parameter is the function we want to call when the event occurs. 3. The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.    1. False 🡪 Event Bubbling 🡪 Bottom to Top    2. True 🡪 Event Capturing 🡪 Top to bottom | | | | |
| **HTML**  <div id="one"> 1  <div id="two"> 2  <div id="three"> 3  <div id="four">4  </div>  </div>  </div>  </div>   * The similar HTML DOM structure has been implemented as shown in the diagram * In the below code, we are binding the click event to all the divs | |  | | |
| **EVENT BUBBLING** | | | | |
| **JS**  document.getElementById("one").addEventListener(  "click", function(e) {  console.log("1");  }, false);  document.getElementById("two").addEventListener(  "click", function(e) {  console.log("2");  }, false);  document.getElementById("three").addEventListener(  "click", function(e) {  console.log("3");  }, false);  document.getElementById("four").addEventListener(  "click", function(e) {  console.log("4");  }, false); | **BOTTOM TO TOP** | | | |
| When we click ON 4 | |  | |
| When we click ON 3 | |  | |
| When we click ON 2 | |  | |
| **EVENT CAPTURING** | | | | |
| document.getElementById("one").addEventListener(  "click", function(e) {  console.log("1");  }, true);  document.getElementById("two").addEventListener(  "click", function(e) {  console.log("2");  }, true);  document.getElementById("three").addEventListener(  "click", function(e) {  console.log("3");  }, true);  document.getElementById("four").addEventListener(  "click", function(e) {  console.log("4");  }, true); | **TOP TO BOTTOM** | | | |
| When we click ON 4 | | |  |
| When we click ON 3 | | |  |
| When we click ON 2 | | |  |

#### EVENT OBJECT

|  |  |
| --- | --- |
| var btnId = document.getElementById("clk");  btnId.addEventListener("click",function(**evtObj**){  console.log(**evtObj**);  },false); | * The DOM events is always passed to the handler as an argument, which contains the details of the event. E.g target , currentTarget, type(eventType) etc.. |
| var btnId = document.getElementById("clk");  btnId.addEventListener("click",function(**evtObj**){  console.log(**evtObj.target**);#1  console.log(**evtObj.currenTarget**);#2  },false); | **#1 o/p** - <button id="clk">Click</button>  **#2 o/p** - <button id="clk">Click</button> |

#### EVENT METHODS

|  |  |  |
| --- | --- | --- |
| **stopPropagation() /**  **stopImmediatePropagation()**   * These method stops the event propagation further(bubbling/Capturing). Example here - is in case event capturing. * The event will not propagate further if it encounters either of the function * Both the function can be interchangeably used with one exception (explained below) | document.getElementById("one").addEventListener(  "click",  function(e) {  console.log("1");  }, true);  document.getElementById("two").addEventListener(  "click", function(e) {  console.log("2");  **e.stopPropagation();**  }, true);  document.getElementById("three").addEventListener(  "click", function(e) {  console.log("3");  }, true);  document.getElementById("four").addEventListener(  "click", function(e) {  console.log("4");  }, true); | o/P |
| **preventDefault()** 🡪The preventDefault() method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur.  For example, this can be useful when:   1. Clicking on a "Submit" button, prevent it from submitting a form 2. Clicking on a link, prevent the link from following the URL   ***Note: The preventDefault() method does not prevent further propagation of an event through the DOM. Use the stopPropagation() method to handle this.*** | Example   * The default behavior of the hyperlink to capture the click event and open the hyperlink * This default behavior can stopped using the preventDefault()   **HTML**  <a id="wLink" href="https://www.w3schools.com/">w3schools</a>  **JS**  document.getElementById("wLink").addEventListener("click", function(e) {  e.preventDefault();  }); | |

### INCLUDING SCRIPTS

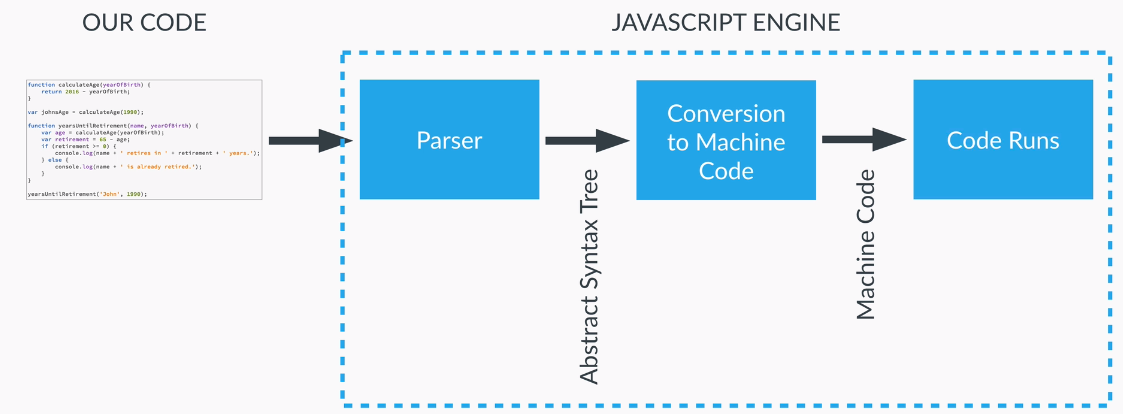
|  |
| --- |
| **NORMAL EXECUTION : <script src="script.js">**  Normal JavaScript Execution. HTML Parsing paused for script fetching and execution   1. When the HTML page loads , it starts the HTML parsing 2. If it encounters a script tag while parsing – the HTML parsing will pause 3. The script is fetched and followed by its execution, till that time HTML parsing will be blocked/paused 4. After the execution of script the HTML parsing resumes. |
| **USING ASYNC ATTRIBUTE : <script async src="script.js">**  Asynchronous JavaScript Execution. HTML parsing is paused only for the script execution   1. When the HTML page loads , it starts the HTML parsing 2. If it encounters a script tag while parsing – the script is fetched asynchronously along with HTML parsing. 3. Once fetched, the script starts executing. While the script is executing the HTML parsing will be blocked/paused 4. After the execution of script the HTML parsing resumes. |
| **USING DEFER ATTRIBUTE : <script defer src="script.js">**  Deferred JavaScript Execution. HTML parsing is never paused. Script execution happens after parsing is complete   1. When the HTML page loads, it starts the HTML parsing 2. If it encounters a script tag while parsing – the script is fetched asynchronously along with HTML parsing. 3. Once the script is fetched the HTML parsing resumes 4. The script will be executed only after HTML parsing is complete. |

### ARRAYS

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| * Stores the values of different data types unlike Java. * Arrays are JavaScript object   Example  var names=['John','Eric','Alex'];  console.log(names); |  | | | | | 1. Here the array’s index are property name and arrays element are the value to it 2. Similar to objects the arrays are access with property names e.g names["1"] 3. name[1] also works in arrays because of type coercion , which converts number into string | |
| **WAYS TO CREATE ARRAY** | | | | | | | | |
| var names =[“John”,”Mark”,”Emily”] | | | var years = new Array(2107,2018,2019); | | | | | |
| **ARRAY FUNCTION** | | | | | | | | |
| **INPUT** | | **FUNCTION** | | | | | | |
| var names =[“John”,”Mark”,”Emily”] | | Adding the last element  name[name.length]=”Joe” **OR** name.push(“Joe”); | | | | | | |
| **USING MAP**   1. The map() method iterates over all the elements and creates a new array from it , without actually modifying the actual arrays 2. If the associated call back function modifies the array element, the new array will store the modified value.   Note:   * map() does not execute the function for array elements without values. This method does not change the original array. | | | | | | | | |
| **EXAMPLE 1**  var arr = [10, 20, 30, 40];  var newArray = arr.map(function(element) {  element++;  console.log(element);  **return element;**  });  console.log(newArray);🡪 [11, 21, 31, 41];  **USING ARROW FUNCTION**  var newArray = arr.map(element => ++element); | | | | | **EXAMPLE 2**  var getAge = function(age) {  return 2020 - age;  };  var yearsOfBirth = [1982, 1992, 2004, 1957];  var listOfAdults = yearsOfBirth.map(function(element) {  return getAge (element);  });  console.log(listOfAdults); 🡪[**38,28,16,63]**  **USING ARROW FUNCTION**  var listOfAdults = yearsOfBirth.map(element => getAge(element)); | | | |
| names.pop() | | returns the last element and removing it from the array | | | | | | |
| names.shift() | | returns the first element and removing it from the array | | | | | | |
| names.push(element) | | Adds the element at the end of an array | | | | | | |
| names.unshift(element) | | Adds the element at the beginning of an array | | | | | | |
| const data = Array(5).**fill**(""); | | Filling arrays with empty string or some default value(passed in the fill method argument) | | | | | | |
| const data = ["1", "2", "3", "1", "4"];  console.log(Array.from(new Set(data))); | | Unique value from an array. | | | | | | |
| **LOOPING ARRAYS USING FOREACH**  var names=['John','Eric','Alex'];  names.forEach(function(value, index){  console.log("index="+index+" "+ "value="+value);  });  **USING ARROW FUNCTION**  var names = ['John', 'Eric', 'Alex'];  **names.forEach((value, index) => console.log(`index=${index} value=${value}`));** | | | | | | | **o/p**  index=0 value=John  index=1 value=Eric  index=2 value=Alex | |
| **ARRAYS TO OBJECT**  var user = ["Alex", "Doe", "XYZ Company", "Manager"];  var userObject = { ...user };  console.log(userObject); | | | | | | | **OBJECT TO ARRAYS**  var user = {  firstName: "Alex",  lastName: "Doe",  company: "XYZ Company",  designation: "Manager"  };  var userArray = Object.values(user);  console.log(userArray); | |
| **DEEP AND SHALLOW COPIES(ARRAYS)** | | | | | | | | |
| **DEEP COPIES** | | | | **SHALLOW COPIES** | | | | |
| Different variables point to different memory location. So changing will not change the other variables. | | | | Different variables point to same memory location. So changing on in-turn changes the other variables too. | | | | |

## JAVASCRIPT EXECUTION

**HOW JS CODE IS EXECUTED - OVERVIEW**



* Code executes in a Javascript engine of the browser
* The code is first parsed by a parser – if it has no syntax error it creates a data structure called Abstract Syntax Tree
* The DS then further translated into machine code which finally runs

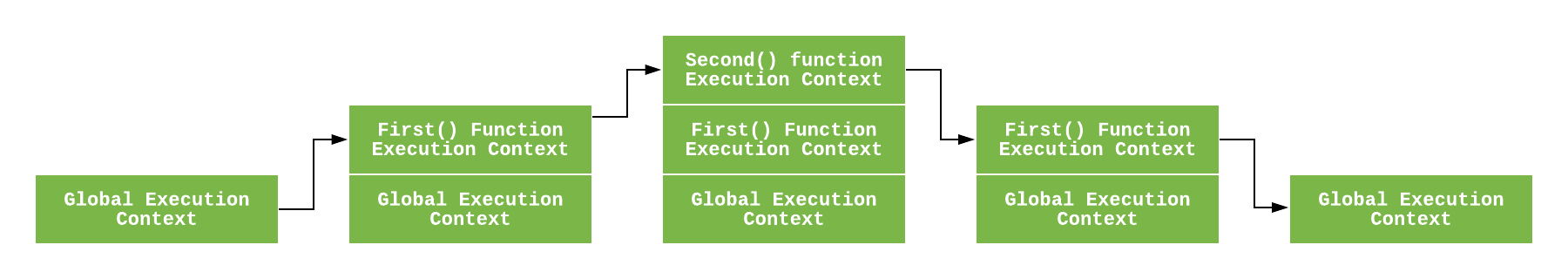
**EXECUTION CONTEXTS AND EXECUTION STACK**

|  |  |
| --- | --- |
|  | * All JS code runs in an environment – That environment is called Execution context. * The default execution context is **Global Execution Context.** * All the JS Code(variable/functions) which is not inside the function runs in **Global Execution Context** * The Global Execution Context code is associated with a global object called “window” object for the browser runtime environment.   var lastName="Hello";  **lastName === window.lastName** 🡨 true |

**EXECUTION STACK – WHEN AN EXECUTION CONTEXT IS CREATED?**

1. Execution stack, also known as “calling stack” in other programming languages, is a stack with a LIFO (Last in, First out) structure, which is used to store all the execution context created during the code execution.
2. When the JavaScript engine first encounters your script, it creates a global execution context and pushes it to the current execution stack. Whenever the engine finds a function invocation, it creates a new execution context for that function and pushes it to the top of the stack.The engine executes the function whose execution context is at the top of the stack. When this function completes, its execution stack is popped off from the stack, and the control reaches to the context below it in the current stack.

|  |  |
| --- | --- |
| let a = 'Hello World!';  function first() {  console.log('Inside first function');  second();  console.log('Again inside first function'); }  function second() {  console.log('Inside second function'); }  first(); console.log('Inside Global Execution Context'); | 1. When the code loads in the browser, the JavaScript engine creates a global execution context and pushes it to the current execution stack. When a call to first() is encountered, the JavaScript engines creates a new execution context for that function and pushes it to the top of the current execution stack. 2. When the second() function is called from within the first() function, the JavaScript engine creates a new execution context for that function and pushes it to the top of the current execution stack. 3. When the second() function finishes, its execution context is popped off from the current stack, and the control reaches to the execution context below it, that is the first() function execution context. 4. When the first() finishes, its execution stack is removed from the stack and control reaches to the global execution context. Once all the code is executed, the JavaScript engine removes the global execution context from the current stack. |



**HOW EXECUTION CONTEXT IS CREATED?**

|  |  |
| --- | --- |
| **OBJECT REPRESENTATION OF EXECUTION CONTEXT** | All execution context is always associated with an object like Global Execution context is associated with “window” object. The execution context object has 3 properties   1. **VARIABLE OBJECT(VO) :** This contains    1. Function arguments    2. Function Declaration    3. Inner variable declaration 2. **SCOPE CHAIN** 3. **“THIS” VARIABLE**   The execution context is created in 2 phases   1. **CREATION PHASE**    1. Creation of VO object    2. Creation of Scope Chain    3. Determine the value of “this” variable 2. **EXECUTION PHASE**    1. The code of the function that generated the current execution context is ran line by line |

**CREATION PHASE IN DETAILS**

Creation phase happens before the actual execution happens

**THE FIRST STEP: CREATION OF VO OBJECT**

* Code is scanned for **function declarations :** for each function a property is created in VO object , pointing to the function
* Code is scanned for **variable declarations :** for each variable a property is created in VO object , and set to undefined

**THIS IS CALLED HOISTING**

|  |  |
| --- | --- |
| **FUNCTION HOISTING**   * In case of function declaration – We can able to call the function before it is declared * This happens because the function declaration is hoisted during the Creation Phase itself. | calculateAge(1982);  function calculateAge(birthYear){  return 2020-birthYear;  } |
| **VARIABLE HOISTING**   * The variable can be accessed before declaring it as the variables are hoisted and set to undefined. | **EXAMPLE 1**  console.log(age);  var age = 23; |
| * The code outside the function and inside the function runs in different execution context. * In this code - The “age” variable is created– one in the Global Execution Context and another Function Execution context. * The code will access the respective variable depending upon the execution context they are executing. * #1 – Declares the age variable in Global execution context * #4 - Accessing the value from Global Execution Context * #2 – It is accessing the hoisted age variable which is in the execution context of the function its where it is undefined (not the Global Execution Context) * #3 – Accessing the age variable which is in the execution context of the function | **EXAMPLE 2**  var age= 30; **#1**  function foo(){  console.log("In foo before declaration="+age); **#2**  var age= 23;  console.log("In foo after declaration="+age); **#3**  }  foo();  console.log("Outside foo="+age); **#4** |

**THE SECOND STEP: SCOPING AND SCOPE CHAIN**

* The scoping answers a question – Where a variable can be accessed?
* Each function creates a scope. In JavaScript the only way to create new scope is to create a function.

**LEXICAL SCOPING –** JavaScript has lexical scoping, means a function inside another function can access to the scope of the outer function.

**SCOPE CHAIN**

|  |  |
| --- | --- |
|  | * The Scope chain propagate from bottom to top |

## OOPS IN JAVASCRIPT

The OOPS in any language has 4 pillars

1. **ENCAPSULATION**
2. **ABSTRACTION**
3. **INHERITANCE**
4. **POLYMORPHISM**

### ENCAPSULATION

|  |  |
| --- | --- |
| In encapsulation we group the related properties and methods that operate on those data together inside a wrapper called Object.  In procedural programming methods will have more number of parameters than object oriented. |  |
| **OBJECT ORIENTED WAY** | **PROCEDURAL WAY** |
| let employee = {  baseSalary : 3000,  overTime : 10,  rate :10,  getWage : function(){  return this.baseSalary+(this.overTime\*this.rate);  }  } | let baseSalary= 3000;  let overTime= 10;  let rate =10;  function getWage(baseSalary,overTime,rate){  return baseSalary+(overTime\*rate);  } |

### ABSTRACTION

Abstraction means hide the details and show/expose only the essentials. These exposed properties or methods can be accessed by client application or consumers.

**IMPLEMENTING ABSTRACTION: PRIVATE PROPERTIES AND METHODS**

|  |  |
| --- | --- |
| The problem with this implementation is that the property “radius” can be modified by client application, so the objective is to hide the “radius” property and expose only the draw() method to client application. This depicts **abstraction**.  There are two ways to achieve this   1. Using Local variables 2. Using setter & getters | function Circle(radius){  this.radius = radius;  this.draw = function(){  console.log("Draw Circle of Radius="+ this.radius);  }  }  let circle = new Circle(1);  circle.radius=2;  circle.draw(); |
| **USING LOCAL VARIABLES** | **SETTER AND GETTERS** |
| function Circle(rad){  let radius = rad;  this.getRadius = function(){  return radius;  }  }  let circle = new Circle(1);  console.log(circle.getRadius());  The local variable “radius” cannot be accessed/ modified directly by client application. It will be only be accessed with exposed method **getRadius()** | function Circle(){  let defaultRadius=0  Object.defineProperty(this,"**defaultRadius**",{  **get**:function(){  return defaultRadius;  },  **set**:function(value){  defaultRadius = value;  }  });  }  let circle = new Circle(1);  console.log(circle.defaultRadius);  circle.defaultRadius=10;  console.log(circle.defaultRadius); |
| Object.defineProperty(<object >,"**<*property\_to\_set*>**",{  **get**:function(){ // },  **set**:function(value){ // }  }); | 1. circle.defaultRadius 🡪The will call the “get” method of the **defaultRadius** property 2. circle.defaultRadius=10; 🡪 The will call the “set” method of the **defaultRadius** property and set its value to 10. |

### INHERITANCE

**INHERITANCE IS MADE POSSIBLE IN JAVASCRIPT USING PROTOTYPE OBJECT.** Refer Prototype Object section

### POLYMORPHISM

### CREATING JAVASCRIPT OBJECTS

|  |  |
| --- | --- |
| ***CREATING OBJECTS USING OBJECT LITERALS***   1. Objects literals in JavaScript are set of key value pairs. 2. The value can be another object or function   const circle = {  radius: 1, 🡨Property  location:{  x: 0,  y:0  },  draw:function(){ 🡨 Methods  console.log(“Draw Circle”);  }  };  **circle.draw(); //Calling function** | ***CREATING OBJECTS USING FACTORY FUNCTION***  function createCircle(radius){  **return** {  radius : radius,  draw: function(){  console.log(“Draw Circle of radius=”+ radius);  }  }  }  let circle1 = createCircle(1);  circle1.draw();  let circle2 = createCircle(2);  circle2.draw();   * **Here each object has own copy of properties and method** |

#### USING OBJECT.CREATE()

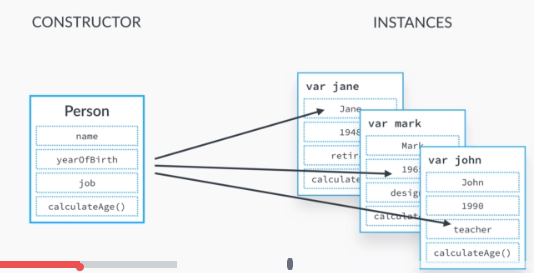
|  |  |
| --- | --- |
| const person = {  isHuman: false,  printIntroduction: function() {  console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);  }  };  **const me = Object.create(person);**  me.name = “Matthew”; // “name” is a property set on “me”, but not on “person”  me.isHuman = true; // inherited properties can be overwritten  me.printIntroduction();  // expected output: “My name is Matthew. Am I human? True” | **DETAILS ON OBJECT.CREATE()**   1. **const me = Object.create(person); 🡪 “me**” object will be created , where person will become the prototype object of **me** object. *Me.\_\_proto\_\_ ==person 🡺 true* 2. **me.name 🡪** This is a property of me object   **me.isHuman 🡪** Inherited property from person object |

#### CREATING OBJECTS USING CONSTRUCTOR FUNCTION

|  |  |
| --- | --- |
| function Circle(radius){  **console.log(this); 🡨 This points to empty object**  this.radius = radius;  this.draw = function(){  console.log(“Draw Circle of Radius=”+this.radius);  }  }  **let circle = new Circle(1);**  circle.draw(); | 1. By convention the first letter of constructor function should be capital. 2. We use “new” operator to create an object. **The new create an empty object and “this” points to the empty object** 3. In the constructor function returns the “this” object implicitly. |
| function Circle(radius) {  **console.log(this); 🡨 This points to global window object**  this.radius = radius;  this.draw = function() {  console.log(“Draw Circle of Radius=” + this.radius);  };  }  **let circle = Circle(1);** | When we a function Circle function constructor without new operator – this points to global window object |

##### CONSTRUCTOR FUNCTION IN DETAIL

1. In other programming languages like Java , we have a concept of classes , which act a template for objects created from that class/ template
2. The classes here are known as constructor which can be used to create multiple instances, as shown below.
3. The constructor act as a blueprint to create multiple instances. Every object created using the constructor function will have its own copy of properties and methods



1. The **function constructor** is the way by which we can create the blueprint

|  |  |
| --- | --- |
| **BLUEPRINT(FUNCTION CONTRUCTOR)** | **CREATING OBJECTS/INSTANCES** |
| var Person = function(firstName, lastName){  this.firstName = firstName;  this.lastName = lastName;  } | var john = **new** Person('john','Doe');  var mary = **new** Person('Mary','jain');  console.log(john);  console.log(mary); |
| When we use the “new” operator JS internally creates a “this” object and return the value at the end – as shown. | var Person = function(firstName, lastName){  **//var this ={} ;**  this.firstName = firstName;  this.lastName = lastName;  **//return this;**  } |

### CONSTUCTOR PROPERTY OF AN OBJECT

***EVERY OBJECT IN JS HAS A CONSTRUCTOR PROPERTY, WHICH REFERS TO THE FUNCTION WHICH IS USED TO CREATE THAT OBJECT.***

|  |  |
| --- | --- |
| **/\* Object Factory \*/**  function createCircle(radius){  return {  radius : radius,  draw: function(){  console.log(“Draw Circle of radius=”+ radius);  }  }  } | **/\* Object Factory \*/**    This Object uses an in-built **Object constructor function** |
| **let circle = createCircle(1);**  **circle.draw();**  **/\* Function Constructor \*/**  function Circle(radius){  this.radius = radius;  this.draw = function(){  console.log(“Draw Circle of Radius=”+this.radius);  }  }  **let anotherCircle = new Circle(1);**  **anotherCircle.draw();** | **/\* Function Constructor \*/**    **We can access the function itself using constructor function**  **anotherCircle.constructor(20);** |
| **/\* Object Literals \*/**  **let obj = {};** | **/\* Object Literals \*/** |

### OBJECTS

* Objects stores the different datatypes value as a key value pair

|  |  |  |
| --- | --- | --- |
| var person ={  firstName:’John’,  lastName: ‘Doe’,  birthYear:1982,  skills:[‘Java’,’C’]  }  **OR**  var person = new Object();  person.firstName=’John’; | **RETRIEVE THE VALUE FROM OBJECT**   * person.firstName 🡪 John * person[‘lastName’] 🡪Doe * var x =’birthYear’; person[x] 🡪1982 | **OBJECT EQUALITY**: Two objects considered to be equal when they pointing to same memory location.  Var myObj = {  prop: “property”  }  var myObJ2 = myObj;  console.log(myObJ2 === myObj); 🡪TRUE |
| **ADDING PROPERTY TO JS OBJECT**   * person.job =’designer’ OR * person[job]= ’designer’ |
| **OBJECT METHOD**  In JS object can have function called methods  var person ={  firstName:’John’,  lastName:’Doe’,  birthYear:1982,  calculateAge : function(){  return new Date().getFullYear() – this.birthYear;  }  }  console.log(person.calculateAge()); 🡨 Calling a function | | **SETTING AN OBJECT PROPERTY**  var person ={  firstName:’John’,  lastName:’Doe’,  birthYear:1982,  calculateAge:function(){  this.age = new Date().getFullYear() – this.birthYear;  }  }  person.calculateAge();  console.log(person); |
| **DELETING AN OBJECT PROPERTY**  var myObj = {  firstName: ‘John’,  middleName: ‘Eric’,  lastName:’Doe’  }  **delete myObj.middleName;**  **This deletes the middleName property from myObj object** | | **DYNAMIC OBJECT PROPERTY**  var company = “companyName”;  const user = {  name: “Alex”,  userid: “AL01”,  [company]: “XYZ Company”  }  console.log(user); |

#### RETRIEVING FROM OBJECTS

|  |  |  |
| --- | --- | --- |
| function Circle(radius){  this.radius = radius;  this.draw = function(){  console.log("Draw Circle of Radius="+this.radius);  }  }  let circle = new Circle(10);  We use dot or square bracket operator to retrieve object properties | | ***ADDING PROPERTY* : circle.location** ={x:10,y:10};  ***DELETING PROPERTY* : delete circle.location** ;  ***USING SQUARE BRACKET NOTATION***  The square bracket notation is used to retrieve the properties of an object   * When the property name is dynamic   let radiusProperty = 'radius';  console.log(circle[radiusProperty]);   * When property is not a valid JavaScript identifier   circle[‘**circle-radius’**]); 🡨 Special characters are not allowed in JS identifier |
| **ITERATING ALL OBJECT PROPERTIES** | for(let property **in** circle)  console.log(property); | |
| **FETCH ALL KEYS OF AN OBJECT** | Object.keys(circle); | |
| **CHECK A PROPERTY PRESENT?** | if('radius' **in** circle)  console.log("radius property exist"); | |
| **GET ALL KEYS AS ARRAY** | Object.keys(Circle) | |
| **GET VALUES AS ARRAYS** | Object.values(Circle) | |
| **GET OBJECT PROPERTIES AS ARRAY**  **(**each key -value pair in the object is an array**)** |  | |

**IMPORTANT POINTS**

* When we pass object in the function parameter – it will be always passed as reference type. So if the function makes any change in the passed object – it will change the original object. Unlike this, primitive types are passed as value type.

#### COPING EXISTING OBJECTS

|  |  |  |
| --- | --- | --- |
| **OBJECT ASSIGNMENT**  var Car= {  color:'Red',  brand:'BMW',  engine:{  make:2020,  power:'10 BP'  }  }  **let myCar = Car;**   * This creates a shallow copy of the existing object * The equality operator points to same memory location – So changing copied object will change the original object too. | **OBJECT. ASSIGN**  var Car= {  color:'Red',  brand:'BMW',  engine:{  make:2020,  power:'10 BP'  }  }  let carCopy = Object.assign({},Car);  console.log(carCopy);  carCopy.color = blue;  carCopy.engine.make ='Ferari';  console.log(Car);  **OUTPUT** | * Object.assign copies the source object to target object and finally return the target object.     **CONCLUSION**   * If we change the top-level property from copied object- it will not update that property in original object * Unlike that if we update the property of nested object property from copied object – it will update the original object also. |
| **SPREAD OPERATOR**  **let newCar = {...Car};**  This has same behavior as “object.assign()” | * DEEP COPY : In deep copy the objects are independent to each other and points to different memory location . * JSON.parse(JSON.stringify(Car)) 🡨 **CREATES THE DEEP COPY** | |
| **OBJECT.CREATE()** |  | |

### FUNCTIONS

* Function is an instance of object type that’s why function behaves like an object

|  |  |  |  |
| --- | --- | --- | --- |
| **WHY FUNCTIONS IN JAVASCRIPT ARE CALLED FIRST CLASS FUNCTION** | |  | |
| function Circle(radius) {  this.radius = radius;  this.draw = function() {  console.log(“Draw Circle of Radius=” + this.radius);  };} | | | * The Function in JS is an object in JavaScript.   **WHO CREATES FUNCTION OBJECT?**   * Circle is a function here and the constructor property return the function which has been used to create the object. * JS use the native Function constructor to create the Circle function as shown. |
| **PASSING FUNCTION AS PARAMETER**  var add = function(a,b){  console.log(a+b);  }  var subtract = function(a,b){  console.log(a-b);  }  var calculator = function (operand1 , operand2 ,operation){  operation(operand1,operand2);  }  **calculator(5,3,add);**  **calculator(5,3,subtract);** | **FUNCTION RETURNING A FUNCTION**  function calculatoController(operation) {  if (operation == “add”) {  return function add(op1, op2) {  return op1 + op2;  };  } else if (operation == “sub”) {  return function sub(op1, op2) {  return op1 – op2;  };  } else {  return function noOperation() {  console.log(“Operation No Supported”);  };  }  }  var addCalc = calculatoController(“add”);  console.log(addCalc(1, 2));  var subCalc = calculatoController(“sub”)(3, 1); 🡨 Can be called like this too  console.log(subCalc); | | |
| **IIFE**  **(**function(){  var a = 10;  var b= 20;  console.log(a+b);  }**)();**  console.log(a+b); 🡨 Here a ,b is not accessible | 1. IIFE stands for – Immediately invoke function expression. 2. IFFE creates only function scope variable (avoids creation of global variables) which get destroyed once the execution is over 3. It an anonymous function .We use if we want to execute logic right away on the page load without explicitly calling it. 4. This help in data privacy and avoid updating any variable in the global execution context. 5. This is called only once | | |
| **PASSING VALUE TO IIFE** | (function(op1, op2) {  console.log(op1 + op2);  })(1, 2); | | |

#### FUNCTION STATEMENTS & EXPRESSION

* Expression always returns some value. E.g [2+3 (o/p = 5) is an expression] so we expect a JS function to return something we should write an expression.

|  |  |  |
| --- | --- | --- |
| **FUNCTION EXPRESSION** | | **FUNCTION STATEMENTS** |
| FUNCTION EXPRESSION | ANONYMOUS FUNCTION EXPRESSION | function add(a,b){  return a+b;  }  console.log(add(1,2)); |
| var add = function **addFn**(a,b){  return a+b;  };  console.log(add(1,2)); | var add = function (a,b){  return a+b;  };  console.log(add(1,2)); |

#### FUNCTIONS IN OBJECT

|  |  |
| --- | --- |
| **LEGACY WAY** | **ES6 WAY** |
| const person = {  firstName: “John”,  lastName: “Doe”,  fullName: function() {  return this.firstName + “ “ + this.lastName;  }  };  console.log(person.fullName()); | const person = {  firstName: “John”,  lastName: “Doe”,  fullName() {  return this.firstName + “ “ + this.lastName;  }  };  console.log(person.fullName());  We can get rid of “**function**” keyword in ES6 |
| **GETTER & SETTERS – THIS HELPS IN IMPLEMENTING ABSTRACTION IN JS** | |
| const person = {  firstName: "John",  lastName: "Doe",  get fullName() {  return this.firstName + " " + this.lastName;  },  set fullName(name) {  let parts = name.split(" ");  this.firstName = parts[0];  this.lastName = parts[1];  }  };  console.log(person.fullName);  person.fullName = "Mike Smith"; 🡨 Call the setter  console.log(person.fullName); 🡨 Calls the getter | 1. **get && set** keywords are used to create getter and setters 2. When we set the value of a property it will call the setter of the property 3. When we access the property it calls the getter of the property 4. The **fullName** property can be accessed as a property not as a function |

#### EXCEPTION HANDLING IN JS

|  |  |
| --- | --- |
| function addNumber(op1 , op2){  if(isNaN(op1) || isNaN(op2))  **throw** "Paramter is not well defined!";  return op1 + op2;  }  window.addEventListener("load", function(){  try{  addNumber("**abc**",2);  }**catch**(exp){  console.log(exp);  }**finally**{  console.log("Method Executed");  }  }); | Exception scenario |

#### CALLING FUNCTION (UNDERSTANDING THIS)

1. **REGULAR FUNCTION CALL**
2. **FUNCTION CONSTRUCTOR.**
3. **CALLING FUNCTION IN CONTEXT TO AN OBJECT**
4. **USING “CALL”, “APPLY” AND “BIND“FUNCTION.**

|  |  |  |
| --- | --- | --- |
| **REGULAR FUNCTION CALL –** When the function called in a regular way. It gets called on the global object i.e. “window” object.  Example:  var print = function(){  console.log("Print Function Called");  console.log(this);  }  print(); | |  |
| **CALLING FUNCTION IN CONTEXT TO AN OBJECT**  Here “this” refers to the object on which function is called.  var myObj ={  prop :'propertyValue',  printProp: function(){  console.log(this);  }  }  myObj.printProp(); | |  |
| **FUNCTION CONSTRUCTOR**.  var foo =function(){  console.log(this);  }  **new foo();** |  | Whenever a function is called using **new** keyword, JS adds the highlighted lines internally, which is empty, so as the output.  var foo =function(){  //var this ={};  console.log(this);  //return this;  } |
| **USING CALL FUNCTION**   1. It can be used to invoke (call) a method with an owner object as an argument (parameter). 2. With call(), an object can use a method belonging to another object. 3. In this example we are assigning the **inflateTire** property (which happens to be a function) to mike object . 4. When **mike** object calls the **inflateTire** function as [mike.inflateTire();] , which hasbelow statement   **this.tirePressure = this.tirePressure + 3;**   1. But mike object has no **tirePressure** property so the value comes out to be **this.tirePressure + 3 = NaN**      1. To call the **inflateTire** we should have bicycle instance . To achieve this we have to call the method using “call()” function 2. **Call** method the object in the parameter on which we call the function   **mike.inflateTire.call(bicycle);** | | This example calls the fullName method of person, using it on person1:  var person = {  fullName: function() {  return this.firstName + " " + this.lastName;  }  }  var person1 = {  firstName:"John",  lastName: "Doe"  }  var person2 = {  firstName:"Mary",  lastName: "Doe"  }  **person.fullName.call(person1);** |
| **Example 2**  function Bicycle(name, speed, tirePressure) {  this.name = name;  this.speed = speed;  this.tirePressure = tirePressure;  this.inflateTire = function () {  this.tirePressure = this.tirePressure + 3;  }  }  function Mechanic(name) {  this.name = name;  }  var bicycle = new Bicycle("Hero", 10, 10);  var mike = new Mechanic("Mike");  mike.inflateTire = bicycle.inflateTire;  //mike.inflateTire();  **mike.inflateTire.call(bicycle);**  console.log(bicycle); |

#### CALL, APPLY AND BIND METHODS

***These methods help us to set the “this” variable manually***

|  |  |
| --- | --- |
| **CALL FUNCTION**   1. The call method helps in function borrowing i.e. An object borrows a function from another object. 2. Using the “call” function calls a function we can invoke the function on the object which we send in the call method parameter. 3. The “eimly” object borrows the “fullName” function from “john” object 4. Using call function we can invoke the fullName() on Emily object. | var john = {  firstName: "John",  lastName: "Doe",  fullName() {  return this.firstName + " " + this.lastName;  }  };  var eimly = {  firstName: "Eimly",  lastName: "Clark"  };  console.log(john.fullName());  **// function borrowing**  console.log(**john.fullName.call(eimly)**); |
| **SHARING FUNCTION AMONG OBJECTS USING CALL/ CALL METHOD WITH ARGUMENT**   1. In general scenario the functions are shared among the objects, but not the part of an object. 2. Using the **call** function we can call the “fullName()” function on the object which we pass in its parameter. We can pass 3. We can pass arguments also using call function. | var john = {  firstName: "John",  lastName: "Doe"  };  var eimly = {  firstName: "Eimly",  lastName: "Clark"  };  **function fullName(location) {**  **return this.firstName + " " + this.lastName + " " + location;**  **}** |
| console.log(**fullName.call(john, "England")**);  console.log(**fullName.call(eimly, "India")**); |
| **APPLY FUNCTION**   1. When we have more number of arguments in call function , so rather than sending them individually we can send it as an array 2. To send parameters as an array we need to use “**apply**” function. | var john = {  firstName: "John",  lastName: "Doe"  };  function fullName(location, state) {  return this.firstName + " " + this.lastName + " " + location + " " + state;  }  var eimly = {  firstName: "Eimly",  lastName: "Clark"  }; |
| **CALL USING APPLY**  console.log(fullName.**apply**(john, ["Laos", "US"]));  console.log(fullName.**apply**(eimly, ["Kolkata", "WB"])); |
| **BIND FUNCTION**   1. Bind function binds a function with an object 2. The bind function actually return the function with which it is bind to and then invoked later (unlike apply and call which are invoked immediately) | var john = {  firstName: "John",  lastName: "Doe"  };  function fullName(location, state) {  return this.firstName + " " + this.lastName + " " + location + " " + state;  }  var eimly = {  firstName: "Eimly",  lastName: "Clark"  };  let johnProfile = fullName.**bind**(john);  let eimlyProfile = fullName.**bind**(eimly);  console.log(johnProfile("Laos", "US"));  console.log(eimlyProfile("Kolkata", "WB")); |
| The **johnProfile** stores the function |
| **FUNCTION CURRYING**  **WHAT IS CURRYING?**  Currying is a process in functional programming in which we can transform a function with multiple arguments into a sequence of nesting functions. It returns a new function that expects the next argument inline.  **We can preset some of the argument of the function** | function checkShape(length) {  return function(breadth) {  return length == breadth ? "Square" : "Rectangle";  };  }  let lengthOfShape = checkShape(10); 🡨 Preseting the value (length of the shape)  console.log(lengthOfShape(5));  console.log(lengthOfShape(10)); |
| **FUNCTION CURRYING USING BIND**   1. We can also achieve function curing using bind method. 2. Here we are preseting the length value 3. Note - “this” is a window object(global Object) | function checkShape(length, breadth) {  return length == breadth ? "Square" : "Rectangle";  }  **let lengthOfShape = checkShape.bind(this, 10);**  console.log(lengthOfShape(5));  console.log(lengthOfShape(10)); |
| **FUNCTION CURRYING USING BIND ON CUSTOM OBJECT**  var john = {  name: "John",  speciesType(speciesType, gender) {  console.log(`${this.name} is ${speciesType} and ${gender}`);  }  };  var mary = {  name: "Mary"  }; | **CALLING**  let johnDetails = john.speciesType.bind(john, "Human");  let maryDetails = john.speciesType.bind(mary, "Human");  johnDetails("Male");  maryDetails("Female");   1. Here we preset the **speciesType** argument and passing the “gender” argument in subsiquent calls. |

## UNDERSTANDING THIS

## PROTOTYPE OBJECTS

**INHERITANCE CAN BE IMPLEMENTED IN JS USING THE PROTOTYPE OBJECT**

|  |  |  |
| --- | --- | --- |
|  | The prototype object can be considered as parent object.    **Here both object inherits the same object base i.e. they have same parent object i.e. prototype object** |  |
|  | * `The x and y object call all the methods it inherits from the parent prototype object. * Even in case of Arrays it has a prototype object – **That’s why we can call different functions like “indexOf” etc., which is actually defined in its prototype object.** |
| function Circle(radius) {  this.radius = radius;  this.draw = function() {  console.log("Draw Circle of Radius=" + this.radius);  };  }  let circle = new Circle(1);  let circle1 = new Circle(1);  console.log(**Object.getPrototypeOf(circle) === Object.getPrototypeOf(circle1)**);   * Here also all the circle object created (using new operator) from same constructor function will have same prototype object. * Any property / method defined in the prototype object is never becomes the part of circle object – They are accessed by circle object via inheritance. | |

|  |  |  |
| --- | --- | --- |
| function User(name) {  this.name = name;  this.greetUser = function () {  return "Hello ! " + this.name;  }  }  var mike = new User("Mike");  var alex = new User("Alex");  console.log(mike);  console.log(alex); |  |  |
| 1. In a typical object oriented programming language , the objects encapsulate the properties and shares the behavior(methods). The “methods” are never becomes the part of objects. 2. In JS when we create an objects using function constructor the functions and properties both becomes part of object (**as shown above**).Each object has their own copy of properties .To solve this issue we create function using **prototype** object. | | |

**ITERATING OBJECT PROPERTY**

|  |  |
| --- | --- |
| function Circle(radius) {  this.radius = radius;  this.draw = function() {  console.log("Draw Circle of Radius=" + this.radius);  };  }  let circle = new Circle(1); |  |
| **QUESTION: WHY WE CANNOT ABLE TO ITERATE PROPERTY DEFINED IN OBJECT’S PROTOTYPE?** | |
|  | |

#### PROTOTYPE OBJECTS

|  |  |  |  |
| --- | --- | --- | --- |
| * Prototype objects are used to create kind of **blueprint** from which we can create objects. * Function in JavaScript is an object. When a JS engine processes a function, it creates 2 objects   1. First - with the name as the function. That’s why functions can be accesses using its name.   2. Prototype object – These prototype object can be accessed using “prototype” property of the function object. | | |  |
|  | Function Object can be accessed using function name | **Function Prototype Object : foo.prototype** | |

#### PROTOTYPE OBJECT IN DETAILS

|  |  |
| --- | --- |
| function Foo(){ }  **Foo.prototype.prop="Hello World!”;**  var object1 = new Foo();  var object2 = new Foo();  console.log(object1.\_\_proto\_\_.prop); 🡪 “Hello World!”  console.log(object2.\_\_proto\_\_.prop); 🡪“Hello World!”  object1.\_\_proto\_\_ == object2.\_\_proto\_\_ 🡪 true  object1.prototype == object2.prototype 🡪true   * When we call a function using “**new**”, JS engine creates a property “\_\_proto\_\_”. This property is called **dunder proto** * We can access the function’s prototype object using dunder proto (\_\_proto\_\_) as show in diagram. |  |

#### SHARING VIA PROTOTYPE OBJECT

|  |  |  |
| --- | --- | --- |
| **PROPERTY SHARING** | | |
| 1. **The User function acts as a template for all the objects created from it.** 2. **User.prototype.greetUser =** This creates a function in the prototype object 3. All the objects created from User function constructor points to the same prototype object (highlighted) 4. If the property and functions are created in prototype object – A single copy of property / function will be shared by all the objects. 5. ***When we call a function on an object [mike.greetUser()] It will be looked up in the current object and if not found it is further looked up in prototype object. This is how the prototype properties are shared between the objects***. | | function User(name) {  this.name = name;  User.prototype.greetUser = function () {  return "Hello ! " + this.name;  }  }  var mike = new User("Mike");  var alex = new User("Alex");  mike.prototype == alex.prototype; 🡨 **TRUE** |
| When we add a property to a prototype object, it will be shared by all objects created using new operator from the function constructor.  function Foo(){ }  Foo.prototype.message ="Hello World!";  var Object1 = new Foo();  var Object2 = new Foo();  console.log(Object1.\_\_proto\_\_.message);  console.log(Object2.\_\_proto\_\_.message)  **OR**  console.log(Object1..message);  console.log(Object2.message);; |  | |
| **FUNCTION SHARING**  function FooBar(){ }  FooBar.prototype.printName =function(name){  return “Hello ”+ name;  };  var Obj1 = new FooBar();  var Obj2 = new FooBar(); | **CALING THE SHARED FUNCTION**  console.log(Obj1.printName("Alex"));  console.log(Obj2.printName("John"));  **OR**  console.log(Obj1.\_\_proto\_\_.printName("Alex"));  console.log(Obj2. .\_\_proto\_\_.printName("John")); | |

#### PROTOTYPE OBJECT - PROPERTY LOOK UP

|  |  |
| --- | --- |
| 1. When we access a property from an object. It do a look up in the object itself 2. If the object don’t have that property – it will do a look up in prototype object |  |
|  |

#### CONSTRUCTOR PROPERTY OF PROTOTYPE OBJECT

|  |  |
| --- | --- |
| The prototype object has a property “constructor” – from which we get hold of the function itself via the dunder proto object.  Step 1: Create a new Object  **var person = new Human();**  Step 2: Get the prototype object  **var proto = person.\_\_proto\_\_;**  Step 3: Access the function using constructor property on Prototype object  **var proto.constructor();** 🡨 This executes the function itself |  |

#### WHO CREATES THE PROTOTYPE OBJECT?

**OBJECT FUNCTION**

**Object function is a global function which gets called whenever an object is created without any specific constructor function**. For example –**var simple ={};**

|  |  |
| --- | --- |
|  | Accesing the Object Function |
|  | Calling the Object Function. Calling the Object function creates an empty object |
| **var objectVar = new Object();** | Creating an object on the Object Function |

|  |  |
| --- | --- |
| The dunder proto of the Object function’s prototype object is **NULL**. This terminates prototype chaining . |  |

|  |  |
| --- | --- |
| **PROTOTYPE CHAINING** | 1. When an object is created using new operator – The JS engine creates a prototype object of the constructor function 2. The Prototype object of Employee function gets created by JS Engine using global Object() constructor function (using new operator)   Like **var empProto = new Object();**   1. Since the Object() function is called using “new” operator – So the Global Object function too have the Prototype object. 2. The Object Function’s prototype object has a **dunder proto** too, which points to null. 3. This here is the point where the Prototype chaining terminates. |

## ES6 CONCEPTS

**ES6 FEATURES**

1. **VARIABLE DECLARATION LET AND CONST**
2. **IIFE V/S BLOCKS.**
3. **ARROW FUNCTIONS**
4. **DE STRUCTURING**
5. **ARRAYS**
6. **SPREAD OPERATOR**
7. **REST AND DEFAULT PARAMETER**
8. **MAPS**
9. **CLASSES AND SUBCLASSES**

### VARIABLES

|  |  |
| --- | --- |
| **CONST**   * const variables **cannot** be re-assigned * It has block scope * Const variables cannot be used before it is declared. * The const varibables are not hoisted | const toDoList = ['reading', 'walking'];  toDoList.push(“eating”); <- Valid  toDoList = []; 🡨 This gives error |
| **LET**   * let variables cannot be re-declared * It has a block scope * let variables cannot be used before it is declared. The let varibables are not hoisted. | let a = 10;  let a = 20; 🡨In-valid it cannot be redeclared  a = 30; 🡨 valid it can be re-assigned |
| “var” has function scope | “let” & “const” has block scope |
| for (var counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter)); 🡨 counter is accessible outside for block and prints **5**. | for (let counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter); 🡨 counter is not accessible outside the for block so it will be undefined here. |
| var counter=20;  for (var counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter) 🡨 o/p = 5 | let counter=20;  for (let counter = 0; counter < 5; counter++) {  console.log(counter)  }  console.log(counter) 🡨 o/p = 20 |

### STRING CONCATENATION – TEMPLATE LITERALS

|  |  |
| --- | --- |
| For String concatenation we usually use “+”.  Now in ES6 we can use a ***back tick*** symbol and use ${} to append dynamic values. They are called “***Template Literals***” | const name = "Adams";  const yearofBirth = 1982;  function calculateAge(yearofBirth) {  return new Date().getFullYear() - yearofBirth;  }  console.log(`My name is ${name} and age is ${calculateAge(yearofBirth)}`); |

### OBJECT LITERALS

|  |  |
| --- | --- |
| To create a object literals , of the property name is same as the parameter name , we can directly the parameters as is. | |
| **OLD WAY** | **ES6 WAY** |
| function getBook(author, title) {  return {  author: author,  title: title  }  }  var book = getBook('JK', "Harry Potter");  console.log(book); | function getBook(author, title) {  return {  author,  title  }  }  var book = getBook('JK', "Harry Potter");  console.log(book); |

### ARROW FUNCTIONS

|  |  |
| --- | --- |
| 1. Arrow function is a shorter version of an anonymous function. 2. We **cannot call an arrow function** before declaring | SYNTAX  () => {statements;} |
| **FUNCTION EXPRESSION**  var myFn = function () {  console.log(“Hello”);  };  **myFn();** | **CORRESPONDING ARROW FUNCTION**  var myFn = ()=> {  console.log(“Hello”);  };  **myFn();** |
| **ARROW FUNTION WITH PARAMETERS** | |
| **ARROW FUNTION WITH ONE PARAMETERS** | **ROW FUNTION WITH 2 OR MORE PARAMETERS** |
| var showData = **data** =>{  console.log(data);  };  showData(10);  When we have one parameter parenthesis are optional so can be removed | var addData = (op1, op2) => {  console.log(op1+op2);  };  addData(1,2);  For more than one parameters parenthesis are mandatory |
| **FOR SINGLE STATEMENT BRACES ARE OPTIONAL**  var showData = **data** => console.log(data);  showData(10); | **RETURN STATEMENTS ARE OPTIONAL**  var addData = (op1, op2) => op1+op2;  **addData**(1,2); |
| **The “this” of arrow function never switches it context. It always refers to the context where it is defined** | |

### DESTRUCTURING

* Destructing helps in extracting the values from a data structure like arrays or object
* Destrructing never changes the actual array or object.

|  |  |
| --- | --- |
| **ARRAY DESTRUCTURING** | In array de-structuring the order matters.  **EXAMPLE 4:**  let numbers =[1,2,3];  **[a, ,b]=numbers;**  console.log(a,b) 🡨o/p a=1; b= 3 |
| **EXAMPLE 1**  const person = ["Adam", "25", "Programmer"];  let [name, age, profession] = person;  console.log(`${name} is ${age} year old and work as a ${profession}`);  **EXAMPLE 2 : Destructuring with REST Parameter**  let names =['Max','rob','Alex'];  let [teacher,...students] = names;  console.log(teacher, students );  **EXAMPLE 3 : Destructuring in declaration**  let [a,b,c] =[1,2,3]; |
| **EXAMPLE 5: DEFAULT VALUE**  In destructuring if the value is not available. It can also be initialized with some default value.  let names =['Max','rob'];  let [student1,student2, student3="Alex"] = names;  console.log(student1,student2,student3 );  **EXAMPLE 6: SWAPING VALUE**  let a= 10;  let b=20;  [b,a]= [a,b];  console.log(a,b); |
| **OBJECT DESTRCUTURING**  const person = {  name: "Adam",  age: "25",  profession: "Programmer"  };  let { name, age, profession } = person;  console.log(`${name} is ${age} year old and work as a ${profession}`);  let { name: n, age: a, profession: p } = person; 🡨**Alias**  console.log(`${n} is ${a} year old and work as a ${p}`); | * In object de-structuring we can give an alias name of the object properties as well and can access the properties using alias names as shown in example |
| function calculateAgeAndRetirementAge(dob) {  let age = new Date().getFullYear() - dob;  return [age, 65 - age];  }  let [age, retirementAge] = calculateAgeAndRetirementAge(1982);  console.log(`Age is ${age} and retirement age is ${retirementAge}`); | De-structuring can be used to return multiple values from a function |

### DEFAULT PARAMETERS

|  |  |  |
| --- | --- | --- |
| * The default parameter is a way to set default values for function parameters if the value is no passed in for that parameter * The default parameters should be the last argument in the function parameters. | function add(op1, op2, op3 = 30) {  console.log(op1);  console.log(op2);  console.log(op3);  }  add(10, 20); | function add(op1, op2 = 20, op3) {  console.log(op1);  console.log(op2);  console.log(op3); 🡨 this will be undefined  }  add(10, 20); |
| * JavaScript also allows the use of arrays and null as default values. | function add(op1, op2, op3 = null){ } | var arr = ["a", "b", "c"];  function add(op1, op2, op3 = arr) {  console.log(op1);  console.log(op2);  console.log(op3[2]); 🡨 this will print “**c**”  }  add(10, 20); |

### SPREAD OPERATOR

|  |  |  |
| --- | --- | --- |
| In a regular function which has multiple parameters. We call the way we are doing . There is a better way of doing in ES5 and ES6 | | function addAges(age1, age2, age3, age4) {  return age1 + age2 + age3 + age4;  }  console.log(addAges(1, 2, 3, 4)); |
| **FUNCTION CALLS** | | |
| **ES5 (using apply)** | | **ES6 : SPREAD IN FUNCTION CALLS** |
| function addAges(age1, age2, age3, age4) {  return age1 + age2 + age3 + age4;  }  let ages = [1, 2, 3, 4];  console.log(addAges.apply(null, ages)); | | function addAges(age1, age2, age3, age4) {  return age1 + age2 + age3 + age4;  }  let ages = [1, 2, 3, 4];  console.log(addAges(...ages)); |
| **ARRAY LITERALS** | | |
| **ES5** | | **ES6: SPREAD IN ARRAY LITERALS** |
| **ARRAY CONCATENATION**  var arr1 = [0, 1, 2];  var arr2 = [3, 4, 5];  arr1 = arr1.concat(arr2); 🡨 Merged Array | | **ARRAY CONCATENATION USING SPREAD**  var arr1 = [0, 1, 2];  var arr2 = [3, 4, 5];  var mergedArray = [...arr1, 10, ...arr2, 12]; 🡨 Merged Array |
| **OBJECT EXPRESSIONS** | | |
| **ES5** | **ES6: SPREAD IN OBJECT EXPRESSIONS** | |
|  | var obj1 = { foo: 'bar', x: 42 };  var obj2 = { foo: 'baz', y: 13 };  var clonedObj = { ...obj1 };  // Object { foo: "bar", x: 42 }  var mergedObj = { ...obj1, ...obj2 };  // Object { foo: "baz", x: 42, y: 13 } | |
| **SPREAD** **DEFINATION**  The Spread operator expand /spread the elements  Spread syntax allows an iterable such as   * an array expression or string to be expanded in places where zero or more arguments (for function calls) or * elements (for array literals) are expected, * or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected. | | |

### REST PARAMETER

|  |  |  |
| --- | --- | --- |
| * The Rest parameters syntax looks exactly same as spread operator, but the functionality is just opposite to it. * The rest parameter syntax allows us to represent an indefinite number of single valued arguments as an array. * With the help of a rest parameter a function can be called with any number of arguments, * ***It should be the last argument in the function.*** |  | |
| **ES5** | | **ES6** |
| function isFullAge() {  var ageArr = Array.prototype.slice.call(arguments);  ageArr.forEach(currentElement => {  console.log(2020 - currentElement >= 18);  });  }  isFullAge(1982, 1997, 2010); | | function isFullAge**(...ages**) {  ages.forEach(currentElement => {  console.log(2020 - currentElement >= 18);  });  }  isFullAge(1982, 1997, 2010); |
| function fun(...input) {  let sum = 0;  for (let i of input) {  sum += i;  }  return sum;  }  console.log(fun(1, 2)); 🡨3  console.log(fun(1, 2, 3)); 🡨6  console.log(fun(1, 2, 3, 4, 5)); 🡨15 | | // rest with function and other arguments  function fun(a,b,...c){  console.log(`${a} ${b}`); //Mukul Latiyan  console.log(c); //[ 'Lionel', 'Messi', 'Barcelona' ]  console.log(c[0]); //Lionel  console.log(c.length); //3  console.log(c.indexOf('Lionel')); //0  }  fun('Mukul','Latiyan','Lionel','Messi','Barcelona'); |

***Note: Spread operators is used in function call but Rest Parameters are used in function declaration***

### FOR OF LOOP

|  |  |
| --- | --- |
| For..og loop is new addition in ES6 for iteration | let names =['Max','rob','Alex'];  for(let name of names)  console.log(name); |

### MAPS

Earlier to store key value pair we were using object , Maps are better to store key value pairs

* In maps we can store anything as a key but in object value are limited to string type.
* Maps are iterable but object are hard to iterate

|  |  |  |  |
| --- | --- | --- | --- |
| **CREATING A MAP** | var studentMap = new Map(); | **ITERATING A MAP** | |
| **SETTING DATA IN MAP** | studentMap.set(1, "Alex"); | **Using forEach Loop** | **Using for of Loop** |
| **GETTING DATA FROM MAP** | studentMap.get(1); | **Using for of Loop**  **var studentMap = new Map();**  **studentMap.set(1, "Alex");**  **studentMap.set(2, "Mary");**  **studentMap.forEach((key, value) => console.log(key + " " + value));** | var studentMap = new Map();  studentMap.set(1, "Alex");  studentMap.set(2, "Mary");  for (let [key, value] of studentMap.entries()) console.log(key + " " + value); |
| **DELETEING FROM MAP** | studentMap.delete(1) 🡨 1 is the key |
| **CLEARING A MAP** | studentMap.clear(); |
| **CHECK KEY EXIST** | studentMap.has(1) 🡨 1 is the key |
| **GETTING ALL KEYS** | studentMap.keys(); |  |  |
| **GETTING ALL VALUES** | studentMap.values(); |  |  |

### WEAKMAP

* The keys of WeakMap are reference types , like object . It cannot be primitive type.
* Having the restriction of keeping the keys of reference types help the browser to identify which items in Weakmap is longer in use and hence can be garbage cleaned. The helps in the memory management
* The WeakMaps are not enumrable i.e. they cannot be iterated because the size of these map are variable (because the unused key/value are always gets garbage collected by the browser if they are not in use)

|  |  |
| --- | --- |
| **CREATING A WEAK MAP** | var student = new WeakMap(); 🡨 Creating a weakmap  var student1 ={rollNo:1}; 🡨 Creating keys of reference type (object)  var student2 ={rollNo:2};  student.set(student1,"Alex");🡨 Setting the values in the Map  student.set(student2,"Erica");  console.log(student.get(student1)); 🡨- Getting the value fron Map |

### SETS

* It is a data structure with no duplicate values

|  |  |  |
| --- | --- | --- |
| **CREATING A SET** | var studentNames = new Set(); | **ITERATING A SET** |
| **SETTING DATA IN SET** | studentNames.add("Alex"); | **Using for of Loop**  for(let studentName of studentNames)  console.log(studentName); |
| **GETTING DATA FROM SET** | studentNames.get(“Alex”); |
| **DELETEING FROM SET** | studentNames.delete(“Alex”) |
| **CLEARING SET** | studentNames.clear(); |
| **CHECK VALUE EXIST IN SET** | studentNames.has(“Alex”); |

### WEAKSET

* WeakSet stores the values of reference types.
* Having the restriction of keeping the values of reference types help the browser to identify which items in WeakSet is longer in use and hence can be garbage cleaned. The helps in the memory management

|  |  |
| --- | --- |
| **CREATING A WEAK SET** | var student = new WeakSet(); 🡨 Creating a weakset  var student1 = {rollNo:1,name:"Alex"};🡨 Creating object of reference type (object)  students.add(student1);🡨 Setting the values in the Set |

## ASYNCHRONOUS JAVASCRIPT

1. The Asynchronous code never block the code , this is something run behind the scene
2. Refer “Event Loop”

|  |  |
| --- | --- |
| **SAMPLE ASYNC CODE** | **CALLBACK** |
| function getDataFromService(requestData, callback) {  setTimeout(function(){  callback({ responseData: 10 });  }, 5000);  }  getDataFromService(5, function (data) {  console.log("Data From Service=" + data.responseData);  }); | var getGitHubData = function(userId) {  $.ajax({  url: "https://api.github.com/users/" + userId,  success: function(response) {  $.ajax({  url: "https://api.github.com/users/" + response.login + "/repos",  success: function(repoResponse) {  console.log(repoResponse);  },  error: function(error) {  console.log(error);  }  });  },  error: function(error) {  console.log(error);  }  });  };  getGitHubData("avishekhsinhaRepo"); |
| 1. In Async calls when the request of a async call depends on the result of previous call , we keep on nesting the calls 2. For more such calls / nesting thing becomes unmanageable. This scenario is called a **“CALLBACK HELL”** 3. ES6 comes with a solution which call solve the problem of Callback hell – The Solution has been catered using PROMISES. 4. Promises are used to deal with asynchronous JavaScript. |

### PROMISES

**WHAT IS PROMISE?**

* Promise is an object which keep track about weather a certain event has happened or not
* And if that event happened what will happen after the event has occurred.

**Here event means aync events like fetching data from server**

|  |  |
| --- | --- |
| **STATES OF PROMISE**   1. Before the event has happened, we call the state has pending 2. Once the event has happened the promise goes in resolved state. 3. if the promise is resolved , it can resolved as successful or error 4. If it is successful it’s called **FULFILLED** and **REJECTED** if it’s failed. |  |

**IMPORTANT POINTS ON PROMISE**

1. With respect to code standpoint we can **PRODUCE PROMISES AND CONSUME PROMISES**
2. When we CREATE A PROMISE, we create an instance of promise and send the result for consumption.
3. When we CONSUME PROMISES, we use callback function to accept the result send by the Promise(in Step2)
4. While consuming we have 2 callback functions
   1. one if the promise is FULFILLED called the success callback
   2. Another if promise in REJECTED called error callback

#### PRODUCING PROMISE

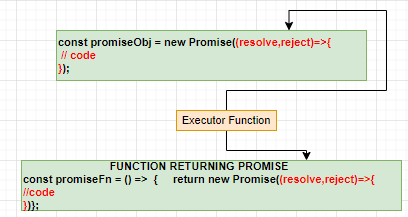
1. The promise object is created using new keyword
2. The function passed in the argument of Promise object is called Executor function

##### EXECUTOR FUNCTION

The Executor function :

* 1. Has two argument “resolve” &” reject”.
  2. In executor function we place the async code.

They are callback functions which will be called based on the Promise States (FULLFILLED OR REJECT)



##### RESOLVING AND REJECTING PROMISES

|  |  |
| --- | --- |
| **RESOLVING PROMISES** | **REJECTING PROMISES** |
| const getGitHubData = new Promise(function (resolve, reject){ **resolve**(response);  });  When the promise is FULFILLED **resolve()** callback is called | const getGitHubData = new Promise(function (resolve, reject) {  **reject**(response);  });  When the promise is REJECTED **reject()** callback is called |
| **EXAMPLE 1**  const promiseObj = new Promise((resolve,reject)=>{  resolve({id:2});  });  promiseObj.then((result)=>{  console.log(result);  }).catch(error =>{  console.log(error);  }); | **EXAMPLE 2**  const promiseFn = () => { return new Promise((resolve,reject)=>{  resolve({id:1});  })};  promiseFn().then((result)=>{  console.log(result);  }).catch(error =>{  console.log(error);  }); |

#### CONSUMING PROMISE

1. To consume promise we use then() method. Then() method returns a promise and can be chained.
2. It takes 2 arguments : a success callback and a error callback.

|  |  |  |
| --- | --- | --- |
| **WHEN PROMISE IS RESOLVED**  getGitHubData.then(function(data){ }); | **WHEN PROMISE IS REJECTED**  getGitHubData.catch(function(data){ }); | |
| **PUTTTING IT ALL TOGETHER** | | |
| var getGitHubData = new Promise(function(resolve, reject) {  **resolve**({ login: "loginId" });  });  getGitHubData.then(function(**data**) {  console.log(data.login);  }) .catch(function(error) {  console.log(error);  }); | var getGitHubData = new Promise(function(resolve, reject) {  **reject**(new Error("Error Occured"));  });  getGitHubData.then(function(data) {  console.log(data.login);  }) .catch(function(**error** ) {  console.log(error);  }); | |
| * When the promise is fulfilled we call resolve() * The result data is received in the argument of resolve * When we call the resolve function the result data is the passed to the callback function of **then**() | * When the promise is rejected we call reject() * The error data is received in the argument of reject() * When we call the reject function the error data is the passed to the callback function of **catch**() | |
| **PUTTING IT ALL TOGETHER** | | |
| **CREATING A PROMISE** | | **CONSUME THE PROMISE** |
| var makeRequest = function(url) {  var request = new XMLHttpRequest(); // Create the XHR request  // Return it as a Promise  return new Promise(function(resolve, reject) {  request.onreadystatechange = function() {  // Setup our listener to process completed requests  if (request.readyState !== 4) return;  // Only run if the request is complete  if (request.status >= 200 && request.status < 300) {  // Process the response  **resolve**(request.response); // If successful  } else {  **reject**({ status: request.status, statusText: request.statusText }); // If failed  }  };  request.open("GET", url, true); // Setup our HTTP request  request.send(); // Send the request  });  }; | | var url = “<https://api.github.com/users/avishekhsinhaRepo>”;  makeRequest(url).then(  data => {  console.log(data);  }).catch( error => {  console.log("Promise rejected."+ error.message);  }); |
| **PROMISE IMPL USING JQUERY AJAX METHOD** | | |
| var gitHubPromise = function(gitUrl) {  return $.ajax(gitUrl);  };  ***The Ajax function returns Promise so we can call “then()” and “catch()” method on that promise object*** | | gitHubPromise("https://api.github.com/users/avishekhsinha Repo") .then(function(data) {  console.log(data);  }) .catch(function(error) {  console.log(error);  }); |

### PROMISES CHAINING

|  |  |  |
| --- | --- | --- |
| const students=[  { id:1, name:'Alex' },  { id:2, name:'John' },  { id:3, name:'Rob' }  ];  const hobbies= [  { id:1, hobbies:[ 'Sports', 'Cooking'] },  { id:2, hobbies:[ 'Hiking', 'Cricket'] },  { id:3, hobbies:[ 'Tracking', 'Chess'] }  ];  const getStudent = (studentName)=> {  return new Promise((resolve,reject)=>{  const student = students.find(element => element.name == studentName);  if(!student) reject(new Error("Student Not Found"));  resolve(student.id);  });  } | | const getHobbies = (studentid)=> {  return new Promise((resolve,reject)=>{  const student = hobbies.find(element => element.id == studentid);  if(!student) reject(new Error("No Associated Hobbies"));  resolve(student.hobbies);  });  }  getStudent('Rob').then(data=> getHobbies(data).  then(hobbies => console.log(hobbies)))  .catch(err =>{ console.log(err.message)}); |
| **CONSUMING PROMISES** | **PRODUCING PROMISES USING AJAX CALL** | |
| var getGitDetailsForUser = function(userId) {  getLoginDetails(userId)  .then(function(data) { #1  let loginId = JSON.parse(data).login;  return getRepoDetails(loginId); #2  }) .then(function(data) {  console.log(data);  }) .catch(function(error) {  console.log(error);  });  };  getGitDetailsForUser("avishekhsinhaRepo");  ***Note:#1 value will become input at #2 and so on*** | **Usually producing promises usually we do via some library like JQuery**  const getRepoDetails = function(loginDetails) {  return $.ajax("https://api.github.com/users/" + loginDetails + "/repos");  };  const getLoginDetails = function(userId) {  return $.ajax("https://api.github.com/users/" + userId);  };   * getRepoDetails & getLoginDetails are Promise Object | |

### ASYNC AND AWAIT

|  |  |
| --- | --- |
| * When we chain the Promises while consuming the promises- it looks still becomes clumsy and hard to maintain for that we use **async and await** * async and await is used when we consume promises. * The function which is consuming the promises is marked with **async and the promises are marked as await.** * ***The async function always return a PROMISE object*** and executes in background without blocking the regular code flow. We need to use await on in async function * For example while consuming the promises .First #1 promise will execute till that time #2 promise will wait . Once #1 is completed #2 will execute. | |
| **IMPLEMENTING ASYNC AND AWAIT**  **The above example in promise chaining can be also achieved by async and await** | |
| const getStudentAsync = async (name) => {  const studentId = await getStudent(name);  const studentHobbies = await getHobbies(studentId);  console.log(studentHobbies);  }  getStudentAsync('Rob'); | **IF ASYNC function returns a value , the returned value can be intercepted using then()**  const getStudentAsync = async (name) => {  const studentId = await getStudent(name);  const studentHobbies = await getHobbies(studentId);  return (studentHobbies);  }  getStudentAsync('Rob').then(hobbies => console.log(hobbies)); |

#### HOSTING

1. Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

|  |  |  |
| --- | --- | --- |
| * In the JS script code since compilation step occurs before it actually get executed. In the compilation step the code is scanned and variable are identified and set into the global /function scope * Due to the compilation step it does actually matter where the variables are declared, it stills holds good. * The variable declarations are hoisted on the top and assigned the value “**undefined**” | ACTUAL CODE | HOSTED CODE BY JS ENGINE |
| a=10;  console.log(b);  b=20  c++;  **var a;**  **var b;**  **var c;** | **var a;**  **var b;**  **var c;**  a=10;  console.log(b); 🡪undefined  b=20  c++; |
| **FUNCTION** | | |
| We call the function expression before declaration. **ITS VALID** | fn();  function fn() {  } | function fn() {  }  fn(); |
| **FUNCTION EXPRESSION** | | |
| We cannot call the function expression before declaration. **ITS INVALID** | fn();  var fn = function(){  } |  |
| For the given code, the highlighted variable prints “**undefined**”. Because the variable in the function is also hoisted inside the function as shown. | var greetings = "Hello!";  function greetMe() {  console.log(greetings);  var greetings = "Hi";  }  greetMe(); | var greetings;  greetings = "Hello!";  function greetMe() {  var greetings;  console.log(greetings);  greetings = "Hi";  }  greetMe(); |

## CLASSES IN JAVASCRIPT

* Classes introduced in ES6. They are just syntactical sugar on the old way of creating object and implementing inheritance using function constructor.
* They also act as a blueprint for all the objects we create using the class.

|  |  |
| --- | --- |
| **CREATING OBJECT** | |
| **ES5 – USING FUNCTION CONSTRUCTOR** | **ES6 – USING CLASS** |
| var Person = function(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  };  Person.prototype.getFullName = function() {  return this.firstName + " " + this.lastName;  };  let person = new Person("Mike", "Clark");  console.log(person); | class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  }  let person = new Person("Mike", "Clark");  console.log(person); |
| **Person.prototype 🡨** Here “**Person**” is Function Constructor | **Person.prototype 🡨** Here “**Person**” is a class |
| * When we create a method in a class it actaully creates a Person’s class prototype. * **The classes in JavaScript are not hosited** , means we can use class only after its declaration unlke functuon constructor. * ***We can only add methods to classes not properties*** * ***The ES6 class body are always execusted in strict mode*** | |

### SYNTAX

|  |  |
| --- | --- |
| **SAMPLE CLASS** | **DETAILS** |
| class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  get fullName() { #2  return this.firstName + " " + this.lastName;  }  set fullName(name) { **#3**  let parts = name.split(" ");  this.firstName = parts[0]; this.lastName = parts[1];  }  static isSamePerson(person1, person2) { #4  return (  person1.firstName == person2.firstName &&  person2.lastName == person2.lastName  );  }  }+  let mike = new Person("Mike", "Doe");  let john = new Person("John", "Clark");  console.log(mike.fullName);  john.fullName = "Dan Wills"; 🡨 **Call to setter**  console.log(john.fullName); 🡨 **Call to getter**  console.log(Person.isSamePerson(mike, john)); 🡨 **Calling a static function** | 1. The class can be declared using “**class**” keyword 2. The class can have constructor (parameterized / non-parameterized)   **GETTERS & SETTERS**   1. Use “**get**” keyword to create a getter.#2- is getter of an “fullName” property. 2. Use “**set**” keyword to create a getter.#3- is setter of an “fullName” property.   **STATIC FUNCTION**   1. #4 – We use “**static’** keyword to declare a static function. 2. Static functions are never a part of an object , but its part of class, so static functions are called using class name(**#11**) 3. The static function are not inherited   **Note :**   1. ***When we set the value of a property it implicitly calls its setter*** 2. ***When we access a value of a property it implicitly calls its getter.*** |

### INHERITANCE

|  |  |
| --- | --- |
| **ES5** | **ES6** |
| function Shape() {  this.x = 0;  this.y = 0;  }  Shape.prototype.move = function(x, y) {  this.x += x;  this.y += y;  console.info('Shape moved.');  };  function Rectangle() {  **Shape.call(this);** // call super constructor.  }  **Rectangle.prototype = Object.create(Shape.prototype); #1**  **Rectangle.prototype.constructor = Rectangle; #2**  **var rect = new Rectangle();**  console.log('Is rect an instance of Rectangle?', rect instanceof Rectangle); 🡪 true  console.log('Is rect an instance of Shape?', rect instanceof Shape); 🡪 true  rect.move(1, 1); 🡪 Outputs, 'Shape moved.' | class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  }  class Programmer extends Person {  constructor(firstName, lastName, skill) {  super(firstName, lastName);  this.skill = skill;  }  }  let mike = new Programmer("Mike", "Wilson", "Java");  console.log(mike);   * Person is the parent class and Programmer is the child class which inherits it. * super() 🡪 Calls the constructor of parent class |
| **DETAILS ON INHERITANCE IN ES5: USING FUNCTION CONSTRUCTOR**   1. Shape is the super class of Rectangle Object 2. **move**() has been created in Shape object (Shape prototype) so that it can be shared by all the object created from Shape Function Constructor. 3. **new Rectangle(); 🡪** This calls the constructor of Rectangle which has a call to constructor of super class [**Shape.call(this);**]. “this” object here will be object of Rectangle type. 4. **#1🡪**subclass extends superclass 5. **#2 🡪**if you don't set **Object.prototype.constructor** to Rectangle, it will take prototype.constructor of Shape (parent). To avoid that, we set the prototype.constructor to Rectangle (child). | |

### POLYMORPHISM

|  |  |
| --- | --- |
| **ES5** | **ES6** |
|  | class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  hobby() {  return "Music";  }  }  class Programmer extends Person {  constructor(firstName, lastName, skill) {  super(firstName, lastName);  this.skill = skill;  }  hobby() {  return "Games";  }  }  let person = new Person("Dan", "Wilson");  console.log(person.hobby());  person = new Programmer("Mike", "Wilson", "Java");  console.log(person.hobby()); |

## HTML 5 WEB WORKER

**WHAT DO WEB WORKERS DO?**

1. This help us in executing multiple threads in parallel
2. The web worker consumes lot of resources. So must use when it’s truly needed.
3. The threads are kernel level thread which runs on client machine

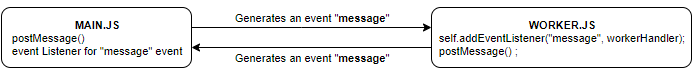
**TYPES OF WEB WORKERS?**

1. Dedicated Web workers
2. Shared Web worker

### DEDICATED WEB WORKERS

1. Dedicated web worker are created and managed by main thread.
2. The Dedicated web worker is forced to quit when the main thread terminates.

**CREATING DEDICATED WEB WORKERS**



**HOW MAIN JS AND WORKER JS COMMUNICATE?**

1. The main JS pass the values from main.js to worker JS via postMessage() function argument
2. Along with passing the value ,the main thread triggers an event “message” as well
3. On the worker we need to implement the event listener for “message” event to receive the vale
4. Once the worker finished it task the workers then triggers a “message” event and pass the value to main thread function using postMessage() function argument
5. The main thread need to implement the event handler for “message” event which receives the result.

|  |  |  |
| --- | --- | --- |
| **HTML** | <input type="text" name="txt" id="txt" />  <input type="button" onclick="getResult()" /> | **STEPS TO CREATE WEB WORKER**   1. #1 🡪Check whether the web worker is supported 2. #2 🡪 Creates an object of Worker with worker js file as argument. 3. The code in the worker file runs in different thread asynchronously. 4. #3 🡪 Success Event handler of “**message**” event 5. #4 🡪 Error Event handler of “**message**” event 6. #5 🡪 postMessage() passes the value to web worker 7. #6 🡪 Event listener of “message” event |
| **MAIN JS** | function getResult() {  if (Worker != undefined) { #1  var txtValue = document.getElementById("txt").value;  var worker = new Worker("worker.js"); #2  worker.onmessage = successHandler; #3  worker.onerror = errorHandler; #4  worker.postMessage(txtValue); #5  }  }  function successHandler(e) {  console.log(**e.data**);  }  function errorHandler(e) {  console.log(**e.data**);  } |
| **WORKER JS** | self.addEventListener("message", workerHandler); #6  function workerHandler **e.data** (e) {  postMessage(+ " " + "World!"); #7  } |

**HANDLING ERRORS - CUSTOM ERRORS**

|  |  |
| --- | --- |
| **MAIN JS** | **WORKER JS** |
| function add() {  if (Worker != undefined) {  var txtValue1 = document.getElementById("number1").value;  var txtValue2 = document.getElementById("number2").value;  var worker = new Worker("worker.js");  worker.onmessage = successHandler;  worker.onerror = errorHandler;  var requestObj = {  num1: txtValue1,  num2: txtValue2  };  worker.postMessage(**requestObj**);  }  }  function successHandler(e) {  if (e.data.errorMsg) {  console.log("Invalid Input");  } else {  console.log("Sum=" + e.data);  }  }  function errorHandler(e) {  console.log(e.message + " in " + e.filename);  } | self.addEventListener("message", add);  function add(e) {  if (isNaN(e.data.num1) || isNaN(e.data.num2)) {  var errorObj = {  errorMsg: "Not a Number"  };  postMessage(errorObj);  } else {  var result = parseInt(e.data.num1) + parseInt(e.data.num2);  postMessage(result);  }  } |
| **HTML**  <input type="text" name="txt1" id="number1" />  <input type="text" name="txt2" id="number2" />  <input type="button" onclick="add()" /> |

**IMPORTING SCRIPTS**

* We can able to import js files in the worker files as well using importScript(“<file\_path>”)
* Below example shows the way to import a JS file which in the same directory , where worker is present.

|  |  |
| --- | --- |
| **WORKER FILE** | **FILE THAT HAS TO BE IMPORTED** |
| **importScripts("workerHelper.js");**  self.addEventListener("message", add);  function add(e) {  if (isNaN(e.data.num1) || isNaN(e.data.num2)) {  var errorObj = {  errorMsg: "Not a Number"  };  postMessage(errorObj);  } else {  var result = addNumbers(e.data.num1, e.data.num2);  postMessage(result);  }  } | function addNumbers(num1, num2) {  return parseInt(num1) + parseInt(num2);  } |
| This helps in centralizing the logic in a common file and the web worker can import the file and use the common code. |

**TERMINATING THE WEBWORKER**

* Since the workers are OS kernel level threads so they consume lot of resources. We should terminate the web worker when it’s not needed. There are 2 ways we can terminate the web worker

|  |  |
| --- | --- |
| 1. **MAIN THREAD TERMINATES THE WEB WORKER**   Main thread can terminates the webworker using terminate() function  **<webwork\_object>.terminate();**  **MAIN.JS**  var worker;  function getResult() {  if (Worker != undefined) {  var txtValue = document.getElementById("txt").value;  worker = new Worker("worker.js");  worker.onmessage = successHandler;  worker.onerror = errorHandler;  worker.postMessage(txtValue);  }  }  function successHandler(e) {  console.log(**e.data**);  **worker.terminate();**  }  function errorHandler(e) {  console.log(**e.data**);  **worker.terminate();**  } | 1. **WEB WORKER TERMINATES ITSELF**   Web worker thread can terminates itself the webworker using close() function  **self.close()**  **WORKER.JS**  self.addEventListener("message", workerHandler);  function workerHandler **e.data** (e) {  postMessage(+ " " + "World!");  **self.close();**  } |

### SHARED WEB WORKERS

|  |  |
| --- | --- |
| **CREATING SHARED WEB WORKER**   1. When the main thread triggers the shared webworker Unlike dedicated worker the shared worked keep on executing even if the main thread terminates 2. Multiple main threads can communicate with shared web worker, but the main thread will share the same instance of Shared worker. 3. Note – Any error in shared web worker is never reported back to the main thread, as they run independently. | For example let’s consider we want to create an application which will increase the count by 1 of every button click |

# TYPESCRIPT

**WHAT IS TYPESCIPT?** - It’s a open source project by Microscoft. It’s a superset of JavaScript

**WHY TYPESCRIPT? -**Typescript help us to overcome the problem in Javascript. For example .

**ISSUES WITH JAVASCRIPT:-**

1. Lack type checking. For example,

var a =10;

a=”Hello” //allowed in JS

1. Issue with Function

**Function Defination** : function add(a,b){ return a+b; }

**Function call** :

* 1. add(1) 🡪 O/P = NaN // Allowed in JS
  2. add(1,2,3) 🡪 O/P = 3 // Third parameter has been ignored

1. **Object Modification –** Lets say we have an object

var person ={ 'firstName': 'John','lastName':'Doe'} **-** This javascript object can be modified at any point of time . We don’t have a way to control

1. Adding new propery is fairly simple = person.age=25

Final object will be : {firstName: "John", lastName: "Doe", age: 25}

These all limitations or quirks can be handled in Typescript.

|  |  |
| --- | --- |
| **HOW TYPESCRIPT WORKS?**   1. We write a code in Typescript 2. Typescript compiler compiles the Typescript to Native Javascipt code. Typescript compiler used Node.js as its compiler 3. The Advantage we get here as follows As the TS compiler compiled to native JS so it is understandable by all the browsers |  |

### TYPESCRIPT DEVELOPMENT ENVIRONMENT SET UP

1. Install Editor Visual Studio code
2. Install the latest stable version **NODE**

**WHAT IS NODE?** Earlier the runtime environment for Javascript is browser. Node provides the runtime environment for Javascript outside the browser.

|  |  |
| --- | --- |
| * Create a folder where you want to create your typescript file * Now to install typescript we need to install it using npm(node package manager). | |
| **CREATING A TS PROJECT**  Step 1 : Create a directory and generate a package.json file : **npm init --yes** | |
| **INSTALLING TS DEPENDENCIES** | **npm install typescript ts-node-dev**  Here **–g** Means we want to install typescript globally.  Otherwise TS will be installed in the folder where we are in |
| **COMPILE TS FILE** | **tsc** <type\_script\_file\_name> . e.g. tsc hello-world – ***the “ts” extension is optional*** |
| **RUN COMPILED JS** | node <js\_file\_name>.js e.g. node hello-world.js |
| **TO WATCH THE CHANGES** | 1. Install the dev dependency : npm install --save-dev **lite-server 🡨** lite server is a development server 2. **Update the start script in package.json**   **"start":"lite-server"** |

|  |  |  |
| --- | --- | --- |
| **Typescript code** | **Compiled JS Code** | When the typescript code is compiled the compiler takes off the types. So, the type declaration in TS comes in picture at compile time only. |
| var a :number;  var b:string;  var c:boolean;  a=10;  b='hello';  c=true; | var a;  var b;  var c;  a = 10;  b = 'hello';  c = true; |
| var a :number; a=10; a='Hello';  In the above declaration a is a number type. We are still assigning the string to it. TS compiler will complain about it at compile time , but it will still generate the JS output file as below . Because it a valid JS syntax  var a;  a = 10;  a = 'Hello'; |

### TYPESCRIPT TYPES

* NUMBER
* STRING
* BOOLEAN
* UNDEFINED
* NULL
* TUPLE
* UNION
* LITERAL
* ANY
* ENUMS

|  |  |  |
| --- | --- | --- |
| **Datatype** | **Declaration** | **Details** |
| Number | var a:number; | “a” can store only number values |
| Boolean | var a:boolean; | “a” can store only boolean values |
| String | var a:string; | “a” can store only string values |
| Undefined | var a:undefined; | “a” can store only undefined |
| Null | var a:null; | “a” can store only null |
| Tuple | var a:[number,Boolean];  **it’s a fixed length and fixed type datatype** | Example :  const user : {  name:string;  age:number;  role:[string,number];  } ={  name:'John',  age:30,  role:['admin',1]  } |
| a can store an arrays of 2 elements . e.g.  var myTuple:[number,boolean];  myTuple=[1,true]; |
| Array | var a:number[]; | a is an array which can store numbers |
| Object type (Nested object) | Let's say you have this JavaScript object:  const product = {  id: 'abc1',  price: 12.99,  tags: ['great-offer', 'hot-and-new'],  details: {  title: 'Red Carpet',  description: 'A great carpet - almost brand-new!'  }  } | This would be the type of such an object:  {  id: string;  price: number;  tags: string[],  details: {  title: string;  description: string;  }  } |

#### ENUM

|  |  |
| --- | --- |
| enum Roles {ADMIN, AUTHOR, READ\_ONLY};  const user = {  name:'Max',  age:'30',  role: Roles.ADMIN  }  if(user.role == Roles.ADMIN){  console.log("Role is Admin");  } |  |

#### FUNCTION TYPE

|  |  |
| --- | --- |
| function add(number1:number, number2:number){  return number1 + number2;  }  let addFn : Function = add;  console.log(addFn(1,3)); |  |

### TS FUNCTIONS

|  |  |
| --- | --- |
| JS WAY | TS WAY |
| function addNumber(a,b){ return a+b; }   1. console.log(addNumber("foo",1)); O/P – **foo1** 2. console.log(addNumber(1,2,3)); O/P - **3**   Here the JS function has no type check for the function parameter passed  #2 – We can pass more parameters to the function. JS function simply ignores the extra parameters. | function addNumber(a :number,b:number){  return a+b;  }   1. console.log(addNumber(2,1)); 2. console.log(addNumber(2));   in TS we have a type check for function parameter types and number of parameters passed to it.#2 will give an error by TS compiler |
| **PASSING VARIABLE PARAMETERS** | |
| function addNumber(a :number, b:number,c?){  return a+b+c;  } | Here c is a optional parameter  console.log(addNumber(2,1,3)); // Valid TS function call  console.log(addNumber(2,1)); // Valid TS function call |

### IMPLICIT TYPING (TYPE INFERENCE)

|  |  |
| --- | --- |
| **EXPLICIT TYPING** | **IMPLICIT TYPING** |
| var a=10:number;  a='hello' 🡨TS compiler gives an error as it’s a string | var a=10  a='hello' 🡨 TS compiler gives an error as it’s a string . Here TS compiler understands that “a” is meant for number type values. |

##### TYPESCRIPT FUNCTION



### TYPESCRIPT CLASSES AND FUNCTION

Member Variable can have both implicit and Explicit Typing as highlighted below.

|  |  |
| --- | --- |
| **CLASS DECALARATION** | |
| class Person{  firstName : string;  lastName :string;  **constructor**(firstName , lastName){  this.firstName = firstName;  this.lastName = lastName  }  getFullName(){  return this.firstName +" "+ this.lastName;  }  } | * firstName and lastName are member variables * getFullName() is a method of the class. * constructor(..) is a constructor of Person class   Note :   * ***In TS classes cannot have overloaded constructor and method***.   **CREATING OBJECT FROM CLASS**  var aPerson = new Person("John", "Doe"); console.log(aPerson.getFullName()); |

#### PRIVATE & PUBLIC ACCESS MODIFIERS

|  |  |
| --- | --- |
| class Employee {  private name:string;  private department:string[]= [];  constructor(name:string){  this.name = name;  }  addDepartment(departmentName:string){  this.department.push(departmentName);  }  printEmployeeInfo(){  console.log(this)  }  }  const mike = new Employee("mike");  mike.addDepartment('Technology');  mike.addDepartment('Sales');  mike.prprintEmployeeInfo(); | 1. The member variables are by default public variables and can be accessed from anywhere. 2. TS has access modifies like private, public and protected. 3. All private variable call be accessed only withing the class. |

###### SHORTHAND FOR INITIALIZATION

|  |  |
| --- | --- |
| class Employee {  private department:string[]= [];  constructor(private name:string, private id:string){ }  addDepartment(departmentName:string){  this.department.push(departmentName);  }  printEmployeeInfo(){  console.log(this)  }  }  const mike = new Employee("mike", 'EMP001');  mike.addDepartment('Technology');  mike.addDepartment('Sales');  mike.printEmployeeInfo(); | * Intilaization of properties can be done in constructor using a shorthand * In this way the parameter passed in constructor and the class property has to be same. |

#### READ ONLY MODIFIERS

|  |  |
| --- | --- |
| class Employee {  private department:string[]= [];  constructor(private name:string, private **readonly** id:string){ }  } | * The readonly modifier allow the property to initialize just once. * There after it cannot be modiflied even in the method inside the class. |

### INHERITANCE

|  |  |
| --- | --- |
| **EXTENDING A CLASS(INHERITANCE)** | |
| class Human {  constructor(private firstName : string, private lastName : string){}  }  class Student extends Human{  constructor(firstName : string, lastName : string,  private id :string,private department :string){  **super(firstName,lastName);**  }  }  let john = new Student('John','Doe','ST001','Engineering');  console.log(john); | * Extends keyword is used to inherit the classes * Super keyword is used to call the implementation of super classes * TypeScript does not support multilevel inheritance. |

#### PROTECTED MODIFIERS

* Protected properties can be accessed within the class and the class extending that base class.

### GETTERS AND SETTERS IN CLASSES

|  |  |
| --- | --- |
| class Account{  private \_balance=0;  set balance(balance:number){ 🡨 Setter  this.\_balance = balance;  }  get balance(){🡨 getter  return this.\_balance;  }  } | const account = new Account();  account.balance= 100; 🡨 Calling Setter  console.log("Balance:" , account.balance); 🡨 Calling Getter |

### STATIC PROPERTIES AND METHOD

|  |  |
| --- | --- |
| class Product{  static currentYear:number =2021;  constructor(private productId:string, private productName:string, private manuFacturingYear:number){}    static isCurrentYearProduct(product:Product){  return Product.currentYear == product.manuFacturingYear;  }  }  const pr1 = new Product("P001","Product1",2021);  console.log(Product.isCurrentYearProduct(pr1)); | * Static properties and method are used to create global constants and utility function * They can be accessed using Class name. * Static member cannot be accessed using “this” keyword |

### ABSTRACT CLASSES

|  |  |
| --- | --- |
| abstract class Shape{  **abstract draw():void;**  }  class Circle extends Shape{  constructor(private radius:number){  super();  }  draw() {  console.log("Draw Circle");  }  }  const circle = new Circle(1);  circle.draw(); |  |

### SINGLETON AND PRIVATE CONSTRUCTORS

|  |  |
| --- | --- |
| class SimpleSingleton{  private static instance: SimpleSingleton;  private constructor(){}  static getInstance(){  if(SimpleSingleton.instance){  return this.instance;  }  this.instance = new SimpleSingleton();  return this.instance;  }  }  const i1 = SimpleSingleton.getInstance();  const i2 = SimpleSingleton.getInstance();  console.log(i1,i2,i1===i2); |  |

### INTERFACES

|  |  |
| --- | --- |
| **INTERFACES** | |
| interface Person {  firstName: string;  lastName: string;  getFullName(): string;  }  ***The Person interface reference can hold the object of classes which implements Person interface*** | class Teacher implements Person {  firstName: string;  lastName: string;  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName(): string {  return this.firstName + " " + this.lastName;  }  }  let mike: Person = new Teacher("Mike", "Holding");  console.log(mike.getFullName()); |
| * Interfaces also help is creating own custom types * Here **confimPassword** is an optional property   This declares an object of type User  user = { userName: "John", password: "Hopkin" }; | interface User {  userName: string;  password: string;  confimPassword?: string;  }  let user: User; 🡨 variable declared to custom type (User)  user = { userName: "John", password: "Hopkin" };  console.log(user); |

### MODULES IN TYPESCRIPT

1. We can modularize the typescript source code into multiple files
2. We have to export the interfaces and classes from one file and import it in another file(using file name)

|  |  |
| --- | --- |
| **EXPORT INTERFACE** | **IMPORTED IN A CLASS** |
| export interface Human {  firstName:string;  lastName :string;  getFullName():string;  } | import {Human} from './typescript-interfaces';  class Person implements Human{  firstName : string;  lastName : string;  constructor(firstName:string, lastName:string){  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() : string {  return "Hello !"+" "+this.firstName + " "+ this.lastName;  }  } |
| When we import a file in another file typescript executes the imported file  **OUTPUT**  From Person  Hello World ! John Doe | export class Person{ }  **console.log("From Person");**  import {Person} from './person';  class Programmer extends Person{  getFullName() : string {  return "Hello World !"+" "+this.firstName + " "+ this.lastName;  }  }  var aPerson: Person = new Programmer("John","Doe");  console.log(aPerson.getFullName()); |

### TYPESCRIPT COMPILER ARGUMENTS

#### WATCH MODE

|  |  |
| --- | --- |
| **DESCRIPTION** | **COMMAND** |
| To create a compiled JS file with desired output file name | tsc .\typescript-compiler.ts --out compiledFile.js |
| To watch the changes in the typescript file   * It constantly monitor the changes of TS file * It picks the changes, compiles it and generate the output | tsc .\typescript-compiler.ts --out compiledFile.js –watch |
| The downside of this approach is that – the above command will only watch one single file for the changes.  The compiler arguments can be configured in a configuration file (tsconfig.json).It helps to ease the compilation process | |

#### CREATE TSCONFIG FILE

1. **The tsconfig.file is basically a configuration file to manage the TS projects**
2. Run the command : **tsc –init**

##### TS CONFIG FILE CONFIGRATIONS

###### WATCHING THE CHANGES IN MULTIPLE FILES

* Creating the tsconfig gives the flexibility to compile all the TS file in one go
* This also enable the watch mode for all files

|  |  |
| --- | --- |
| Step 1: Generate the tsconfig.json file. | **tsc --init** |
| Step 2: To compile all the files [tsc] | **tsc** |
| Step 3 : To watch all the files | **tsc --w** |

###### EXCLUDING AND INCLUDING FILES

|  |  |
| --- | --- |
| {  "compilerOptions": {  /\* Basic Options \*/  },  "exclude": [  /\*Path of file to be excluded from compilation \*/  "analytics.ts",  "\*.dev.ts",  "\*\*/\*.dev.ts",  "node\_modules"  ],  "include": [  /\*Include the file and file path to include  ]  } | * Tsconfig.json also helps in configuring the exclusion and inclusion of files and folders in the compilation * We can use folder path and regex to define the paths |

###### SETTING UP COMPILATION TARGET

|  |  |  |
| --- | --- | --- |
| {  "compilerOptions": {  "target": "es5"  /\* Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES2015', 'ES2016', 'ES2017', 'ES2018', 'ES2019' or 'ESNEXT'. \*/  }  } | | |
| const employee ={  name:'Dan',  age:30  }  let empCopy = employee;  console.log(empCopy); | When Target is es5. The ts file will be compiled to es5 javascript  "use strict";  var employee = {  name: 'Dan',  age: 30  };  var empCopy = employee;  console.log(empCopy); | When Target is es5. The ts file will be compiled to es6 javascript  "use strict";  const employee = {  name: 'Dan',  age: 30  };  let empCopy = employee;  console.log(empCopy); |

###### SETTING UP OUTPUT FOLDER

To set the location(output folder) of compiled script.



### CREATING A TYPESCRIPT PROJECT

To create a typescript project we need to follow below steps

1. **CREATE A TS.CONFIG FILE**
2. **CREATE A NODE PROJECT**

# CREATE A NODE PROJECT

|  |  |
| --- | --- |
| 1. Run a command “**npm -init**” to create a **package.json** file. 2. Press “Enter” to create npm project with default values.   **SAMPLE PACKAGE.JSON FILE**  **{**  **"name": "typescript-basics",**  **"version": "1.0.0",**  **"description": "",**  **"main": "index.js",**  **"scripts": {**  **"test": "echo \"Error: no test specified\" && exit 1"**  **},**  **"author": "",**  **"license": "ISC"**  **}**   1. **main** 🡪 is the starting point of the application. 2. So in TS we can create a index.ts file which when compiled creates a index.js file, which ultimately becomes the starting point of the application. |  |

#### COMPILING AND EXECUTING TS FILES

To understand the compilation and execution, lets create 2 files

|  |  |
| --- | --- |
| person.ts | Index.ts |
| **export** class Person {  firstName: string;  lastName: string;  constructor(firstName: string, lastName: string) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName(): string {  return this.firstName + " " + this.lastName;  }  } | i**mport { Person } from "./person";**  let per = new Person("Mike", "Clark");  console.log(per.getFullName()); |
| 1. Person class has been exported from person .ts which in turn imported in index.ts |
| **COMPILATION** :  Command For compilation : **tsc**  This will create the compiled JS file in output folder configured in tsconfig.json(outputFolder) in our example |
| 1. **EXECUTION**   **Command for execution : node ./outputFolder/index.js** |
| This seems to be a lengthy process we can simply the compilation and execution step using package.json configuration   1. Add “start“ property and assign the chained command , which will compile and execute the index.js file   TO COMPILE & RUN USE COMMAND : **npm start** | **{**  **"name": "typescript-basics",**  **"version": "1.0.0",**  **"description": "",**  **"main": "index.js",**  **"scripts": {**  **"start": "tsc && node outputFolder/index.js",**  **"test": "echo \"Error: no test specified\" && exit 1"**  **},**  **"author": "",**  **"license": "ISC"**  **}** |

#### MANAGING DEPENDENCIES

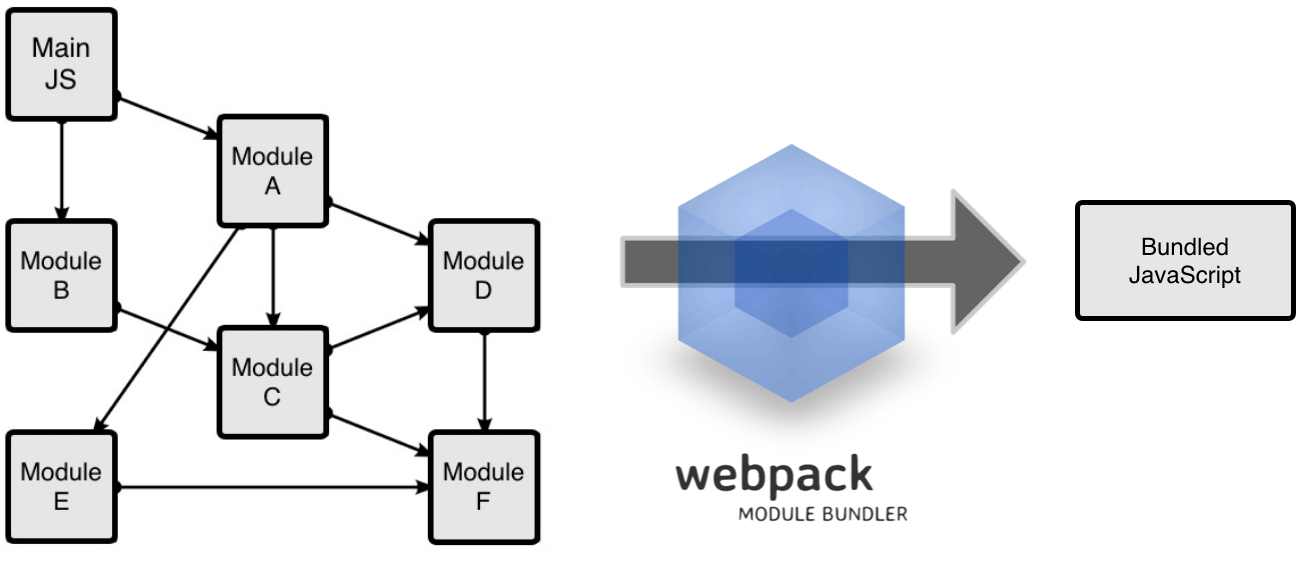
* One of the using a node project is that its ease the process of managing dependencies . Below are the steps how node help us in managing dependent libraries. Let’s say we want to use “lodash” library(this libabry has lot of utility functions)

|  |  |
| --- | --- |
| **INSTALLING LODASH LIBRARY**  **Generic Command to install a library**  **Npm install <Library\_name> --save** | **FOR LODASH**  **npm install lodash --save** |
| This command will do the following   1. Add the lodash library in the node\_module 2. Add a dependency in the package.json file. |  |
| **INSTALLING LODASH TYPE DEFINITION**  **Generic Command to install the type definition library**  **Npm install @type/<Library\_name> --save-dev** | **FOR LODASH**  **npm install @types/lodash --save-dev**  **Note ,**   * Installing type definition give suggestions in the IDE and type checking functionality for the depended libraries. * The type definitions are not the TS reimplementation of lodash JS library , it just contains the type definition. |
| **USING LIBRARY IN TS FILES** | import \* as **\_lodash** from "lodash";  let arr = [5, 4, 3];  console.log**(\_lodash.reverse(arr));** |

#### SAMPLE TYPESCRIPT PROJECT

# WEB PACK

* WEBPACK is a module web bundler.
* The webpack becomes more important for SPA’s because- This need a huge amount of Javascript code , which is responsible for dynamically showing up the HTML/Content on client browser unlike server side rendering(HTML is rendered by the server itself)



## MODULES IN JAVASCRIPT

* A module is just a file. One script can be one module.
* For large application – we logically divide the application based on functionality. Those logical divisions are called Modules.
* ***A module is a reusable piece of code that encapsulates implementation details and exposes a public API so it can be easily loaded and used by other code.***

**WHY MODULES?**

* **ABSTRACTION** : To delegate functionality to specialized libraries so that we don't have to understand the complexity of their actual implementation
* **ENCAPSULATION: T**o hide code inside the module if we don't want the code to be changed
* **REUSABILITY**: To avoid writing the same code over and over again
* **DEPENDENCY MANAGEMENT**: To easily change dependencies without rewriting our code

**PROBLEMS WITH MODULES**

|  |  |  |
| --- | --- | --- |
|  |  | * When we break the Javascript in modules we need to load the dependent modules should be load prior .So order of loading the modules on the page is important * **WEBPACK helps us such kind of dependency management among the modules** |
| ***Along with dependency management - Webpack also bundle all the modules together into a single file hence, minimize the number of network requests.*** |

**WEBPACK IN ACTION**

To understand webpack let build a small project to leverage webpack. We will follow below steps to do so

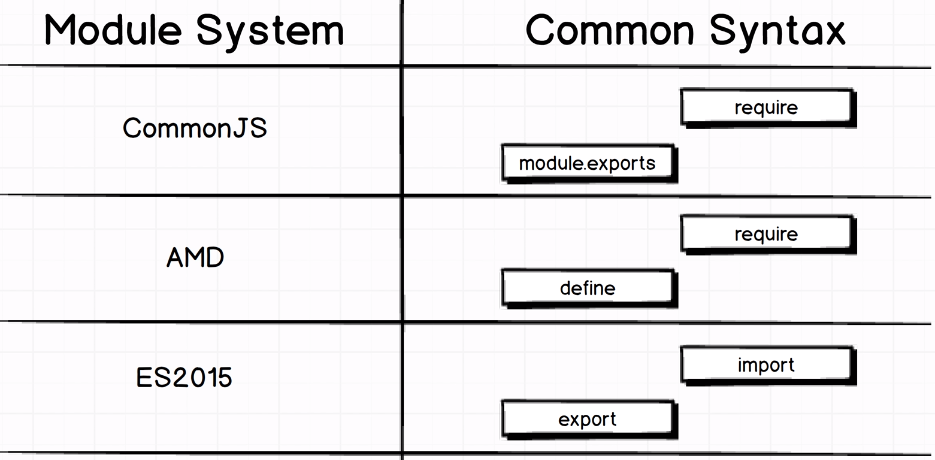
1. **Create new NPM project**
2. **Create 2 JS modules**
3. **Install and configure webpack**
4. **Run webpack**

|  |  |  |
| --- | --- | --- |
| **STEP 1: CREATING NPM PROJECT** | **npm init**  This will create a npm project with package.json file. This file is used for dependency management and configure sone build script | |
| **STEP 2: CREATING JS MODULES** |  | Here index.js is a dependent file which will call the function fron sum.js |
|  |

**After we create modules (as above) we need to link them up. So the way we can link all the modules we need to follow some pattern, which is know as MODULE FORMAT or MODULE SYSTEM**

### MODULE FORMATS/SYSTEM

* IN ES5 we didn’t have concept of modules, so developer came up with different approaches to modularize the code. The pattern/syntaxes to define a module are called Module Formats. The popular module formats used in ES5 are
  + **AMD**
    - Asynchronous Module Definition.
    - Loads the modules asynchronously
  + **COMMON JS**
    - This module system is used by Node JS
  + **ES 2015** – Universal Module definition – Used in Browsers and Node JS



### IMPLEMENTING COMMON JS MODULE FORMATS

|  |  |  |
| --- | --- | --- |
| sum.js (Module 1) | Index.js(Module 2) | |
| const sum = (a, b) => a + b;  **module.exports = sum;** | **const sum = require("./sum");**  const total = sum(10, 5);  console.log(total); | * All the export statement go at the bottom of the module * The “require” statement go at the top of the module * **The “require” function always takes the relative path of the module which it is importing** |

### INSTALLING AND CONFIGURING WEBPACK

|  |  |
| --- | --- |
| **INSTALLING WEBPACK (2.0) AS DEV DEPENDENCIES** | **npm install --save-dev webpack** |
| This will add web pack as project dependencies in package.json file | {  "name": "webpack-exercise",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "author": "",  "license": "ISC",  **"devDependencies": {**  **"webpack": "^2.7.0"**  **}**  } |
| **CREATING WEBPACK CONFIG FILE**   * To know the webpack know regarding the bundling, entry point etc..configuration – we need to create a file names ”**webpack.config.js**” * This is kind of instruction file for webpack. |  |
| **webpack.config.js**  const path = require("path");  const config = {  **entry: "./src/index.js",**  output: {  **path: path.resolve(\_\_dirname, "build"),**  **filename: "bundle.js"**  }  };  module.exports = config; | 1. The entry property give the entry point JS file. We need to provide the relative path (from project directory) of the script file 2. Output property – This property will help us to configure the output file 3. **Path property** – To configure path of the o/p file . 4. This is the absolute path .We are using node to get the absolute path of the project(path.resolve()) 5. “build” is the name of the parent folder where “**bundle.js**” will be created 6. **\_\_dirname is a node variable which is the path of current working directory.** |

### INSTALLING AND CONFIGURING WEBPACK

|  |  |  |
| --- | --- | --- |
| UPDATE PACKAGE.JSON | RUNNING WEBPACK | INCLUDING THE FILE IN HTML |
| {  "name": "webpack-exercise",  "version": "1.0.0",  "description": "",  "main": "index.js",  **"scripts": {**  **"build": "webpack"**  **},**  "author": "",  "license": "ISC",  "devDependencies": {  "webpack": "^2.2.0-rc.0"  }  } | **npm run build**  This will create a bundle.js file in a bundle folder | <html class="no-js">  <head></head>  <body>  **<script src="./build/bundle.js"></script>**  </body>  </html> |
|  |

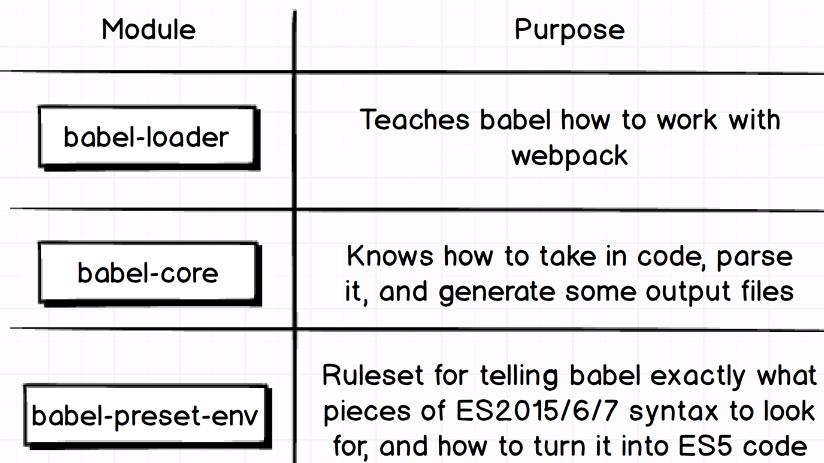
### MODULE LOADERS IN WEBPACK

* **Loader are used to do some pre-processing before bundling the code**
* Preprocessing like transpiling ES6 code(Babel Loader), preprocessing CSS and images

#### BABEL LOADER

**Babel loader transpiles the ES6 code to ES5 code**

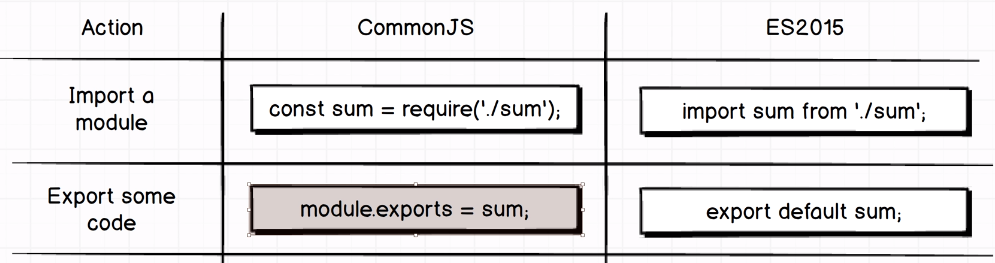
1. Babel Loader – Babel support multiple build sytem.This helps in compartibility with the build system as webpack
2. Babel Core –
3. Babel preset env – ES6 🡺 ES5



|  |  |  |  |
| --- | --- | --- | --- |
| **INSTALLING BABEL LOADER** | npm install --save-dev **babel-loader @babel/core @babel/preset-env** | | |
|  | | | |
| * To configure any loader we have to tell webpack on which type of files loader has to be applied. It can be applied to any type of files like JS, CSS, images etc. * We provide this configuration in webpack.config.js using some regex | | | |
| **CONFIGURATIONS** | | | |
| **STEP 1 :**  **CONFIGURATION IN WEBCONFIG JS**  const path = require("path");  const config = {  entry: "./src/index.js",  output: {  path: path.resolve(\_\_dirname, "build"),  filename: "bundle.js"  },  module: {  rules: [  {  **use: "babel-loader",**  test: **/\.js$/**  }  ]  }  };  module.exports = config; | | **STEP 2 :**  We need to create a file with name **.babelrc** in the root folder of the project  **PRESET CONFIGURATION IN .baberc**  {  "presets": ["@babel/preset-env"]  }   * **test :** This takes a regex value .This property tell webpack that babel loader will be applied to JS file only * **use:** This property makes babel compartible with webpack. * The preset-env module transpiles the ES6 code to ES5 code so that the transpiled code is compartible to lower version of browsers too. |  |

**CONVERTING TO ES6 MODULES SYSTEM**

Let’s convert the above commonJs module system into ES6 module system. Below are few highlight of the difference between them



|  |  |
| --- | --- |
| **Sum.js** | **Index.js** |
| const sum = (a, b) => a + b;  **export default sum;** | **import sum from "./sum";**  const total = sum(10, 10);  console.log(total); |

### ES6 MODULES SYSTEM

Modules can load each other and use special directives **export and import** to interchange functionality, call functions of one module from another one:

* export keyword labels variables and functions that should be accessible from outside the current module.
* import allows to import functionality from other modules.

#### EXPORT

The export statement is used when creating JavaScript modules to export functions, objects, or primitive values from the module so they can be used by other programs with the import statement. There are two types of exports:

1. Named Exports (Zero or more exports per module)
2. Default Exports (One per module)

We can have multiple **named** exports but one and one only **default** export per module.

#### DEFAULT EXPORT

|  |  |
| --- | --- |
| **DEFAULT** CLASS EXPORT (depend.js) | **DEFAULT** CLASS IMPORT (script.js) |
| **VARIATION 1**  class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  }  export default Person; | import **per** from "./depend";  class Programmer {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  }  let p1 = new Programmer("Mike", "Clark");  let person = new per();  console.log(person.getFullName.call(p1)); |
| **VARIATION 2**  export default Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  } | **ADDING SCRIPT TO HTML**  <script **type="module"** src="js/script.js"></script>   1. For export default we use the keyword “export default” 2. In the imported file default exported modules can be imported by any name (i.e “**per**”) |

|  |  |  |
| --- | --- | --- |
| **DEFAULT** FUNCTION EXPORT (depend.js) | | **DEFAULT** FUNCTION IMPORT (script.js) |
| **VARIATION 1**  function sayHello() {  console.log("Hello World!");  }  export default sayHello; | **VARIATION 2**  export default sayHello () {  console.log("Hello World!");  } | import hello from "./depend.js";  hello(); |

|  |  |  |
| --- | --- | --- |
| **DEFAULT** VARIABLE EXPORT (depend.js) | | **DEFAULT** VARIABLE IMPORT (script.js) |
| **VARIATION 1**  const name = ["Alex", "Maria"];  export default name; | **VARIATION 2 IS INVALID**  ~~export default~~ const name=["Alex", "Maria"]; | import n from "./depend.js";  console.log(n); |

#### NAMED EXPORT

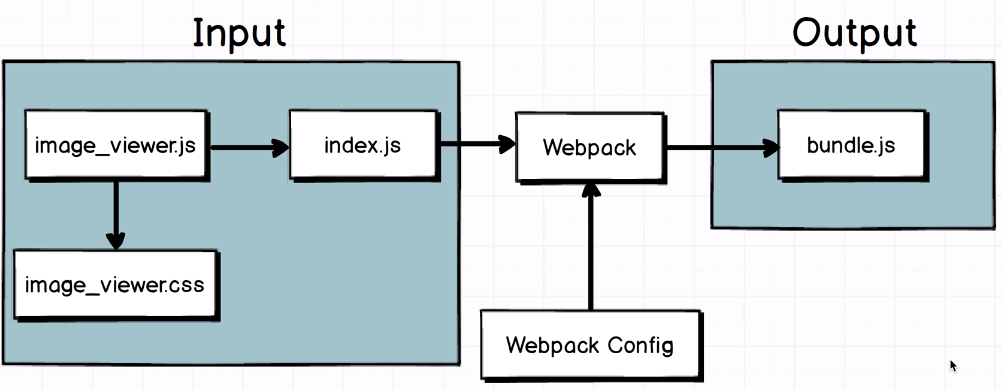
|  |  |
| --- | --- |
| **NAMED** CLASS EXPORT (depend.js) | **NAMED** CLASS IMPORT (script.js) |
| **VARIATION 1**  class Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  }  export {Person}; | import { Person } from "./depend.js";  class Programmer {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  }  let p1 = new Programmer("Mike", "Clark");  let person = new Person();  console.log(person.getFullName.call(p1)); |
| **VARIATION 2**  export Person {  constructor(firstName, lastName) {  this.firstName = firstName;  this.lastName = lastName;  }  getFullName() {  return this.firstName + " " + this.lastName;  }  } | **ADDING SCRIPT TO HTML**  <script **type="module"** src="js/script.js"></script>   1. For export default we use the keyword “export default” 2. In the imported file default exported modules can be imported by any name (i.e “**per**”) |

|  |  |  |
| --- | --- | --- |
| **NAMED** FUNCTION EXPORT (depend.js) | | **NAMED** FUNCTION IMPORT (script.js) |
| **VARIATION 1**  function sayHello() {  console.log("Hello World!");  }  export {sayHello}; | **VARIATION 2**  export sayHello () {  console.log("Hello World!");  } | import { sayHello } from "./depend.js";  sayHello(); |

|  |  |  |
| --- | --- | --- |
| **NAMED** VARIABLE EXPORT (depend.js) | | **NAMED** VARIABLE IMPORT (script.js) |
| **VARIATION 1**  const name = ["Alex", "Maria"];  export {name}; | **VARIATION 2**  export const name = ["Alex", "Maria"]; | import {name} from "./depend.js";  console.log(name); |

|  |  |  |
| --- | --- | --- |
|  | **EXPLICIT IMPORTING** | **IMPORTING USING ALIAS NAME** |
| class Person {  sayName() {  console.log("Name is Alex");  }  }  function sayHello() {  console.log("Hello World");  }  const mobile = 123;  export { Person, sayHello, mobile }; | import { Person, sayHello, mobile } from "./depend.js";  let person = new Person();  console.log(person);  sayHello();  console.log(mobile); | import \* as details from "./depend.js";  let person = new details.Person();  console.log(person);  console.log();  details.sayHello();  console.log(details.mobile); |

### CSS FILES IN WEBPACK



* To bundle the CSS file using webpack we need to couple of css-loader
* To make the CSS file part of the bundled file we need below two css loader

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **INSTALLING CSS LOADER**  **npm install --save-dev css-loader style-loader** | | **PACKAGE.JSON UPDATED**  {  ….  },  …  "devDependencies": {  "@babel/core": "^7.9.6",  "@babel/preset-env": "^7.9.6",  "babel-loader": "^8.1.0",  **"css-loader": "^3.5.3",**  **"style-loader": "^1.2.1",**  "webpack": "^2.7.0"  }  } | | |
| **WEB CONFIG**   1. To plug in the css loaders we need to add a rule. Note that the order of loading of the loader is from right to left 2. The css loader is only applied to css files only (test property) | | const path = require("path");  const config = {  entry: "./src/index.js",  output: {  path: path.resolve(\_\_dirname, "build"),  filename: "bundle.js"  },  module: {  rules: [  {  use: "babel-loader",  test: /\.js$/  },  {  **use: ["style-loader", "css-loader"],**  **test: /\.css$/**  }  ]  }  };  module.exports = config; | | |
|  | **Image\_viewer.css** | **image\_viewer.js** | | **Index.js** |
| img {  border: 10px solid red;  } | import "./image\_viewer.css";  const image = document.createElement("img");  image.src = "https://i.picsum.photos/id/1/400/400.jpg";  document.body.appendChild(image); | | **import sum from "./sum";**  **import "./image\_viewer";**  const total = sum(10, 10);  console.log(total); |
| 1. Lets create two files “image\_view.js” and “image\_viewer.css” 2. Image\_viewer.js file creates an image and append it to DOM. It also imports the css file into it. 3. Finally this file is imported to the index.js(entry file). 4. Note – The way of importing the image\_viewer.js file is different from “sum” module import, because- we don’t need anything from image\_viewer.js- We just want it to exexute when index.js is executed.when we import any file, it actually gets executed when index.js file executes. | | | |
|  | | * The style dynamically added to head section of the HTML document. * This is the JS file which actually dynamically add it in the DOM * ***Note here styles are not loads as a separate entity (we cannot see any network call for CSS)*** | |
| **FLOW DIAGRAM** | | | | |

#### GENERATING THE CSS AS SEPARATE ENTITY

1. To load the css file separately we need to use a plugin “extract-text-webpack-plugin”
2. **INSTALLING THE PLUG-IN** : **npm install --save-dev extract-text-webpack-plugin**

|  |  |  |
| --- | --- | --- |
| **WEB CONFIG JS**  const path = require("path");  **const ExtractTextPlugin = require("extract-text-webpack-plugin");**  const config = {  entry: "./src/index.js",  output: {  path: path.resolve(\_\_dirname, "build"),  filename: "bundle.js"  },  module: {  rules: [  {  use: "babel-loader",  test: /\.js$/  },  **{**  **use: ExtractTextPlugin.extract({**  **loader: "css-loader"**  **}),**  **test: /\.css$/**  **}**  ]  },  **plugin: [new ExtractTextPlugin("style.css")]**  };  module.exports = config; | 1. We use a plug-in here not a loader 2. The bundled css will be created in the **bundle** folder as “**style.css**”   **HTML FILE**  <head>  **<link rel="stylesheet" href="./build/style.css" />**  </head>  <body>  <script src="./build/bundle.js" async defer></script>  </body>  </html> | |
|  | The webpack build process will create a separate css file which can then can be included in HTML |

### CODE SPLITING USING WEBPACK

Webpack usually bundles all the code as on single file . The complete file gets loaded even it not needed on a particult page

1. Code splitting means loading code when it is needed
2. Code splitting can be done using webpack using ”System.import()”

|  |  |
| --- | --- |
|  | Lets consider an example   1. Page will have a “click” button 2. On clicking the button 🡪 it will load the JS file 🡪 which in turn loads the image 3. This whole process is asynchronous |
|  |