Javascript

BY Sumit Kohli

Advanced JavaScript

So far we have learned JavaScript that was used to do basic validations, DOM manipulations on for web pages However JavaScript is much more deep than this. JavaScript today is object based and is used to create big platforms and applications.

Unlike other programming languages, Javascript is not fully object oriented. It is object based language.

We can create classes as we do in Java or C++, but the same is not possible in JavaScript However with the clever use of functions and objects, we imitate the object oriented behavior

Before we dive into object oriented JavaScript, Lets understand more about functions and few other concepts in JavaScript

Parsing data

When we take input from forms, the data format is always string. This creates problem if we want to do some mathematical operations like adding/multiplication In order to solve this issue we use parseInt() function. This function takes any type of input (mostly string) and tries to make an integer out of it.

If it fails, it returns NaN:

parseFloat()

parseFloat() is similar to parseInt(), but it also looks for decimals to figure out a number from input.

Ex:

```
>parseFloat('12');
  Returns 12

>parseFloat('1.23');
 Returns 1.23
```

isNaN()

Using isNaN() functio nwe can check if an input value is a valid number.This is useful to find if the number can be used in arithmetic operations

eval()

eval() function takes a string input and executes it as a JavaScript code:
Ex:

```
> eval('var i = 2;alert(i);');
```

This alerts 2 on the screen.

Note that It's slower to evaluate live code using eval than to have the code directly in the script. Also its dangerous because it directly executes any code provided to it.

About alert() function

Note that this function blocks the browser thread, meaning
that no other code will be executed until the alert is closed. It is because of this reason, its not preferred to use this function.

Variable hoisting

In JavaScript variable declarations no matter wherever they occur, are processed before any code is executed.

Because variable declarations are processed before any code is executed, declaring a variable anywhere is same as declaring it at the top. This also means that a variable can appear before it's declared. This behavior is called hoisting.

```
var i = 212
var i;
```

is same as

```
var i;
var i = 212
```

What is function in Javascript

Functions are given much more power in javascript. A function in JavaScript is a special variable. A function is actually an object with special properties.
Since functions in JavaScript are variables, we can treat them like normal variables. We can pass them to other functions, we can overwrite them return and many more.
Lets explore these properties in details

More about Function

As discussed,functions are actually objects.
There is a built-in constructor function called Function() that can create a function.

Below are different ways to define a function:
> function sum(a, b) { // function declaration
return a + b;
}

var sum = new Function('a', 'b', 'return a + b;');
> sum(1, 2)
 source code evaluation suffers from the same drawbacks as the eval() function, so defining functions using the Function() constructor should be avoided

Anonymous functions

```
var f = function (a) {
return a;
};
```

The above notation is often called an anonymous function because it doesn't have a name.
Anonymous function is used mostly for callbacks where we do a async task and we need to execute the function to save memory.
Ex: setTimeout

```
setTimeout(function(){ console.log(“Task done”); }, 2000);
```

Passing function to function

function is just like any other data assigned to a variable. Here we are passing function to other function. Have a look at the code

```
function invokeSum(a, b) {
return a() + b();
}
function returnOne() {
  return 1;
}
function returnTwo() {
  return 2;
}
alert(invokeSum(returnOne, returnTwo))
```

function code within the call

We can pass function as shown below too. Here we are passing the entire function code within the call

```
function invokeSum(a, b) {
  return a() + b();
}
 invokeSum(
      function () { return 5+4; },
      function () { return 7+9; }
);
```

Immediate function

Immediate function is another application of anonymous function.
Immediate function means calling a function immediately after it's defined
We done need to invoke it. This type of function invokes automatically

```
(
function () {
  alert('Hey u called me without invoking.');
}
)();
```

Passing parameter to immediate function

We can pass parameters to immediate function also. Have a look at the code below

```
(
function (name,location) {
alert('Hello  name is ' + name + ' and location is '+location);
}
)('smith','Bangalore');
```

OR

```
(function () {
// ...
}());
```

Passing parameter to immediate function

We can pass parameters to immediate function also. Have a look at the code below

```
(
function (name,location) {
alert('Hello  name is ' + name + ' and location is '+location);
}
)('smith','Bangalore');
```

OR

```
(function () {
// ...
}());
```

Alternative syntax for immediate function

For convenience JavaScript lets us declare the immediate function as shown below

```
(
function (name,location) {
alert('Hello  name is ' + name + ' and location is '+location);
}
)('smith','Bangalore');
```

OR

```
(
function (name,location) {
alert('Hello  name is ' + name + ' and location is '+location);
}('smith','Bangalore')
);
```

Passing parameter to immediate function

We can pass parameters to immediate function also. Have a look at the code below

```
(
function (name,location) {
alert('Hello  name is ' + name + ' and location is '+location);
}
)('smith','Bangalore');
```

OR

```
(function () {
// ...
}());
```

Immediate function advantages

Application of immediate anonymous functions is when we want to have some task done without creating extra global variables.This saves memory.
Another practical use is when trying to use jquery with conflicts. We write jquery code with the $ specifier. However there are times when the $ variable is used by other JavaScript libraries. In that case we can use the jQuery global variable instead of $. But this is quite inconvenient because the developer wrote the code using $ global variable. Means we have to now change $ to JQuery every where in the code. Lets see the solution using immediate function in the next slide.

Immediate function advantages

TO solve the problem previously discussed, We can wrap the entire code in an immediate function and pass the jQuery as parameter to the anonymous function. Below is the example. Note that within the anonymous function we can still use the $ notation because we have passed it to the anonymous function.

```
(
Function ($) {
//write the code using $ notation
}
)(jQuery)
```

Immediate function advantages

Application of immediate anonymous functions is when we want to have some task done without creating extra global variables. This saves memory.
Another practical use is when trying to use jquery with conflicts. We write jquery code with the $ specifier. However there are times when the $ variable is used by other JavaScript libraries. In that case we can use the jQuery global variable instead of $. But this is quite inconvenient because the developer wrote the code using $ global variable. Means we have to now change $ to JQuery every where in the code. Lets see the solution using immediate function in the next slide.

Inner function

We can have functions inside function. Why is so? Pretty simple. Remember we discussed that functions are variables in JavaScript If we can have variables inside a function then why not functions. They just behave like local variables inside a function. They even have their scope that is limited to the function. Means these functions cant be accessed outside the function.
See the code below

```
function outerFunc(param) {
	function innerFunc(input) {
		return input * 4;
	}
	return 'The result is ' + innerFunc(param);
}
outerFunc(12);
```

Inner Anonymous function

Have a look at the example below. We have written an anonymous function inside. Since the innerFunc() is local, it's not accessible outside outerFunc(), we call such a function as private function.

```
var outerFunc = function (param) {
var innerFunc = function (input) {
    return theinput * 4;
};
    return 'result is ' + innerFunc(param);
};
```

Functions returning functions

Since function is a variable, it can also be returned. Have a look at the example below

```
function test() {
	alert('inside test');
	return function () {
	alert('inside returned function');
	};
}
var newFunc = test();
> newFunc();
```

Test this function in firebug or chrome console to see the output

Function can rewrite rewrite itself!

use the new function to replace the old one.

```
test = test();
```

Callback functions

function is just like any other data assigned to a variable, it can be defined,
copied, and also passed as an argument to other functions.

```
function invokeMsg(a, b) {
  return a() + b();
}
function returnHi() {
  return 'Hi';
}
function returnHello() {
  return 'Hello';
}
> invokeMsg(returnHi, returnHello);
```

Closures

A closure is a special kind of variable that combines a function, and the environment in which that function was created. The environment consists of all local variables that were in-scope at the time that the closure was created. Lets understand it in detail. Note that though innerFunc is a private function if we return it we can access it from outside the outterFunc. Along with it, this returned innerFunc comes up with the environment of the function that created it. Thats why it has access to the private variables and variable param.

```
function outerFunc(param){
        var privateVar = 5;
  var innerFunc = function (input){
     alert('param passed to function  is '+ param + 'and private variable is '+privateVar );
    var sum = privateVar+input+param;
     alert('addition of private var and input and param is '+sum);
  }
  return innerFunc;
}
var closureFunc = outerFunc(12);
closureFunc(4);
```

Creating different functions using closure

As discussed closure is a special kind of variable that contains the environment in which that function was created. Lets now create a ticket generator function that generates different functions and returns as closures.

```
function TicketGenerator(transportMode){
  var generateTicket = function(destination){
    return transportMode+"--"+destination+"123";
  }
  return generateTicket;
}
//creating two different functions
var flightTicketGenerator = TicketGenerator("Flight");
var shipTicketGenerator = TicketGenerator("Ship");
alert(flightTicketGenerator("Chennai"));
alert(shipTicketGenerator("singapore"));
```

Modifying the ticket generator function

Lets see the modified version of ticket generator that has 2 variables within it namely transportMode and class(business class, executive class etc)

```
function TicketGenerator(transportMode,classType){
  var generateTicket = function(destination1,destination2){
    return classType+transportMode+"--"+destination1+"123"+destination2;
  }
  return generateTicket;
}

var flightTicketGenerator = TicketGenerator("Flight","Business");
var shipTicketGenerator = TicketGenerator("Ship","Executive");
alert(flightTicketGenerator("Chennai","Mysore"));
alert(shipTicketGenerator("singapore","USA"));
```

Objects in JavaScript

To Start with , lets know how to create objects.
In JavaScript objects can be created on the fly. We just need to instantiate the class Object which is the parent class for object.Below is the example.

```
var ob = new Object();
```

Note that we can also create objects using the curly braces notation.
So we can also write like this

```
  var ob = {};
```

Now that we have created the object we can add properties on the fly!Lets add the name and age property to the object we have created.

```
ob.name = “Smith”;
ob.age = 29;
```

Note that we use the dot operator for accessing and adding properties to objects.
We can also add functions to the object too.

```
ob.sayName = function(){
  alert(“The name is “+this.name);
}
```

Note that we can use the this property to access the properties within the function

Accessing an object's properties

We can access an Object's properties using the two notations. We can either access using the bracket or using the dot notation.
Ex ob.name and ob[“name”] both will give the value of the name property of the object.

Object Literals

Object literal notation is creating object using the curly braces notation. We can have key value pairs for the properties or functions.
Below is the example code.

```
var ob = {
name: “Smith”,
age: 29,
sayName:function(){
                    alert(“The name is “+this.name);
                    }
}
```

Note that we can modify the existing properties and add new properties on the fly
Now that we have created the object we can add properties on the fly!Lets add the name and age property to the object we have created. Lets modify the property age to 32. See the example below

```
  ob.age = 32;
```

Lets now add a property location to the object. Simply use the dot notation to set the property.

```
  ob.location = “Bangalore”;
```

Creating object using Constructor functions

We have created object earlier. It serves the purpose of creating objects on the fly. But how about a template or a class type technique to create objects. We might want to create similar objects with different instance properties like we do in other programming languages.
Yes, we can do this using a constructor function.
So what is a constructor function. Nothing different. Its just a simple function. But we can create objects with functions also. Lets see how we can do this below
Below is the example code.

```
function Hero(name,planet){
	this.name = name;
	this.planet = planet;
	this.sayName = function(){
		alert(this.name);
	}
}

//creating object of type Hero
var obOne = new Hero("SpiderMan","Earth");
var obTwo = new Hero("IronMan","Earth");
//calling the sayName function for obOne object
obOne.sayName();
//calling the sayName function for obTwo object
obTwo.sayName();
```

Note that we can modify the existing properties and add new properties on the fly
Now that we have created the object we can add properties on the fly!Lets add the name and age property to the object we have created. Lets modify the property age to 32. See the example below

```
ob.age = 32;
```

Lets now add a property location to the object. Simply use the dot notation to set the property.

```
ob.location = “Bangalore”;
```

Public and private properties

We can have public and private properties.
Below is the example code.

```
function Hero(name,planet){
        this.name = name;
        this.planet = planet;
        this.sayName = function(){
                alert(this.name);
        }
}

//creating object of type Hero
var obOne = new Hero("SpiderMan","Earth");
var obTwo = new Hero("IronMan","Earth");
//calling the sayName function for obOne object
obOne.sayName();
//calling the sayName function for obTwo object
obTwo.sayName();
```

Note that we can modify the existing properties and add new properties on the fly
Now that we have created the object we can add properties on the fly!Lets add the name and age property to the object we have created. Lets modify the property age to 32. See the example below

```
ob.age = 32;
```

Lets now add a property location to the object. Simply use the dot notation to set the property.

```
ob.location = “Bangalore”;
```

Public and private properties

We can have public and private properties.
Below is the example code.

```
function Hero(name,planet){
        this.name = name;
        this.planet = planet;
        this.sayName = function(){
                alert(this.name);
        }
}

//creating object of type Hero
var obOne = new Hero("SpiderMan","Earth");
var obTwo = new Hero("IronMan","Earth");
//calling the sayName function for obOne object
obOne.sayName();
//calling the sayName function for obTwo object
obTwo.sayName();
```

Note that we can modify the existing properties and add new properties on the fly
Now that we have created the object we can add properties on the fly!Lets add the name and age property to the object we have created. Lets modify the property age to 32. See the example below

```
ob.age = 32;
```

Lets now add a property location to the object. Simply use the dot notation to set the property.

```
ob.location = “Bangalore”;
```

Accessing an object's properties

- Using the square bracket notation, for example hero['occupation']
- Using the dot notation, for example hero.occupation

constructor property

Every object has a constructor property.
The constructor property contains a reference to the function that created the Object
Say we have our Hero constructor function we created earlier.

```
var obOne = new Hero(“spiderman”, “earth”);
var constructorFunc = obOne.constructor;
```

Now we can create objects using this returned function.

```
var obTwo = new constructorFunc(“ironman”,”earth”);
```

prototype

The prototype is the property of a function object and it may point to another object

We can implement inheritance using the prototype property.

All objects created with the function keep a reference to the prototype object. All these created objects also have access to the prototype object's property.

When we create a function we can verify that it automatically has a prototype property that points to a new object.

```
> function location() {}
> typeof location.prototype;
"object"
```

In the coming slides we will understand a lot more about the prototype property and its uses.

Enumerable property

Properties of objects that show up are called enumerable.
Example, length of
arrays and constructor properties don't show up. we can check which ones are enumerable
using propertyIsEnumerable() method that every object Provides.

We can also check if a property is an object's own property
or its prototype's inherited property using the hasOwnProperty() method.
Note the prototype's properties are the properties that are inherited from the prototype object.

Javascript objects also have the isPrototypeOf() method. This method tells whether a specific object is used as a prototype of another object.

__proto__ property

Note that prototype is a property of the function. Objects dont have prototype property. If objects dont have prototype property, then how the objects are able to look access their prototype object's properties. The answer is using the __proto__ property that points to the prototype object.The __proto__ allows methods and properties of the prototype object to be used as if they belonged to the newly-created object.
Consider a object called Bike and use it as a prototype when creating objects with the Car() constructor.

```
var Bike = {
        TyresNo:2,
}
> function Motorbike() {}
> Motorbike.prototype = Bike;
Now, let's create a hondabike object and give it some properties.
> var hondabike = new Motorbike();
> hondabike.fuel = 'petrol';
> hondabike.topspeed = 250;

> hondabike.__proto__  //gives  Bike
```

__proto__ is
a property of the instances (objects), whereas prototype is a property of the constructor functions used to create those objects.

What is prototype chain?

We know aht when a function is invoked with new operator, an object is created and this new object links to the prototype object using __proto __ link.
The prototype object to which current object points may also links to another prototype object and this chain continues. This chain of prototype objects is called **Prototype chain.**

SO where does this chain finally end?
The answer is with the Object.prototype
Object which is the grandparent of every object. Every object inherits from it.
Lets say object we have object A points proto points to object B and object B points to C.
If object A doesn't have a property but B has it, A can still access this property as its own using prototype chain. Same applies if B also doesn't have the required property, but C has it. This is how inheritance takes place in javascript: an object can access any property found somewhere down the inheritance chain.

Lets test these lines in console.

For the Motorbike function example type the following

```
> typeof hondabike.__proto__;
"object"
> typeof hondabike.prototype;
"undefined"
> typeof hondabike.constructor.prototype;
"object"
```

Inheritance for objects

We can implement inheritance for object literals first.
Lets see the example code below

```
var parentOb = {
	parentProperty:"Parent Object Literal",
	sayHi: function(){
		console.log("hi");
	}
}
//we create objecct using Object.create(). The childOb now has all properties of parentOb.
var childOb = Object.create(parentOb);

//we are adding new property for the child object.
childOb.name = "child object";
childOb.sayHello = function(){
	console.log("Hello");
}
```

Inheritance in Javascript

Using prototype we can implement inheritance in JavaScript Implementing inheritance using prototype property is called prototype based inheritance.
Lets now implement inheritance using prototype.
For our example we will have a base function Human.
Lets first create the function Organism which is our base class. We create object of Organism class and extend it for the function Human which is the child class. Below is the code.

```
function Organism(planet){
        this.planet = "Earth";
}
Organism.prototype.sayPlanet = function(){
        alert("the planet is"+this.planet);
}
var organismOb = new Organism("Earth");

function Human(){
}
Human.prototype = organismOb;
Human.prototype.breathes = "air";
Human.prototype.constructor = Human;


Human.prototype.breathesWhat = function(){
        alert("Humans breathe "+this.breathes);
}
var humanOb = new Human();
```

Resetting the constructor

Using prototype we can implement inheritance in JavaScript but its important to note that when we use prototype, the constructor property of the function points to the constructor property of the prototype. In order to reset the constructor property we again point the constructor value again to the child function. Try running this code and see the output. you will see that It still points to the Organism function.

```
function Organism(planet){
        this.planet = "Earth";
}
Organism.prototype.sayPlanet = function(){
        alert("the planet is"+this.planet);
}
var organismOb = new Organism("Earth");

function Human(){
}
Human.prototype = organismOb;
Human.prototype.breathes = "air";
console.log(Human.constructor);
```

Chain inheritance in JavaScript

Lets us see the example using chain inheritance.

```
function Organism(planet){
        this.planet = "Earth";
}
Organism.prototype.sayPlanet = function(){
        alert("the planet is"+this.planet);
}
var organismOb = new Organism("Earth");

function Human(){
}
Human.prototype = organismOb;
Human.prototype.breathes = "air";
Human.prototype.constructor = Human;


Human.prototype.breathesWhat = function(){
        alert("Humans breathe "+this.breathes);
}
var humanOb = new Human();
function Programmer(name,sleepHours){
        this.name = name;
        this.sleepHours = sleepHours;
}
Programmer.prototype =  humanOb;
Programmer.prototype.sayName = function(){
        alert(this.name);
}
Programmer.prototype.constructor = Programmer;
var programmerOb = new Programmer("smith",10);
console.log(programmerOb);
programmerOb.sayPlanet();
programmerOb.breathesWhat();
```

inheritance using temporary constructor

In the previous examples, we implemented inheritance by creating object of the parent class. This its a good approach but we have a more efficient way of implementing inheritance. There is a general rule that the instance level properties are kept inside the constructor functions. The rest properties which we want to inherit are moved to the prototype. This way we can separate the instance level properties form the prototype properties A better way is creating a temporary function, inheriting prototype to the parent class and extend the object of this temporary function to the child class prototype. This is the standard approach used by modern libraries. Lets see the working example.

```
function Organism(planet){
        this.planet = "Earth";
}
Organism.prototype.sayPlanet = function(){
        alert("the planet is"+this.planet);
}
var organismOb = new Organism("Earth");

function Human(){
}
var tempFunc = function(){};
tempFunc.prototype = Organism.prototype;
Human.prototype = new tempFunc();
Human.prototype.breathes = "air";
Human.prototype.constructor = Human;
```

Modify extend function

As we have observed, inheritance is all about reusing
Code. Then why not simply copy the properties we like from one object to another?

```
function extend(ChildConstructor, ParentConstructor) {
var p = Parent.prototype;
var c = Child.prototype;
for (vari in p) {
c[i] = p[i];
}
c.uber = p;
}
```

Unlike the previous example though, it's not
necessary to reset the Child.prototype.constructor because here the child prototype is augmented, not overwritten completely, so the constructor property points to the initial value.
 this is only true for
properties containing primitive types. All objects (including functions and arrays) are not duplicated, because these are passed by reference only.

Creating the inherit function

We can also create a inherit function that can inherit the parent properties for the child class.
Below is the example
Note that most of the libraries use this technique to implement inhertiance.

```
var  extend = function(childConstructor, parentConstructor) {
  function tempConstructor() {};
  tempConstructor.prototype = parentConstructor.prototype;
  childConstructor.superClass_ = parentConstructor.prototype;
  childConstructor.prototype = new tempConstructor();
  childConstructor.prototype.constructor = childConstructor;
};
```

Modify extend function

As we have observed, inheritance is all about reusing
Code. Then why not simply copy the properties we like from one object to another?

```
function extend(ChildConstructor, ParentConstructor) {
var p = ParentConstructor.prototype;
var c = ChildConstructor.prototype;
for (var i in p) {
        c[i] = p[i];
}
        c.superclass = p;
}
```

Now it's not necessary to reset the Child.prototype.constructor because here the child prototype is augmented, not overwritten fully. Thats why the constructor property points to the initial value.

Note that this is only true for properties containing primitive types. All objects are not duplicated, because these are passed by reference only.

Be Careful when copying by reference

```
 function Human() {}
function Developer() {}
Human.prototype.name = 'smith';
Human.prototype.hobbies = ["sleeping", "eating", "travelling"];
```

Now, let's have Developer inherit from Human. Lets use the modified extend function

```
extend(Developer, Human);
```

We can see that Developer function's prototype inherited the properties of Human prototype as its own.

```
Developer.prototype.hasOwnProperty('name');
true
Developer.prototype.hasOwnProperty('hobbies');
True
```

Since name property is primitive type so a new copy is created. The property hobbies is an array object so it's copied by reference!
Changing the Developer function's owns property,affects Human because both properties point to the same array in memory. In the code below we try to pop an array out of the developer prototype.But the same also effects the array of the Human prototype.

```
>Developer.prototype.hobbies.pop();
sleeping
>Human.prototype.hobbies;
["eating", "travelling"]
```

Lets copy objects using shallow copy

you can start by copying all of the properties of an existing object. Here's a
function that does exactly that: it takes an object and returns a new copy of it.

```
function extendCopy(p) {
var c = {};
for (vari in p) {
c[i] = p[i];
}
c.superClass = p;
return c;
}
```

Solution using Deep Copy

We can use the deep copy solution to solve this problem. Have a look at the code below.

```
function deepCopy(p, c) {
    c = c || {};
    for (var index in p) {
    if (p.hasOwnProperty(index)) {
        if (typeof p[index] === 'object') {
            c[index] = Array.isArray(p[index]) ? [] : {};
            deepCopy(p[index], c[index]);
        } else {
            c[index] = p[index];
        }
    }
}
return c;
}
```

Multiple inheritance in Javascript

As we have understood that inheritance is all about copying properties from parent to child prototype, then why not copy properties from multiple parents. This is how we can implement multiple inheritance in JavaScript Have a look at the code below.
Note that instead of passing parents we can retrieve the parameters using the arguments inbuilt array.

```
function multiInheritance() {
	var n = {}, stuff, j = 0, length = arguments.length;
	for (j = 0; j <length; j++) {
		stuff = arguments[j];
		for (var index in stuff) {
			if (stuff.hasOwnProperty(index)) {
				n[index] = stuff[index];
			}
		}
	}
	return n;
}
```

Parasitic inheritance

This Pattern as suggested by Douglas Crockford, is called parasitic inheritance. It's about a
function that creates objects by taking all of the functionality from another object
into a new one, augmenting the new object, and returning it.
Parasitic inheritance is different from prototypal inheritance which we have discussed so far.
Prototypal inheritance is used more often because its more efficient, But there are scenarios where we can go for Parasitic inheritance Lets understand more about parasitic inheritance in the coming section.

Parasitic inheritance

Let us see the inheritance example using prototype inheritance. Note that using prototype inheritance we have a private variable which is common in the two objects ob1 and ob2. Also you can see that any changes made to the myObject is ob1 is reflected in ob2!! because objects are referred by reference in JavaScript

```
function ParentConstructor() {
    var privateVariable = false;

    this.flipPrivateVariable = function() {
        privateVariable = !privateVariable;
    };

    this.showPrivateVariable = function() {
        alert( privateVariable);
    };
}

ParentConstructor.prototype.myObject = {count:1};

function ChildConstructor(){
}
ChildConstructor.prototype = new ParentConstructor();
var ob1 = new ChildConstructor();
var ob2 = new ChildConstructor();
ob1.flipPrivateVariable();
ob1.myObject.count++;
console.log(ob2.myObject.count);
ob2.showPrivateVariable();
ob1.flipPrivateVariable();
ob2.showPrivateVariable();
```

Parasitic inheritance

Let us see the inheritance example using parasitic inhertance.Note that flipping private variable of one object doesnt effect other private variable. Also the object changed for one doesnt change for other.

```
function ParentConstructor() {
    var parent = new Object();
    var privateVariable = false;
    parent.flipPrivateVariable = function() {
        privateVariable = !privateVariable;
    };
    parent.showPrivateVariable = function() {
        alert( privateVariable);
    };
    parent.myObject = {count:1};
    parent.name = 'ParentConstructor';
    return parent;
}
function ChildConstructor() {
    var child = new ParentConstructor();
    child.name = 'ChildConstructor';
    return child;
}
var ob1 = new ChildConstructor();
var ob2 = new ChildConstructor();

ob1.flipPrivateVariable();
ob2.showPrivateVariable(); //The private variable is not changed for object 2
ob1.myObject.count++;
console.log(ob2.myObject.count);  //the count still remains 1 for object 2.
```

Parasitic inheritance vs prototype

Both approaches have their own benefits and shortcomings.

Parasitic approach stops you from using **instanceof**. Everything is just an object; you can’t detect its class taxonomy. That because we create an empty object and add properties to it.

In Parasitic inheritance all the properties are duplicated. It is this reason that parasitic inheritance can potentially be memory inefficient provided there are lot of large fields.
Again it depends on the scenario and depending upon the requirement we can decide which inheritance approach to use.