**Tutorial Series 1: Introduction to JavaScript for Developers by Koushik**

**Tutorial 1: Introduction**

**Tutorial 2: Unit Introduction/Tutorial 3: What is JavaScript/Tutorial 4: JavaScript as a scripting Language?**

1. JavaScript Definition: JavaScript is a light weight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it’s used in many non-browser environments as well.
2. Buzzwords for JavaScript:
3. Lightweight - Small memory footprint, easy to implement
4. Interpreted – No Compilation. Instructions executed directly (In true sense there is a compilation step, but it is different from the languages like C++ and Java)
5. Object-oriented – Modelled around objects
6. First-class functions – Functions as values
7. Scripting Language – Instructions are written for a runtime environment
8. HTML has different tags forming a tree of objects which is called as Document Object Model
9. Runtime environment interprets the HTML and builds a tree.
10. HTML is essentially a static language; view of the browser is rendered using HTML DOM Tree.
11. By using JavaScript you can use the DOM Tree nodes and then you have opportunity to modify the DOM Tree nodes, you can edit existing nodes or even add and remove the DOM tree nodes dynamically.
12. JavaScript is a way to execute dynamic functionality; this is how JavaScript is typically used.

**Tutorial 5: Why Learn JavaScript**

1. Why learn JavaScript?
2. Client side web development. This can be done in below given ways:
3. Native JavaScript – Pure JavaScript no frameworks to access the DOM
4. jQuery
5. AngularJS, React, Ember
6. Server side Development
7. Node JS
8. Express (Another Framework which works with Node JS)
9. Browser Extensions
10. Desktop Applications
11. Mobile Applications
12. IOT Applications

**Tutorial 6: Some thoughts on learning JavaScript**

1. JavaScript is most loved and most hated language

**Tutorial 7: A brief history of JavaScript**

1. Created by Brendan Eich at Netscape
2. In 90s two popular browser vendors were NETSCAPE with their Netscape Navigator and Microsoft with Internet explorer.
3. JavaScript was created to complement Java
4. JavaScript was created roughly in about 10 days – it was rushed to production filled with bugs.
5. JavaScript OOB is very forgiving language.
6. JavaScript is forgiving language
7. Standardized as ECMAScript
8. Latest version of JavaScript is ECMAScript 6. As of now ECMAScript 5 is fully supported by all the browsers.

**Tutorial 8: Setting up our development environment**

1. For JavaScript the runtime environment is Web Browser.
2. Use Firefox, because Firefox has some tools which come as very handy while writing JavaScript code
3. With Firefox tools we can write JavaScript code and debug it.
4. JavaScript can be directly run from console.
5. To print anything in the console use console.log(); //Example: console.log(“Hello World”);
6. Open Menu > Developer > Scratchpad; then write code
7. Right click > reload and run

**Tutorial 9: Variable Declaration**

1. var value = 42; //method 1
2. var value; value = 42; //method 2

**Tutorial 10: Number primitive type**

1. Primitive types – Number, String, Boolean, undefined, null, symbol (Symbol primitive type is defined in the ECMAScript 6)
2. Number – Numbers in JavaScript are “double-precision 64-bit format IEEE 754 values”; which means we don’t have integers as such, all are floating point double precision 64 bit numbers.

**Tutorial 11: String and Boolean**

1. String – Sequences of Unicode (16-bit); there is no character data type in java script i.e. (No Character type! A character is just a String of length 1).
2. **Loose typing or week typing** means that we can assign string or Boolean value to a variable holding a number value.

**Tutorial 12: Understanding undefined**

1. Declaration and definition: Declaration -> var value; Definition -> value =42;
2. Undefined -> you have declared it but have not assigned any value to it.
3. JavaScript has created a separate type called “Undefined”, it has one value “undefined”

**Tutorial 13: Understanding null**

1. Now the primitive types become -> Number, String, Boolean, undefined, null, symbol (Symbol primitive type is defined in the ECMAScript 6)

**Tutorial 14: Difference between undefined and null**

1. Null is a not applicable value i.e. if user wants to say that this field is not applicable to him; he will put its value as null or not applicable.
2. So, null is non value.
3. Symbol is kind a like Enumeration in C++ and Java

**Tutorial 15: Types Summery**

1. No need to declare variable type in JavaScript
2. Same variable can be assigned the values of different types
3. There is no scoping information in the variable declarations.
4. Variables and values can be “interrogated”
5. “interrogated ” will be explained in detail in next Tutorial

**Tutorial 16: The typeofoperator**

1. In JavaScript we have typeof operator to know that the variable is of what type.
2. var a = 10; console.log(typeof a); 🡪 number is the output
3. a = “Hello”; console.log(typeof a); 🡪string is the output
4. a = null; console.log(typeof a); 🡪object
5. Yes, type of null is 🡪 Object

**Tutorial 17: Type Coersion and the === operator**

1. There are sometimes when compiler or interpreter has to make some automatic type conversions for you to make that operation work generally when we use “==” operator to compare.
2. Type coercion example is 123+”4” 🡪 “1234”// Here the number is first converted to string then string concatenation happens.
3. “====” is the strict comparison; which means that while comparison, type conversion, cannot happen or take place.

**Tutorial 18: Type Coersion summary**

1. var a = 10; if(a){console.log(“a is true”);} 🡪 a is true
2. 0 is a false value. If a is any other number than 0; the value of the if condition is a truthie.
3. If I pass any string in the if condition it will be true i.e. “sukhmeet”; if I pass empty string it will be false i.e. “”.
4. Null and undefined are also falsie.
5. Summarising type coercion:
6. JavaScript is “flexible” with typing.
7. Values of all types have an associated Boolean value
8. Always use === for precise checks i.e. both value and type at the same time

**Tutorial 19: Objects**

1. JavaScript is an object oriented programming language
2. However, it’s not a class based programming language
3. Simplest object value that I can give is the empty object.
4. var myString = “Hello”; var myObject = {}; //Empty object
5. console.log(myObject); 🡪 Object { } //output is an object which is an empty object
6. Data in the object consists of the property, adding property in the object
7. myObject.prop = “Hello”;
8. Now, if I do console.log(myObject); 🡪 Object {prop: “Hello”}
9. myObject.prop2 = 123;
10. Now, if I do console.log(myObject); 🡪 Object {prop: “Hello”, prop2: 123}
11. Now, I can access the properties as console.log(“The number property is ”+ myObject.prop2); 🡪 The number property is 123

**Tutorial 20: The Object Literal**

1. var myObj = {“prop”: “Hello”, “prop1”: 123, “prop3”: true};
2. There is no accessor in JavaScript i.e. public, default etc.
3. Here I can always access object.property and access any property of the object as such everything is public
4. All properties are accessible to anybody
5. If we try to access a property form an object in JavaScript which does not exist in the object we will get “undefined” as the output.

**Tutorial 21: Object Characteristics Summary**

1. Free-form – not bound to a class i.e. Object is Free-form
2. Object literal notation to create objects
3. Object properties can be accessed directly i.e. there is no private and public here; everything is public
4. New properties can be added on objects directly
5. Objects can have methods.

**Tutorial 22: The dot and bracket notations**

1. Now, there are 2 types of notations to access the properties of JavaScript Object
2. The Dot notation
3. The [] i.e. square bracket notation.
4. Examples: console.log(“The number property is ”+ myObj.prop1); //using Dot notation

console.log(“Accessing using square bracket notation ”+ myObj[“prop1”]); //Square bracket notation; note here that the property name in the square bracket notation is given as string inside the square brackets.

**Tutorial 23: Difference between Dot and Bracket Notations**

1. The [] notation:
2. Use [] notation when:
3. Property name is a reserved word or invalid identifier
4. Property name starts with a number
5. Property name is dynamic
6. Property name in the object should start with a letter and not with a number, however, if there is a scenario in which the property name is number then we need to use square bracket notation.
7. Ex. var myObj = {“1” : “one”};
8. Console.log(myObj[“1”]); // one gets printed
9. Now, we will show an example when the property name is dynamic with an example the use of the square bracket notation.
10. Ex. var myObj = {“prop” : “Hello”, “prop1”: 123, “prop3”: true};
11. Now, suppose I need to I need to pull up a property based on some external factor and we don’t know what the property is gonna be.
12. Now, suppose we get the property name in the string ex. var propertyName = “prop1”;
13. Now, somebody gives me this variable propertyName and I don’t know what its value is gonna be; I will use square bracket notation to access it
14. console.log(myObj[propertyName]); //here I don’t have to know what is the value in this line i.e. what will be the value of propertyName variable but still we would be able to access it and print the value.
15. Dot notation is faster to access than the square bracket notation.
16. There are some cases where we have to use the square bracket notation, however, the cases where we have can use dot notation we should prefer dot notation over square bracket notation.
17. Runtime engine may not be able to optimize [] i.e. square bracket notation, that’s why it may take more time to execute the square bracket notation.
18. In the rest of the cases Dot and [] notations can be interchanged.

**Tutorial 24: Nested Objects**

1. var myObj = {};
2. Here myObj variable is pointing to the location where object is located instead of holding its value.
3. If I have another variable say myObj2 and we assign myObje1 to it; then both the objects would be pointing to same location.
4. In true sense variables are pointing to the memory location of the object.
5. When one object has a property which is an object, that becomes nested object.
6. Dot operator can be used to fetch the property of inner objects i.e. if there are nested objects then we can get the property as outerObject.innerObject.Property;
7. Similarly, we can also create new properties in the inner object.
8. In the similar way we can also fetch the inner property by using square brackets i.e. []

**Tutorial 25: Revisiting === operator for Objects**

1. If I assign one object to a variable i.e. var myObj = {“myProp”:”Hello”}; var myObj2 = myObj1;
2. Now, both of the above objects will have same property.
3. Now, if we compare two objects, the while making comparison it will be checked if both objects point to the same memory location.
4. myObj1 === myObj2 🡪 true

**Tutorial 26: Revisiting undefined vs null**

1. Null is an empty value or a non-value that is set to a variable.
2. If we try to access a property from an object which does not exist in the object, then we will get undefined as the output.
3. Suppose we define a property to which we don’t want to give any value or a non-value then we can assign it null as the value which is non value.

**Tutorial 27: Deleting Properties with the delete operator**

1. Suppose we assign a property of an object as undefined i.e. person.age = undefined;
2. Now the new value of the property of the object will be undefined, however, we want to remove the property from the object
3. So, we will use 🡪 delete person.age;

**Tutorial 28: Introducing Arrays**

1. An array can be defined in JavaScript using the syntax – var myArray = [100,200, 300];
2. The way to access these values is by using the square bracket i.e. myArray[0]; myArray[1];
3. If we try to access a value from an array and we put the index of the array which is out of bound i.e. beyond the count of the number of values, we will get the output as undefined.
4. To add another value to the array simply take the next index to the index of last element and add the value i.e. my Array[3]=30;//suppose the array was having 3 elements before now it will become values.

**Tutorial 29: The secret behind JavaScript Arrays**

1. To get the length of the array, we can use the length property of the array i.e. myArray.length
2. In true sense an Array is an object in JavaScript.
3. Every JavaScript array is secretly a JavaScript Object.
4. Array just has some special properties to it., however, underneath it’s just an object.
5. Now, even length is a property of the object Array.
6. Array has the properties like 1, 2 …. Etc. So in true sense array indexes are properties.
7. But here we cannot use Dot notation to access the properties of array which are numbers. We can only access by the [] brackets notation the properties which are numbers.
8. Here when we pass a number property to an object, JavaScript internally converts it into a String.
9. Suppose I give a large index in the array and add a value there i.e. without adding a value at the index locations which are in between i.e. myArray[100]=”Sukhmeet”;
10. If we check the length property of this array we will get 101 bit the array has 4 elements.
11. So, the length doesn’t count how many properties you have in the array, it just counts the index of the last element of the array i.e. the greatest element as a number
12. We can also add other properties i.e. string properties in an array; but if you want to treat array as an array always use number properties.

**Tutorial 30: Wrapper Objects**

1. Is string a primitive?
   1. String has the length property. Ex. var greeting = “Hello World”; 🡪greeting.length = 11
   2. String has a length property so, is it an object or is it a primitive?
   3. JavaScript has equivalent objects for the primitive datatypes. Ex. String primitive has an equivalent string object.
   4. When we call the object property of string, then JavaScript converts it into String object; when it converts it to object the length property becomes available. Then it calls the length property of the object String.
   5. The new string object that gets created for primitive object is a temporary object, but it is not assigned to the primitive string, if you later check this primitive type, it would still be the primitive type.
   6. Below are the primitive types which have equivalent objects:
      * 1. String
        2. Boolean
        3. Number
        4. Symbol

**Tutorial 31: Introduction to Functions**

1. function sayHello(){console.log(“Hello”);}
2. Execute above function by calling it 🡪 sayHello();
3. While passing the parameters in the function you don’t need to define that variable or its type i.e. function sayHello(a,b){console.log(“Hello”);}
4. In the above function look how the parameters a and b are passed.
5. Another example 🡪 function sayHello(name){console.log(“Hello ”+name);} //Parameter name received
6. sayHello(“Sukhmeet”); //Argument Sukhmeet Passed

**Tutorial 32: Flexible arguments counts**

1. If a function accepts 2 parameters and you pass 1 argument while calling the function, in JavaScript it will then also work. In other languages like C++ and Java compilation error would happen.
2. Suppose a function accepts 2 parameters and we pass 1 argument instead of 2 then the second parameter in the function would be undefined.
3. Even if I pass one extra argument while calling the function that would be ignored.
4. Functions in JavaScript are very flexible.
5. Overloaded functions are not possible in JavaScript

**Tutorial 33: Return Values**

1. By using return key word, we can return value from a function which can be later stored in a variable.
2. function sayHello (name, timeOfDay){

return “Hello “+name + “ Time of Day is ”+ timeOfDay;

}

1. var returnValue = sayHello(“Sukhmeet”,”evening”, 42); //the return value is assigned //to the returnValue variable.
2. If we don’t send any value in the return but only put a return keyword in the end of the function, then the value that the variable returnValue will have will be undefined.

**Tutorial 34: Function Expressions**

1. The way in which we wrote function in previous tutorial was using **function declaration** i.e. we wrote the function keyword.
2. Function declaration is one of two ways in which you can write a function; these two ways are called the primary ways of creating function.
3. The second method to create a function is by using **function expression**.
4. Functions in JavaScript are called the first class values, which mean the functions are actually values in JavaScript; just as much as a string is a value and the number 100 is a value. Similarly, functions are also
5. var f = function foo(){

Console.log(“Hello”);

};

1. Now, in the previous line we assigned the function to a variable. Now, we can use that variable to execute a function.
2. f(); //executed the function using a variable.
3. For function declaration we don’t need semi colon but for function expression we must put semi-colon at the end i.e. after the curly brackets.

**Tutorial 35: Anonymous Function Expressions**

1. We don’t even need to give the name of the function in the function expression, we can write the function expression as given below:
2. var f = function(name){

console.log(“Hello ”+ name);

};

1. f(); //executing the variable to which anonymous function expression is assigned
2. This way of executing functions is called the anonymous function expression.

**Tutorial 36: Functions as Arguments**

1. Now, if a variable is a function, you can actually pass that variable to another function as an argument.
2. Then we will have another function which will execute the function which we passed as a value.
3. Now, we will pass the function variable created in the previous tutorial in the new function called executer
4. var executer = function(fn, name){

fn(name);

};

1. executer(f, “Sukhmeet”);

**Tutorial 37: Functions on Objects**

1. The functions can also be added as an object property.
2. var myObj = {

“testProp”: true

};

1. myObj.myMethod = function(){

console.log(“Function in object”);

};

**Tutorial 38: Understanding the this keyword**

1. var person = {

“firstName”:”Sukhmeet”

“lastName”:”Singh”

“getFullName”: function(){ //here the function only gets initialized

return person.fullname + ” ” + person.lastName;

}

};

1. var fullName = person.getFullName(); //here the function gets executed
2. console.log (fullName);
3. Variables are only pointing to the object; we can also call them the pointers.
4. var person2 = person; //now, the person2 also points to the same i.e. person object
5. person = {}; //now only person2 will point to that object.
6. Now, if we execute console.log(person2.getFullName()); //it will print undefined undefined; because we have hard coded the person object to get the firstName and lastName.
7. Now, we need to define that, we need to get the details from the current object for that instead of person we will define this key word.
8. Now, we will define above object again using this key word.
9. var person = {

“firstName”:”Sukhmeet”

“lastName”:”Singh”

“getFullName”: function(){ //here the function only gets initialized

return this.fullname + ” ” + this.lastName;

}

};

**Tutorial 39: Code Exercise and 40 Exercise solution**

1. var person = {

“firstName”:”Sukhmeet”

“lastName”:”Singh”

“getFullName”: function(){ //here the function only gets initialized

return this.fullname + ” ” + this.lastName;

},

“address”:{ //nested object

“street”: “123 JS street”

“city”: “JS”

“state”: “CA”

},

“isFromState”: function(state){

return (this.address.state===state);

}

};

1. console.log(person.isFromState(“ABC”));

**Tutorial 41: Default function arguments**

1. There are 2 arguments for any function that we get for free
2. var add = function (a, b){

return a+b;

};

1. console.log(add(10, 30));
2. You can pass as many arguments you want in a function; those all arguments will always be available to you from the **arguments** i.e. **arguments** is a hidden argument.

It comes OOB free.

1. var add = function (a, b){

console.log(arguments); //**arguments** are an implicit argument; it comes OOB for //every function

return a+b;

};

1. console.log(add(10, 30, 3, 4, 80)); //all the other arguments which the function does //not accept are stored in the implicit argument **arguments**.
2. There is another or the second argument that is available to all functions i.e. **this** argument
3. **arguments** is not an array, it looks like an array, it is actually an object.

**Tutorial 42: Unit Summary**

1. JavaScript functions:
2. Functions can be written in literal form.
3. A function is a value that can be assigned to a variable.
4. Can be called by passing in arguments
5. Functions are objects!!!
6. Flexible argument count
7. No function overloading in JavaScript
8. Default arguments
9. The **arguments** argument
10. Function Declaration:
11. function addNumbers(a, b){

return a+b;

}

var number= 1;

var result = addNumbers(number, number);

1. JavaScript function expression:
2. var additionFn = function addNumbers(a, b){

return a+b;

};

1. Anonymous function expressions:
2. var additionFn = function (a, b){

return a+b;

};

result = additionFn(number, number);

1. Functions as object property:

var mathObj = {};

mathObj.add = function(a, b){

return a+b;

};

result = mathObj.add(number, number);

**Tutorial 43: Array Methods**

1. Arrays are objects under the hood. Every array is actually a JavaScript Object.
2. JavaScript also has some methods for the array which come OOB; some of the common functions for array are:
3. myArray.push(500);
4. myArray.pop(); //pops the last element
5. myArray.shift();
6. myArray.unshift(42);

**Tutorial 44: Array for each Method**

1. var myArray = [10, 20, “Hello”,{}];
2. myArray.forEach(function(item, index, array){ //forEach accepts the function as an //argument

console.log(“For an element ”+ item);

});

1. forEach executes the function for each element of the array.

**Tutorial 45: Reading Exercise**

1. <https://developer.mozila.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math>
2. These are couple of useful objects; the first object that we are talking about is the math object. This object is useful for doing mathematical calculations using JavaScript.
3. Other object which is essential is the date object when you are dealing with current date and time.
4. You can use a new date operator to create a new date.
5. var today = new Date();

**Tutorial 46: Next Steps**

1. Next Concepts:
2. Scopes and Closures
3. Objects and Prototypes
4. Asynchronous JavaScript – Call-backs and Promises
5. Client side frameworks
6. Server side frameworks
7. JavaScript is a single threaded language. You cannot do things in parallel. Because of this you need to write Asynchronous JavaScript if you wanna have responsive

Programs.

**JavaScript Tutorial 2 series: JavaScript Scopes and Closures In-depth**

**Tutorial 1: Introduction**

1. Unit 1: Understanding Scopes
2. Unit 2: Compilation and Interpretation
3. Unit 3: Closures
4. For reading purpose use – You Don’t Know JS by Kyle Simpson (youdontknowjs.com)

**Tutorial 2: JavaScript Functions Primer**

1. Function foo(){ //function declaration

Console.log(“Function foo called”);

}

1. Functions are also objects.
2. Now there is another way in which you can create functions. JavaScript has firstclass functions; you can assign these functions to a variable like string, integer etc.
3. var bar = function(){

console.log(“Function bar called”);

};

1. Calling above functions:
2. foo();
3. bar();

**Tutorial 3: Understanding Scopes and Block Scoping**

1. The concept of scope is not new in JavaScript; it is present in all the programming languages.
2. A scope is place in program where a particular variable is accessible and is not accessible at other places i.e. if the variable is accessed outside the scope the variable is not available.
3. var **a** = 10; // variable a is accessible at every line of the program.
4. Here variable **a** is present at the global scope.
5. No, not all the variables are accessible at every place in every program; they have certain scope.
6. If every variable is accessible at everywhere in the program, then there would be chaos.
7. There is 1 global scope which is accessible across the board for that program.
8. Then, there are these small pockets of scopes where you access certain variables.
9. Now, we will isolate **a** to a particular place or say box so that it is not accessible everywhere else.
10. Inside the scope all the variables of that scope achieve some result, now, that result can be stored somewhere and returned.
11. Scopes typically have a hierarchy means that you can have 1 scope inside other scope.
12. The inside scope can access the variable that is present outside that scope, however, the scope which is outside cannot access the scope that is inside that scope which is outside.
13. Now, we will create a scope. Scopes can be created by a **function**.

**Tutorial 4: Function Scoping in JavaScript**

1. **In JavaScript scoping based on Functions.** JavaScript is supposed to have function scoping not block scoping.
2. var name = “Sukhmeet”;

if(name===”Sukhmeet”){

var department = “Engineering”;

}

console.log(name);

console.log(department); //this will get printed because **if, for doesn’t create a new scope in JavaScript.**

1. JavaScript is not a block scoped; the only way to create scope in JavaScript is by using function. So, in JavaScript there is only one way to create scopes that is by functions.
2. var name = “Sukhmeet”;

function allocateDepartment(){

if(name===”Sukhmeet”){ // here name is declared outside the function, so is //available inside also.

var department = “Engineering”;

}

}

1. allocateDepartment();
2. console.log(department); //here we will get runtime error, because JavaScript has //function scope; the variable is not available outside of the function.
3. Function (name====”Sukhmeet”){}; //first the function will look inside the function, //if it doesn’t find it inside then it will look at the outer scope of that function
4. A variable defined in top scope is available for the child scope also, however, one defined in child scope is not available in outer scope.

**Tutorial 5: Scope Exercises**

1. var a = 10; var b = 10;
2. console.log(a + b);
3. Now, we need to restrict the scope of a and b. So, all I have to do is wrap this in a function.
4. function myFn(){

var a = 10;

var b = 10;

console.log(a+b);

}

1. In order to run the function, we need to call it:

myFn();

1. var name = “Sukhmeet”;
2. function printGreeting (name) {

console.log(“Hello ”+ name);

}

1. printGreeting(“Arthur Dent”); //this will be printed instead of Sukhmeet because local scope is sorted first.

**Tutorial 6: IIFE (Immediately invoked function expression)**

1. We talk to so many developers; we often hear that global variables are bad.
2. An application may have many JavaScript files, each of this file may be written be a different JavaScript programmer.
3. Now, the global variables are accessible by all scripts; so, if one of the scripts accesses it and changes it, it may corrupt the functionality of the application.
4. The name of the global variable may clash the name of some other global variable, which may lead to corrupted application.
5. function myFn(){

var a = 10;

var b = 10;

console.log(a+b);

}

1. myFn();
2. Now, I have created a new problem i.e. I have removed 2 globals i.e. a and b; and have added one new global i.e. myFn
3. Now, we will create an anonymous function
4. (function (){

var a = 10;

var b = 10;

console.log(a+b);

})();

1. Now, the anonymous function will be executed right away.

**Tutorial 7: Read and Write operations**

1. var = 10;
2. console.log(a); // read operation on a
3. var a = 10; var b ;
4. b = a; //Here, I am reading from a and writing to b
5. function greet(name){ // at this line there is a write operation on the variable name

console.log(name); //at this line there is a read operation on the variable //name

}

1. greet(“Sukhmeet”);

**Tutorial 8: Implications of Read and Write**

1. var foo;
2. console.log(foo); // the value of the variable will be printed as undefined.
3. foo=10; //here we have not declared foo and directly assigned value to it
4. console.log(foo); //value 10 will be printed
5. **If we assign a value to a variable without declaring it, then a global variable will be created; even if you do it inside a child scope.** Because it will try find that variable in the current scope, if it doesn’t find; then it will find in scope one level above the current scope. If it doesn’t find at the last level i.e. the global scope it will create one.
6. And we know that variables at global scope are very bad.

**Tutorials 9: The window object**

1. Let’s understand the concept of global object
2. Now, there is a global object also which holds all the global variables. Global object or the root object depends on which runtime you are using.
3. **When you run JavaScript in the browser, the global object is the window object.**
4. There are the bunch of properties that window object holds.
5. Whenever you create a global variable, it gets added as a property in the global object.
6. **Every global variable is a property on the global object.**
7. **It is not only for variables but it is same for functions also.**
8. Window is the starting point for the page to render and to execute the scripts.
9. In node the global object is global; so it totally depends on the runtime in which you are executing the JavaScript.

**Tutorial 10: Compilation and Interpretation**

1. In JavaScript what you execute is your source code; now the browser executes the code without this intermediatory file i.e. file created after compilation is not created in JavaScript.
2. Does that mean that JavaScript code is not compiled?
3. When a browser executes the JavaScript code, it actually does the 2 things:
   1. **Compilation:** Browser looks at your code and looks at particular things in order to execute it.
   2. **Interpretation:** Once the compilation step is completed, then the interpretation step is executed where the actual code is executed.
   3. These 2 steps happen very quickly and together.
   4. When JavaScript gets a file it runs the compilation step a fraction of second before the interpretation step. So, these two steps are very close to each other in terms of timeframe. But still you cannot deny the fact that there is compilation step.
4. Now, we will understand what the browser does in these 2 steps.

**Tutorial 11: Understanding the compilation phase**

1. var a = 10; var b = 20;
2. console.log(a + b);
3. Now, on every variable declaration we one global variable gets registered at the global scope; on the global scope here are 2 variables i.e. a and b.
4. var a = 10;
5. function myFn(){

var b = 20;

var c = b;

console.log(a+b);

}

1. myFn();
2. Now the global scope have below variables registered:
   1. a
   2. myFn
3. myFn scope:
   1. b
   2. c
4. So, the compiler while compiling the code will create 4 variables i.e. a, myFn, b,c
5. var myName = “Sukhmeet Singh”;
6. function greet(name){

console.log(“Hello ”+ name);

}

1. greet(myName);
2. Here 3 variables are created i.e. myName, greet, name
3. Variables at global scope:
4. myName
5. greet
6. at greet/function scope are:
7. name

**Tutorial 12: Understanding the interpretation step**

1. Execution step doesn’t look at var i.e. variables. It’s not gonna worry about declarations. It will only worry about assigning the values.
2. Ex. myName = “Sukhmeet”;
3. So at this step all the values will be assigned.

**Tutorial 13: The Global Scope Problem**

1. var a = 10;
2. function myFn(){

var b = a;

console.log(b);

console.log(c); //printing the variable which is not declared – compiler throws //exception.

d = 100; // **a new variable named d will be created at global scope**

}

1. myFn();
2. Variables created at global scope are:
   1. a
   2. myFn
   3. d
3. myFn() scope
   1. b
4. Above all the creation steps happen at compilation step; and then the values are assigned at interpretation step

**Tutorial 14: Some Exercises and a surprising result**

1. var a = 10;
2. function outer(){

var b = a;

console.log(b);

function inner(){

var b = 20;

var c = b;

console.log(c);

}

Inner();

}

1. outer();
2. function outer(){

var b = a;

console.log(b);

function inner(){

var c = b; ///still the value of b would be **undefined** because at compilation //step it was defined that at the inner scope b is there i.e. that a b is there at //the inner scope but interpreter has not assigned any value to b yet

console.log(c);

var b = 20;

}

Inner();

}

**Tutorial 15: Hoisting**

1. With the knowledge of Compilation and Interpretation; the concept of hoisting becomes very easy. First the compiler looks at only the vars and then interpreter executes what vars are doing.
2. I seem like the variables that we declare are hoisted to the very top and then there is the code we write.
3. a = 10;

console.log(b);

c++;

var a;

var b;

var c;

1. Now, as there is the compilation step which checks only the declaration of the variables, so even if the variables are declared at the bottom they will be compiled first because running part is done by interpreter. **This is called Hoisting.**
2. Hoisting is also true for functions.
3. Recursion is function calling itself; now recursion is also possible with more than 1 function in which function 1 calls function 2 and function 2 calls function 1.
4. Example

function fnA(){

fnB();

}

function fnB(){

fnA();

}

**Tutorial 16: Using strict mode**

1. var myName = “”;
2. myname = “Sukhmeet Singh”; //misspelled myName; so this will create a variable in global //scope, which can be very dangerous
3. To stop the creation of global variable (when we assign value to the variable without declaring it) behaviour of JavaScript one can use **Strict Mode**.
4. To use the strict mode we need to write “use strict”; in the very first line of the JavaScript file.
5. When we use strict mode, below code will through error:
6. “use strict”;

var myName = “”;

myname = “Sukhmeet”;

1. We can also, only apply strict mode inside the function:

function myCode(){

“use strict”

var myName = “”;

myname = “sukhmeet”;

}

1. Try to use strict mode everywhere.

**Tutorial 17: Introducing Closures**

1. var a = 10;
2. function outer(){

var b = 20;

var inner = function (){

console.log(a);

console.log(b);

}

return Inner; //this variable will help us execute this inner function outside of outer

}

1. var innerFn = outer(); //here the inner function object will be returned
2. Compiler creates the scope chain and the interpreter looks at the scope chain
3. innerFn(); //now when we execute the inner function; the a is global variable and b is a //local variable of outer function; so, what should get executed for b?
4. Now, when we call innerFn(); there is no function outer present at this very moment
5. innerFn(); will only execute inner and will not execute outer, so, how would it get the value of variable b?
6. Now, we know that JavaScript creates scope chain, what JavaScript does is it creates a function declaration/function expression when it creates the inner function object - here it also remembers the scope chain when the function was declared i.e. when the function object was assigned. Now the function object inner not only contains the function object, it also contains the scope information.
7. So, the function will always have the snapshot of the scope chain. It knows where each variable is and where it needs to point.
8. So, the inner function will always know a and b both, no matter, where ever you call it.
9. **Definition of closure is:**
   1. **A function which remembers its scope or lexical scope.**

**Tutorial 18: Closures in detail**

1. var a = 10; // here only 1 copy of a would be created
2. Now, if I run this script for second time, another copy of a would be created, which may be different from the **a** that got created the first time.
3. So, everytime I run that script new copy of a would be created.
4. var a = 10; //when this script gets executed one copy of a is created
5. function outer(){

var b = 20; // when this function gets executed one copy of a is created

var inner = function (){

console.log(a);

console.log(b);

}

return Inner; //now this function variable remembers the copy of a and b in current //execution

}

1. That’s what we mean that the function remembers where the copy of a and b is there.
2. Now, in this function object i.e. inner there is another property which remembers the state of that scope it remembers the creation of a and b.
3. Now, the function inner has reference the variable b – which is part of function outer, so, the garbage collector won’t be able to delete the variable b even after the function outer has already executed. Ideally when the function execution completes all its local variables are garbage collected.
4. Now, point to remember is that if the variable outer gets executed second time a new b will be created.
5. var = innerFn2 = outer(); //new inner function created and new b created; now the second new inner function will have same value of a as it is global variable but the new copy of b because it is created again inside outer.
6. var a = 10;
7. function outer(){

var b = 20; // when this function gets executed one copy of a is created

var inner = function (){

a++;

b++;

console.log(a);

console.log(b);

}

return Inner; //now this function variable remembers the copy of a and b in current //execution

}

1. innerFn();
2. innerFn2();
3. Output 🡪 11

21

12

21

**Tutorial 19: Closures in Call-backs**

1. JavaScript has a feature called callback where you send a function to another function and you have that function execute the first function.
2. As you know JavaScript execution is single threaded. When the page executes we have the option of 1 thread execution, because we cannot make multiple threads to execute in JavaScript.
3. The JavaScript language doesn’t have the features like wait and pause. In JavaScript we have a function called setTimeout();
4. .
5. a = 10;
6. setTimeout(fn(), 1000); //this function comes with JavaScript; first parameter is the function and //the second parameter is time in milliseconds
7. It helps in execution after sometime; the time in this function is excepted in milliseconds.
8. var fn = function(){

console.log(a);

} ;

1. setTimeout(fn(), 1000);
2. Variable which is passed in setTimeout(); function is executed after the time which is provided in the
3. Now, this is practical usage of closures, here the function fn is passes to another library function

setTimeout(fn(), 1000); Now the executing function is setTimeout(fn(), 1000); it doesn’t know about the variable a etc. Here the concept of closure comes into the picture because the instance of a is stored inside the function variable fn.

**Tutorial 20: The Module Pattern**

1. In JavaScript there is nothing like public and private. You can create functions as the properties of objects, then it is accessible as any other property.
2. Practically Closure is what is called a module pattern in JavaScript.
3. var person = {

“firstname”=”Sukhmeet”;

“lastName”=”Singh”;

“getFirstName”= function(){

return this.firstname;

},

“getLastName”= function(){

return this.firstname;

};

1. In JavaScript something is called module pattern used for creating the private data. Now, we know that something inside the scope will not be available outside. So, this is the concept we will use to hide things in JavaScript.
2. function createPerson(){

“firstname”=”Sukhmeet”;

“lastName”=”Singh”;

var returnObj = {

“getFirstName”= function(){

return firstname;

},

“getLastName”= function(){

return lastname;

},

“setFirstName”= function(name){

firstname = name;

},

“setLastName”= function(name){

lastName = name;

},

return returnObj;

}

1. var person = createPerson();
2. console.log(person.getFirstName);
3. person.setFirstName(“Foo”);
4. Any variables declared in a function get created everytime the function is called.

**Tutorial 21: Closures In async Callbacks**

1. var i;
2. for (i=0; i<10; I++){

console.log(i); //it will print first 10 numbers

}

1. Var print = function(){

console.log(i);

};

1. for (i=0; i<10; I++){

setTimeout(print, 1000); //here it will print 10 – 10 times because when it waited for 1 //sec; the final value of I was 10

}

**Tutorials 22: Solving async with closures**

1. for (i=0; i<10; I++){

setTimeout(print, 1000); //Now, here we need multiple copies of I i.e. the value of the i //which was stored in at the time setTimeout(print, 1000); was called

}

1. This we can do by storing the value each time in a local variable of a function.
2. Easiest way to this here is to use IIFE
3. for (i=0; i<10; I++){

(function(){

var currentValueOfI = i;

setTimeout(function(){

console.log(i);

}, 1000); //Now, here we need multiple copies of I i.e. the value of the i //which was stored in at the time setTimeout(print, 1000); was called

})(); //there is a scope that gets created for this IIFE

}

1. Now, when we execute the above IIFE, there well be 1-10 counting printed.
2. Below expression also works in a similar way
3. for (i=0; i<10; I++){

(function(currentValueOfI){

setTimeout(function(){

console.log(i);

}, 1000); //Now, here we need multiple copies of I i.e. the value of the i //which was stored in at the time setTimeout(print, 1000); was called

})(i); //there is a scope that gets created for this IIFE

}

**Tutorial 23: Conclusion**

1. We read this course of JavaScript Scopes and closure in depth, we read in this course:
2. What scopes are and how JavaScript creates scopes i.e. by function scoping
3. We learnt how to use the concept of function scoping using **IIFE (immediately invoked function expression)**
4. We also learnt about the difference between the read operation and write operation in JavaScript
5. How the undefined variable manifests in the read vs write expression.
6. We also understood the problem of the Global Scope in JavaScript and also how to prevent the typo on the local variable which would lead to creation of the global variable i.e. by using the **strict mode.**
7. We also learnt the concept of **Hoisting** in JavaScript
8. We learnt what closures are.
9. We also looked at few of the patterns i.e. module pattern.

**Tutorial Series 3: JavaScript Objects and Prototypes In-depth**

**Tutorial 1: Introduction**

1. JavaScript is an object oriented programming language. Java Script will be used to create objects in the code solve problems using object oriented paradigm.
2. Now, there are multiple ways in which we can create objects.
3. We have constructors in JavaScript which are different from the constructor model of other programming languages like Java/C++
4. We will also look at multiple ways in which a function can be executed.
5. Various uses of this reference.
6. this reference is fairly confusing concept used in JavaScript.
7. Prototypes are also one of the confusing concepts in Java Scripting.
8. We will learn what prototypes are and how it lets you make an object based on a particular template.
9. JavaScript does not have the concept of classes. In core language there is not concept of class like C++ and Java. But prototypes have features which lets you build something which resembles with classes.
10. There is a short cut which lets us build something which resembles classes.
11. We will understand what prototypes are and how we can modify the behaviour of prototype to act like a blueprint of classes. And then create objects which follow a particular blueprint
12. Then we will write some code which uses all the concepts that we have used in this course.

That will be semantically understanding of these concepts.

**Tutorial 2: Objects basics**

1. Object is a collection of multiple values and variables which are primitive or we can also have an object which bundles multiple values. You can put one more object, array or a function in an object. So an object is collection of multiple values.
2. An object can be created in multiple ways in JavaScript; the simplest way is to create an object inline – as shown below
3. var myObj = {};
4. console.log(myObj); 🡪 Object{ }
5. myObj.foo = “value”;
6. console.log(myObj.foo); 🡪 “value”
7. Objects in JavaScript are not class based so there is no blueprint
8. Object is sort of key-value pair
9. var myObj = {

“foo”:”Value”,

“age”: 30,

“address”:{

“foo”:”123 JS”,

“city”:”JS”,

“pincode”: 12345

}

};

1. Console.log(myObj); 🡪 Object{foo: “Value”, age: 30, address: Object}
2. Now, you can access the properties of an object by dot notation or [] notation.
3. Console.log(myObj.foo); 🡪 “Value” //Dot notation
4. Console.log(myObj[“foo”]); 🡪 “Value”// square bracket notation

**Tutorial 3: Creating Objects**

1. JavaScript is considered an object oriented programming language in the sense that you can create objects to solve your programming problems.
2. If you are using object then the objects will represent your entities/things in your system and you can add logic so that they become the behaviours of the system. We can have objects interacting with other objects to solve your computing problem – that is one way in which you can write code in JavaScript. Let’s take an example of employee management system. You will have objects for employees, you will have objects for managers and some kind of method i.e. the person reports to a particular kind of manager or not.
3. var emp 1 = {};
4. emp1.firstName = “Michael”;

emp1.lastName = “Scott”;

emp1.gender =”M”;

emp1.designation = “Regional Manager”;

1. var emp2 = {};
2. emp2.firstName = “Dwight”;

emp2.lastName = “Schrute”;

emp2.gender =”M”;

emp2.designation = “Assistant to the Regional Manager”;

1. Now there would be so many employees so we cannot repeat this all again and again.
2. Now, we would write a code in such a way that we can reuse it at multiple places. So, we will create a function to handle this situation.
3. function createEmployeeObject(firstName, lastName, gender, designation ){

var newObject = {};

newObject.firstName = firstName;

newObject.lastName = lastName;

newObject.gender = gender;

newObject.designation = designation;

return newObject;

}

1. var emp3 = createEmployeeObject(“Jim”,”Halpert”,”M”,”Sales Representative”);

**Tutorial 4: JavaScript Constructors**

1. You can write functions which create objects; these functions are called constructor functions.
2. Constructers are functions which let you populate the objects you want to create.
3. So, everytime we create an object we use a **new** keyword to create an object and that function prepares the object and gets everything ready and then returns the prepared object to you.
4. Now, think of the function given below which creates an object
5. function createEmployeeObject(firstName, lastName, gender, designation ){

var newObject = {};

newObject.firstName = firstName;

newObject.lastName = lastName;

newObject.gender = gender;

newObject.designation = designation;

return newObject;

}

1. Now, let’s say there are 100 different functions like this in your application; Now, if we ask you what is the common thing in all these different functions?; i.e. what are the common lines of code in all these functions

Now, if we analyse properly we will find that the first line and the last line are more of the common in all these objects i.e. var newObject = {}; and return newObject;

1. So when we are creating the object what would be common is the first line creating an empty object and the last line returning the object. Now, these 2 lines are common in all the functions. **We will write a shortcut which will let us not have to write these lines when you are creating functions that create objects. The way to do this is to call the above function in the constructor mode.**
2. We need to tell JavaScript that this not an ordinary function, this is the function that we are using to create an object.
3. The way to tell JavaScript that the function you are calling is a Constructor function is by adding a **new** keyword while creating the function, as shown in the example given below.
4. var emp3 = new createEmployeeObject(“Jim”,”Halpert”,”M”,”Sales Representative”);
5. Now, as we are calling the above function in constructor mode, we need to write the function as given below:

function createEmployeeObject(firstName, lastName, gender, designation ){

newObject.firstName = firstName;

newObject.lastName = lastName;

newObject.gender = gender;

newObject.designation = designation;

}

1. Word that follows new keyword has to be the name of the function and not the name of the class like other languages like C++/Java
2. What new key word does is that it switches it into the construction mode; this enables JavaScript to let you do a shortcut i.e. it eliminates the first line and the last line in the functions that we define common.
3. What JavaScript does is, its makes available the new object as this, as shown in the function given below:

function createEmployeeObject(firstName, lastName, gender, designation ){

//var this = {}; //JavaScript adds this line in the beginning by itself when a function is //called in constructor mode

this.firstName = firstName;

this.lastName = lastName;

this.gender = gender;

this.designation = designation;

//return this; //JavaScript adds this line in the end by itself when a function is //called in constructor mode

}

1. The commented lines above are the lines that JavaScript is going to do for you. Now, instead of using the object name now we will use this, as this object is created by function in the constructor mode.

**Tutorial 5: Difference between regular functions and constructors**

1. In this video we will demonstrate the difference between constructor and non-constructor ways of creating objects in JavaScript.
2. Now, we will create a bicycle object with couple of properties on it, first with normal function and then in the constructor mode.
3. var bicycle = {

“cadence”= 50,

“speed”= 20,

“gear”=4

};

1. Now, we don’t want to create the object inline i.e. directly but we want to create it by using a function or a constructor.
2. function createBicycle (cadence, speed, gear){

var newBicycle = {};

newBicycle.cadence = cadence;

newBicycle.speed = speed;

newBicycle.gear = gear;

return newBicycle;

}

1. Now, instead of creating the object inline, I can call this createBicycle function.
2. var bicycle = createBicycle (50, 20, 4);
3. var bicycle2 = createBicycle (20, 5, 1); //creating one more
4. Now, we will create the bicycle object by using constructor.
5. function bicycleCostructor (cadence, speed, gear){ // bicycleCostructor is not the best //naming convention because the name of the constructor should start with capital letter for //now we will use this name

//var this = {}; //this line is added by JavaScript internally

this.cadence = cadence;

this.speed = speed;

this.gear = gear;

//return this; //this line is added by JavaScript internallys

}

1. var bicycle3 = new bicycleCostructor (cadence, speed, gear); //function called in constructor //mode.
2. The constructor should always be named with a capital letter, as shown in the function below:
3. function Bicycle (cadence, speed, gear){ // bicycleCostructor is not the best naming //convention because the name of the constructor should start with capital letter for now //we will use this name

//var this = {}; //this line is added by JavaScript internally

this.cadence = cadence;

this.speed = speed;

this.gear = gear;

//return this; //this line is added by JavaScript internally

}

1. var bicycle3 = new Bicycle (cadence, speed, gear);

**Tutorial 6: Switching function types and calls**

1. If we call a constructor function without this keyword then it will not work, however, if we call a normal function with this keyword, it will still work.
2. When a normal function is called with new keyword:
3. var bicycle3 = new bicycleCostructor (cadence, speed, gear);
4. function createBicycle (cadence, speed, gear){

//var this = {}; //this line goes wasted

var newBicycle = {};

newBicycle.cadence = cadence;

newBicycle.speed = speed;

newBicycle.gear = gear;

return newBicycle;

// return this; // this line also goes wasted

}

1. when the constructor is called without new keyword:
2. var bicycle3 = Bicycle (cadence, speed, gear);

function Bicycler (cadence, speed, gear){ // first and last line not defined i.e. //definition of this and return of this is not defined.

this.cadence = cadence;//this will be created as a global object

this.speed = speed;

this.gear = gear; //if you don’t return anything explicitly then undefined will be //returned.

}

**Tutorial 7: Function Execution Types**

1. There are 4 different ways in which we call a function in JavaScript. Till now we have seen the 2 ways i.e.
   1. Regular mode
   2. Constructor Mode
2. Once you come to know all the 4 ways to create functions, we would be able to understand this keyword very easily.
3. Let’s start with a function foo
4. function foo(){

Console.log(“Hello”);

}

1. foo(); //**Method 1 of calling functions**
2. obj.foo = function (){

console.log(“Hello”);

};

1. obj.foo(); //**Method 2 to execute functions**
2. As we saw above method one executed function directly; however, method 2 executes it as a function property.
3. The third way to use function is calling it as a **Constructor**. If we use a new keyword while calling a function, then it would get called in constructor mode.
4. new foo(); //**Method 3 to execute a function i.e. constructor way**
5. There is method 4 also to call a function, we will discuss it later. For this method we need to understand the concept of execution context.

**Tutorial 8: The this argument values**

1. Here we will understand the concept of execution context. This is actually the concept in most programming languages. When there is a function which gets called it is always called in a particular context. The stacks of variables containing different scopes and the call chains. There are bunch of other stuff also.
2. The idea is there is some context i.e. some group of data which helps the execution of the particular function and the function might need to refer to it. So, the JavaScript also has the concept of execution context. The content of the execution context depends on the method in which the function is called by any 1 one of the 4 ways explained above.
3. One of the elements of the execution context is **this**, as we know that there are 2 implicit arguments of a function, one of them is **this** and other one is **arguments.**
4. Anytime we write a function there is always an implicit argument called **this**.
5. So, anytime you write a function this will always point to something, what that thing is it really depends on which of the above ways of calling the function you are using.
6. However, once the function call has been identified, the value of **this** is very predictable.
7. Now, let’s identify those different values of this.
8. function foo(){

Console.log(“Hello”);

Console.log(this); // points to the window object of the browser.

}

1. foo(); //**Method 1 of calling functions**
2. Here, the value of the **this** keyword is the global object. The global object depends on the runtime environment; if you are running the JavaScript on the browser then the global object would be the window object. This means the window or the tab in which you are running the JavaScript.
3. Now, let’s move on to the second approach for calling the function
4. var obj ={};
5. obj.foo = function (){

console.log(“Hello”);

};

1. obj.foo(); //**Method 2 to execute functions; here the function is called as a property of the //object**
2. this here refers to the object itself i.e. the object on which the called function property is there. Here this would be obj object itself.
3. Now, we will look at method 3 i.e. constructor way of calling the method and will examine the value of this
4. new foo(); //**Method 3 to execute a function i.e. constructor way**
5. In the constructor mode call this always refers to the newly created object.
6. So, in this case the value of this is an empty object because with new keyword an empty object gets created which doesn’t contain any property.

**Tutorial 9: Working on objects with reference**

1. //Function meant to be called in constructor mode

Function Bicycle(cadence, speed, gear, tirePressure){

this.cadence = cadence;

this.speed = speed; //this reference here is the new object created

this.gear = gear;

this.tyrePressure = tyrePressure;

this.tirePressure= function(){ //this reference here points to bicycle object

this. tyrePressure +=3;

}

}

1. //calling the function in constructor mode
2. var bicycle = new Bicycle(50, 20, 4, 25);
3. bicycle.inflateTires();
4. var bicycle2 = new Bicycle(50, 20, 4, 25); //this for this object will refer to bicycle2 //object
5. bicycle.inflateTires();
6. We can see that this reference on the constructor function Bicycle is different from this reference of the inner function inside the constructor.
7. This in the constructor is the new object created; whereas this reference inside the inner function is the object from which that function is called.
8. Every function could have different this reference; it doesn’t matter if it is inside another function or not.

**Tutorial 10: Using the call function**

1. //Function meant to be called in constructor mode

Function Bicycle(cadence, speed, gear, tirePressure){

this.cadence = cadence;

this.speed = speed; //this reference here is the new object created

this.gear = gear;

this.tyrePressure = tyrePressure;

this. inflateTires = function(){ //this reference here points to bicycle object

this. tyrePressure +=3;

}

}

1. //calling the function in constructor mode
2. var bicycle = new Bicycle(50, 20, 4, 25);
3. bicycle.inflateTires();
4. var bicycle2 = new Bicycle(50, 20, 4, 25); //this for this object will refer to bicycle2 //object
5. bicycle.inflateTires();
6. function Mechanic(name){

this.name = name;

}

1. var mike = new Mechanic(“Mike”);
2. Now, we need to give mike rites to inflate tyres of any bike. Right now the inflate tyres functionality is in the Bike constructor.
3. mike.inflateTires = bicycle1.inflateTires;
4. mike.inflateTires();
5. Now, the point is that, in the function inflateTires this is referring to mike object, however, this inside the Bicycle constructor refers to the new this object created.
6. Now, we will circumvent the default this value with new this value which we will send from the where the function is called. Now, this will refer to the new object we are sending.
7. So, I wanna tell the interpreter to use another object for this.
8. **4th Way of calling the function is below:**

**4th way to call a function is by using the property on the function object, as shown in the example below:**

**function foo(){};**

**//one of the property in the function object is call, a function in JavaScript is an object //which can have a property.**

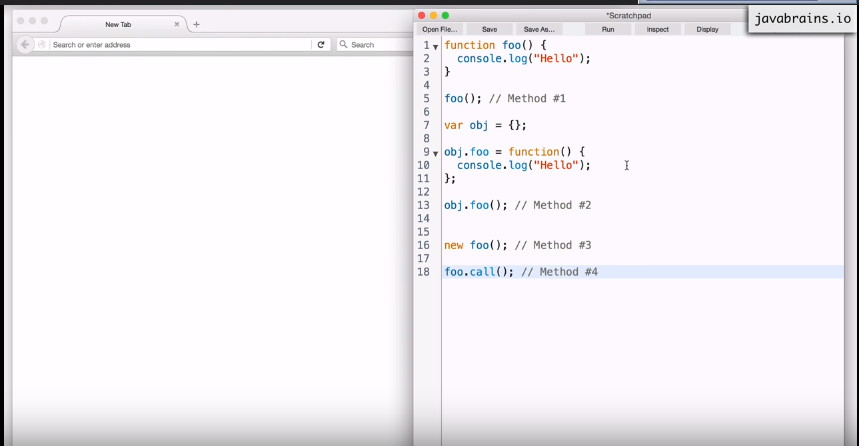
**Any, object in JavaScript has an out of the box property called call – which is another function.**

**foo.call(); //this line is exactly the same as just calling the function itself foo();**

**Now, in the call property of the function we can pass an object.**

**What the call property does is take this object and bind that object to the this reference.**

1. **Function foo(){ this.abc = def;} //this here is the object passed in the call function**
2. **foo.call({});//object passed in the call function which would be the new this**
3. Now, we will modify the above example where Mechanic inflates the tyres.
4. mike.inflateTires.call(bicycle1); //now this would be bicycle1
5. 4 ways to call an object are given below:



**Tutorial 11: when Constructors aren’t good enough**

1. Here we will learn about prototypes, prototypes are used as a blueprint in JavaScript. A lot of object oriented languages have the concept of classes.
2. Prototypes are used to build some kind template/blueprint which influences the objects.
3. **It is not exactly equal because you don’t create the instances of prototypes., but you can create behaviours which influences many objects**.
4. //Function meant to be called in constructor mode

Function Bicycle(cadence, speed, gear, tirePressure){

this.cadence = cadence;

this.speed = speed; //this reference here is the new object created

this.gear = gear;

this.tyrePressure = tyrePressure;

this.tirePressure= function(){ //this reference here points to bicycle object

this. tyrePressure +=3;

}

}

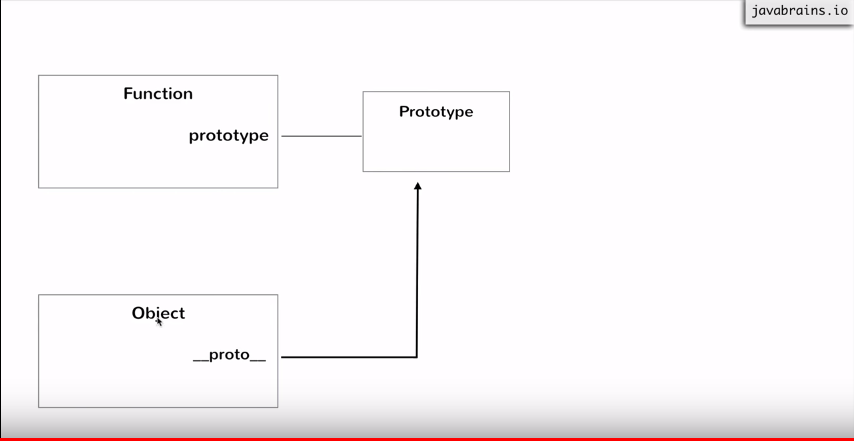
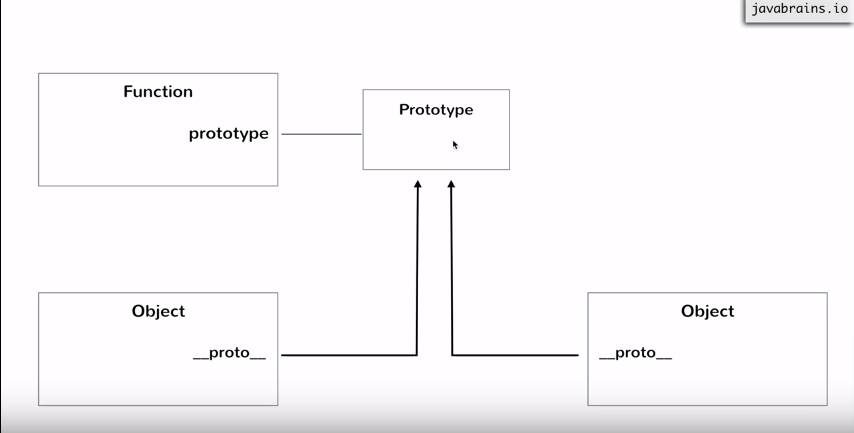
1. var bicycle1 = Bicycle (50, 20, 4, 25);
2. In constructors when a new object is created, all objects have a separate copy of variable instances. However, in other languages which use the concept of classes have a same copy as the class has.

**Tutorial 12: Introducing the prototypes**

1. function foo(){};
2. When JavaScript processes the function, it doesn’t create only one object – it creates 2 objects.
3. The second object which gets created for every object is called prototype object.
4. What is the point of the prototype object – this we will explain in next tutorial?
5. We can access the prototype object by using the .operator i.e.

Foo.prototype

1. Whenever JavaScript creates the new function it creates the property prototype and the prototype property points to the prototype object.
2. When an object is created by using a new keyword, then a property is created in it i.e. \_\_proto\_\_ which points to the prototype object of the parent.



**Tutorial 13: Property lookup with prototypes**

1. If we create a property on prototype object of function; then all the objects created by using new keyword from that object would have that property form the \_\_proto\_\_ object.
2. Foo.prototype === newObject.\_\_proto\_\_ // 🡪 true
3. **If we try to fetch a property on an object which does not exist on that object then it will try to find that property on the parent i.e. prototype object**

**Tutorial 14: Object behaviours using prototypes**

1. The advantage of having prototype object is that, there can be many objects created from the prototype object; maybe they require the common property, which can be present on the prototype object.
2. By prototype we can add a couple of properties to all the objects created from that function.
3. Lookup for prototype and its properties happens dynamically at runtime
4. You can put a property on the prototype at the runtime and the minute you do that every object having that prototype object will have that property.

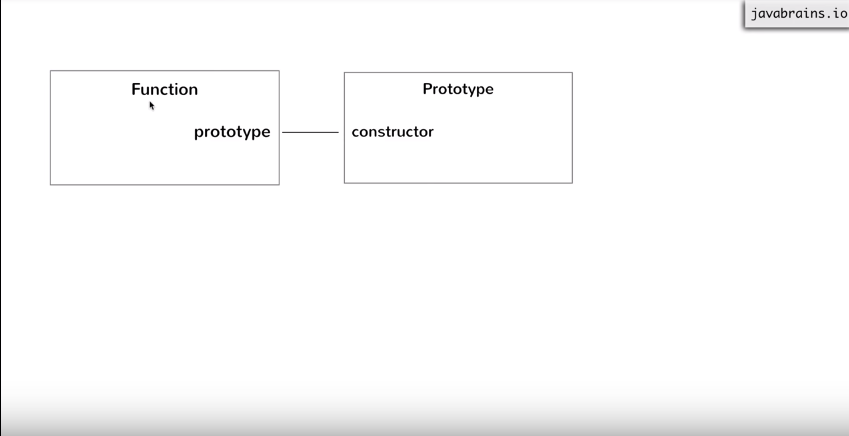
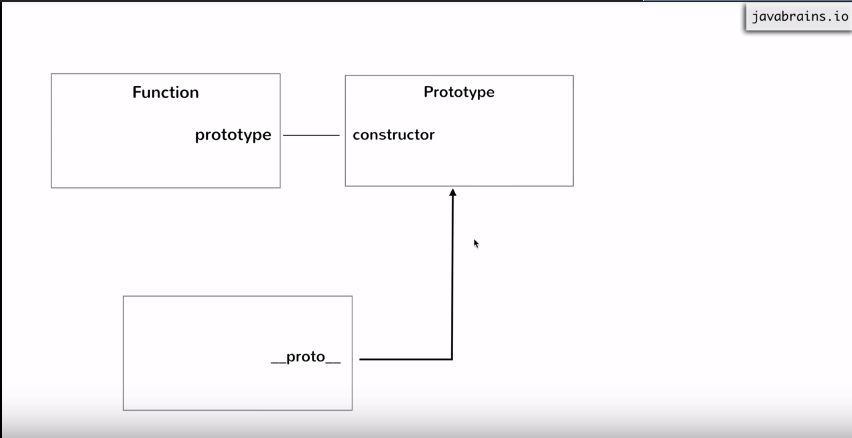
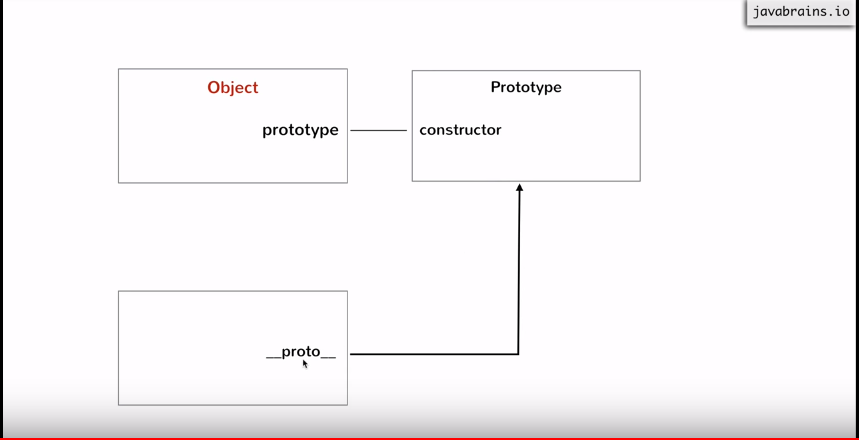
**Tutorial 15: Object Links with Porotypes**

1. Getting a property = \_\_proto\_\_ from the object created by using new keyword is difficult to write i.e. \_\_proto\_\_ is difficult to write; so there is another name of this property i.e. dunder-proto
2. In the object there is property \_\_proto\_\_ which points to the prototype object but inside the prototype also there is a property which points to the function i.e. constructor i.e. foo.prototype.constructor will point to function object.
3. Suppose there is a sukhmeet object created from a function so this will point to function i.e. **sukhmeet.\_\_proto\_\_.constructor**
4. **Now, if we want to create a new object mike which needs to be created from the same function from which the object sukhmeet was called then we can use the below code:**

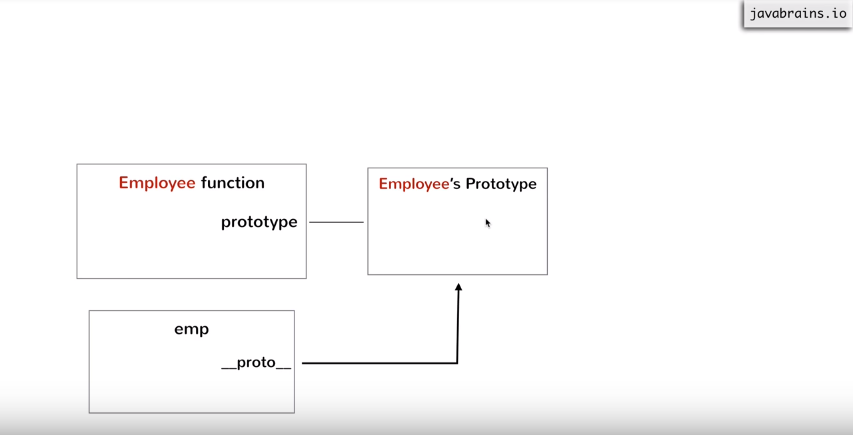
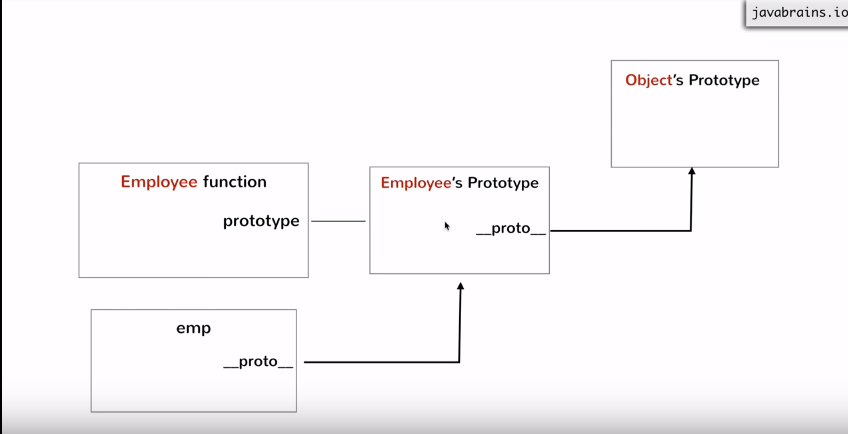
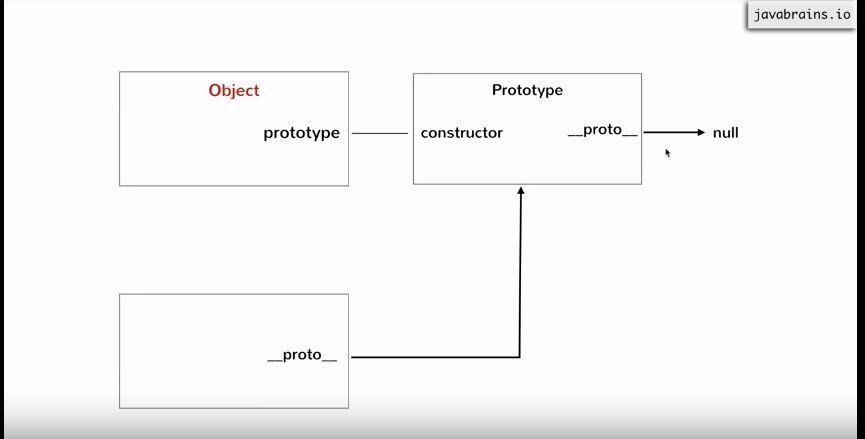
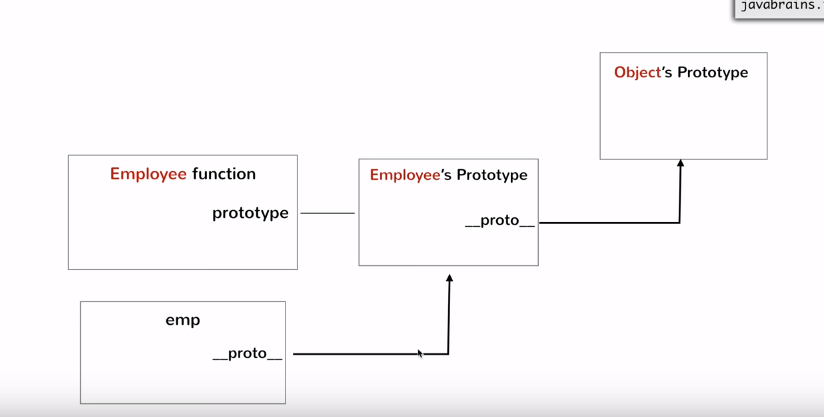
**var mike = new sukhmeet.\_\_proto\_\_.constructor(); //this will create a new object from // the same function from which sukhmeet object was created**

1. We can also change the properties provided., we can also change the function where constructor points.

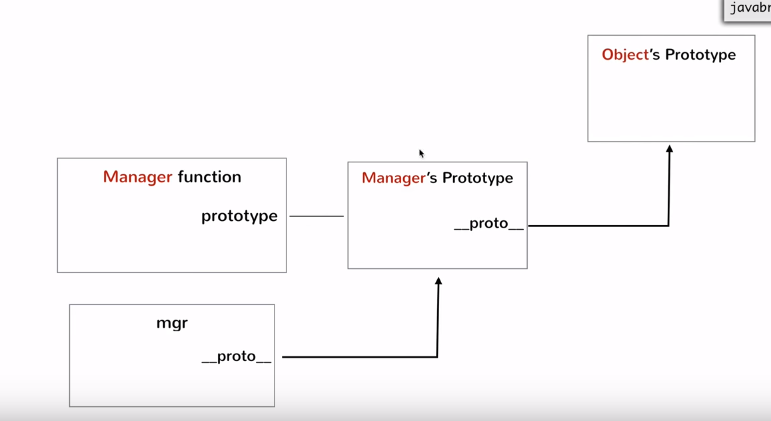
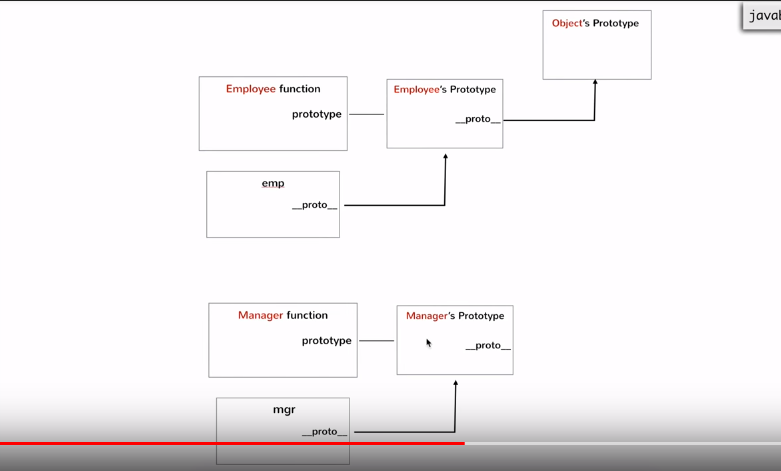
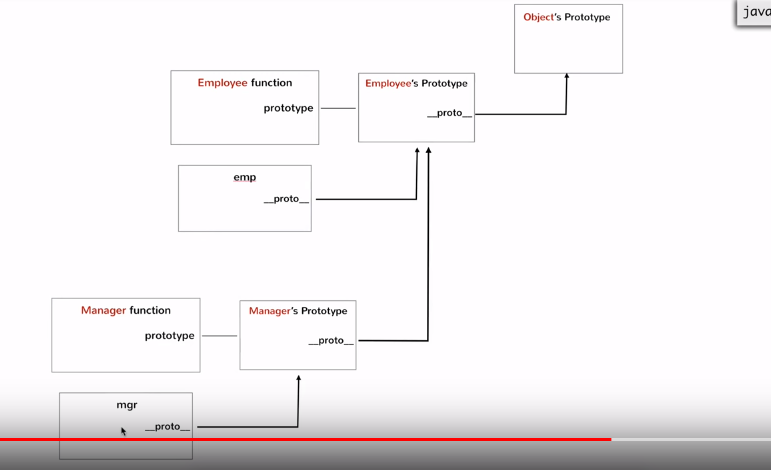
**Tutorial 16: The Object function**

1. Is a function object – yes it’s both
2. Just like we have global window object we also have global functions.
3. Just like a function is an object in JavaScript, this global function which is called Object is also an object.
4. Object; //type this on console 🡪 function Object()
5. Object(); //🡪 Object{}
6. 
7. 
8. var simple = {}; //simplest way to create an object
9. var obj = new Object(); //this is same as var simple = {};
10. simple.\_\_proto\_\_ === obj.\_\_proto\_\_ ; // 🡪 true
11. 

Tutorial 17: The Prototype object

1. function Employee(){}
2. var emp = new Employee();
3. 
4. When we use new Object(); then what happens is shown in the screen shot below:
5. 
6. If a property is not found at the bottom level then it tries to find it at one level above it, id it still doesn’t find it tries to find it in Objects Porotype.
7. var emp = new Employee();
8. emp.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype; // 🡪 true
9. If you create a property on Object’s Prototype; then that property will be available to all the objects in the program. Its kind of global property.
10. Prototype of object points to null, as shown in the screenshot below:
11. 
12. 

**Tutorial 18: Inheritance in JavaScript**

1. To change the behaviour we can make the dunderproto of one object to point to some other object not point to the prototype object of Object.
2. function Employee(name){this.name = name;}
3. Employee.prototype.getName = function(){return this.name;}
4. var emp1 = new Employee (“Jim”);
5. emp1.getName(); // 🡪 Jim
6. Now, we will create some manager objects:
7. Function Manager(name, dept){this.name = name; this.dept = dept;}
8. Manager.prototype.getDept = function(){ return this.dept; };
9. Var mgr = new Manager(“Michael”,”Sales”);
10. mgr.getDept();
11. 
12. 
13. Now, we will change the pointing direction of dunderproto property of Manger object.
14. 
15. **Mgr.\_\_proto\_\_.\_\_proto\_\_ = Employee.prototype;// no the dunderptoto will point o //empoyees prototype**

**Tutorial 19: Conclusion**

1. Read, you don’t know JS book.