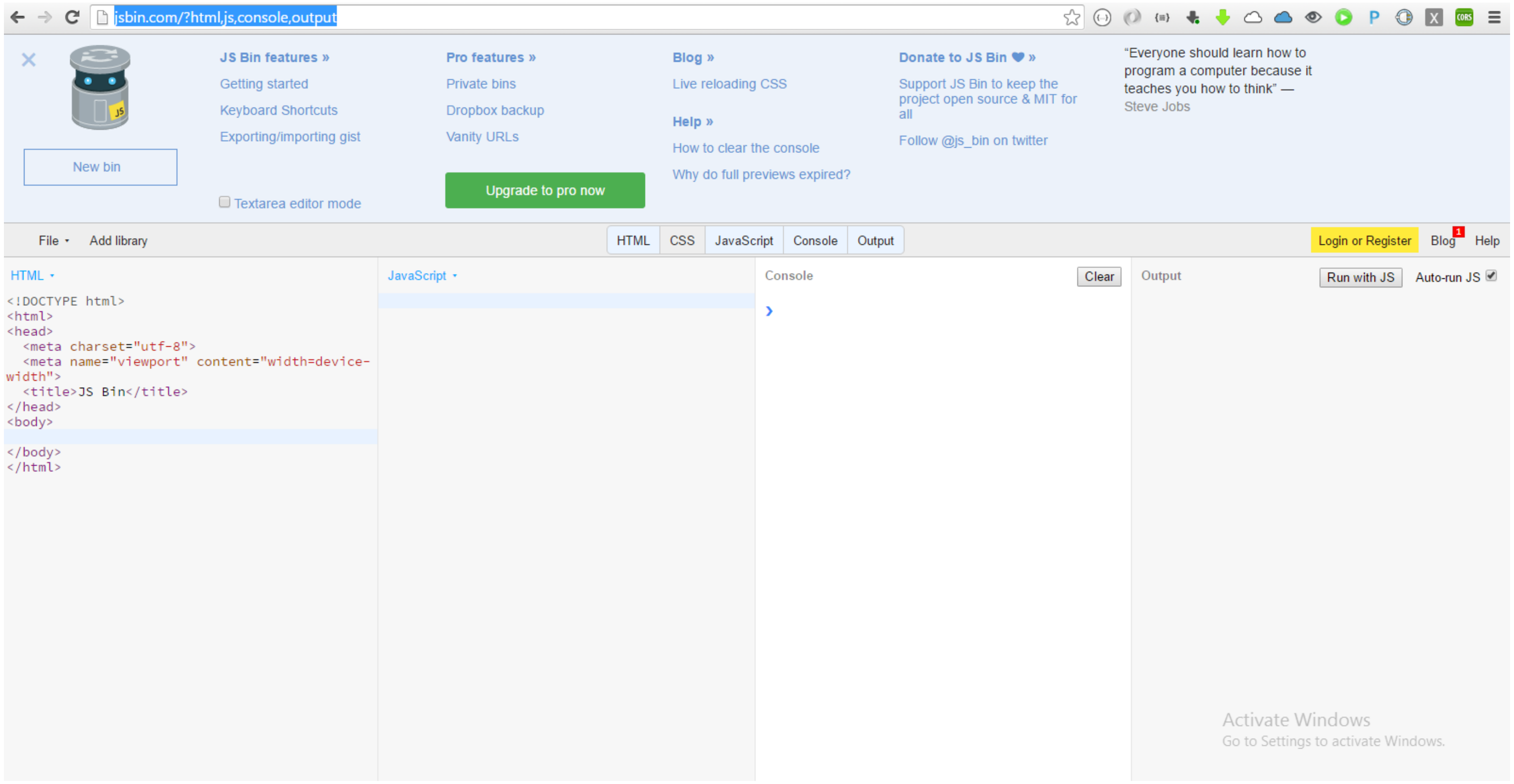
2. OOP IN JAVASCRIPT

Jsbin.com



jsbin.com/?html,js,console,output

JS Bin features »
Getting started
Keyboard Shortcuts
Exporting/importing gist

Pro features »
Private bins
Dropbox backup
Vanity URLs

Blog »
Live reloading CSS

Help »
How to clear the console
Why do full previews expired?

Donate to JS Bin ♥ »
Support JS Bin to keep the project open source & MIT for all
Follow @js_bin on twitter

"Everyone should learn how to program a computer because it teaches you how to think" — Steve Jobs

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-
width">
  <title>JS Bin</title>
</head>
<body>

</body>
</html>
```

JavaScript

A cross-platform, object-oriented scripting language. Small and lightweight. It contains a standard library of objects such as Array, Date, Math and a core set of language elements.

- Client-side JavaScript
  - Extends the core language by supplying objects to control a browser and its DOM
- Server-Side JavaScript
  - Extends the core language by supplying objects relevant to running javascript on a server.

Built-in Data types in JavaScript

Not everything in JS is Object

Primitive /Value Types

- String
- Number
- Boolean
- Undefined
- Null

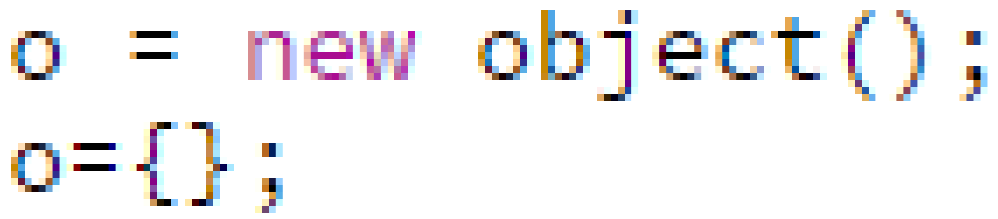
Objects/Reference Types

- Object
- Array
- Function
- Date
- RegExp

Objects in JS

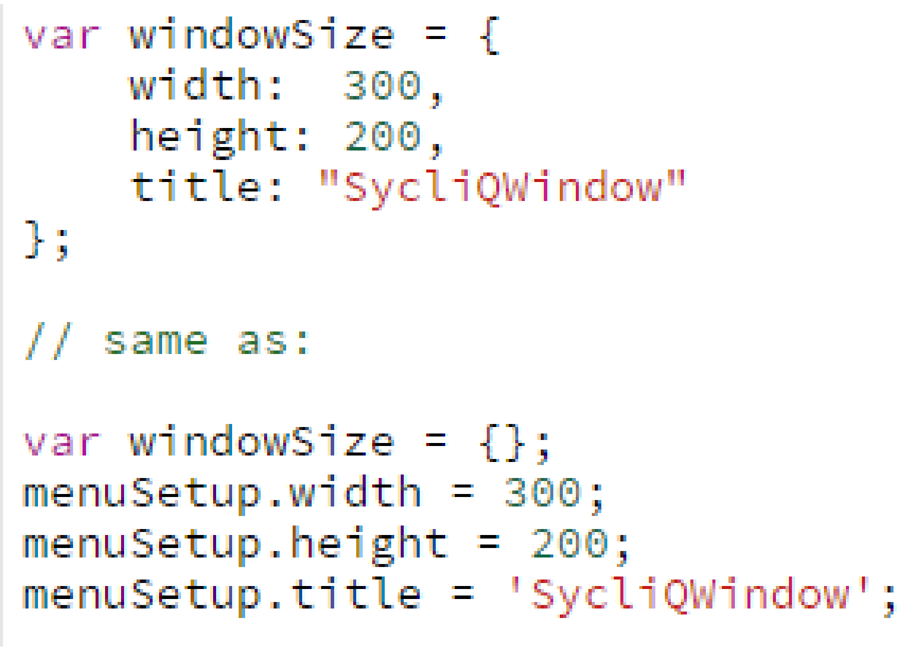
- Objects in JavaScript are kind of two-faced

1. An object is an associative array (called hash in some languages). It stores key-value pairs
2. On the other side, object are used for object-oriented programming.

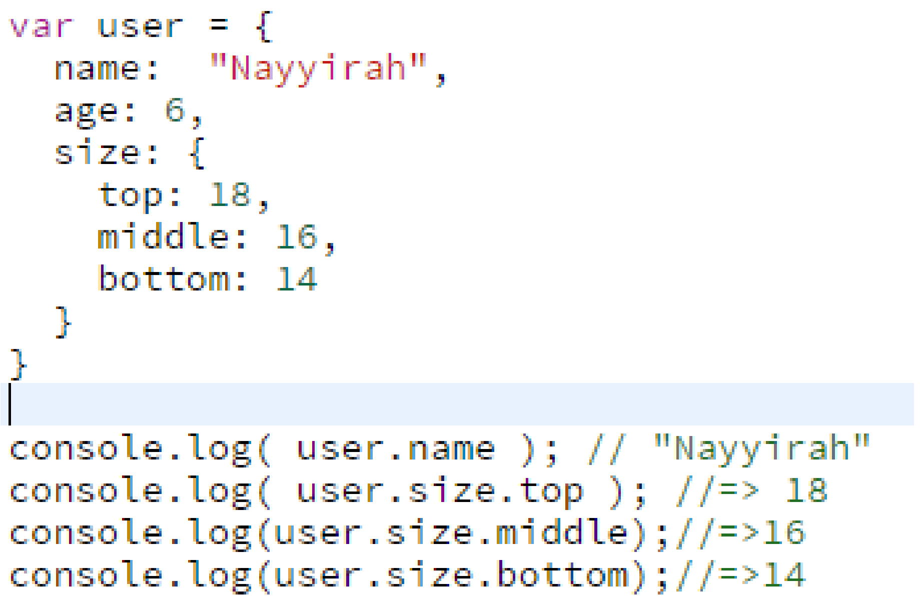


```
o = new object();
o={};
```

Objects

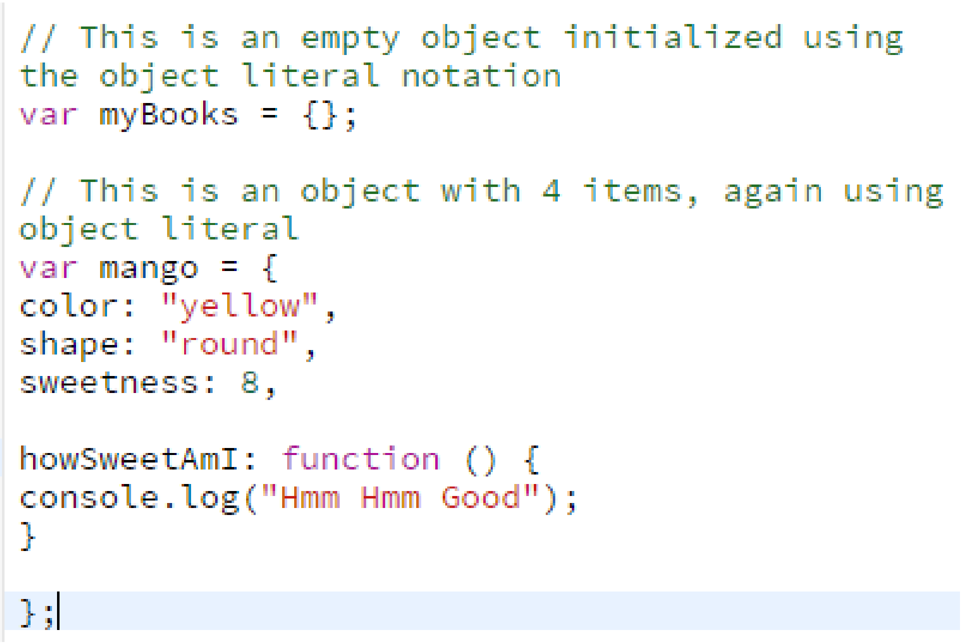


```
var windowSize = {
    width:  300,
    height: 200,
    title: "SycliQWindow"
};

// same as:

var windowSize = {};
menuSetup.width = 300;
menuSetup.height = 200;
menuSetup.title = 'SycliQWindow';
```



```
var user = {
  name:  "Nayyirah",
  age: 6,
  size: {
    top: 18,
    middle: 16,
    bottom: 14
  }
}

console.log( user.name ); // "Nayyirah"
console.log( user.size.top ); //=> 18
console.log(user.size.middle);//=>16
console.log(user.size.bottom);//=>14
```

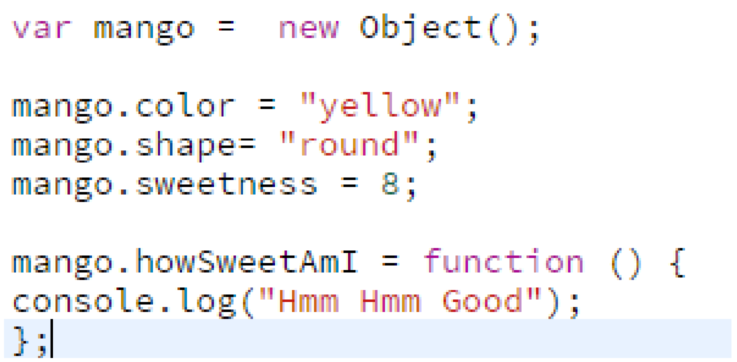
Object Literals



```
// This is an empty object initialized using
the object literal notation
var myBooks = {};

// This is an object with 4 items, again using
object literal
var mango = {
color: "yellow",
shape: "round",
sweetness: 8,

howSweetAmI: function () {
console.log("Hmm Hmm Good");
}

};
```

Object Constructor

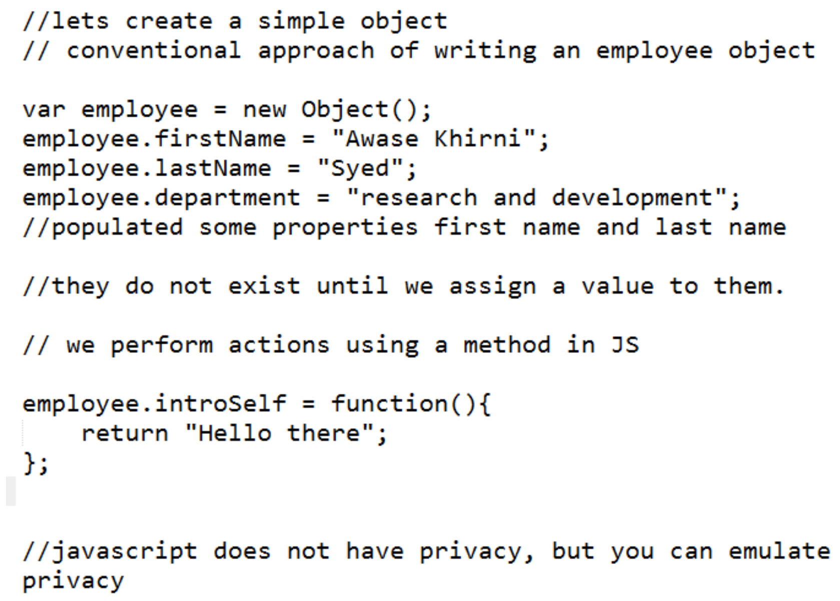
- A constructor is a function used for initializing new object and you use **new keyword** to call the constructor



```
var mango =  new Object();

mango.color = "yellow";
mango.shape= "round";
mango.sweetness = 8;

mango.howSweetAmI = function () {
console.log("Hmm Hmm Good");
};
```

Conventional approach

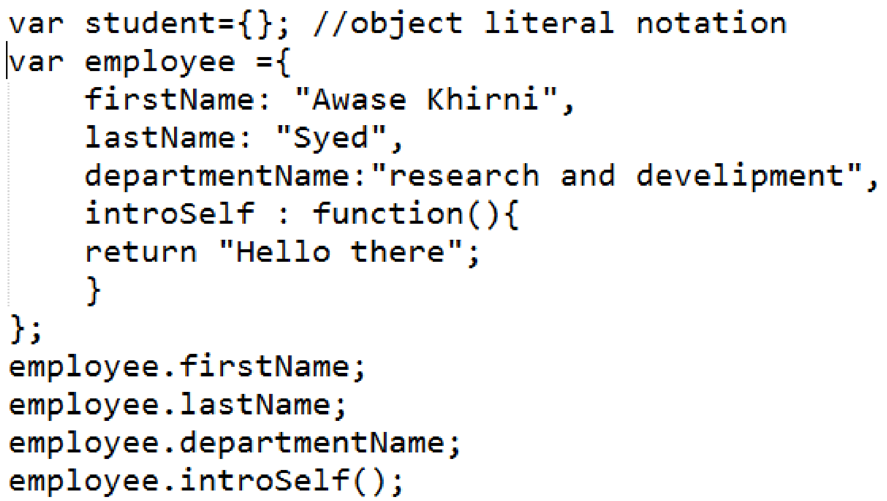
- Properties for an object do not exist until we assign a value to them
- We perform actions using a method in javascript
- Javascript does not have privacy , but we have a way to emulate privacy.



```
//lets create a simple object
// conventional approach of writing an employee object

var employee = new Object();
employee.firstName = "Awase Khirni";
employee.lastName = "Syed";
employee.department = "research and development";
//populated some properties first name and last name

//they do not exist until we assign a value to them.

// we perform actions using a method in JS

employee.introSelf = function(){
    return "Hello there";
};


//javascript does not have privacy, but you can emulate
privacy
```

Object Literal Notation

- Better Performance
  - Javascript can execute a single complex statement alot faster than multiple statements
- Better organization of code
- Easier to read and understand the code
- Less amount of code

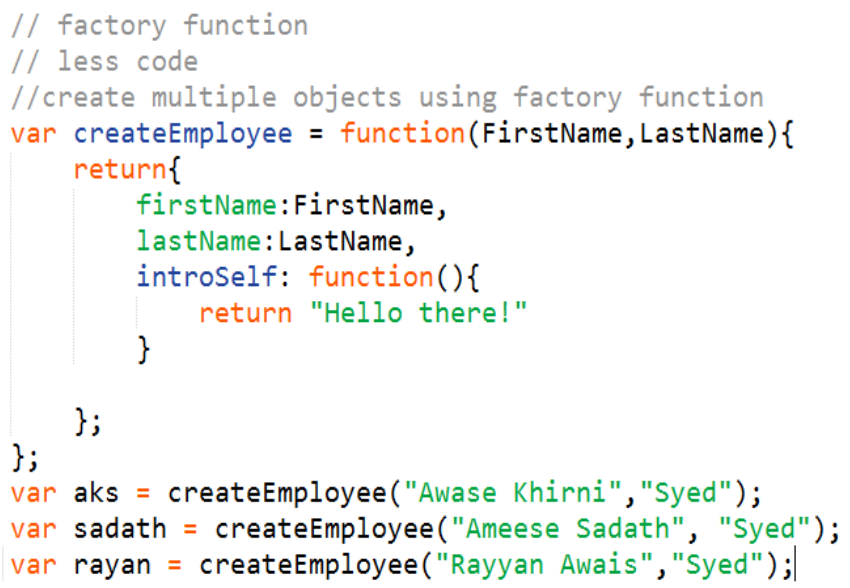


```
var student={}; //object literal notation
var employee ={
    firstName: "Awase Khirni",
    lastName: "Syed",
    departmentName:"research and develipment",
    introSelf : function(){
    return "Hello there";
    }
};
employee.firstName;
employee.lastName;
employee.departmentName;
employee.introSelf();
```

Factory constructor pattern

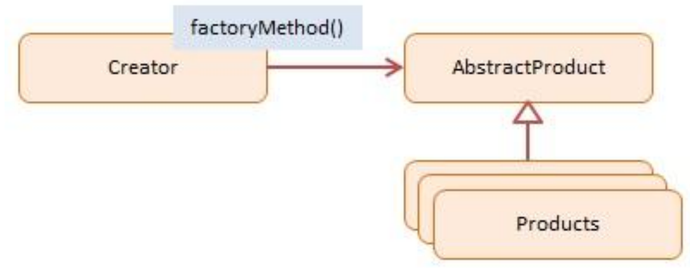
Factory Method

- This pattern is special because it does not use “new”
- The object is created by a simple function call.
- The constructor is defined as a function which return a new object
- The factory constructor uses a function which creates an object on its own without “new”.
- Inheritance is done by creating a parent object first and then modifying it
- Local methods and functions are private. The object must be stored in closure prior to returning if we want to access it’s public methods from local ones



```
// factory function
// less code
//create multiple objects using factory function
var createEmployee = function(FirstName,LastName){
    return{
        firstName:FirstName,
        lastName:LastName,
        introSelf: function(){
            return "Hello there!"
        }

    };
};
var aks = createEmployee("Awase Khirni","Syed");
var sadath = createEmployee("Ameese Sadath", "Syed");
var rayan = createEmployee("Rayyan Awais","Syed");
```

Factory Method



```
function Animal(name){
    return{

        run: function(){
    console.log(name+"is running!");
        },

    talk: function(){
            console.log(name+"is talking");
        },

    eat: function(){
            console.log(name+ "is eating");
        }
    };

}

//usage
var animal = Animal("tiger");
animal.run();
animal.talk();
```

Factory Method

- A factory method creates new objects as instructed by the client.
- One way to create objects in JavaScript is by invoking a constructor function with the new operator.
- **There a situations, however where the client does not , should not , know which one of several candidate objects to instantiate**
- The factory method allows the client to delegate object creation while still retaining control over which type to instantiate.

- Key objective is **extensibility.**
- **Frequently used in applications that manage, maintain or manipulate collections of objects that are different but at the same time have many characteristics (i.e. Methods and properties in common.)**

Factory pattern

- The factory pattern generally supplies an interface for developers to create new objects through the use of a factory rather than invoke the **new** operator on an object.
- Disadvantages
  - Unit testing can be difficult as a direct result of the object creation process being hidden by the factory methods.
- Advantages
  - Makes complex object creation easy through an interface that can bootstrap this process for you
  - Great for generating different objects based on the environment.
  - Practical for components that require similar instantiation or methods.
  - Great for decoupling components by bootstrapping the instantiation of a different object to carry out work for particular instances

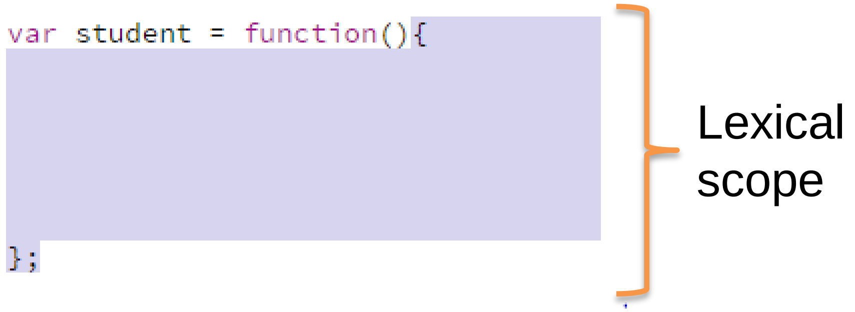
5. SCOPE

Scope

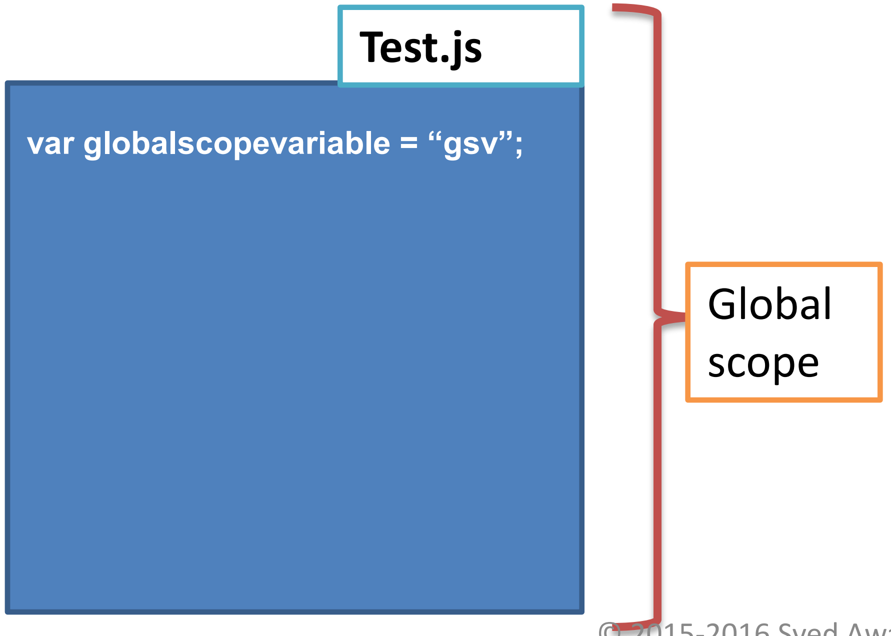
- Scope is the set of variable, objects and functions you have access to.
- Javascript has function scope: the scope changes inside functions.
- Variables declared within a javascript function, become **local** to the function
- Variables declared outside a function, becomes **global.**
- **A global variable has global scope and all scripts and functions have access to it**
- If you assign a value to a variable that has not been declared, it will automatically become a **global** variable.
- In javascript, all local variables and functions are properties of special internal object called **lexicalEnvironment (also called as global object)**

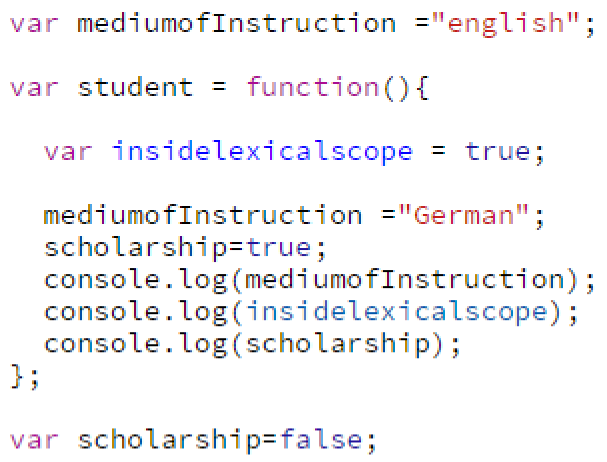
Lexical (Statically) Scope

- A new scope is created only when you create a **new function this is called lexical scope.**



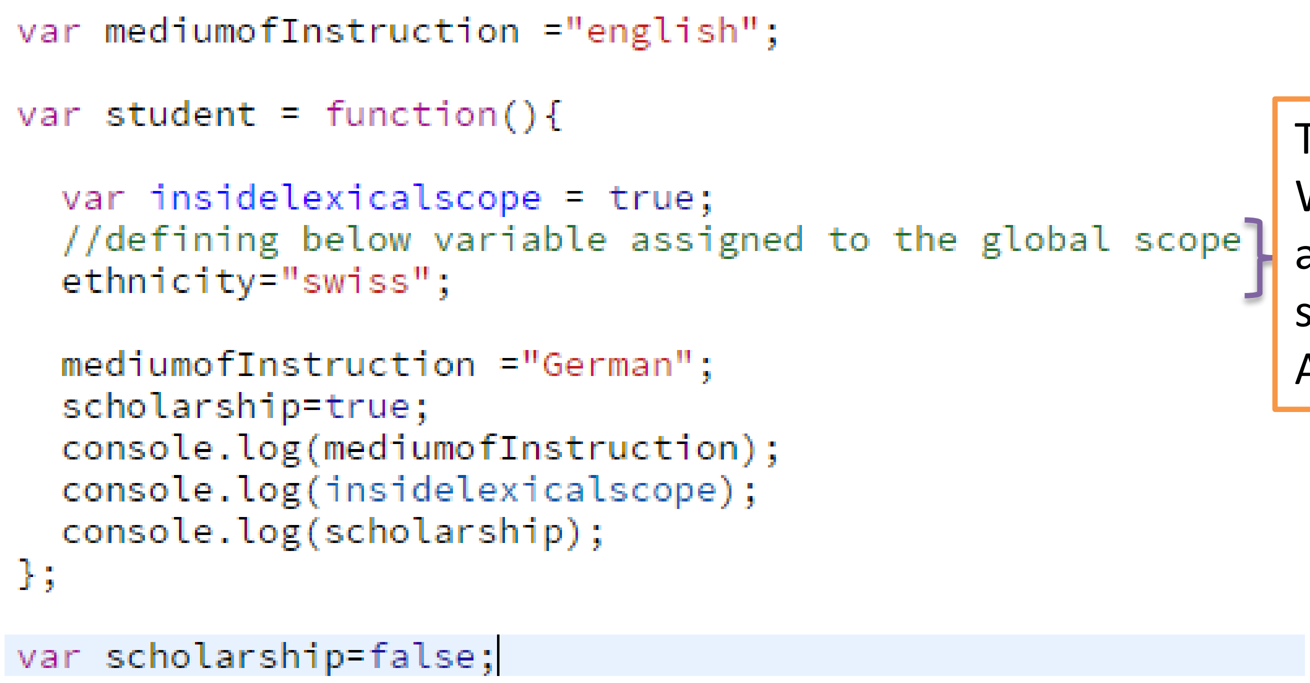
```
var student = function(){

};
```





```
var mediumofInstruction ="english";

var student = function(){

  var insidelexicalscope = true;

  mediumofInstruction ="German";
  scholarship=true;
  console.log(mediumofInstruction);
  console.log(insidelexicalscope);
  console.log(scholarship);
};

var scholarship=false;
```

Scoping

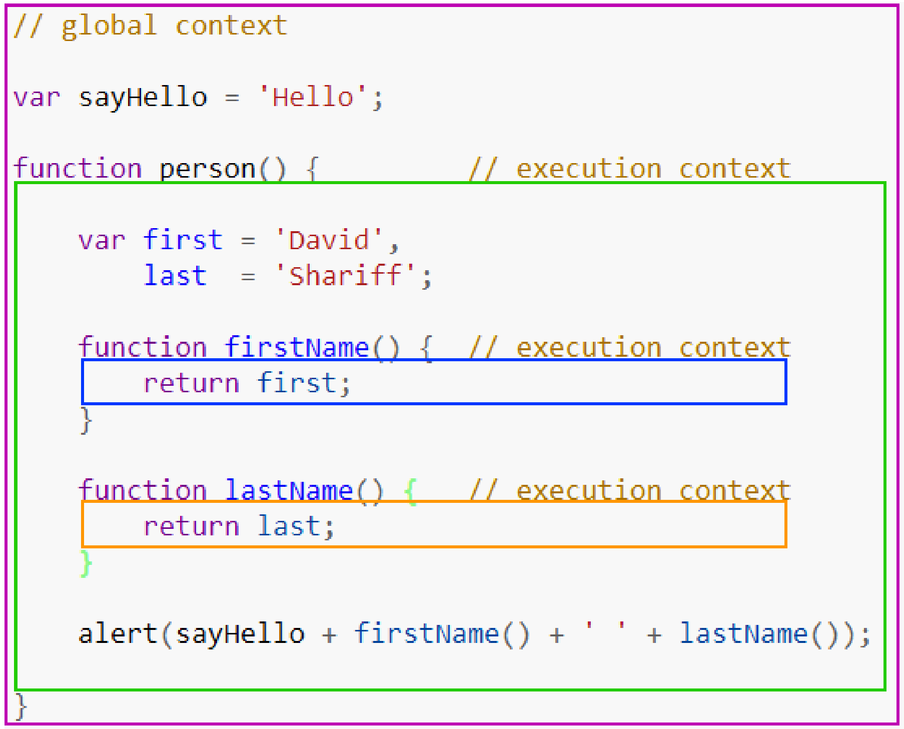


```
var mediumofInstruction ="english";

var student = function(){

  var insidelexicalscope = true;
  //defining below variable assigned to the global scope
  ethnicity="swiss";

  mediumofInstruction ="German";
  scholarship=true;
  console.log(mediumofInstruction);
  console.log(insidelexicalscope);
  console.log(scholarship);
};

var scholarship=false;
```

Technically this works! Without **var** this is assigned to the global scope . But we should Avoid this

Execution Contexts

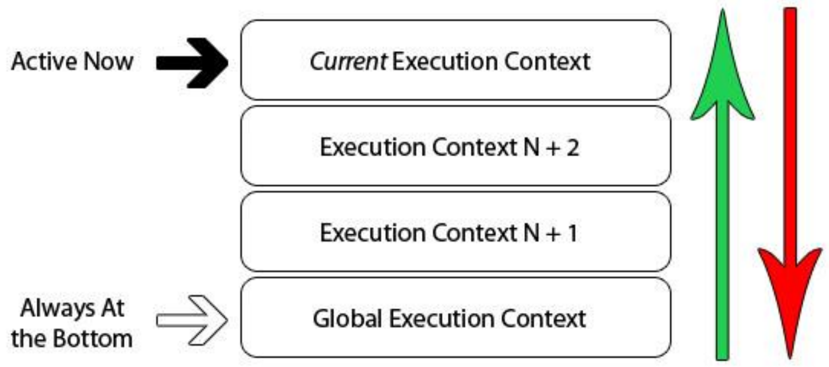
- When a code is run in JavaScript, the environment in which it is executed is very important
  - Global code: the default environment where your code is executed for the first time
  - Function code: whenever the flow of execution enters a function body
  - Eval code: text to be executed inside the internal eval function



```
// global context

var sayHello = 'Hello';

function person() {          // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {  // execution context
        return first;
    }

    function lastName() {   // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

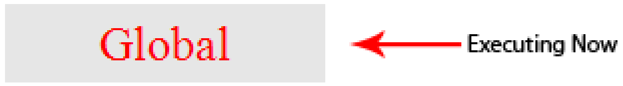
Execution Context Stack

- JavaScript interpreter in a browser is implemented as a single thread.
- Only one thing can ever happen at one time in a browser, with other actions or events being queued in what is called the **Execution Stack**



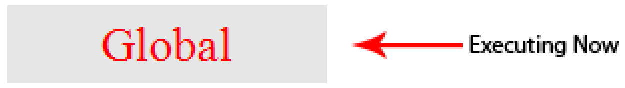
Execution Stack

```
(function foo(i) {
    if (i === 3) {
        return;
    }
    else {
        foo(++i);
    }
}(0));
```



Execution Stack

- Single threaded
- Synchronous execution
- 1 Global Context
- Infinite function context
- Each function call creates a new **execution context, even a call to itself.**

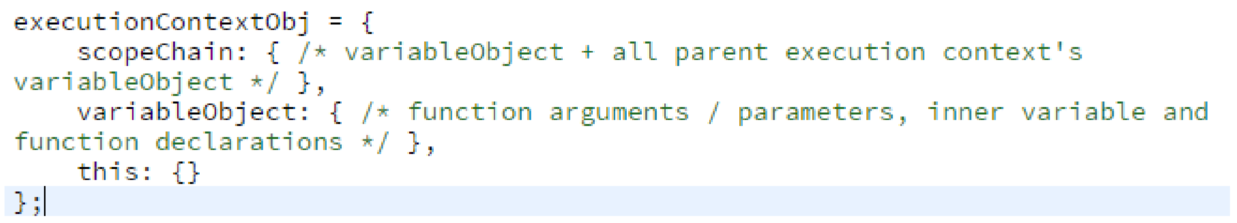


Execution Context stages

- JavaScript interpreter has 2 stages for every call to an execution context
  1. Creation stage : when the function is called, but before it executes any code inside
     - Create the **Scope Chain**
     - Create variables,functions and arguments
     - Determine the value of **“this”**.

2. Activation/Code Execution stage

- Assign values, references to functions and interpret/execute code



```
executionContextObj = {
    scopeChain: { /* variableObject + all parent execution context's
variableObject */ },
    variableObject: { /* function arguments / parameters, inner variable and
function declarations */ },
    this: {}
};
```

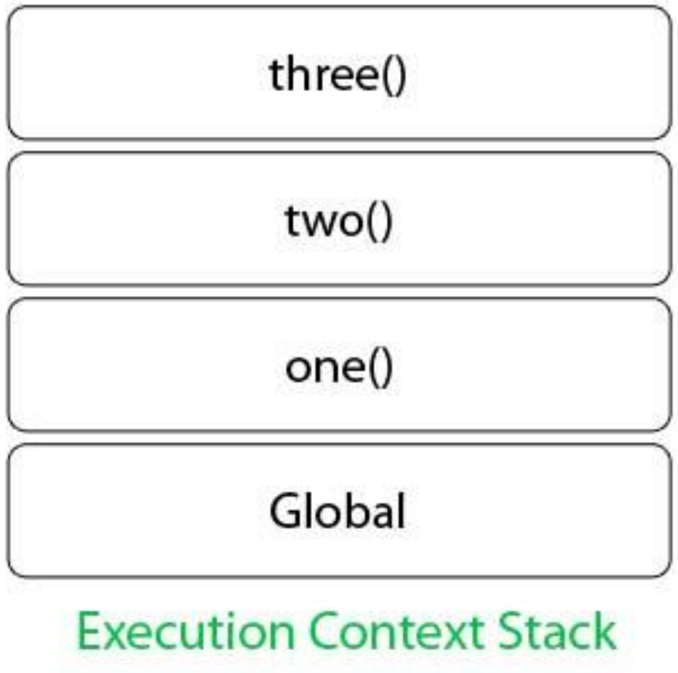
Scope Chain

- Every function has an associated **execution context** that contains a **variable object[vo]** which is composed of all variables, functions and parameters defined inside that given local function.
- **Scope Chain:** is simply a collection of the **current context's variable object[vo] + all parent's lexical [vo]s**

```
Scope = VO + All Parent VOs
Eg: scopeChain = [ [VO] + [VO1] + [VO2] + [VO n+1] ];
```

Scope Chain

```
function one() {

    two();

    function two() {

        three();

        function three() {
            alert('I am at function three');
        }

    }

}

one();
```



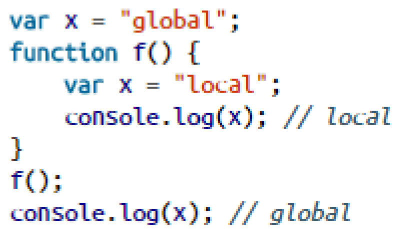
```
three() Scope Chain = [ [three() VO] + [two() VO] + [one() VO] + [Global VO] ];
```

Dynamic Scope

- Examining the scope of the variables, functions etc.. While executing the program **at runtime.**

Shadowing

- If a scope declares a variable that has the same name as one in a surrounding scope, access to the outer variable is blocked in the inner scope and all scopes nested inside it.
- Changes to the inner variable do not affect the outer variable, which is accessible again after the inner scope if left.



```
var x = "global";
function f() {
    var x = "local";
    console.log(x); // local
}
f();
console.log(x); // global
```

Declarations, Names and Hoisting

- In javascript, a name enters a scope in one of four basic ways.
  - Language define – All scopes are by default, given the names **this** and **arguments**
  - Formal parameters – functions can have named formal parameters, which are scoped to the body of that function.
  - Function declarations: these are of the form function **Person(){}**
  - Variable declarations: these take the form **var Person;**

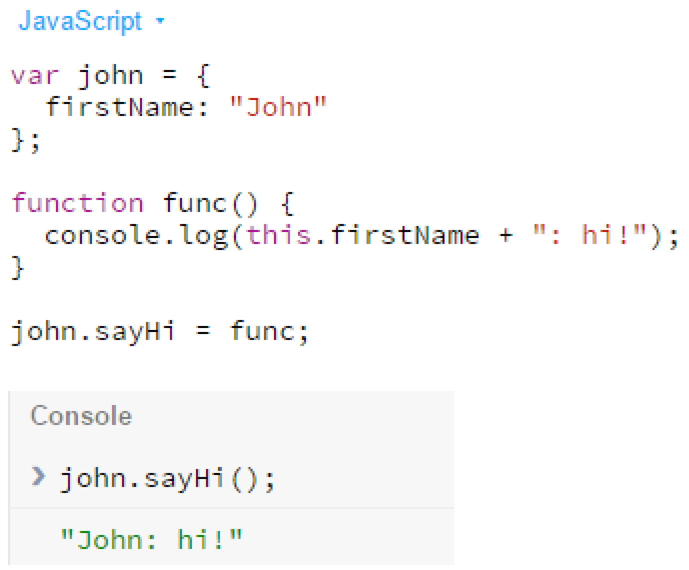
- Function declaration and variable declarations are always moved (“hoisted”) invisibly to the top of their containing scope by the JavaScript interpreter.

this keyword

- The value of **this** is dynamic in JavaScript
- It is determined when function is called, not when it is declared.
- Any function may use **this. It doesn't matter if the function is assigned to the object or not.**

When called as a method

- A function is called from the object (either dot or square bracket will do), **this** refers to this object



```
var john = {
  firstName: "John"
};

function func() {
  console.log(this.firstName + ": hi!");
}

john.sayHi = func;
```

```
> john.sayHi();
  "John: hi!"
```

this keyword

When called as a function

- If a function uses this, then it is meant to be called as a method. A simple func() call is usually a bug.

```
obj = {
  go: function() { alert(this) }
}

obj.go();
//result=> object

(obj.go)(); //result=> object

(a = obj.go)(); //result=> window

(0 || obj.go)(); //result=>window
```

In new

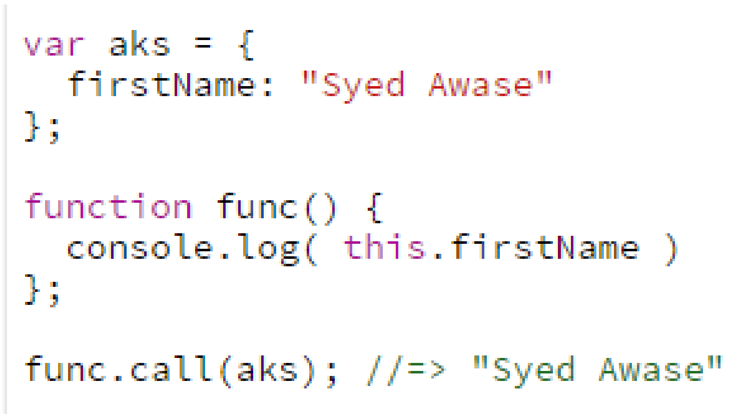
- When a new function is called, this is initialized as a **new object.**

```
function Animal(name) {
  this.name = name,
  this.canWalk = true
};

var animal = new Animal("beastie");

console.log(animal.name);
console.log(animal.canWalk);
```

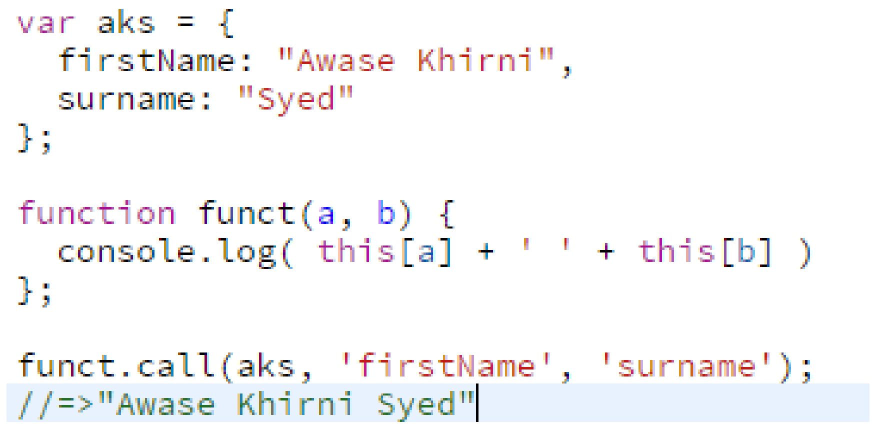
this Keyword

Explicit this

- A function can be called with explicit **this** value.
- This is done out by one of the two methods : call or apply.
- Funct is called with this=aks and given arguments, so it outputs aks['firstName] and aks['surname']
- **funct.call(context,args....) is essentially same as the simple call func(args..) but additionally sets this**



```
var aks = {
  firstName: "Syed Awase"
};

function func() {
  console.log( this.firstName )
};

func.call(aks); //=> "Syed Awase"
```

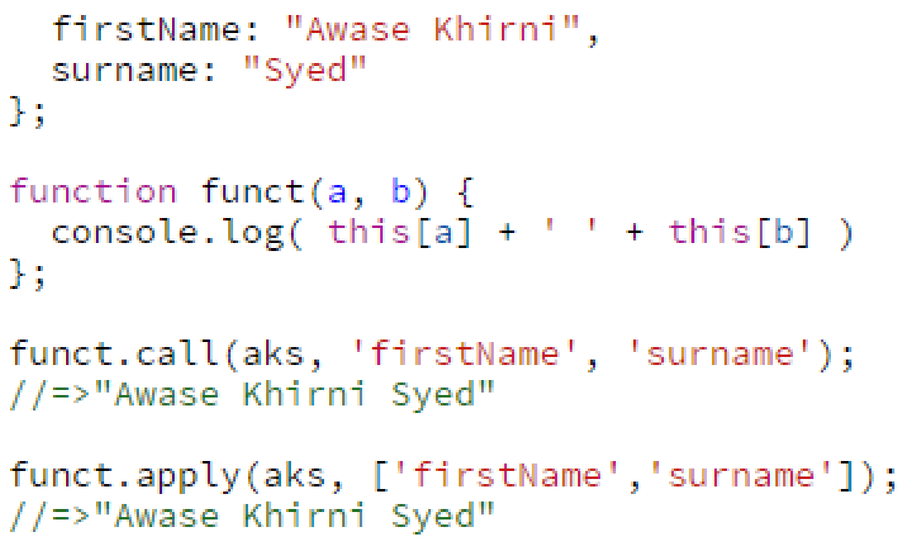


```
var aks = {
  firstName: "Awase Khirni",
  surname: "Syed"
};

function funct(a, b) {
  console.log( this[a] + ' ' + this[b] )
};

funct.call(aks, 'firstName', 'surname');
//=>"Awase Khirni Syed"
```

this keyword

apply

- Func.apply is same as func.call, **but it accepts an array of arguments instead of a list.**



```
  firstName: "Awase Khirni",
  surname: "Syed"
};

function funct(a, b) {
  console.log( this[a] + ' ' + this[b] )
};

funct.call(aks, 'firstName', 'surname');
//=>"Awase Khirni Syed"

funct.apply(aks, ['firstName','surname']);
//=>"Awase Khirni Syed"
```

Anatomy of Object Properties

- An object is considered to be an unordered set of properties, similar to a HashMap
- We can access an Object’s properties using **“dot” notation**
  - **var o = obj.prp; // use dot notation to access object properties**
- **Or bracketed notation**
  - **var o = obj[‘prp’] // property is a name (string identifier) associated with a property descriptor.**
- **hasOwnProperty**
  - **To find out if an object has specific property as one of its own property.**
  - **Useful to enumerate an object and extract the properties from own properties, not the inheritec ones.**
  - **Console.log(Object.hasOwnProperty(“propName”);**

Property descriptor

- Attributes are used to define the state of the properties.
- These attributes are available as fields on a property descriptor object.
- Fields vary, depending on the property descriptor type
  - Value
  - Get
  - Set
  - Writable
  - Configurable
  - enurmerable

Object

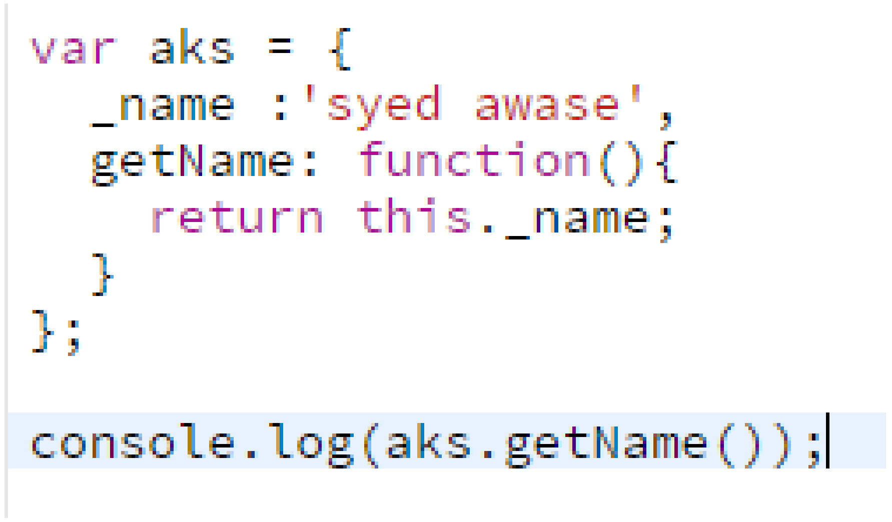
- Object Data Properties Have Attributes
  Each data property (object property that store data) has not only the name-value pair, but also 3 attributes (the three attributes are set to true by default):
  — Configurable Attribute: Specifies whether the property can be deleted or changed.
  — Enumerable: Specifies whether the property can be returned in a for/in loop.
  — Writable: Specifies whether the property can be changed.

ECMAScript 5 Property Types

- Internal properties
- Data properties
- Accessor properties

DATA Properties

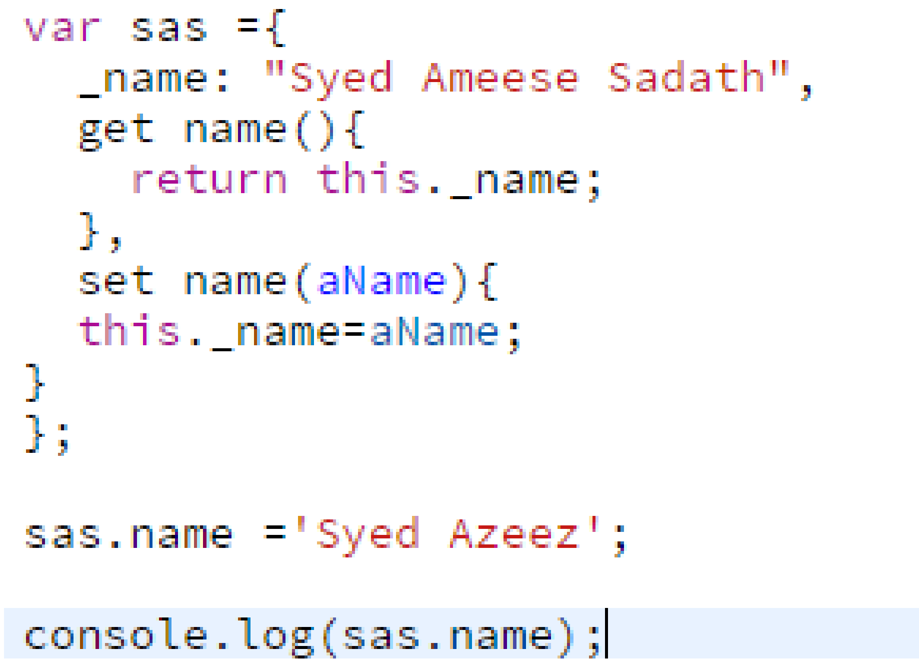
- It directly associates a name to a value through the value field of the property descriptor.
- When a data property’s descriptor value directly references a **function type** it is referred to as a method.



```
var aks = {
   _name :'syed awase',
   getName: function(){
      return this._name;
   }
};

console.log(aks.getName());
```

Accessor Properties

- Introduced in ECMAScript 5
- They are described through user provided getter and setter functions. Where the user provided functions are assigned to get and set attributes on the Accessor Property's property descriptor object.



```
var sas ={
  _name: "Syed Ameeese Sadath",
  get name(){
    return this._name;
  },
  set name(aName){
  this._name=aName;
}
};

sas.name ='Syed Azeez';

console.log(sas.name);
```

Property Descriptor Types

Attribute	Description	Default
Value	The value associated with the property (Data descriptor only)	Undefined
Writeable	Boolean:when false, attempts by cod to change the value are ignored	FALSE
Enumerable	Boolean: if true, property can be enumerate by for ... In statement	FALSE
Configurable	Boolean : if true, property can be changed or property can be deleted, if false, only property value can change	FALSE

Accessor Property Descriptor

- It includes any fields named either [[GET]] or [[SET]]

Attribute	Description	Default
Get	The function that returns the property value. The function has no parameters	Undefined
Set	A function that has one parameter that contains the value assigned. The set accessor function is used to set the property value	Undefined
Enumerable	Boolean: if true, property can be enumerate by for ..In statement	FALSE
Configurable	Boolean: if ture, property attributes can be changed or property can be deleted, if false, only property value can change	FALSE

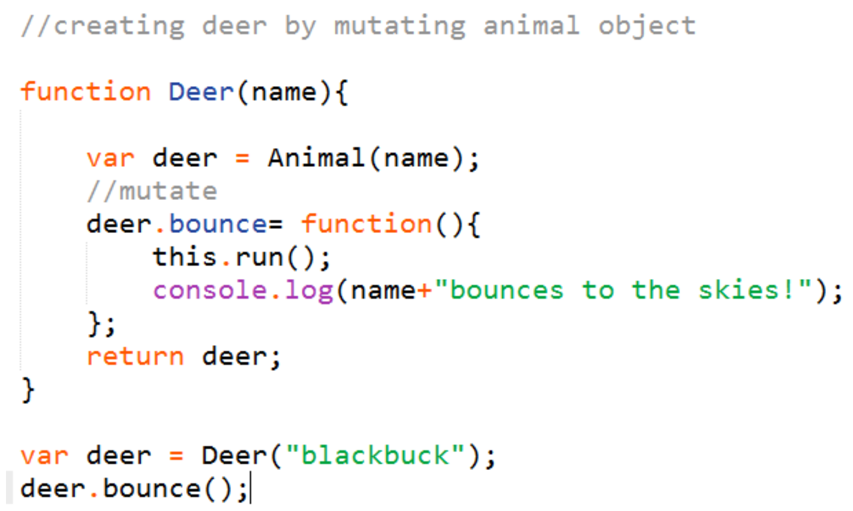
Parasitic Inheritance

By Douglas Crockford

- With parasitic inheritance you create an object using a regular function instead of a constructor.
- This function simply copies the object that you want to inherit, adds the extra properties and methods you want the new object to have and then returns the new object
- Makes the “new” keyword optional.

Inheritance

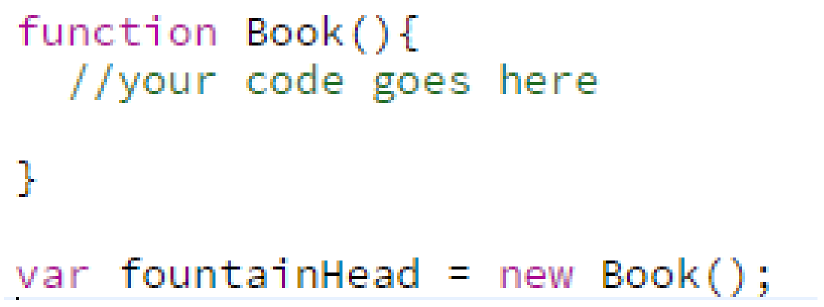
- Created an animal – deer by inheriting the properties and methods and then mutating it.



```
//creating deer by mutating animal object

function Deer(name){

    var deer = Animal(name);
    //mutate
    deer.bounce= function(){
        this.run();
        console.log(name+"bounces to the skies!");
    };
    return deer;
}

var deer = Deer("blackbuck");
deer.bounce();
```

Constructor Function()

- JavaScript does not support classes, but it has constructors to bring similar functionality to JavaScript.
- Constructors are like regular functions, but we use them with the”new” keyword.
- Two types of constructors:
- Built-in like Array and Object
- Custom constructors- which define properties and methods of your own type of object
- A constructor is useful when you want to create multiple similar objects with the same properties and methods.
- It’s a **convention to capitalize** the name of constructors to distinguish them from regular functions.



```
function Book(){
  //your code goes here

}

var fountainHead = new Book();
```

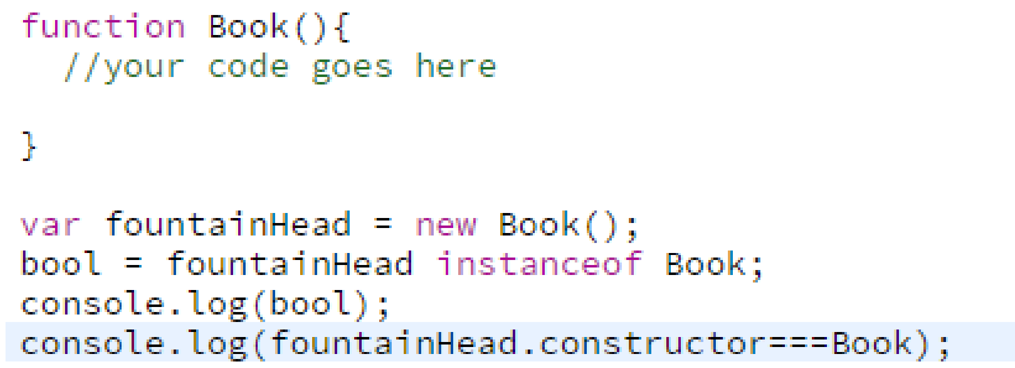
? Happens when a Constructor is called

- Creates a new object
- Sets the constructor property of the object to ObjectName defined
- Sets up the object to delegate to ObjectName.prototype
- It calls ObjectName() in the context of the new Object

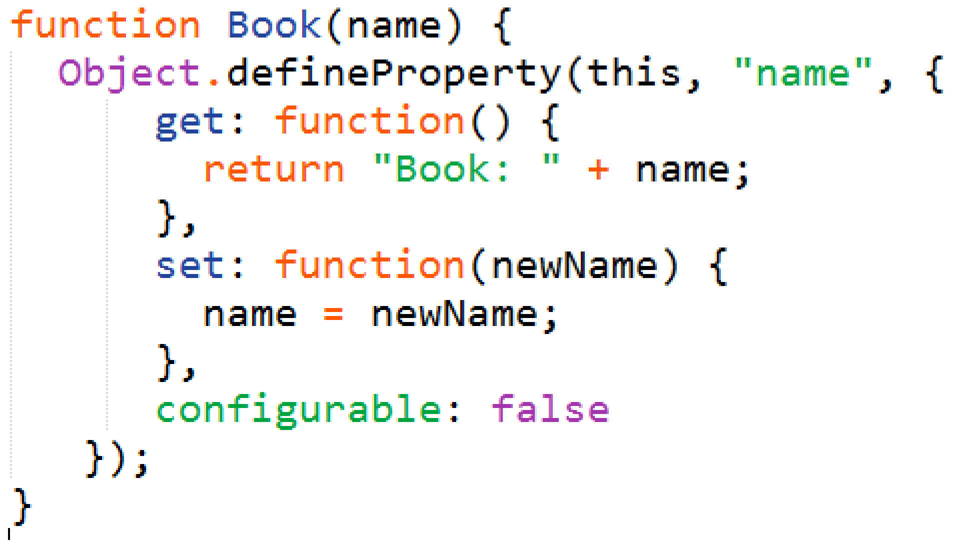
- A function is just a special kind of object and like any object a function can have properties.
- Functions automatically get a property called **prototype, which is just an empty object**

Determining the type of an Instance

- **Instanceof** is used to find out whether an object is an instance of another one.
- **Object.defineProperty()** can be used inside of a constructor to help perform all necessary property setup.



```
function Book(){
  //your code goes here

}

var fountainHead = new Book();
bool = fountainHead instanceof Book;
console.log(bool);
console.log(fountainHead.constructor===Book);
```

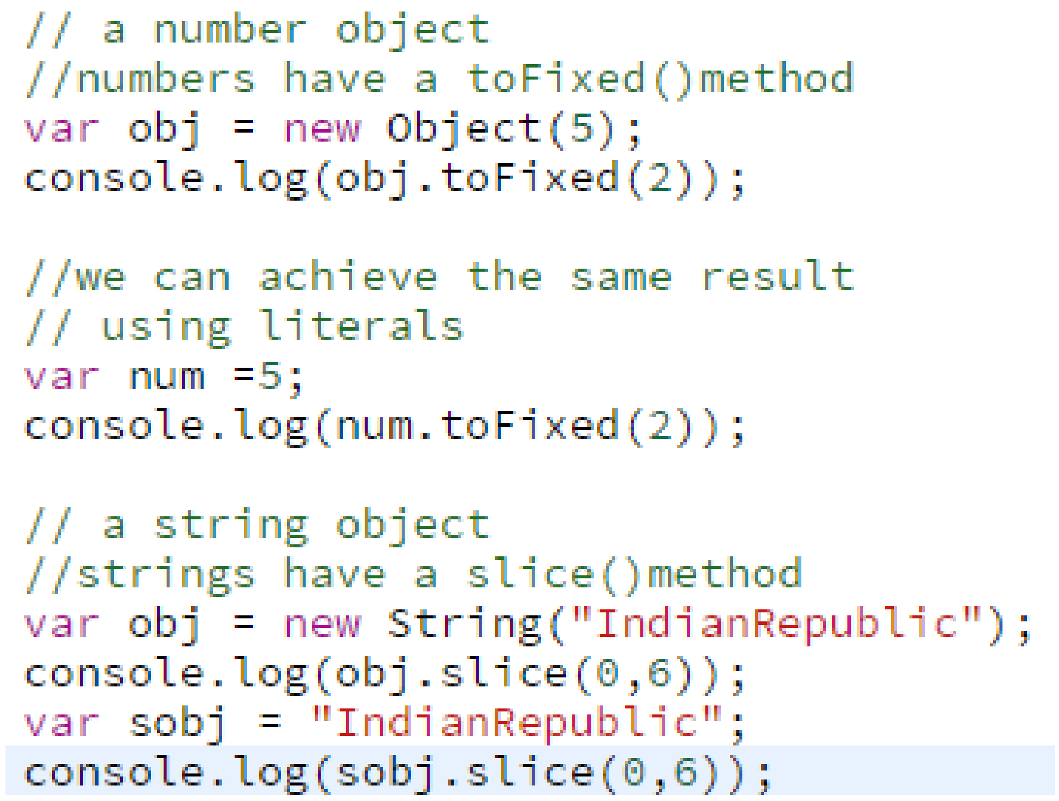


```
function Book(name) {
  Object.defineProperty(this, "name", {
      get: function() {
        return "Book: " + name;
      },
      set: function(newName) {
        name = newName;
      },
      configurable: false
    });
}
```

Object Literal Notations

JavaScript has nine built-in constructors

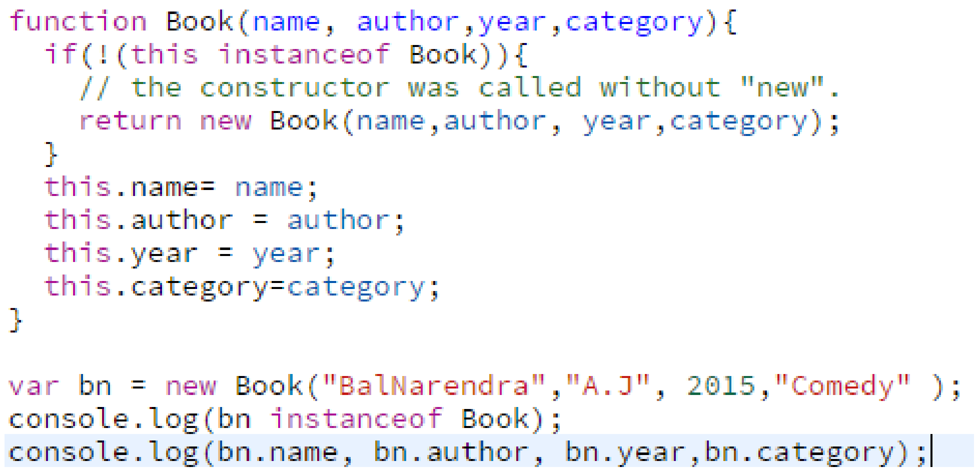
1. Object()
2. Array()
3. String()
4. Number()
5. Boolean()
6. Date()
7. Function()
8. Error()
9. RegExp()



```
// a number object
//numbers have a toFixed()method
var obj = new Object(5);
console.log(obj.toFixed(2));

//we can achieve the same result
// using literals
var num =5;
console.log(num.toFixed(2));

// a string object
//strings have a slice()method
var obj = new String("IndianRepublic");
console.log(obj.slice(0,6));
var sobj = "IndianRepublic";
console.log(sobj.slice(0,6));
```

Scope-Safe Constructors

- Since a constructor is just a function, it can be called without the **new** keyword, leading to unexpected results/errors
- Call scope-safe constructors with or without **new** keyword and they return the same result in either form
- Most of the built-in constructors such as Object, Regex and Array are scope-safe



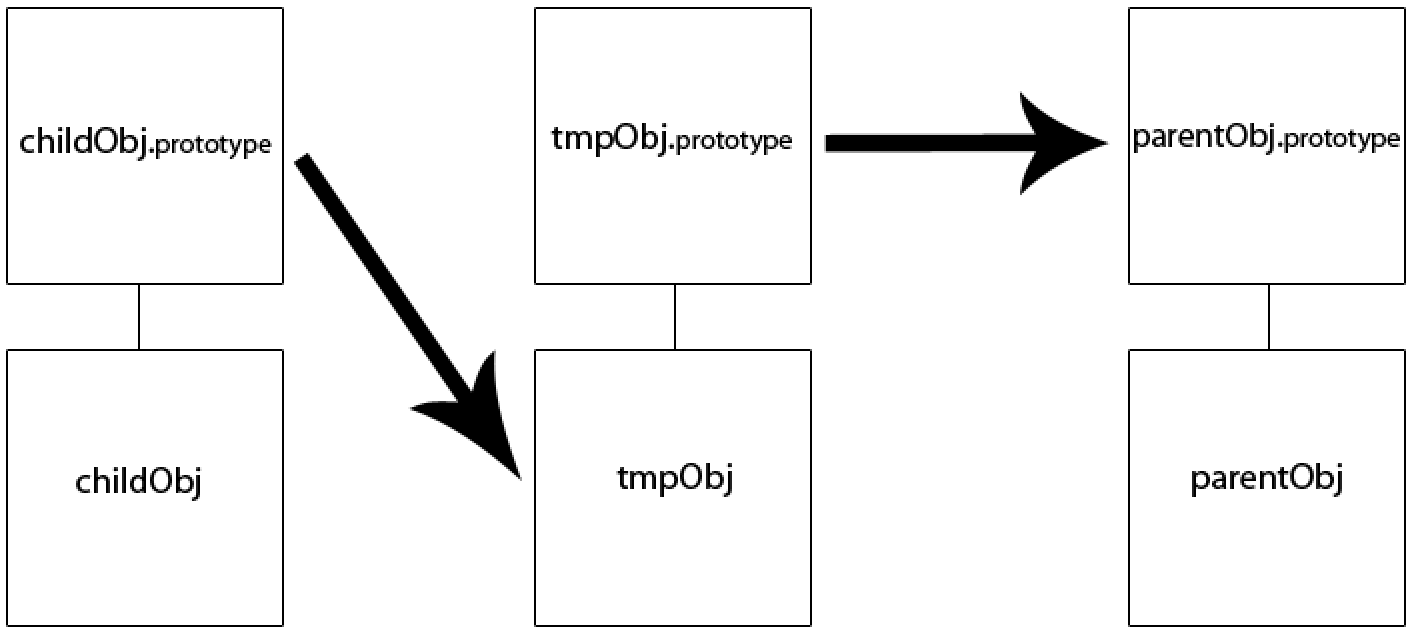
```
function Book(name, author,year,category){
  if(!(this instanceof Book)){
    // the constructor was called without "new".
    return new Book(name,author, year,category);
  }
  this.name= name;
  this.author = author;
  this.year = year;
  this.category=category;
}

var bn = new Book("BalNarendra","A.J", 2015,"Comedy" );
console.log(bn instanceof Book);
console.log(bn.name, bn.author, bn.year,bn.category);
```

7. OBJECT ORIENTATION WITH PROTOTYPES

Prototype Inheritance

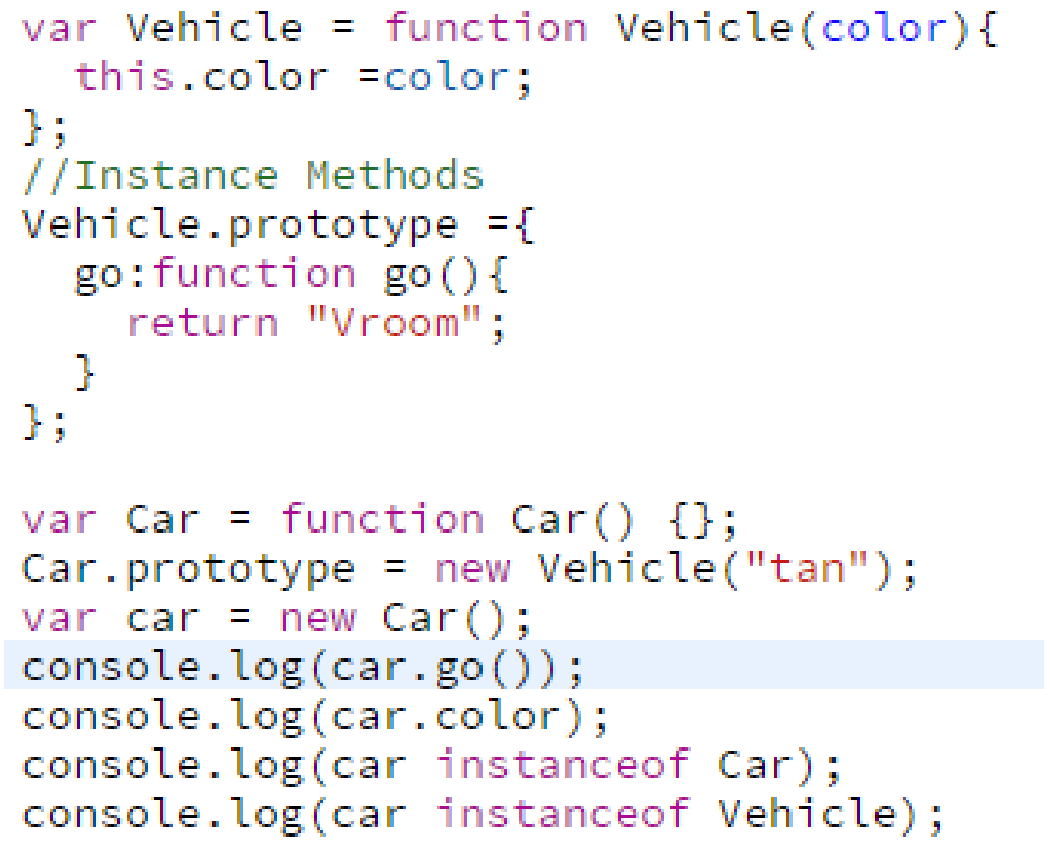
- When an object is constructed, **it inherits all of the properties of its constructor's prototype.**
- **Now this"inheritance" is more than simply copying properties to the new objects.**
- The object is set up to delegate any properties which haven't been explicitly setup to the **constructor's prototype.**
- **Which means that we can change the prototype later, and still see the changes in the instance.**

Flow of extendObj function



src:http://davidshariff.com/blog/javascript-inheritance-patterns/

Prototype Inheritance



```
var Vehicle = function Vehicle(color){
  this.color =color;
};
//Instance Methods
Vehicle.prototype ={
  go:function go(){
    return "Vroom";
  }
};

var Car = function Car() {};
Car.prototype = new Vehicle("tan");
var car = new Car();
console.log(car.go());
console.log(car.color);
console.log(car instanceof Car);
console.log(car instanceof Vehicle);
```

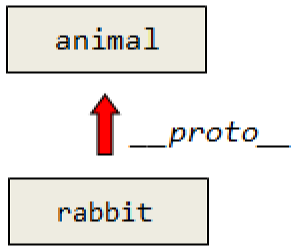
Constructor Function Benefits over factory functions

- Instanceof operator
- Better resource management
- Less memory usage
- Better performance and less execution time

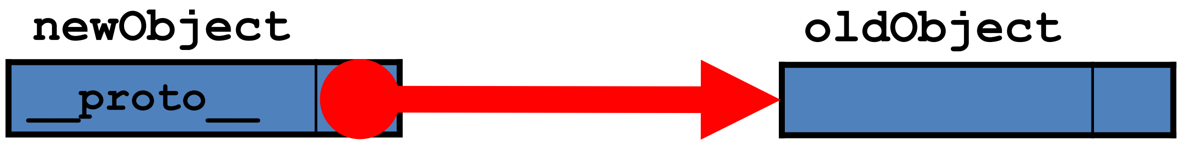
Prototypal Inheritance

- Class-free.
- Objects inherit from objects.
- An object contains a **secret link** to the object it inherits from.
- Mozilla calls it __proto__.

- Changes in old object may be immediately visible in newObject
- Changes to newObject have no effect on oldobject
- OldObject can be the prototype for unlimited number of objects which will inherit its properties



```
var newObject = object(oldObject);
```



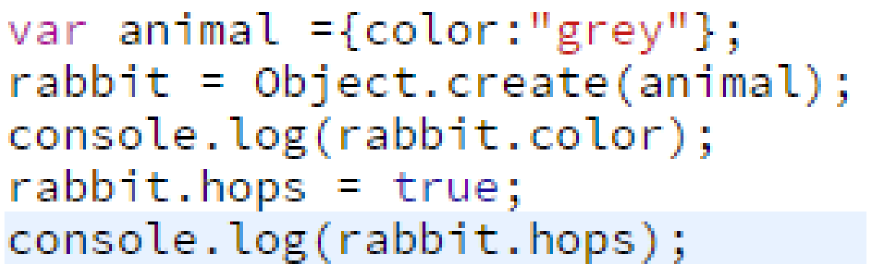
Prototype Inheritance

- Using the Object function, we can quickly produce new objects that have the same state and behaviour as existing object
- We can then augment each of the instances by assigning new methods and members
- Available for Constructor Functions
- In JavaScript, the inheritance is prototype-based. That means that there are no classes. **Instead, an object inherits from another object**

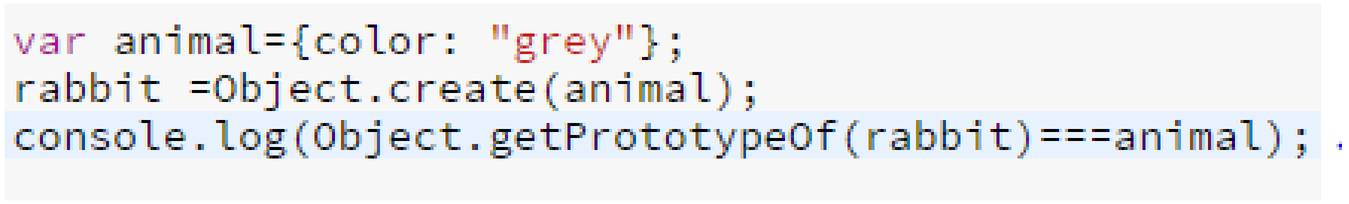
Standard methods for working with Prototypes

- Object.create(proto[,props])
  - Create an empty object with given __proto__
  - Creates empty rabbit with animal __protot__

- Object.getPrototypeOf(obj)
  - Returns the value of obj.__proto__



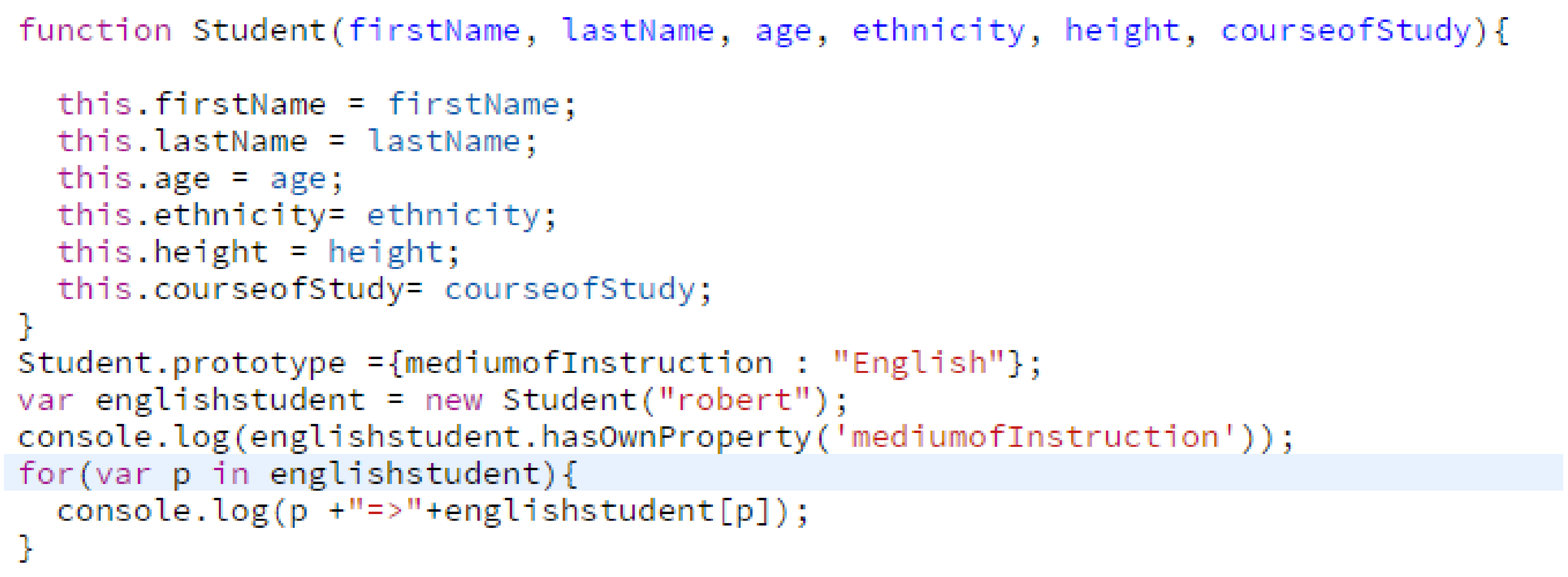
```
var animal ={color:"grey"};
rabbit = Object.create(animal);
console.log(rabbit.color);
rabbit.hops = true;
console.log(rabbit.hops);
```



```
var animal={color: "grey"};
rabbit =Object.create(animal);
console.log(Object.getPrototypeOf(rabbit)===animal);
```

Looping with/without inherited properties

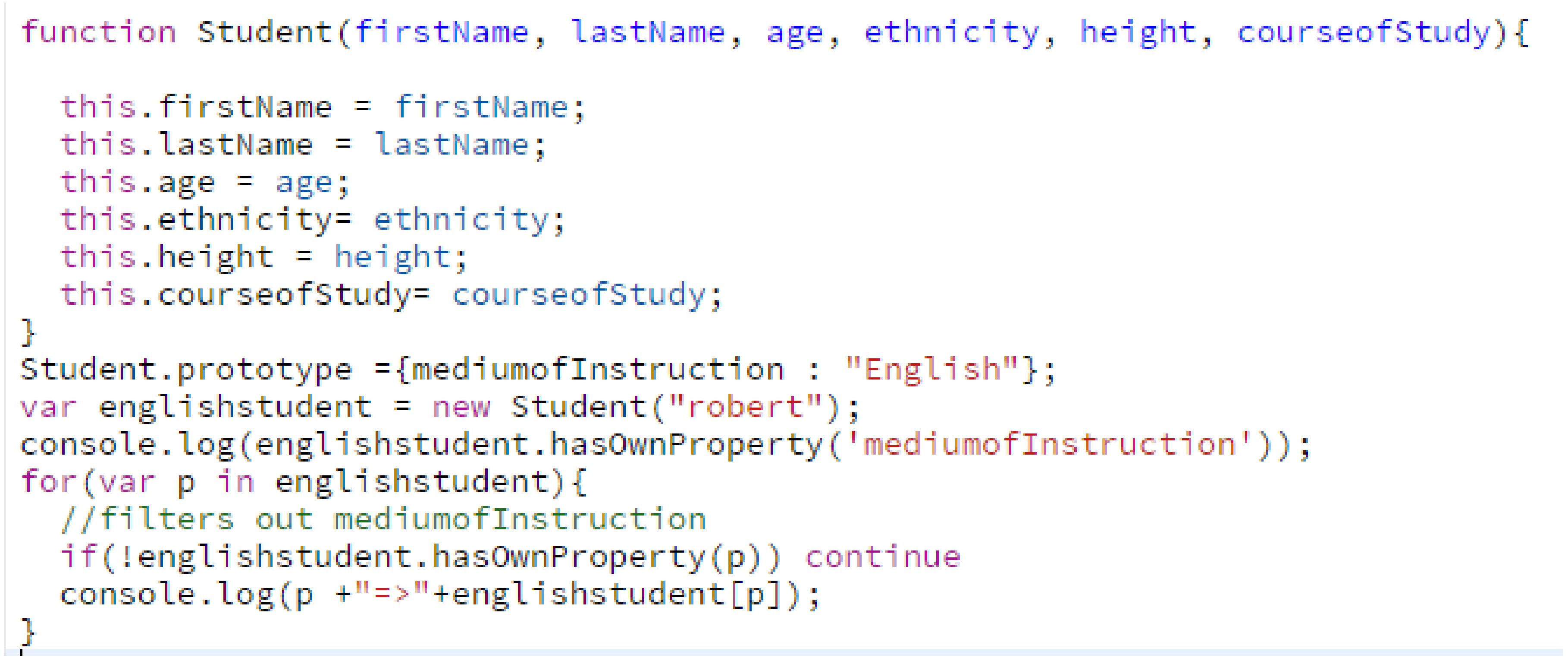
- For..in loop outputs all properties from the object and its prototype



```
function Student(firstName, lastName, age, ethnicity, height, courseofStudy){

  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.ethnicity= ethnicity;
  this.height = height;
  this.courseofStudy= courseofStudy;
}
Student.prototype ={mediumofInstruction : "English"};
var englishstudent = new Student("robert");
console.log(englishstudent.hasOwnProperty('mediumofInstruction'));
for(var p in englishstudent){
  console.log(p +"=>"+englishstudent[p]);
}
```

hasOwnProperty

To get a list of properties, which are not in prototype, we shall filter them through **hasOwnProperty**



```
function Student(firstName, lastName, age, ethnicity, height, courseofStudy){

  this.firstName = firstName;
  this.lastName = lastName;
  this.age = age;
  this.ethnicity= ethnicity;
  this.height = height;
  this.courseofStudy= courseofStudy;
}
Student.prototype ={mediumofInstruction : "English"};
var englishstudent = new Student("robert");
console.log(englishstudent.hasOwnProperty('mediumofInstruction'));
for(var p in englishstudent){
  //filters out mediumofInstruction
  if(!englishstudent.hasOwnProperty(p)) continue
  console.log(p +"=>"+englishstudent[p]);
}
```

Prototype Inheritance Summary

- Inheritance is implemented through a special property __proto__ (named[[prototype]]
- When a property is accessed, and the interpreter can't find it in the object, it follows the __proto__ link and searches it there
- The value of this for function properties is set to the object, not its prototype
- Assignment **obj.prop= val**
- Deletion **delete obj.prop**

- An empty object with given prototype can be created by
- A constructor function sets __proto__ for objects it creates to the value of its prototype property
- **Object.create(proto)**

Or

```
function inherit(proto){
Function F(){}
F.prototype = proto
Return new F
}
```

The”constructor” Property

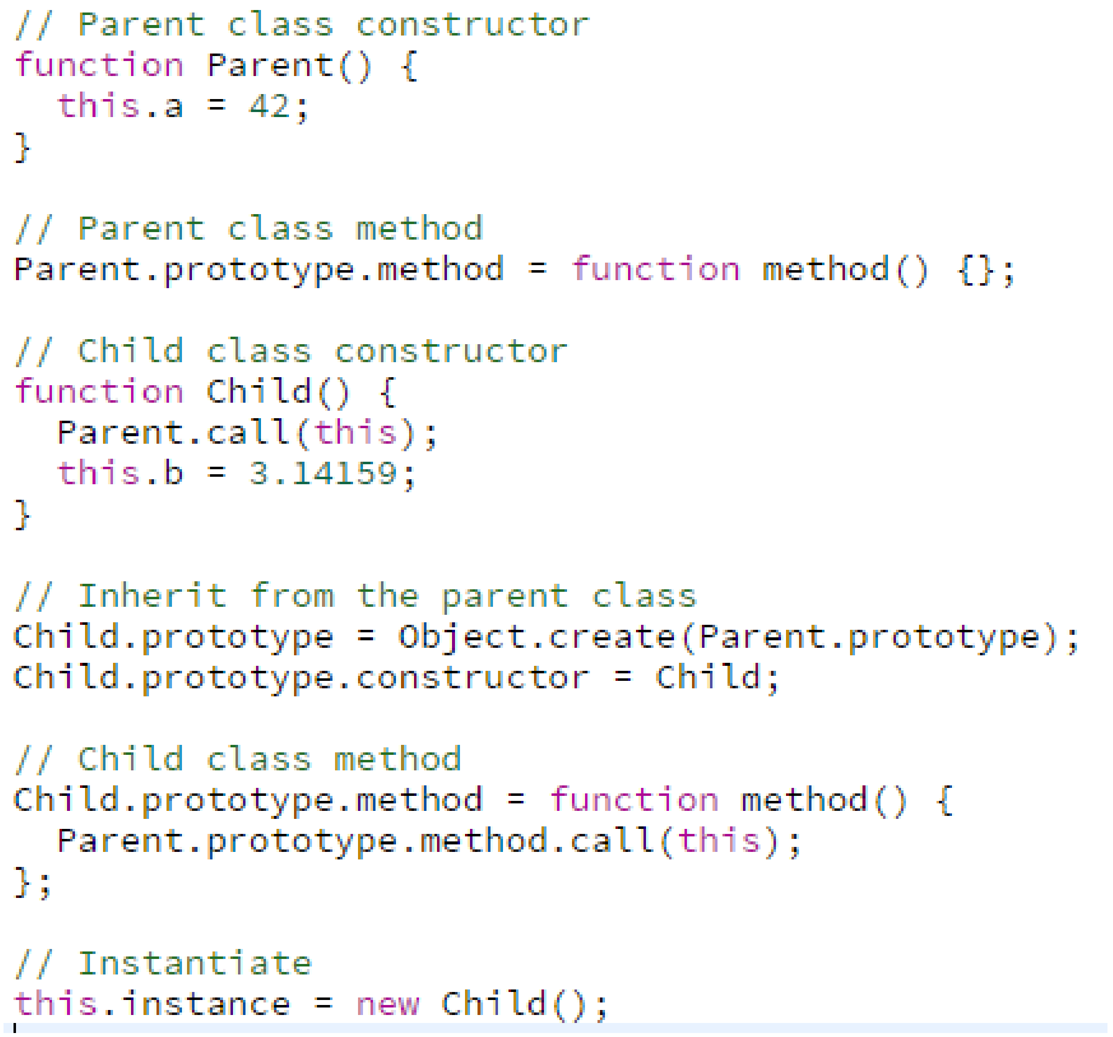
- The interpreter creates the **new** function object from your declaration. It’s prototype property is created and populated.
- The default value the prototype is an object with property **constructor,** which is set to the function itself

```
function Student(){}

var estudent = new Student();

console.log(estudent.constructor === Student);
```

```
function Student(){}

var estudent = new Student();

console.log(estudent.constructor === Student);
console.log(estudent.hasOwnProperty('constructor'));
console.log(Student.hasOwnProperty('constructor'));
console.log(Student.prototype.hasOwnProperty('constructor'));
```

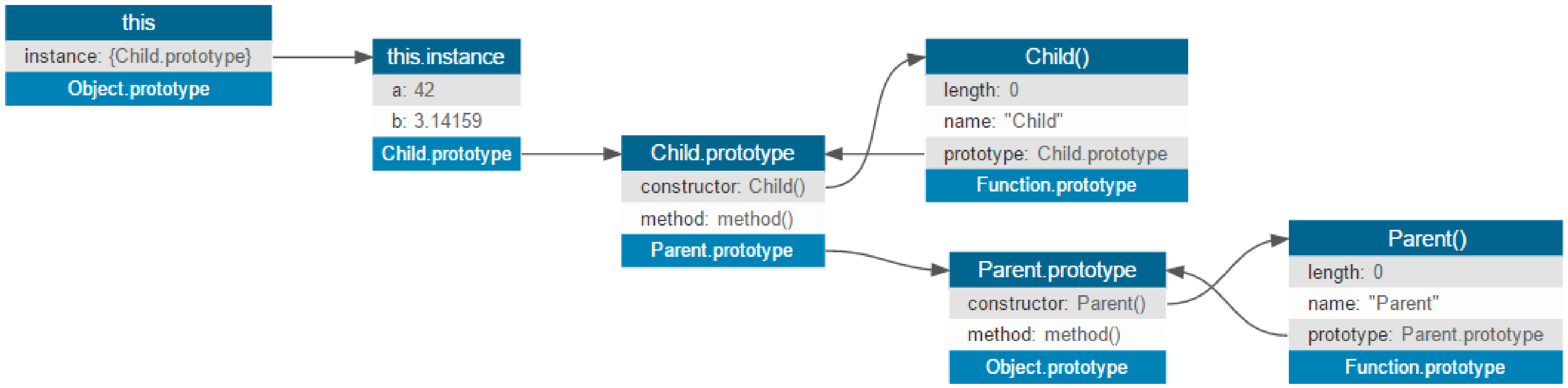
Student.prototype ={constructor: Student}

Parent Child



```
// Parent class constructor
function Parent() {
  this.a = 42;
}

// Parent class method
Parent.prototype.method = function method() {};

// Child class constructor
function Child() {
  Parent.call(this);
  this.b = 3.14159;
}

// Inherit from the parent class
Child.prototype = Object.create(Parent.prototype);
Child.prototype.constructor = Child;

// Child class method
Child.prototype.method = function method() {
  Parent.prototype.method.call(this);
};

// Instantiate
this.instance = new Child();
```

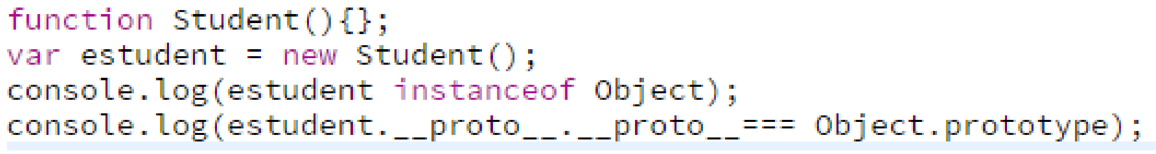
Prototype Inheritance



src: http://www.objectplayground.com/

The “instanceof” Operator

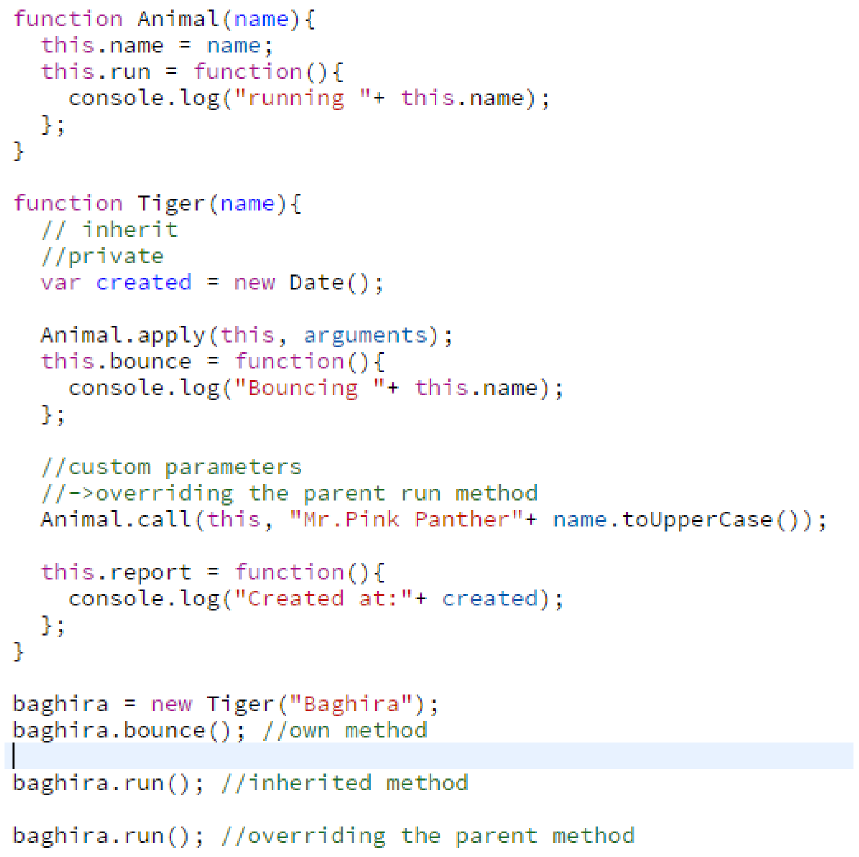
- It follows the __proto__ chain to check if the object is created by given constructor.
- Steps
  1. Get obj.__proto__
  2. Compare obj.__proto__ against F.prototype
  3. If no match then set temporarily obj= obj.__proto__ and repeat step 2 until either match is found or the chain ends.



```
function Student(){};
var estudent = new Student();
console.log(estudent instanceof Object);
console.log(estudent.__proto__.__proto__=== Object.prototype);
```

All-in-one Constructor Pattern

- The object is declared solely by it's constructor
- Inheritance
  - We use apply constructor to **this.**
  - Modify **this,** add more methods to derived function
  - Overriding a parent method is as easy as overwriting it in **this**.
- Private methods and properties are supported really good in this pattern
  - A local function or variable is private.
  - All constructor arguments are private automatically



```
function Animal(name){
  this.name = name;
  this.run = function(){
    console.log("running "+ this.name);
  };
}

function Tiger(name){
  // inherit
  //private
  var created = new Date();

  Animal.apply(this, arguments);
  this.bounce = function(){
    console.log("Bouncing "+ this.name);
  };

  //custom parameters
  //->overriding the parent run method
  Animal.call(this, "Mr.Pink Panther"+ name.toUpperCase());

  this.report = function(){
    console.log("Created at:"+ created);
  };
}

baghira = new Tiger("Baghira");
baghira.bounce(); //own method

baghira.run(); //inherited method

baghira.run(); //overriding the parent method
```

Factory constructor pattern

- Uses a function which creates an object on its own without **new**
- **Inheritance is done by creating a parent object first and then modifying it**
- **Local methods and functions are private. The object must be stored in closure prior to returning if we want to access it’s public methods from local ones**

Section VI

JavaScript

6. CLOSURE IN-DEPTH

Closure

- A closure is an inner function that has access to the outer(enclosing) function’s variables – scope chain.
- The closure has **3 scope chains**
  1. **It has access to its own scope(variables defined between its curly brackets)**
  2. **It has access to the outer function’s variables**
  3. **It has access to the global variables**

Closure

A function that remembers what happens around it

Any function that keeps reference to variable from its parent's scope even after the parent has returned.

Closures

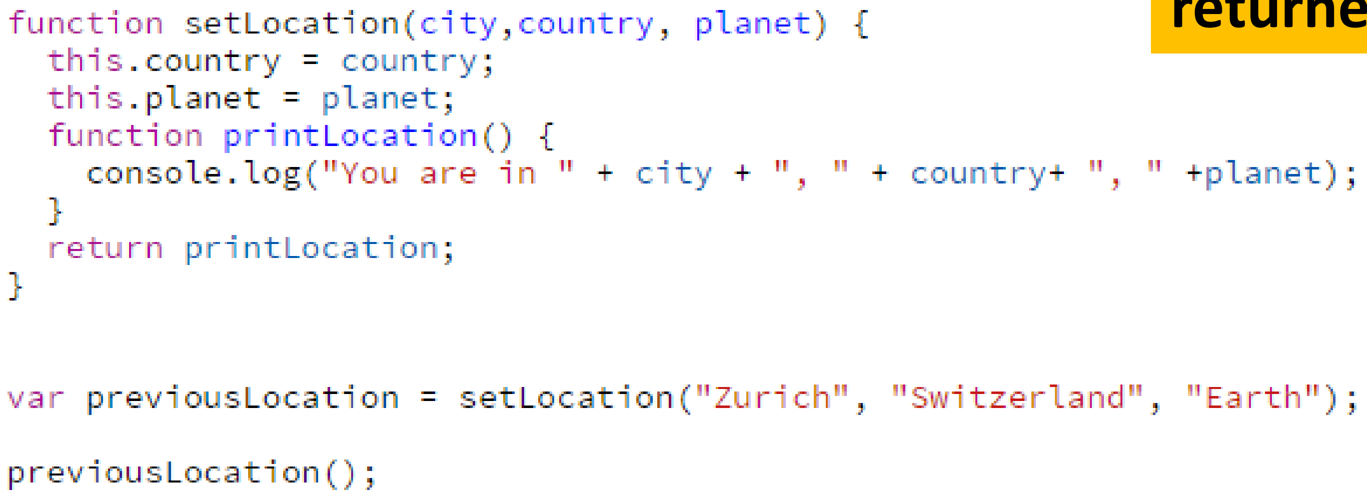
- A function can refer to, or have access to
  1. Any variable and parameters in its own function scope
  2. Any variable and parameters of outer(parent) functions
  3. Any variables from the global scope.

Refer to variables defined outside of the current function

```
function setLocation(city,country, planet) {
  this.country = country;
  this.planet = planet;
  function printLocation() {
    console.log("You are in " + city + ", " + country+ ", " +planet);
  }
  printLocation();
}
setLocation("Bangalore","India", "Earth");
```

A closure (inner function) is able to remember its surrounding scope (outer functions) even when it's executed outside it's lexical scope

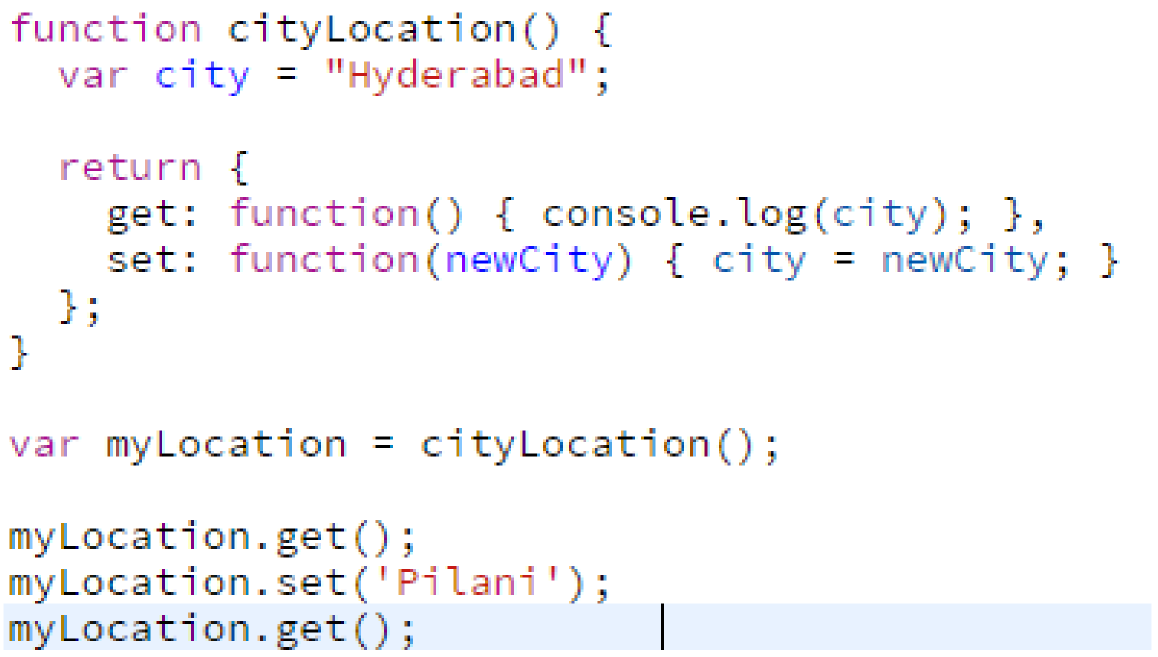
Inner functions can refer to variables defined in outer functions even after the latter have returned



```
function setLocation(city,country, planet) {
  this.country = country;
  this.planet = planet;
  function printLocation() {
    console.log("You are in " + city + ", " + country+ ", " +planet);
  }
  return printLocation;
}


var previousLocation = setLocation("Zurich", "Switzerland", "Earth");

previousLocation();
```

//printLocation() is executed outside its lexical scope

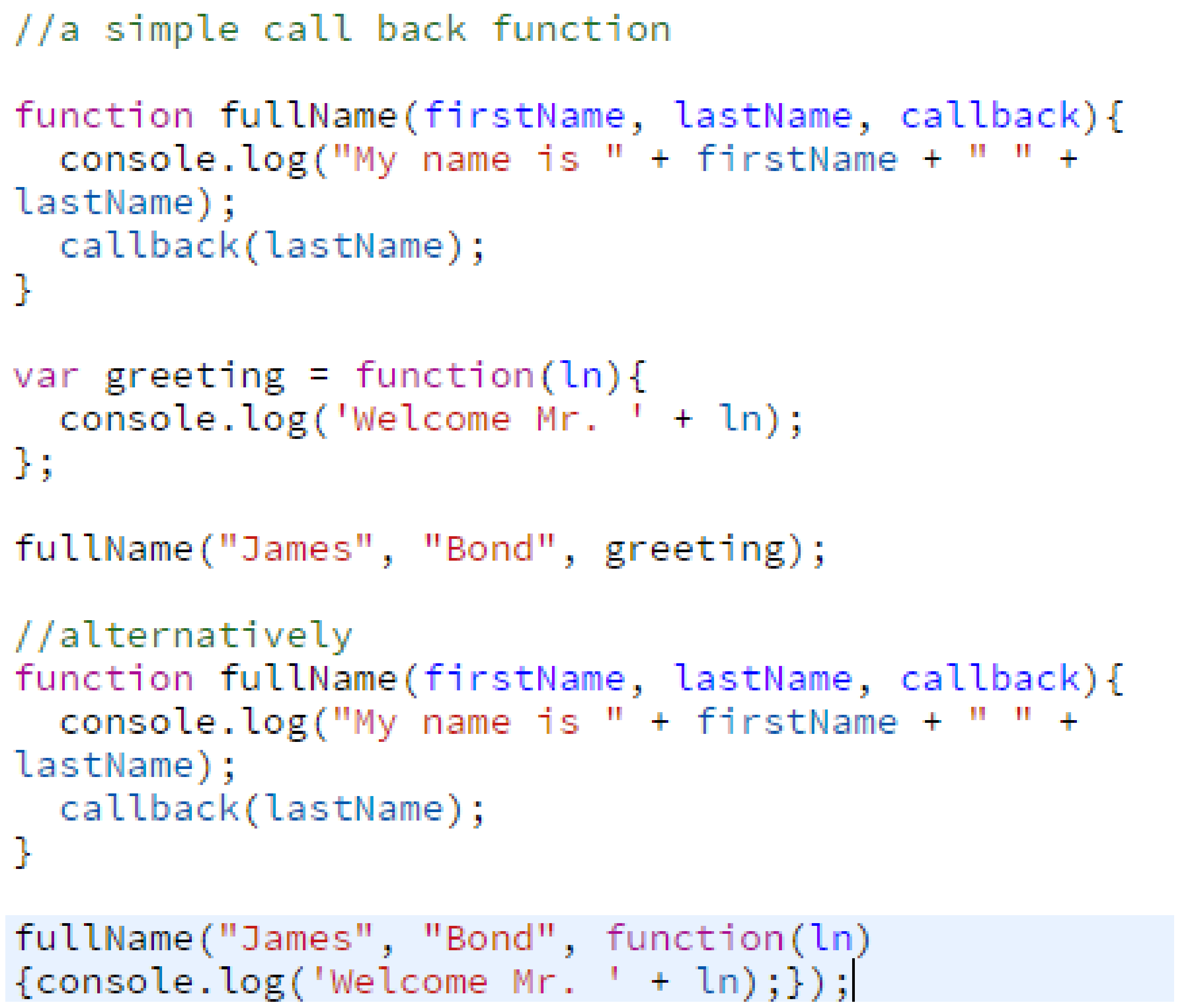
Inner functions store their outer function’s variables by reference, not by value



```
function cityLocation() {
  var city = "Hyderabad";

  return {
    get: function() { console.log(city); },
    set: function(newCity) { city = newCity; }
  };
}

var myLocation = cityLocation();

myLocation.get();
myLocation.set('Pilani');
myLocation.get();
```

Callbacks

- Functions are first-class object. Functions can be passed as arguments to other functions and can also be returned by other functions.
- A function that takes other functions as arguments or returns functions as its result is called higher-order function and the function that is passed as an argument is called a callback function/callback, because at some point in time it is “called back” by the higher-order function
- Callbacks are heavily used to provide generalization and reusability. They allow the library methods to be easily customized and/or extended.



```
//a simple call back function

function fullName(firstName, lastName, callback){
  console.log("My name is " + firstName + " " +
lastName);
  callback(lastName);
}

var greeting = function(ln){
  console.log('Welcome Mr. ' + ln);
};

fullName("James", "Bond", greeting);

//alternatively
function fullName(firstName, lastName, callback){
  console.log("My name is " + firstName + " " +
lastName);
  callback(lastName);
}

fullName("James", "Bond", function(ln)
{console.log('Welcome Mr. ' + ln);});
```

http://benalman.com/news/2010/11/immediately-invoked-function-expression/

Immediately Invoked Function expression (IIFE)

Ben Alman

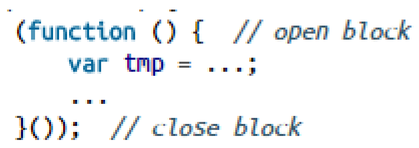
- Every function, when invoked creates a new execution context, because variables and functions defined within a function may only be accessed inside, but not outside, that context, invoking a function provides a **very easy way to create privacy**
- **Javascript interpreter encounters the function keyword in the global scope or inside a function, it treats it as a function declaration(statement) and not as a function expression,by default**
- Unless explicitly informed to the javascript interpreter to expect and expression, it sees what it thinks to be a **function declaration without a name and throws a SyntaxError exception because function declarations require a name**
- **Parenthesis placed after an expression indicate that expression is a function to be *invoked, parenthesis placed after a statement are totally separate from the preceding statement and are simply a grouping operator i.e used as a means to control precedence of evaluation.***

IIFE

- Arguments may be passed when functions are invoked by their named identifier, they may also be passed when immediately invoking a function expression.
- By the definition of closure, any function defined inside another function can access the outer function's passed-in arguments and variables

- IIFE can be used to "**lock in**" values and effectively save state
- One of the most advantageous side effects of IIFE is that, because this **unnamed, or anonymous, function expression is invoked immediately, without using an identifier, a closure can be used without polluting the current scope**

IIFE



```
(function () {  // open block
    var tmp = ...;
    ...
}());  // close block
```

It’s a function expression that gets invoked immediately

Introduces a new scope to restrict the lifetime of a variable

IIFE

- Enable you to attach private data to a function.
- Users don't have to declare a global variable and can tightly package the function with its state.
- Users can avoid polluting the global namespace

- Creating fresh environments, avoiding sharing
- Keeping global data private to all of a constructor
- Attaching global data to a singleton object
- Attaching global data to a method
- Avoiding global variables, hiding variables from global scope

Module

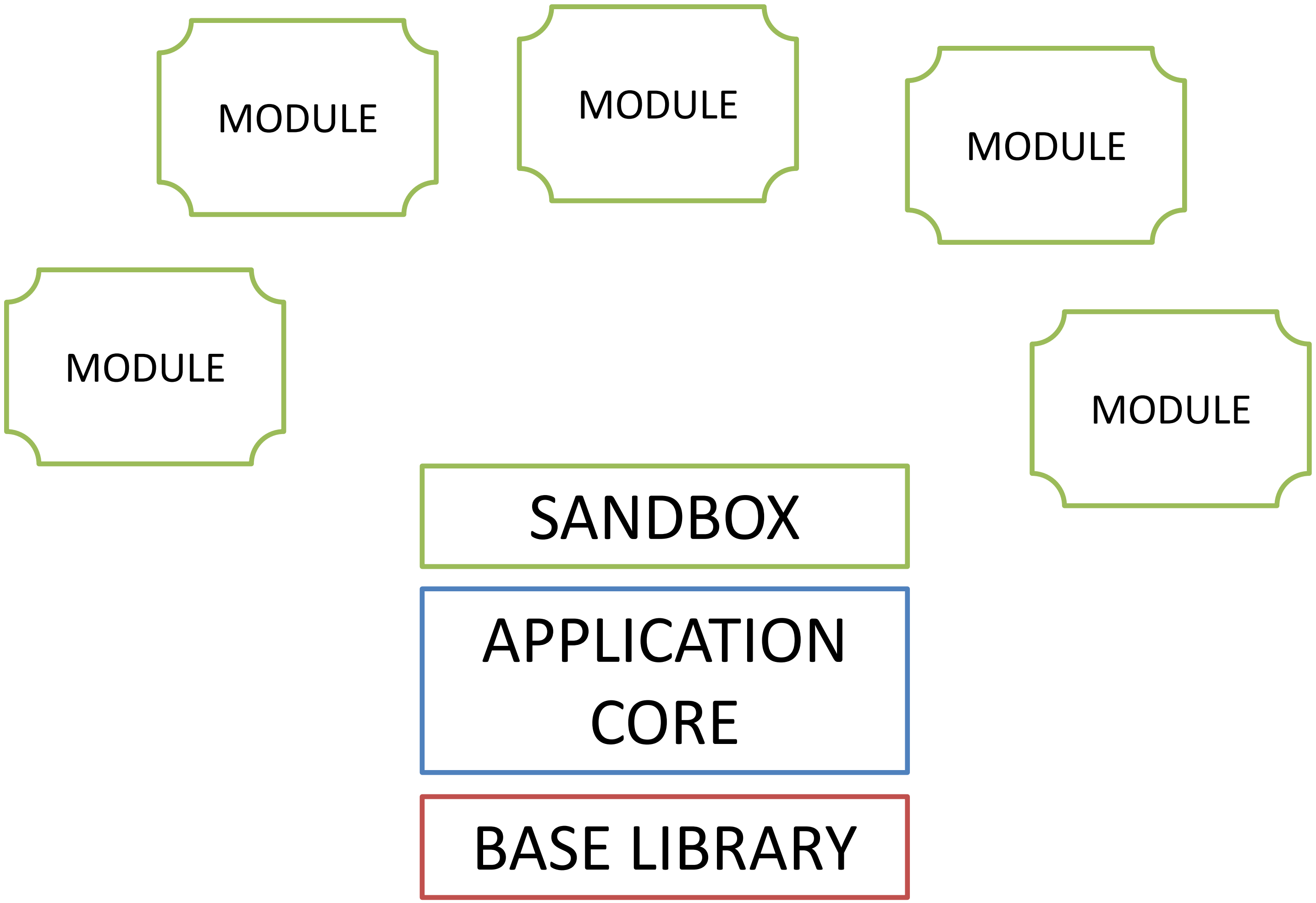
An independently operable unit that is part of the total structure

Any single module should be able to live on its own

Loose coupling allows you to make changes to one module without affecting the others

Each module has its own sandbox. An interface with which the module can interact to ensure loose coupling

Modules have limited knowledge


MODULE
MODULE
MODULE
MODULE
MODULE
SANDBOX
APPLICATION CORE
BASE LIBRARY

Module Rules

- Hands to yourself
  - Only call your own methods or those on the sandbox
  - Don’t access DOM elements outside of your box
  - Don’t access non-native global objects
- Ask, don’t take
  - Anything else you need, ask the sandbox
- Do not create global objects
- Donot directly reference other modules
- The sandbox ensures a consistent interface. Sandbox also acts as security layer

- Role of Sanbox
  - Consistency
  - Communication
  - Security
- Application Core Jobs
  - Manage module lifecycle
  - Enable inter-module communication
  - General error handling
  - Be extensible
- Extensions?
  - Error handling
  - Ajax communication
  - New Module capabilities
  - General utilities

Module Pattern

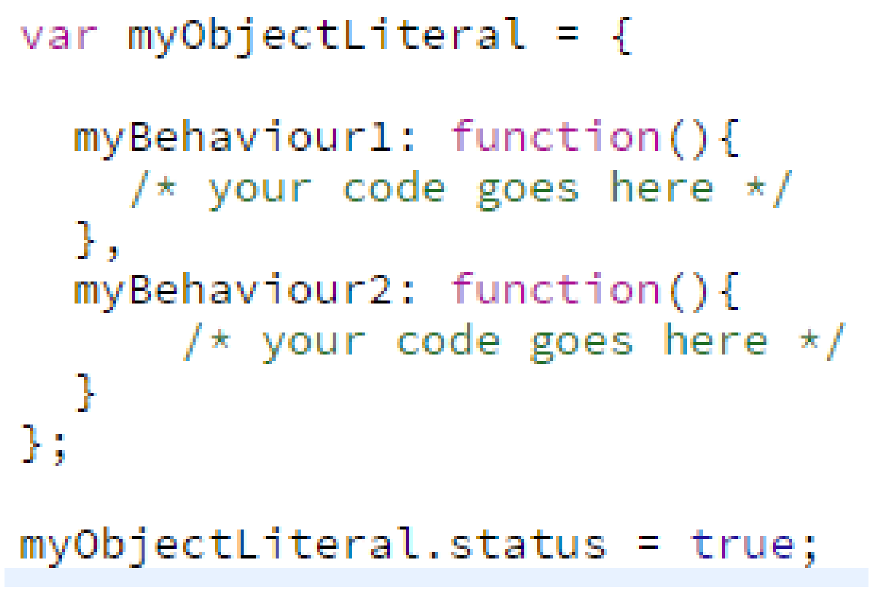
Modules are an integral piece of any robust application's architecture and typically help in keeping the units of code for a project both cleanly separated and organized

Implementation in javascript

- The Module pattern
- Object Literal Notation
- AMD modules
- CommonJS modules
- ECMA Script Harmony modules

Object Literal Notation

- They do not require instantiation using the **new operation**
- **Outside of an object, new members may be added to it using assignment**
- **They assist in encapsulating and organizing your code**



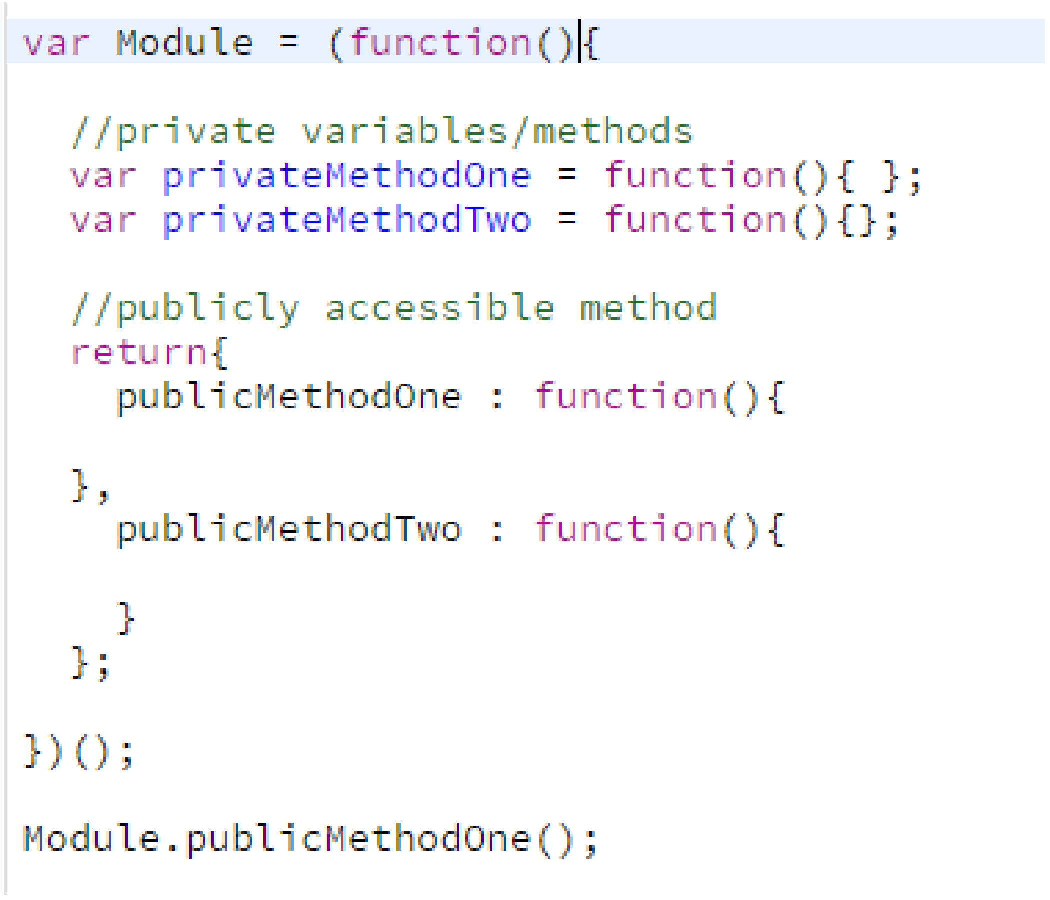
```
var myObjectLiteral = {

  myBehaviour1: function(){
    /* your code goes here */
  },
  myBehaviour2: function(){
      /* your code goes here */
  }
};

myObjectLiteral.status = true;
```

The Module Pattern

- DisAdvantages
  - Private methods are unaccessible
  - Private methods and functions loose extendability since they are unaccessible
- Advantages
  - Cleaner approach for developers
  - Supports private data
  - Less clutter in the global name space
  - Localization of functions and variable through closures

Creating a module

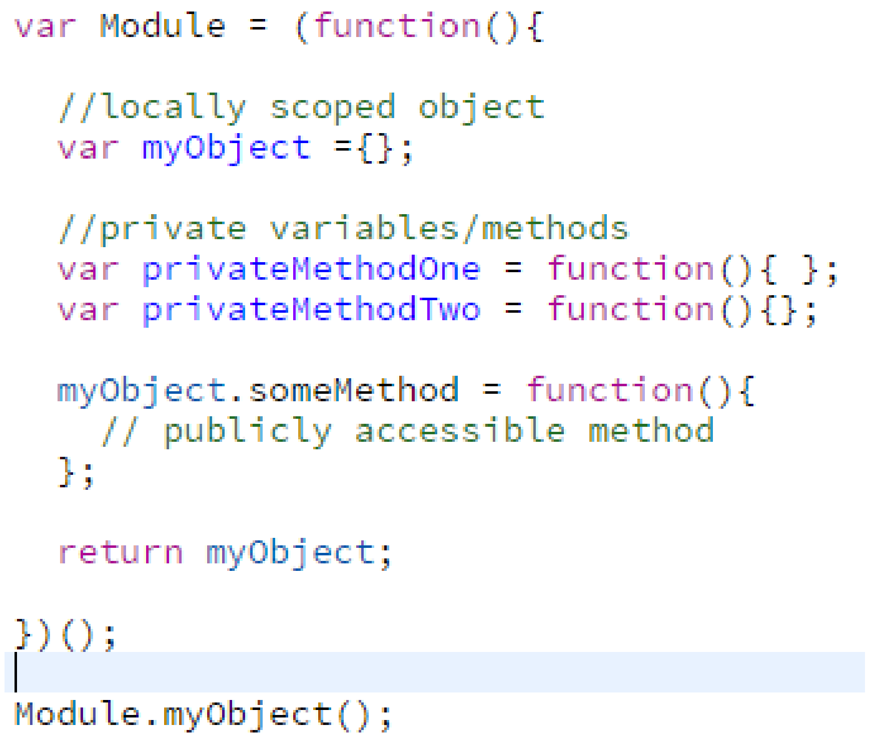
1. Define an anonymous IIFE
2. Assign it to a module
3. Define private variables/methods within the lexical scope of IIFE
4. Return public methods
5. Access public methods using module.objectliteral



```
var Module = (function(){

  //private variables/methods
  var privateMethodOne = function(){ };
  var privateMethodTwo = function(){};

  //publicly accessible method
  return{
    publicMethodOne : function(){

  },
    publicMethodTwo : function(){

    }
  };

})();

Module.publicMethodOne();
```

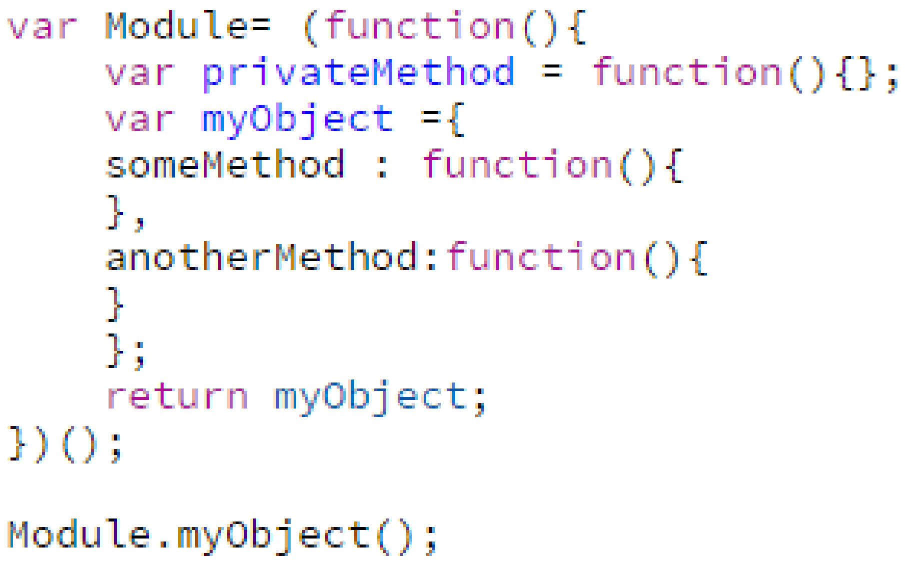
Module Example

Locally scoped object literal



```
var Module = (function(){

  //locally scoped object
  var myObject ={};

  //private variables/methods
  var privateMethodOne = function(){ };
  var privateMethodTwo = function(){};

  myObject.someMethod = function(){
    // publicly accessible method
  };

  return myObject;

})();

Module.myObject();
```

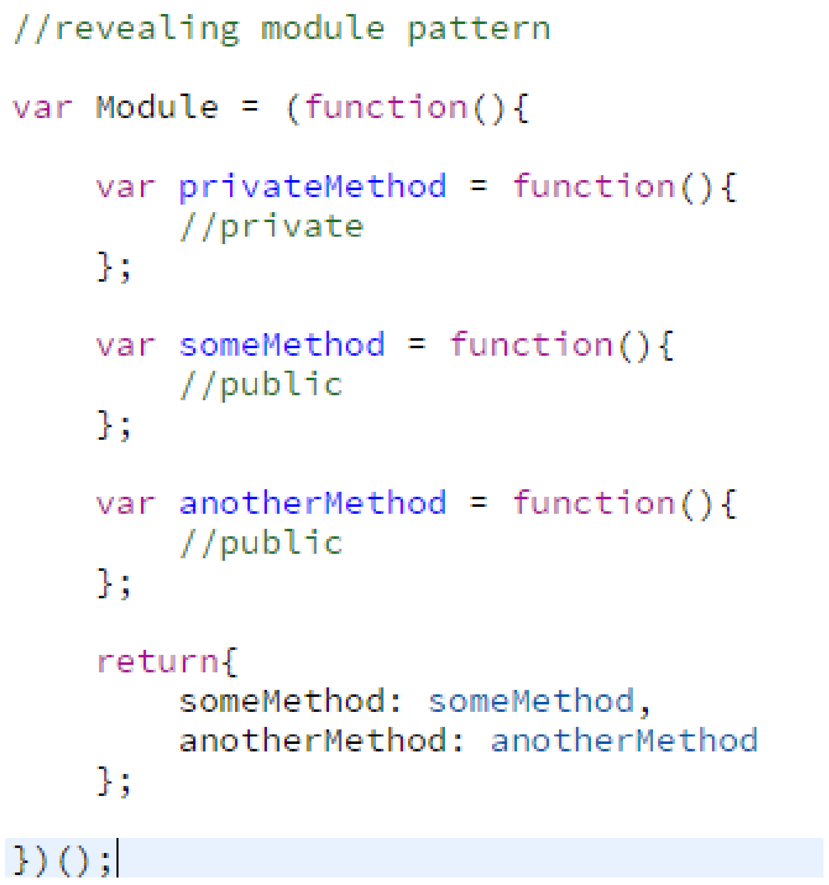
Stacked locally scoped object literal



```
var Module= (function(){
    var privateMethod = function(){};
    var myObject ={
    someMethod : function(){
    },
    anotherMethod:function(){
    }
    };
    return myObject;
})();

Module.myObject();
```

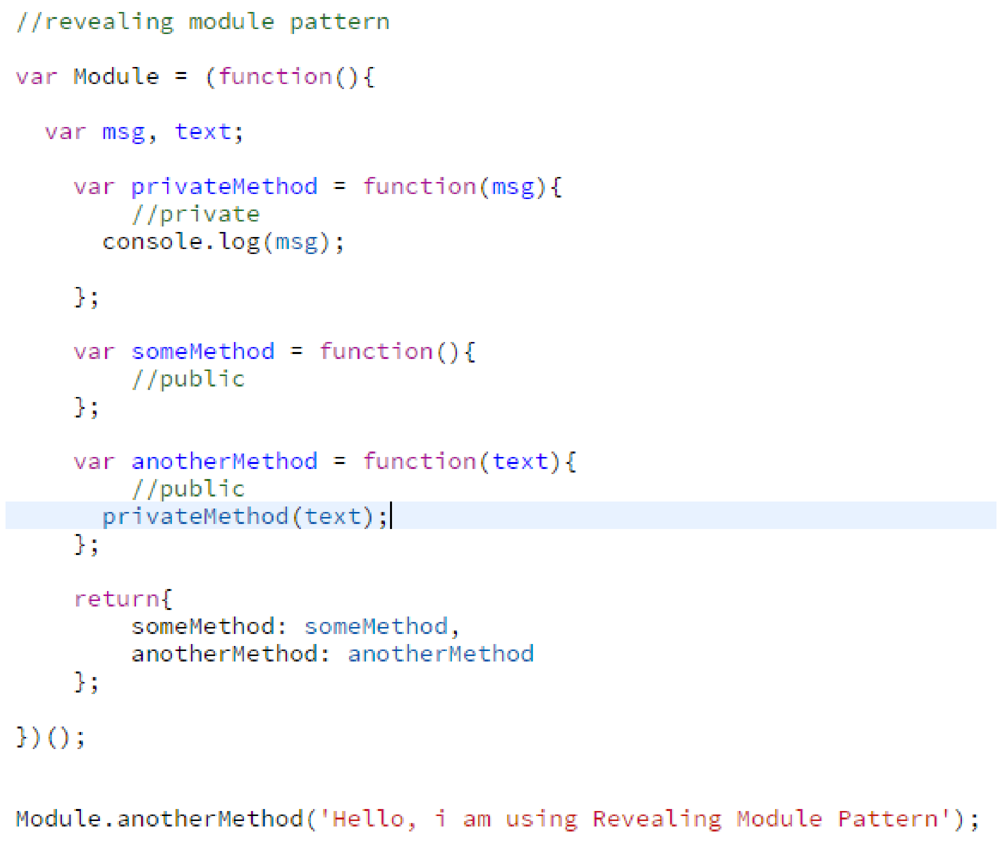
Revealing Module Pattern

- A **NEAT variant** of module pattern, which reveals public points to methods inside the **Module's SCOPE.**
- **Helps create nice code management system in which you can clearly see and define which methods are shipped back to the Module**

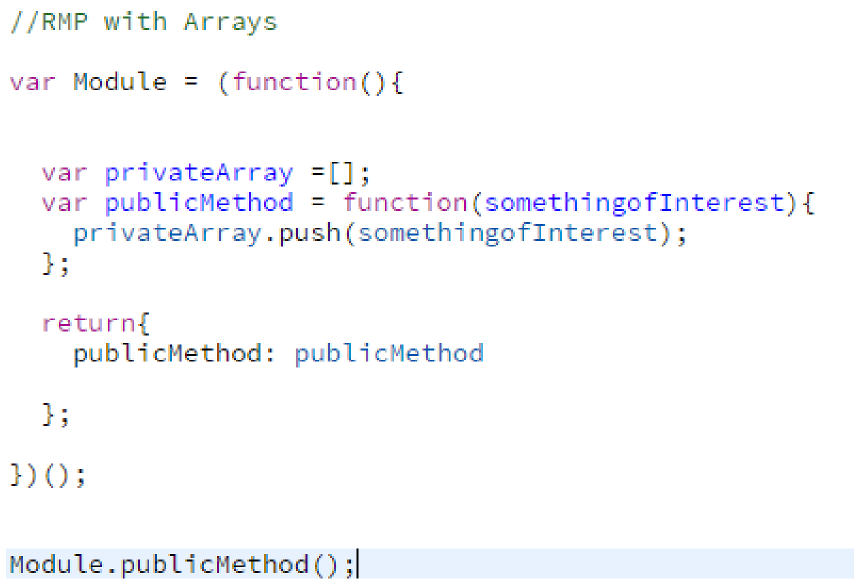


```
//revealing module pattern

var Module = (function(){

    var privateMethod = function(){
        //private
    };

    var someMethod = function(){
        //public
    };

    var anotherMethod = function(){
        //public
    };

    return{
        someMethod: someMethod,
        anotherMethod: anotherMethod
    };

})();
```

Revealing module Pattern



```
//revealing module pattern

var Module = (function(){

  var msg, text;

    var privateMethod = function(msg){
        //private
      console.log(msg);

    };

    var someMethod = function(){
        //public
    };

    var anotherMethod = function(text){
        //public
      privateMethod(text);
    };

    return{
        someMethod: someMethod,
        anotherMethod: anotherMethod
    };

})();


Module.anotherMethod('Hello, i am using Revealing Module Pattern');
```



```
//RMP with Arrays

var Module = (function(){


  var privateArray =[];
  var publicMethod = function(somethingofInterest){
    privateArray.push(somethingofInterest);
  };

  return{
    publicMethod: publicMethod

  };

})();


Module.publicMethod();
```

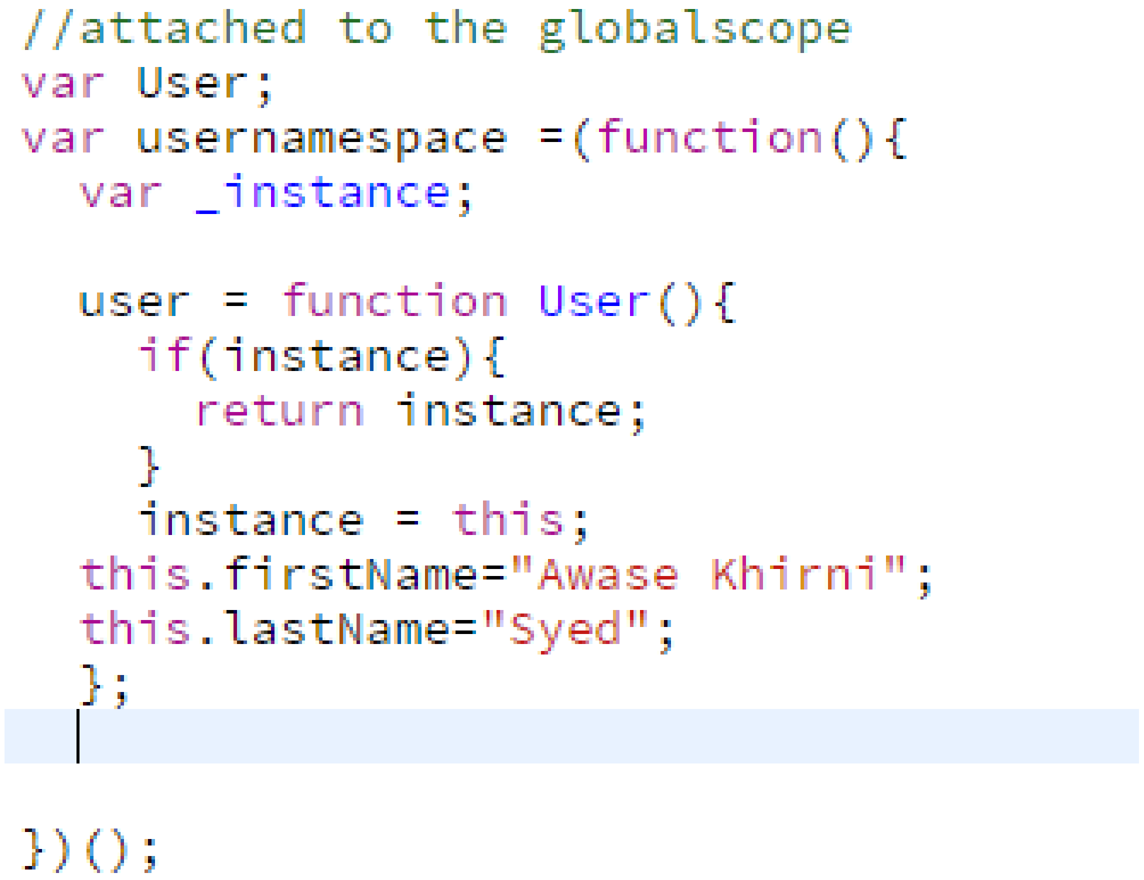
The Singleton Pattern

This pattern restricts instantiation of an object to a **single reference thus reducing its memory footprint and allowing a "delayed" initialization on an as-needed basis**

- **Advantages**
  - **Reduced memory footprint**
  - **Single point of access**
  - **Delayed initialization that prevents instantiation until required**

- Disadvantages
  - Once instantiated, they are hardly ever "reset"
  - Harder to unit test and sometimes introduces hidden dependencies

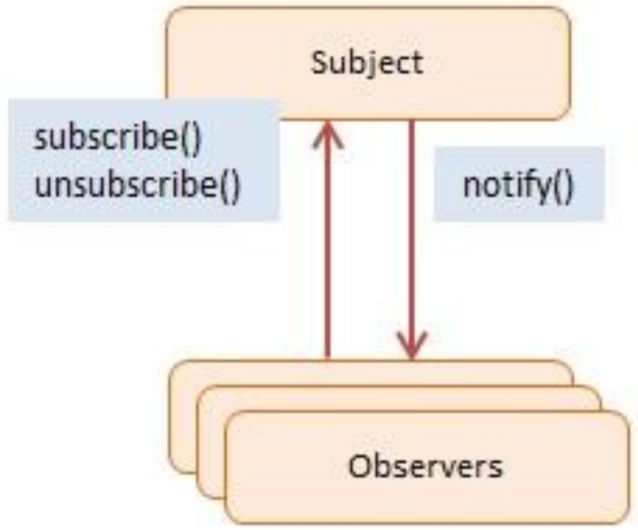
singleton



```
//attached to the globalscope
var User;
var usernamespace =(function(){
  var _instance;

  user = function User(){
    if(instance){
      return instance;
    }
    instance = this;
  this.firstName="Awase Khirni";
  this.lastName="Syed";
  };


})();
```

Observer pattern

- It implements a single object(the subject) that maintains a reference to a collection of objects (known as “observers) and broadcasts notifications when a change to state occurs.
- When we don’t want to observe an object, we simply remove it from the collection of objects being observed.
- “Define a one-to-many dependency between objects so that one object changes state, all its dependents are notified and updated automatically”



The Observer Pattern

Advantages

- Requires deeper-level thinking of the relationship between the various components of an application
- Helps us pinpoint dependencies
- Excellent at decoupling objects which often promotes smaller, reusable components

Disadvantages

- Checking the integrity of your application can become difficult
- Switching a subscriber from one publisher to another can be costly

Observer Pattern

- Objects participating in this pattern are
  - Subject
    - Maintains list of observers. Any number of observer objects may observe a subject.
    - Implements an interface that lets observer objects subscribe or unsubscribe
    - Sends a notification to its observers when its state changes
- Observers
  - Has a function signature that can be invoked when **subject** changes (i.e. Event occurs)

Coercion

In computer science, type conversion, typecasting, and coercion are different ways, of implicitly or explicitly changing an entity of one data type into another

http://jscoercion.qfox.nl/

Coercion

Primitive data types

- Undefined (variables has been declared but not yet assigned a value)
- String
- Number
- Boolean
- Object
- **Function**
- **Null (empty values)**

- Javascript can convert data types behind the scenes to complete an operation, known as coercion/type coercion
  - Implicit coercion
  - Explicit coercion

Natives

- String
- Number
- Boolean
- Function
- Object
- Array
- RegExp
- Date
- Error

- Sub-type of the Object