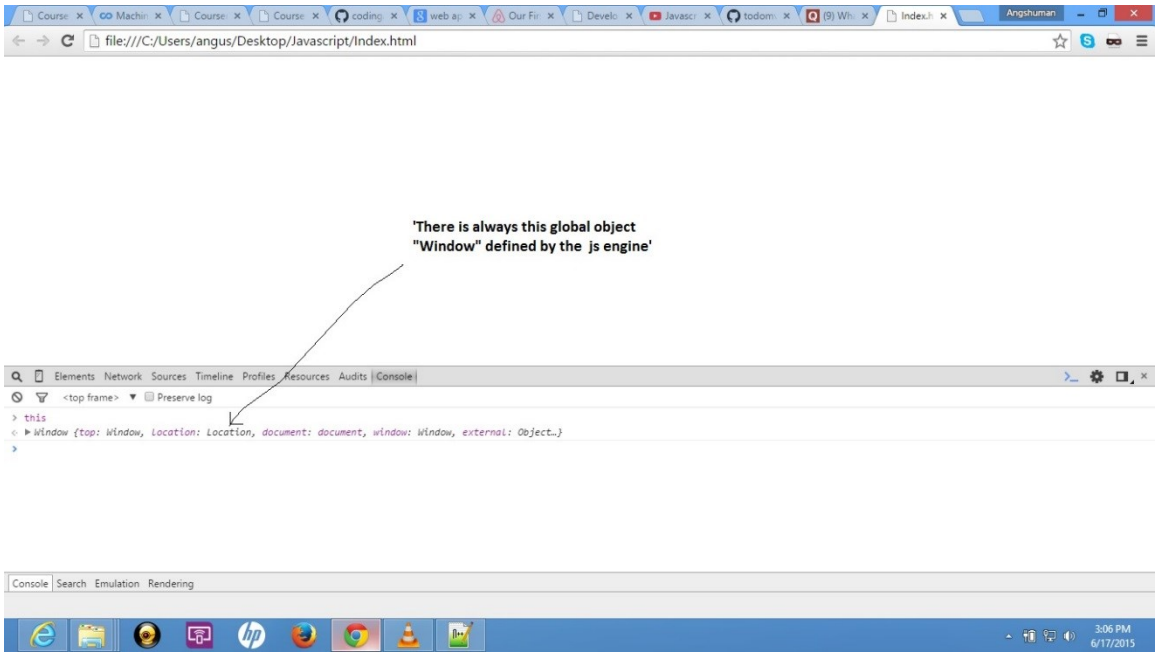
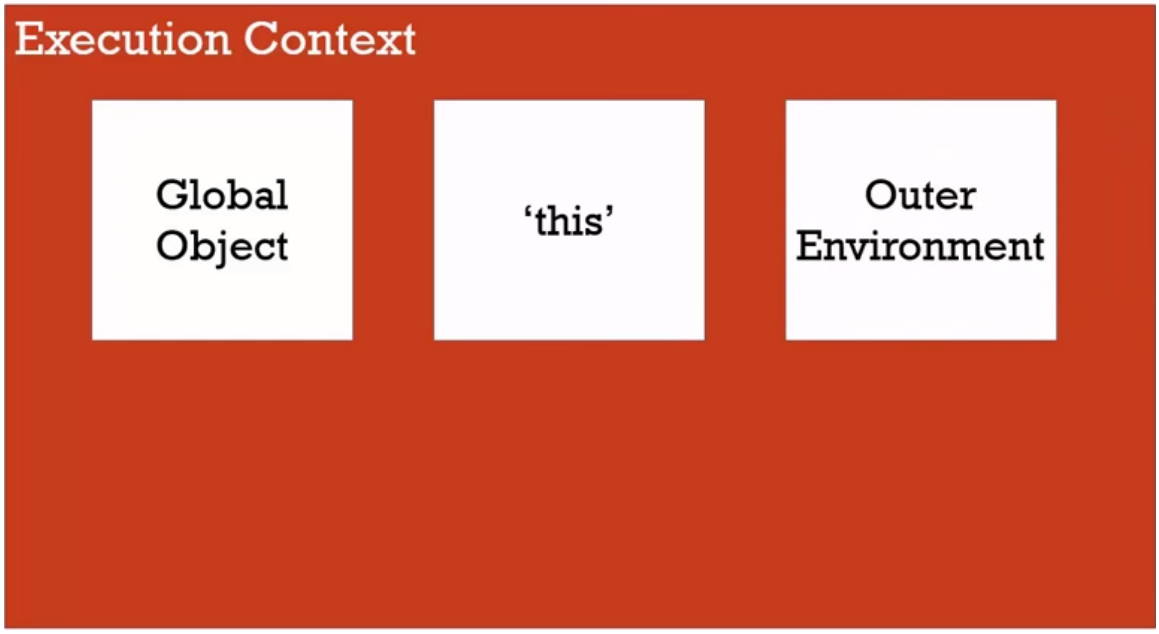
Chapter 1: Introduction (Preview)

-Compiled by: Angshuman Gupta

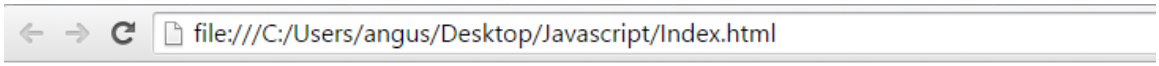


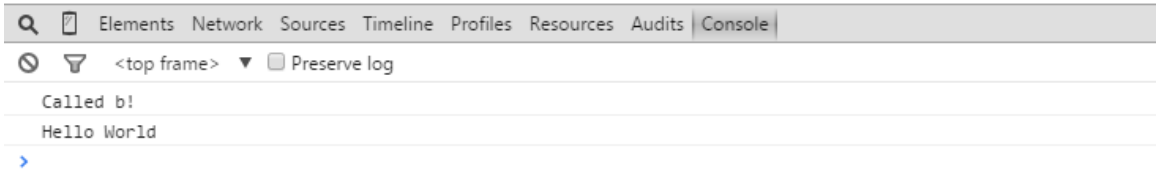

```
var a = 'Hello World';

function b ()
{

}
```



This Execution Context is the wrapper that wraps the code.

```
var a = 'Hello World';

function b ()
{
    console.log('Called b!');
}

b();
console.log(a);
```





Hoisting: Acts from the top

```
b();
console.log(a);

var a = 'Hello World';

function b ()
{
    console.log('Called b!');
}
```

Elements Network Sources Timeline Profiles Resources Audits Console

<top frame> Preserve log

```
Called b!
undefined
>
```

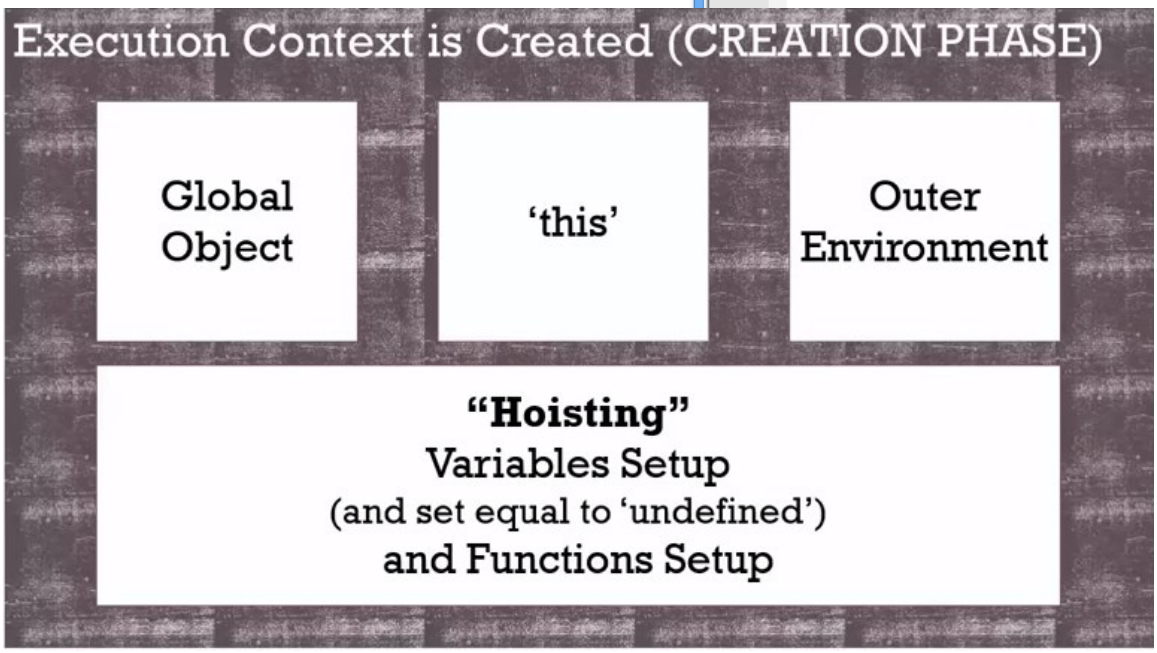
Elements Network Sources Timeline Profiles Resources Audits Console

<top frame> Preserve log

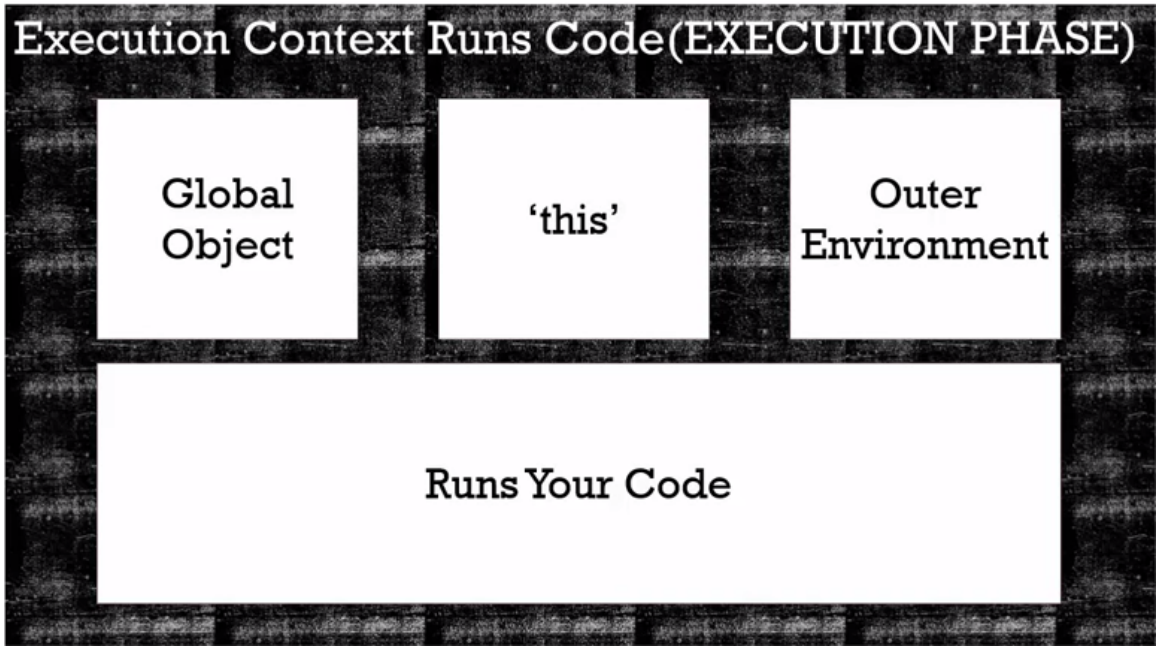
```
Called b!
Uncaught ReferenceError: a is not defined
>
```

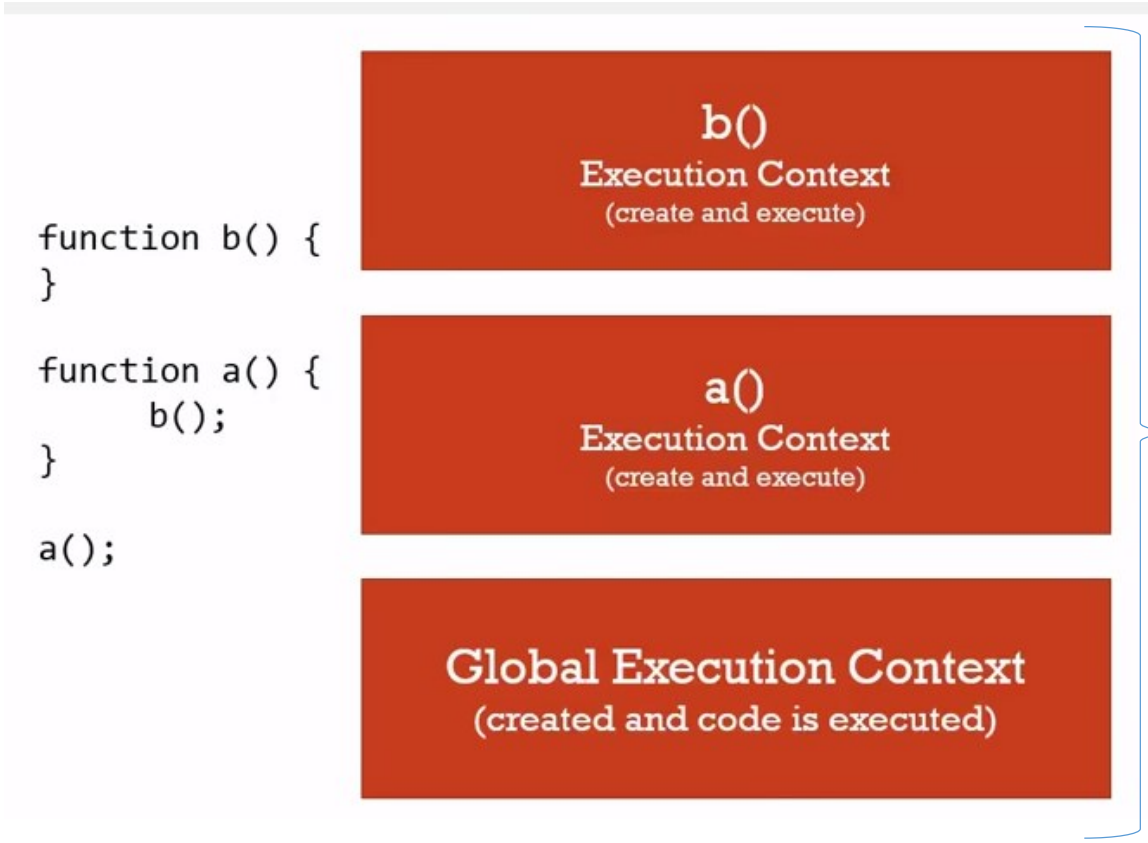
wp-login.php YouNoodle Entry Details.html logo.xml

```
b();
console.log(a);

//var a = 'Hello World';

function b ()
{
    console.log('Called b!');
}
```



JavaScript makes a placeholder and defines all the variables and functions as 'undefined' before its execution.


Execution Context Runs Code(EXECUTION PHASE)
Global Object
'this'
Outer Environment
Runs Your Code


function b() {
}
function a() {
b();
}
a();
b()
Execution Context
(create and execute)
a()
Execution Context
(create and execute)
Global Execution Context
(created and code is executed)
Execution Stack (for function)

Elements Network Sources Timeline Profiles Resources Audits Console

<top frame> Preserve log

```
1
2
undefined
>
```

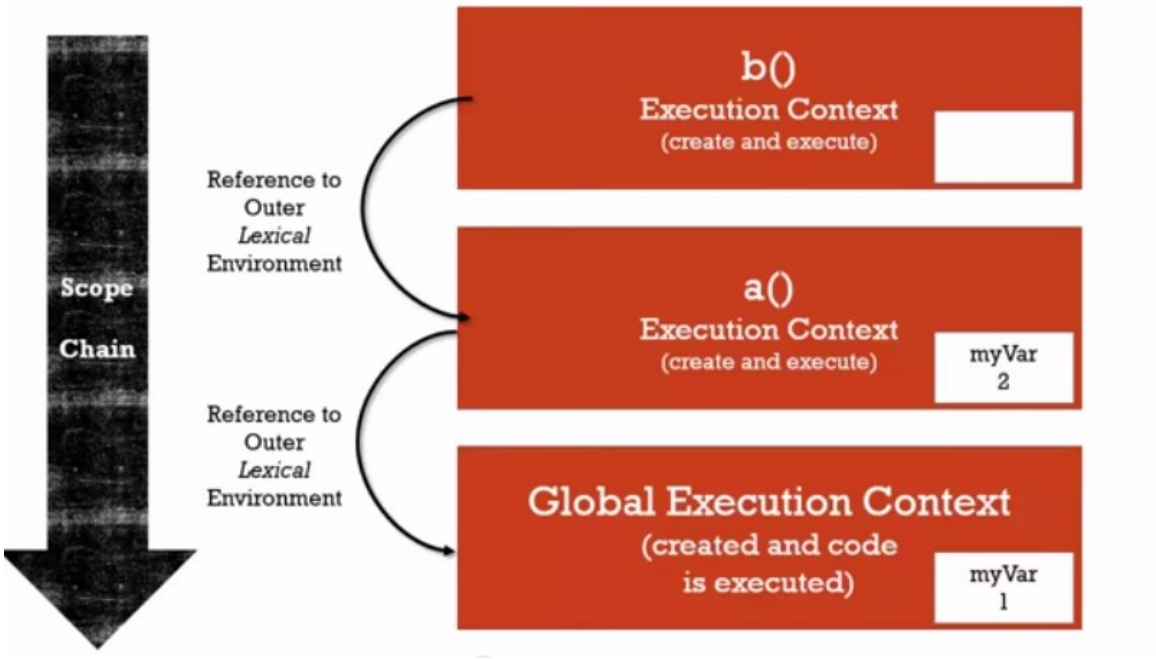
```
function b() {
    var myVar;
    console.log(myVar);
}

function a()
{
    var myVar=2;
    console.log(myVar);
    b();
}

var myVar = 1;
console.log(myVar);
a();
```

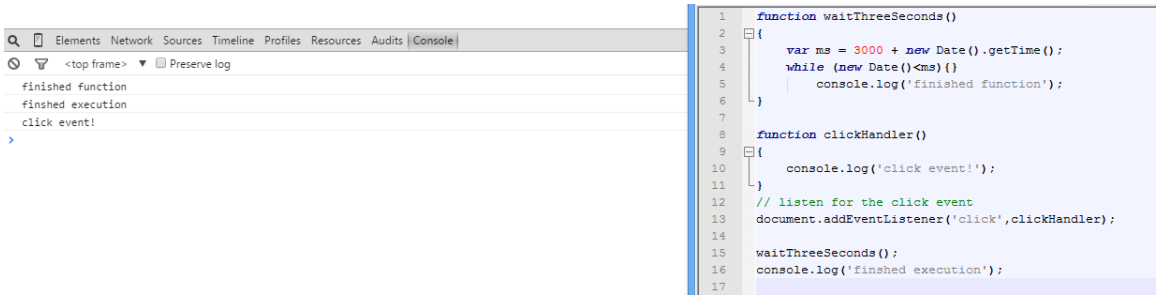
Elements Network Sources Timeline Profiles Resources Audits Console

<top frame> Preserve log

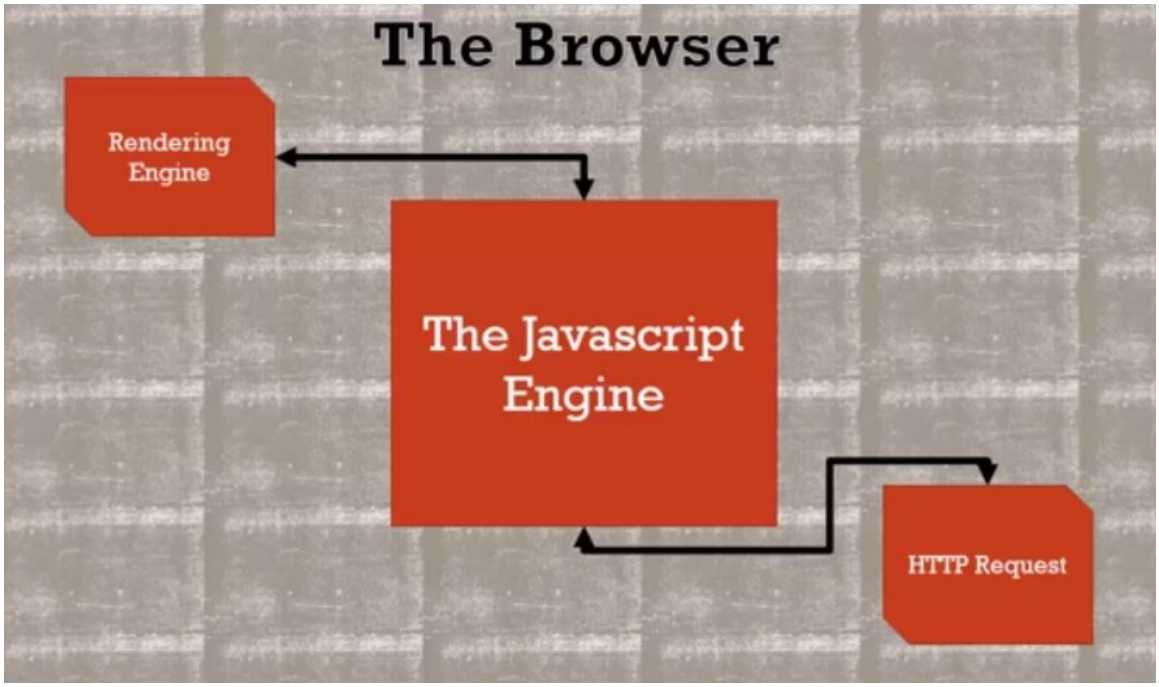
```
1
>
```

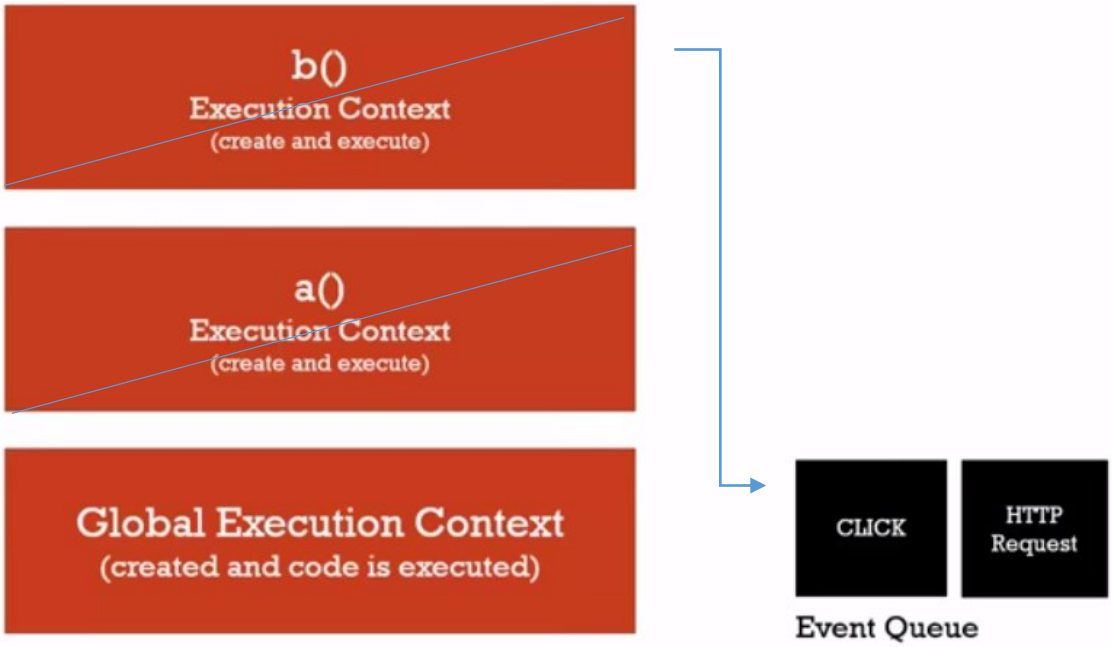
```
function b() {
    //var myVar;
    console.log(myVar);
}

function a()
{
    var myVar=2;
    //console.log(myVar);
    b();
}

var myVar = 1;
//console.log(myVar);
a();
```





After completing the Execution Stack it looks in the Event Queue.





DYNAMIC TYPING:

YOU DON'T TELL THE ENGINE WHAT TYPE OF DATA A VARIABLE HOLDS, IT FIGURES IT OUT WHILE YOUR CODE IS RUNNING

Variables can hold different types of values because it's all figured out during execution.

Static Typing

```
bool isNew = 'hello'; // an error
```

Other Programming Languages

Dynamic Typing

JS

```
var isNew = true; // no errors
isNew = 'yup!';
isNew = 1;
```

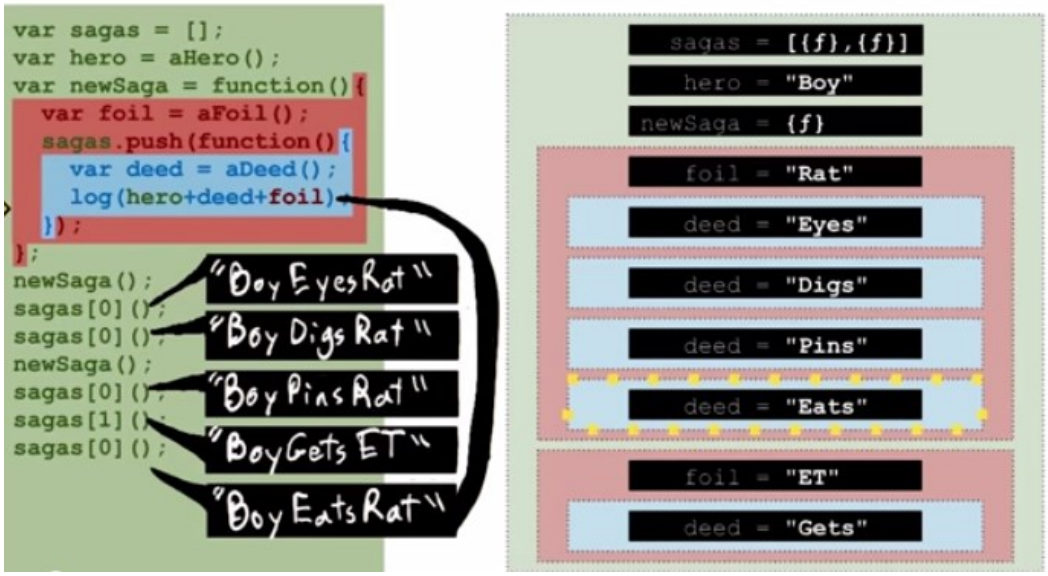
Chapter 2 : Scopes & Closures

`

1. Global Scope
2. Lexical Scope
3. Execution Context

Closure: Remains available to all the functions.

- Passing
- Return
- Global save



Chapter 3: This Keyword

This:The object found to the left of the dot where the containing function is called

If - object.method(a,y);
so, this = object value aka focal object

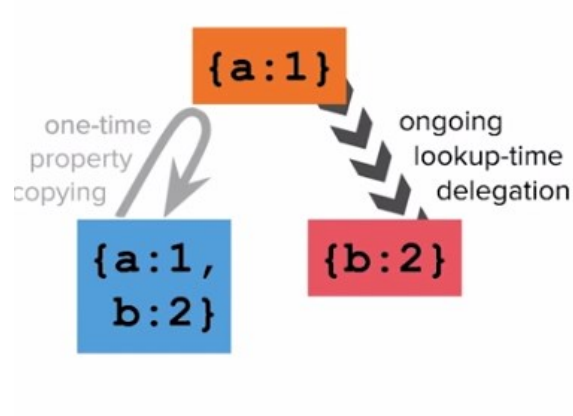
If a function is defined in the global scope, this would refer to the global scope i.e. its lexical scope.

Function.call(r,x,y) → this would get bind to r
Object.method.call(r,x,y) : this → r

Chapter 4 : Prototype Chain

Its inheritance.

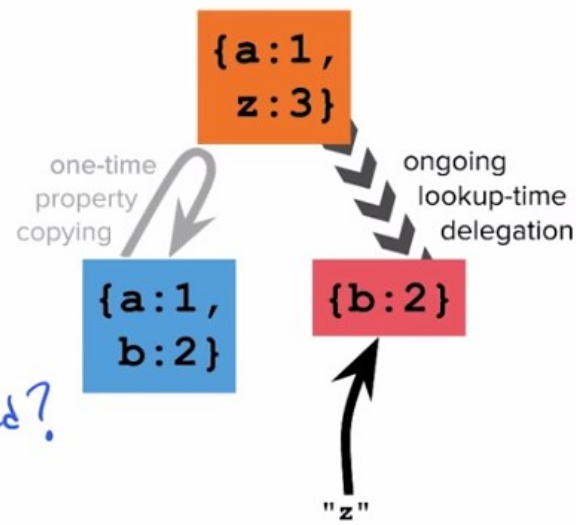
```
Var gold = {a:1};
Var blue = extend({},gold); //copying all aka one time property copying
Var rose = Object.create(gold); //linkage aka lookup
```




```
var gold = {a:1};
log(gold.a); // 1
log(gold.z); // undefined

var blue = extend({}, gold);
blue.b = 2;
log(blue.a); // 1
log(blue.b); // 2
log(blue.z); // undefined

var rose = Object.create(gold);
rose.b = 2;
log(rose.a); // 1
log(rose.b); // 2
log(rose.z); // undefined

gold.z = 3;
log(blue.z); // undefined
log(rose.z);
```

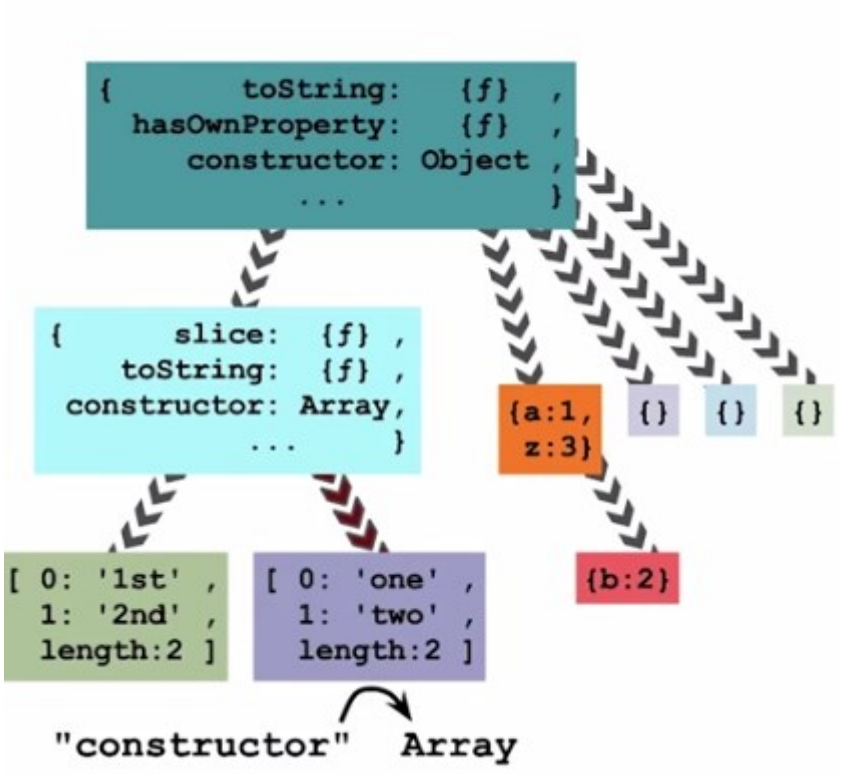
What will be logged?

3



Object Prototype:

Rose.toString();

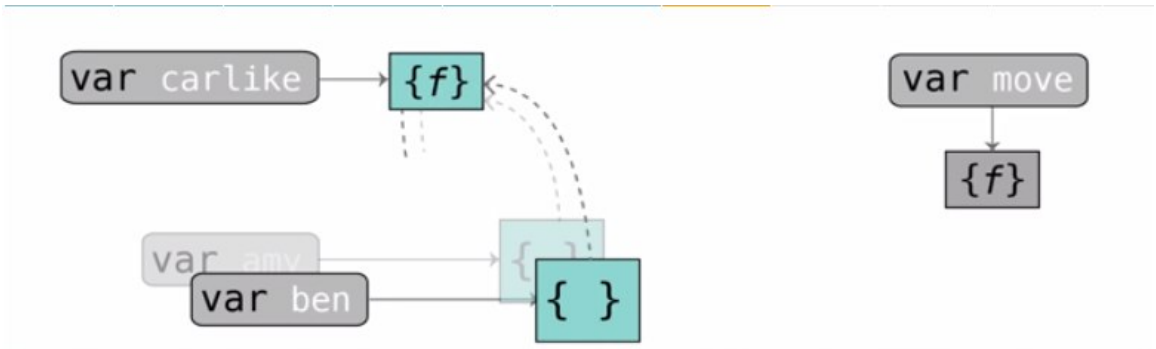

{ toString: {ƒ} ,
hasOwnProperty: {ƒ} ,
constructor: Object ,
... }
{ slice: {ƒ} ,
toString: {ƒ} ,
constructor: Array,
... }
{a:1,
z:3}
{}
{}
{}
[0: '1st' ,
1: '2nd' ,
length:2]
[0: 'one' ,
1: 'two' ,
length:2]
{b:2}
"constructor"
Array

Chapter 5 : Object Decorator Pattern> Code Reuse

```
var move = function(car){
  car.loc++;
};
```

```
var amy = {loc:1};
move(amy);
var ben = {loc:9};
move(ben);
```

Decorator Function- Naming Convention: Camel Case, here it is CarLike:




```
var carlike = function(obj, loc){
  obj.loc = loc;
  return obj;
};

var move = function(car){
  car.loc++;
};
```

```
var amy = carlike({}, 1);
move(amy);
var ben = carlike({}, 9);
move(ben);
```

```
var carlike = function(obj, loc){
  obj.loc = loc;
  obj.move = move;
  return obj;
};

var move = function(){
  this.loc++;
};
```

```
var amy = carlike({}, 1);
amy.move();
var ben = carlike({}, 9);
ben.move();
```

this:

```
var ob1 = {};
var ob2 = {};
ob1.example = function(arg1){
  log(this, arg1);
};
ob1.example(ob2); // logs ob1 and ob2
```

```
library.js
var carlike = function(obj, loc){
  obj.loc = loc;
  obj.move = function(){
    this.loc++;
  };
  return obj;
};
```

```
run.js
var amy = carlike({}, 1);
amy.move();
var ben = carlike({}, 9);
ben.move();
```

```
library.js
var carlike = function(obj, loc){
  obj.loc = loc;
  obj.move = function(){
    obj.loc++;
  };
  return obj;
};
```

```
run.js
var amy = carlike({}, 1);
amy.move();
var ben = carlike({}, 9);
ben.move();
```

Chapter 6 : Introduction to Classes

Difference between decorator function and classes: Decorator function accepts the object which it wants to augments but a class builds the objects which it wants to augments.

Class: Is a construct that can build a fleet of objects. Naming Convention- Capitalized Noun, Proper Noun

```
library.js
var Car = function(loc){
  var obj = {loc: loc};
  obj.move = function(){
    obj.loc++;
  };
  return obj;
};
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

The function of the Class is called a constructor function as it constructs the member of the Class.

The arrow represents instantiating (instance of a class).

To avoid duplicity of creating different object creation for each call.

```
library.js
var Car = function(loc){
  var obj = {loc: loc};
  obj.move = move;
  return obj;
};

var move = function(){
  this.loc++;
};
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

To avoid multiple declaration of a new object inside the class for a new function is better to declare them as object.

```
library.js
var Car = function(loc){
  var obj = {loc: loc};
  extend(obj, methods);
  return obj;
};
var methods = {
  move : function(){
    this.loc++;
  },
  on  : function(){ /*...*/ },
  off : function(){ /*...*/ }
};
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

Note Extend doesn't work like that refer link: http://www.2ality.com/2012/01/js-inheritance-by-example.html

Using methods as a property of Car:

```
library.js
var Car = function(loc){
  var obj = {loc: loc};
  extend(obj, Car.methods);
  return obj;
};
Car.methods = {
  move : function(){
    this.loc++;
  }
};
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

Chapter 7: Prototypal Class

```
library.js
var Car = function(loc){
  var obj = Object.create(Car.methods);
  obj.loc = loc;
  return obj;
};
Car.methods = {
  move : function(){
    this.loc++;
  }
};
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

We don't need to copy properties anymore, prototyping is giving a linkage.

Steps for prototyping:

1. Function that allows to make instances
2. Line in that function that generate the instance object
3. Delegation from new object to some prototype object
4. And some logic to augment that object to make it unique from all other objects of same class.

Since it's so common, JavaScript creates a default container attached to an object whenever a function is created

```
library.js
var Car = function(loc){
  var obj = Object.create(Car.prototype);
  obj.loc = loc;
  return obj;
};
Car.prototype.move = function(){
  this.loc++;
};
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

Don't get confused with the keyword prototype, it doesn't literally acts as a prototype but as a reference call.

```
library.js
var Car = function(loc){
  var obj = Object.create(Car.prototype);
  obj.loc = loc;
  return obj;
};
Car.prototype.move = function(){
  this.loc++;
};
console.log(Car.prototype.constructor);
console.log(amy.constructor);
log(amy instanceof Car);
```

```
run.js
var amy = Car(1);
amy.move();
var ben = Car(9);
ben.move();
```

Chapter 8: Pseudo Classes

JavaScript doesn't have the traditional "classes" that lower-level languages like C++ and Java have. Instead, JavaScript does some tricks to allow you to write code as though it had these traditional classes. We call these "pseudo-classes".

Use of new operand generate the extra code:

```
library.js
var Car = function(loc){
  this = Object.create(Car.prototype);
  var obj = Object.create(Car.prototype);
  obj.loc = loc;
  return obj;
  return this;
};
Car.prototype.move = function(){
  this.loc++;
};

run.js
var amy = Car(1);
amy.move();
var ben = new Car(9);
ben.move();
```

So finally the code turn to:

```
library.js
var Car = function(loc){
  this = Object.create(Car.prototype);

  this.loc = loc;

  return this;
};
Car.prototype.move = function(){
  this.loc++;
};

run.js
var amy = new Car(1);
amy.move();
var ben = new Car(9);
ben.move();
```

```
library.js
var Car = function(loc){
  this.loc = loc;
};
Car.prototype.move = function(){
  this.loc++;
};

run.js
var amy = new Car(1);
amy.move();
var ben = new Car(9);
ben.move();
```

Chapter 9: Super Classes and Subclasses

"Subclass" one object to another: This will give our new object the attributes of the original object. It will allow us to make further modifications to the new object without affecting the original object.

library.js

```
var Car = function(){
  var obj = {loc: loc};
  obj.move = function(){
    obj.loc++;
  };
  return obj;
};

var Van = function(loc){
  var obj = Car(loc);
  obj.grab = function{ /*...*/ };
  return obj;
};

var Cop = function(loc){
  var obj = Car(loc);
  obj.call = function(){ /*...*/ };
  return obj;
};
```

run.js

```
var amy = Van(1);
amy.move();
var ben = Van(9);
ben.move();
var cal = Cop(2);
cal.move();
cal.call();
```

Car here is the Super Class.

Creation of subclasses from pseudo classes:

library.js

```
var Car = function(loc){

  this.loc = loc;
};
Car.prototype.move = function(){
  this.loc++;
};

var Van = function(loc){
  this = Object.create(Van.prototype);
  Car.call(this, loc);
};
```

run.js

```
var zed = new Car(3);
zed.move();

var amy = new Van(9);
amy.move();
amy.grab();
```

Below is an incorrect definition:

```js
var Car = function(loc){
  this.loc = loc.valueOf();
};
Car.prototype.move = function(){
  this.loc++;
};

var Van = function(loc){
  Car.call(this, loc);
};
Van.prototype = new Car();
```

```js
var zed = new Car(3);
zed.move();

var amy = new Van(9);
console.log(amy.loc)
amy.move();
amy.grab();
```

What would happen when we ran this code?

- this.loc would equal undefined
- all cars would share the same .loc property forever
- the constructor would throw an error

Correct definition:

```js
var Car = function(loc){
  this.loc = loc;
};
Car.prototype.move = function(){
  this.loc++;
};

var Van = function(loc){
  Car.call(this, loc);
};
Van.prototype = Object.create(Car.prototype);
```

```js
var zed = new Car(3);
zed.move();

var amy = new Van(9);
console.log(amy.loc)
amy.move();
amy.grab();
```

Now amy.move() can instantiate from Car.prototype.move

Final code:

```js
var Car = function(loc){
  this.loc = loc;
};
Car.prototype.move = function(){
  this.loc++;
};

var Van = function(loc){
  Car.call(this, loc);
};
Van.prototype = Object.create(Car.prototype);
Van.prototype.constructor = Van;
Van.prototype.grab = function(){ /*...*/ };
```

```js
var zed = new Car(3);
zed.move();

var amy = new Van(9);
console.log(amy.loc)
amy.move();
amy.grab();
console.log(amy.constructor);
```

While we created a new Object. Create it destroyed the default .constructor prototype and its .constructor so new we had to create it again.

Useful Link: http://eloquentjavascript.net/