Part 1: Basics

What is Javascript??

Is a Scripting language (a language where instructions are written for a runtime environment. Example: shell script, windows power shell programming.When to execute a pile of commands/instructions or code we can put them into a scripting file ask the runtime environment to excuse the file and that is called scripting file.)

Interpreted language (no compilation, instructions executed directly by the runtime environment)

A Brief History of JavaScript

created by Brendan Eich in 1995 during his time at Netscape Communications(was rival to Microsoft's Internet explorer then). It was inspired by Java, Scheme and Self.Created as a compliment of JAVA for in-reaching client side experience.Designed to be easy and forgiving and welcoming for the newcomers. Due to its rivalry with Microsoft, Netscape felt the urge to standardize Javascript so that any browser claiming to support JS must follow the standards.
A committee called ECMA was formed for the standardization process.Thats what JS often referred as ECMA script.

Variables and Javascript Type system

Javascript doesn't have the concept of types variables.There is no pre-declaration of types required to create a variable.

There are some primitive types though that was designed naturally like any other languages.

Numbers: no int like java or others. All floating point double precession 64 bit numbers.

Boolean: true or false.

String: Sequence of Unicode char(16bit). Everything is a string no concept of char data type JS.

undefined : It is a type and value in JS. Like between declarations (var val)and defining(val = 43); a variables type is undefined and value is undefined.So until u assign some value into a variable JS does not know its type or value so both remain undefined then.Different language has different ways of handling this.

null: It is a type and value in JS like undefined.

Symbol: new permeative data types supported in Ecmascript 6.More like enums.

Difference between undefined and null

They both are non-valued.but something to have a value null it has to be put into it, where as undefined is assigned as soon as you declare a var and it remains until u assign some value into that var.

Summary of Vars and Types

I. No need to declare types.
II. One variable. Can be assigned values of many types.
III. No scoping info.
IV. The typeof operator can be used to interrogate the type of a var.

***type of null returns object which is a bug and still not corrected for its dependencies.

Type Coercion and the === operator

== operator checks if 2 variable's type can be converted into a common type and then it checks their value. Thats why string and int comparison gives true if their value are same unlike other programming language.
=== operator however does not do that and compares variables only if they are of same type and returns false otherwise.

Type Coercion:

Every value has its corresponding boolean value in JS.

- If a var contains **number** then its corresponding boolean value will be true for all non-zero values and false if zero.
- Incase **of string** if Its empty string then corresponding boolean value is false and true otherwise.
- Null and undefined values will be false.

Objects:

Javascript is an object oriented programming language but it is not class based.So you can created objects without classes.Javascript objects are **free-formed**; there is no predefined structure, new properties can be added to an object at any point of time.There also no accessor property.Objects can have **methods**.

The Object Literal Notation :

Rather adding properties later to an empty object you can create an object with properties using **object literal notation** {}. For defining properties and values u have to use this format “property” : value then “,” comma and next properties.
While accessing a non existing property u will get undefined.

The dot and bracket notations:

dot notation is use to access properties of an object.e.g. : object.property ;
We can also use **square bracket notation** by using square bracket [] to access objects property.e.g. : object[“property”]

Difference between dot and bracket notations:

There are certain cases where the name of the property is invalid. So to access an invalid property identifier u can use bracket notation as you can not use dot notation on those cases. e.g. var obj = { “1” : “one”}
Here property name 1 is invalid and you cannot use obj.1 to access it rather you have to use obj[“1”].

Some other cases are :
- Property name is a **reserved key word/** invalid identifier.
- Property name starts with a **number**.
- Property name is **dynamic**.

=== for objects
Returns true if two object’s memory address is same or they point to the same location in memory heap/stack.

Deleting property

delete person.age is the right way. Now person.age will return undefined.

Assigning person.age = undefined would also return the value for person.age undefined but it will not remove the property form the object.

The Secret behind JavaScript Arrays

That like everything else they are Object and it has every properties like an object.
var myArr = [0,1,1,4]

- **myArr.length** is just an object.property call.
- **myArr[0]** can be interpreted like myArr[“0”] as JS interpreter does the type cohesion of 0 (number) to string “0” when it sees a number is being used to access an object’s property.

- Can we use dot notation here? No we cannot use dot notation where property name is number and for array object index properties are number that is why we have to use square bracket notation.

Wrapper Objects

Javascript has en equivalent object for the primitive data types (string, number, boolean, symbol).Whenever we try access any object property (like string.length) JS converts the primitive variable into its equivalent object.Only after conversion the length property becomes available to the variable and we can access those properties. This conversion is not persistent.It happens every time we try to access any property then JS wraps the variable into its equivalent object let us use the property of the wrapper object.The original type of the variable is not changed to the object.So the wrapping object gets created only for a fraction of second.

string is primitive and has primitive object wrapper String.
Others are Number,Boolean,Symbol.

Introduction to Functions:

Functions are also object like array in JS.
No Function overloading.
The **default argument**, and there is **arguments** and **this.**

Function Expressions:

Functions in JS are first class values like number 100 or a string “dummy string” is a value.So you can assign function to a variable.Mainly two ways u can declare function in JS

Function Declaration:

```
function foo(){
return “Hello world”;
}
```

Function Declaration By Function Expression:

```
var f = function foo(){
return “Hello world”;
};
```

Function Declaration By Anonymous Function Expressions:

```
var f = function (){
return “Hello world”;
};
```

Anonymous function will get lost if we assign anything else in the variable after. Like var f = 1;

Number Of Params:

Flexible about count of list of arguments sent to a function.If u send more argument than ie expects the extras are ignored and if you send less remaining are assigned undefined.

Overloaded functions are not possible in JS.(may be because it does not have a compiler??)

Functions on Object

Declare a function and assign it to a property of an object, thus object can have functions/methods on them.

Understanding the this keyword

This refers to the calling object.

Default function arguments

arguments is an Array-like object accessible inside functions that contains the values of the arguments passed to that function.

Array Methods

Under the hood every array is a javascript object.As objects can have method/func, JS has provided some built in methods and properties with Array object.

- push() => adds elements to end of an array
- push() => removes elements from end an array
- unshift() => adds elements to the start of an array
- shift() => removes elements from start an array
- forEach() => runs for each element of an array. Takes a function as an arg. That function can receive the item, index and array by its arguments/parameters.
  example:

```
myArray.forEach(function(item, index, myArray){

})
```

Some other objects like Array are Date,Math etc…

Part 2 : Scopes and Closures In-depth

JavaScript Functions Primer:

JS has first class functions meaning you can create a function and assign to a variable.

Understanding Scopes , Block Scoping and Function Scoping:

Function Scoping in JavaScript

JS does not create scopes for blocks.It is function scoped meaning, whatever declared inside a function is not available outside of it.But variables declared inside a block (like if/else) has scope to be used outside of that block.Whatever declared outside the function like within a global scope is available to use inside the function or block like any other programming language.

Scope hierarchy : Top level scope (Global), Child Scope(Inside functions)
Inside a child scope child variables value will override the global var.

IIFE

Immediately Invoked Function Expression : Anonymous function that gets executed right-away.It is a design pattern which is also known as a Self-Executing Anonymous Function and contains two major parts: The first is the anonymous function with lexical scope enclosed within the Grouping Operator ().

This prevents accessing variables within the IIFE idiom as well as messing up the global scope.

Use Case: Avoid polluting the global namespace.

```
(/** IIFE pattern */

(function add(c) {
    var a = 10;
    var b = 20;
    a = a * b;
    console.log("a:", a);
    console.log("c:", c);
})(10);
```

Read and Write operations

```
/** Read and Write Operation */

var a = 10 ; // assignment operation
console.log( a) // read operation

function greet (name){ // write operation
    console.log(name); // read operation
}
```

Implications of Read and Write operations:

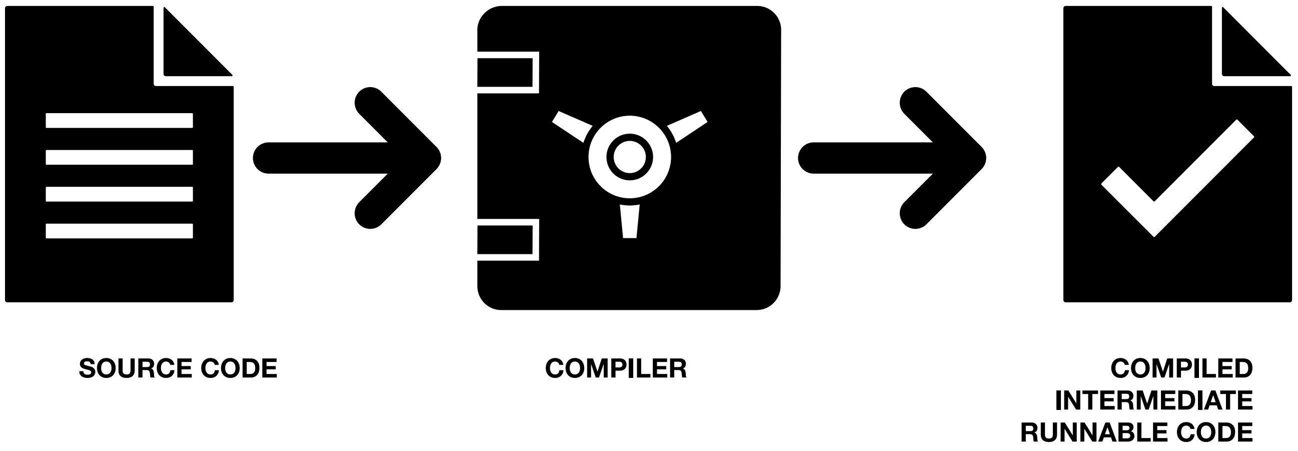
It is ok to do a write operation on a variable without declaring it but not read operation.

The window object

When you create a global variable you actually creating a property on the global object. In case of JS/Browser runtime this global object is window.In case of node runtime this is an object called global.

Compilation and Interpretation:

Compiler phenomenon:



JS runtime does not create an intermediate executable file like traditional compiling process (javac/c++ compiler) but that does not mean there is no compilation process at all.
But it definitely means that JS is an interpreted language as the JS runtime does not create any intermediate code/file. Incase of an interpreted language the runtime execute the source code directly.
However the execution process happens in two steps.First the JS runtime looks for certain signs and make notes of things which it needs to execute the source code. Again the goal of the first/compilation step is not to generate any executable compiled code.
Once the compilation step is done then interpretation step executes and actual src code is executed.
These two steps happens very quickly and together.

Understanding the Compilation step:

In the compilation step scopes are created and variables are assigned to a scope. These scopes forms a scope chain for later use in interpretation step.

```
var myName = "Sadaf Fatin"; // var greeting declared in the
global scope

// var greet declared in the global scope. although it is a
function but JS treats it like a property in the global object.
function greet(greeting) { // var greeting declared in the greet
function scope
    console.log("Hello " + greeting + myName);

}
greet("Good Morning!!”)
```

Understanding the interpretation step:

Code example:

```
var myName = "Sadaf Fatin"; // var greeting declared in the
global scope
// var greet declared in the global scope. although it is a
function but JS treats it like a property in the global object.
function greet(greeting) { // var greeting declared in the greet
function scope
    console.log("Hello " + greeting + myName);
    //** nothing called console in the greet function scope,
    //so the interpreter looks one level up in the scope
hierarchy.
    //(in this case : global scope ) for console object. */
}
greet("Good Morning!!”)
```

Rules: Whenever an object/method/variable is not present within a scope the interpreter keeps looking for it by going one level up all the way up to the global scope. in the scope hierarchy until it finds it.

The Global scope problem:

While the interpreter executes a block if finds a write operation on a variable which has not been declared neither within the function block nor the global scope, it creates the variable as JS normally does, but not in the function scope but in the global scope.

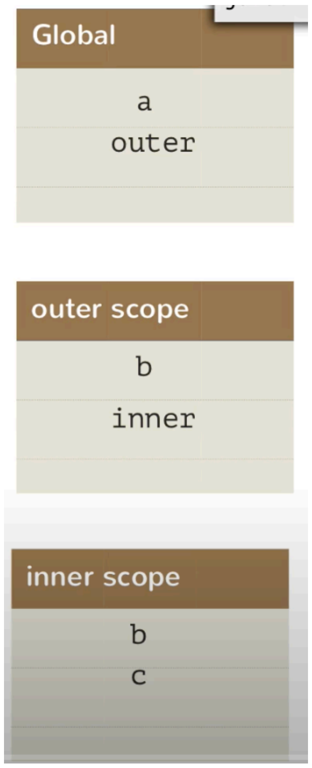
code:

```
/** The Global scope problem: */
var myName = "Sadaf Fatin"; // var greeting declared in the
global scope
// var greet declared in the global scope. as it is a function
but JS treats it like a property in the global object.
function greet(greeting) { // var greeting declared in the greet
function scope
    myAge = 28; // at compilation step these are ignored.
    //But while interpreting var myAge gets declared at the
global scope rather than function scope
    console.log("Hello " + greeting + myName+" your
age:"+myAge);
}
greet("Good Morning!!")
```

Some Exercises and a surprising result:

```
var a = 10;
function outer() {
    var b = 10;
   function inner() {
        var c = b;
        console.log(c); // *** this line prints undefined
        var b = 50;
    }
    inner();
}
outer();
```

The fig in the right shows a scope chain.This scope chain gets created in the compilation step.



The console log command in the above code prints undefined.

Reason for that at the compilation step variables are only added to the scope chain.
As for the inner() function; variable b is just added to its scope at compilation step. The interpreter while executing the inner() function finds the read operation on var b but no value is yet assigned to it up until that line. Thats why it assign undefined value to the variable b.

Below is another example:

```
//** another example of this case */
console.log(temp);
var temp = 12;
```

Hoisting

If there are two interlinked function a and b both call each other form within themselves which one u need to declare first. Code example:

```
/** Function hoisting */
function a(){
    b();
}
function b(){
    a();
}
```

Well it does not really matter as the compilation steps handles the declaration in the scope at the first step and when the interpreter tries to execute it finds both the method ready.So the order of declaration does not matter here. This is called function hoisting.

This happens for **functions by declaration only not for functions by expressions.** A function declaration results in a function object being created in the scope at the compilation step.

Using Strict mode

For read operation on a variable that has not been declared JS throws an error but for write operation JS creates that variable.The way JS manipulates Global scope can be a problem sometimes.Also its not like other programming languages.

A programmer can be strict about these manipulation by using strict mode in JS.
This concept was introduced in Ecma Script version 5.
The way to use strict mode is to use the string “use strict” at the top of the program.These stops JS from doing the manipulations which at times may seem to be crazy for new developers.

Introducing Closures:

What is closures?? To answer this lets follow some observations :

1. JS has the concept of first class function.That is that you can declare a function, pass it to a variable and execute it somewhere else in your program.
2. So a function can be executed in a completely different scope or context from where it was declared.
3. But what if it needs some variables and values that was present in the scope chain of that function while it was declared but not in the scope chain where it is being executed??

Lets follow the code example below:

Code Examples:

```
//** Closures */
var a = 20; // a in global scope
function outer() { // outer in global scope
    var b = 10; // b in outer scope
    var inner = function() { // function by exp. inner is a var 
in outer scope.
        console.log(a); // remembers a from global scope;
        console.log(b); // remembers the scope the function was 
declared in which the
            // had the var b in it.
    }
    return inner;
}

var innerFunction = outer(); // function inner returned by outer
innerFunction(); //inner() is being executed out of the context 
it was declared 
// but it remembers the scope chain at the time of its 
declaration
```

Above code is an example of closures in JS. When function inner() is being executed outside the scope of the function within which it was declared (outer function) it still remembers the the scope chain of that function and thus prints the value of var b.

So what JS does that whenever it creates a function object, whether it is a function by declaration or by expression it remembers the scope chain when that function was declared or that function was assigned.

Not only it remembers the scope chain it actually holds a pointer to the variables.So inside the function if any var gets manipulated it is not the snapshot or copy of that original var but it is actually that variable.

***One thing to remember though that every time you create a new function object; new set of variables/properties are attached to that function object which does not have any tie with the previous object.

```
var innerFunction = outer(); // function inner returned by outer
innerFunction(); //inner() is being executed out of the context
it was declared
// but it remembers the scope chain at the time of its
declaration
```

```
var innerFunction2 = outer(); // this will create a new function
object inner which
//will be pointing towards newly created set of variables
innerFunction2();
```

CLOSURES: The Definition of a closure is a function that remembers its scope.

Closures In-depth::Closures In Callbacks:

Below code is a real world example of a function remembering its scope chain or closures.

```
//**Real world closure example as callback function**//
// continuing form previous example...
setTimeout(innerFunction, 5000); //sending the function object
as a function parameter
//where setTimeout function will execute it in its context where
variable a or b
//doesn't even exists;
```

Closures In-depth 20 - The Module Pattern:

```
//** Closures In-depth-The Module Pattern**//
function createPerson() {
    var firstName = "";
    var lastName = "";
    var personObject = {
        getFirstName: function() {
            return firstName;
        },
        getLastName: function() {
            return lastName;
        },
        setFirstName: function(name) {
            firstName = name;
        },
        setLastName: function(name) {
            lastName = name;
        }
    }
    return personObject;
}
var person = createPerson();
person.setFirstName("Sadaf");
console.log(person.getFirstName()); // the person object inside
createPerson() remembers its scope
//using Closures
```

***Any variable declared inside function gets created every time the function executes.

Closures In-depth-Closures In async Callbacks::

```
//** Closures In-depth - Closures In async Callbacks **//

var i; // var i in global scope
var print = function() {
    console.log(i); //function print remembers the scope at the 
time of its declaration;
    //remembers i from global scope/scope of the function was in
}
for (i = 0; i < 10; i++) {
    setTimeout(print, 1000); //set time out takes print function 
as parameter as callback function
    // takes the same function print() and waits 1 sec.
    // by the time it executes print() all iteration of loop is 
done
    // now the global var i has the value 10.
    // as print() function remembers its scope 
    // which had i declared it prints the value of that var i.
}

//** Closures In-depth - Solving async with closures **//

var printVal = function(val) {
    console.log(val); //function print remembers the scope at 
the time of its declaration;
    //remembers i from global scope/scope of the function was in
}
for (i = 0; i < 10; i++) {
    (function(currentVal) {
        setTimeout(() => {
            printVal(currentVal);
        }, 1000);
    })(i)
}
```

Part 3: Objects and Prototypes

Objects

Objects are a collection of multiple values and properties. In JS objects are not class based, so there is no blue print that an object must follow, you can create any property on any object whenever you want.There is no data type restrictions also.

Creating Objects in JS

Creating by functions

```
//Object creation by function
function employee(name, age) { // this function construct an
empty object and add properties to it
    var employee = {}; // In-line object
    employee.name = name;
    employee.age = age;
    return employee;
}
```

Javascript Constructor

From above code we can create object like below:

```
var employee = employee("Sadaf", 28);
```

JS provides a feature to skip the object creation and return statements. If you create an object using **new** keyword like below you can skip the empty object creation and return statement in the **constructor function.**

```
//JavaScript Constructors
function Student(name, age) { // this functions are called
constructor function
    //var this = {}//JS is taking care of this line
    this.name = name;//changed the var name with standard var
name this
    this.age = age;
    //return this //JS is taking care of this line
}
var student = new Student("Sadaf", 29);
```

Difference between regular functions and constructors:

A function gets called in constructor mode not for how it is implemented, rather how you call it using **new** keyword.
So the convention for writing constructor functions is in **PascalCase** not in **camelCase**.

You can call a regular function using new keyword, it will still work.Just the extra work of initializing an empty object with a standard var name **this** and return statement done by JS will be of no use.

You can do opposite of this.Calling a constructor function without **new** keyword and will get undefined in return which is default return value by JS if anything is not returned explicitly.

Function Execution Types

There are 4 ways to call a function

1. Directly call the function object.
2. Call the function as a property of another object.
3. Using **new** keyword to call the function in **constructor mode**.
4. Fourth one is a bit complex using **call** to change the reference **this** from calling object to some other custom object.

See the code examples for 1, 2 & 3 and number 4 is what we are gonna discuss.

```
function foo() {
    console.log("This is function by declaration");
}
var foo = function() {
    this.type = "Expression";
    console.log("This is function by Expression");
}
foo(); // #Method 1 : calling/executing the function object 
directly.

var temp = {
    "foo": function() {
        console.log("This function object is a property of 
object temp");
    }
}
temp.foo() // #Method 2 : calling/executing the function object 
as a property of another object.

var dummy = new foo(); // Method 3: calling using new in 
constructor mode
```

Concept building for 4

There are 2 default arguments to each JS function one is called **arguments** and other one is **this**.
At execution time a function might refer to some data or reference to other functions/objects which are usually provided by the object that calls the function.The calling object is referred by **this** keyword.
So a function needs some sort of context to run within which it operates.

JS has the concept of global object and global object depends on the runtime environment.If running inside browser the global object is **window**, incase of node runtime its an object called **global**.

So, Incase of **#method 1** in above example a this reference gets created in the function **foo()** which refers to the global object since it is called directly.

Incase of **#method 2** in above example a this reference gets created inside the function property **foo** which refers to the calling object **temp.**

So, Incase of **#method 3** in above example a this reference gets created and an empty object gets created and this points to that empty objects which gets returned ate the end, when a method gets called in constructor mode.

So, Incase of **#method 4** we use **call** for function execution.
Lets see another example:

```
function Car() {
    this.velocity = 10;
    this.accelerate = function() {
        this.velocity = this.velocity + 6; // this refers to the 
calling object.
        //this.property refers to the calling objects property
        console.log("Velocity after acc:" + this.velocity);
    }
}
function Runner() {
    this.runningSpeed = 10;
}

var toyotaCar = new Car();
var athlete = new Runner();
athlete.accelerate = toyotaCar.accelerate;
console.log(athlete.accelerate());
console.log(toyotaCar);
```

Running this code will provide following result:

```
Velocity after acc:NaN
undefined // Remember every function in JS has an automatic return value, undefined
Car { velocity: 10, accelerate: [Function (anonymous)] }
```

So , when we called the accelerate() function on athlete object then **this** points to the calling object.But the athlete object does not have any property called velocity which is being accessed inside the method accelerate().
As a result value is NaN when we try to access velocity inside accelerate() with athlete object's reference.

To solve this we can do following:

```
//solve using call function
console.log(athlete.accelerate.call(toyotaCar));
//changing the reference object to toyotaCar object using call()
console.log(toyotaCar);
```

Now it gives following result:
Velocity after acc:16
undefined
Car { velocity: 16, accelerate: [Function (anonymous)] }

Intro to Prototype:

Unlike other programming languages JS does not have the concept of class.Class acts as a blue print for object creation.JS ECMAScript-5 introduced concept of class but not like other object oriented languages.So in JS objects can be created by simulating the action of classes using functions.Every time we call functions in constructor mode using **new** it basically creates a new function object.

Classes
Classes are a template for creating objects. They encapsulate data with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are not shared with ES5 class-like semantics.

Defining classes
Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.

Hoisting
An important difference between function declarations and class declarations is that function declarations are hoisted and class declarations are not. You first need to declare your class and then access it, otherwise code like the following will throw a ReferenceError:

Prototype
There are two objects created every time we declare a function.One is that function object and another one is the **prototype** object.
JS interpreter creates a property to the actual function object for accessing the prototype object.

Property name is **prototype**.
It only comes to seen if we create a new instance of that function object using **new** keyword.Otherwise it just sits there in the function object as an empty object.
Newly created object will have a **__proto__** property which will point to the prototype object. **__proto__** property is also called **dunder-porto**.

Since **ECMAScript 2015**, the prototype is accessed using the accessors Object.getPrototypeOf() and Object.setPrototypeOf(). This is equivalent to the JavaScript property **__proto__** which is non-standard but de-facto implemented by many browsers.
It should not be confused with the functions prototype property of function objects which is accessed by the key prototype.

Prototype object will remain same for all the instances just new properties will be added or removed to it.Below code will explain details.

```
function PrototypeExample() {
    this.type = "This object demonstrate the features of 
prototype";
}
PrototypeExample.prototype.type = " This custom key explains the 
presence prototype " +
    "object that gets created for every function 
declaration"; // Adding property to functions prototype object
console.log(PrototypeExample.prototype);
var protoTestObj1 = new PrototypeExample();
var protoTestObj2 = new PrototypeExample();
Object.getPrototypeOf(protoTestObj1).foo = "Adding foo to the 
prototype object";
console.log(Object.getPrototypeOf(protoTestObj1)); // accessing 
prototype object of an instance; __proto__ is replaced 
by .getPrototypeOf() as of ES5
```

```
console.log(Object.getPrototypeOf(protoTestObj2).foo); // 
property foo of prototype object is automatically accessible by 
different object.
console.log(Object.getPrototypeOf(protoTestObj2) === 
Object.getPrototypeOf(protoTestObj1)); // Proof that there is a 
single instance of Prototype object which is attached 
// to every new object and gets overwritten every time we add/
remove/manipulate.
```

Property lookup with prototypes

If a property doesn't exist in an object then JS interpreter looks into the prototype object if we try to access that property.If the property found in the prototype object then that value gets returned.

From the above example below code explains the fact:

```
console.log(protoTestObj2.foo); //property lookup
```

Since the prototype object has foo property so that value gets returned by prototype object.

Object behaviors using prototypes
Some behaviors of class based object oriented programming can be imitated using **prototyping** in JS.

For example you can declare a common function in the prototype object then automatically every instance created with **new operator** of that function will be able to call that function automatically. If any object needs a different implementation, it also can be done, thus the concept of overwritten functions/dynamic/runtime polymorphism can be achieved.

Difference between class and prototype implementation is, you can add properties to the prototype object at runtime, every object will automatically be able to access those properties where in class you can not add attribute at runtime.

Object Links With Prototypes

As from the previous paragraphs we learned that a function object has a key called prototype to access its prototype object.An instance of that function, created with new keyword will also have a property called __proto__ (**dunder proto**) to access it's copy of that original prototype object.

Interestingly each prototype has a key called **constructor** to traceback to the function object for which the prototype object was created in the first place. Following code example shows the case:

```
//** Object Links With Prototypes **//
var prototypeObj = PrototypeExample.prototype;
console.log(prototypeObj.constructor);
console.log(Object.getPrototypeOf(protoTestObj1).constructor);
```

** note: these are all references which can be changed at runtime by the developer or js engine.

The Object function:

Like global object concept (called **window** in JS runtime, **global** in Node runtime) JS also has some global functions.One of the global function is called **Object**.

If no explicit constructor is used for creating an object(inline object) the default Object function is used to create that object.

The same rule applies for in creation of prototype object.We don't create prototype object explicitly as JS does that for us, it uses the Object function as constructor for creating the prototype object.Thats why the prototype object's constructor property will point to the Object function.

In objects property lookup we have seen how JS looks into prototype object for properties accessed but not present in the object.

There is an another step to this hierarchy which the **Object** functions prototype object.