2017

Object-Oriented JavaScript

WRITING REUSABLE AND MAINTAINABLE CODE

HELI WANG

Based on https://www.udacity.com/course/object-oriented-javascript--ud015

Chapter 1: Prerequisite Knowledge

Scopes: Only new function create a new scope (not for while/if statements)

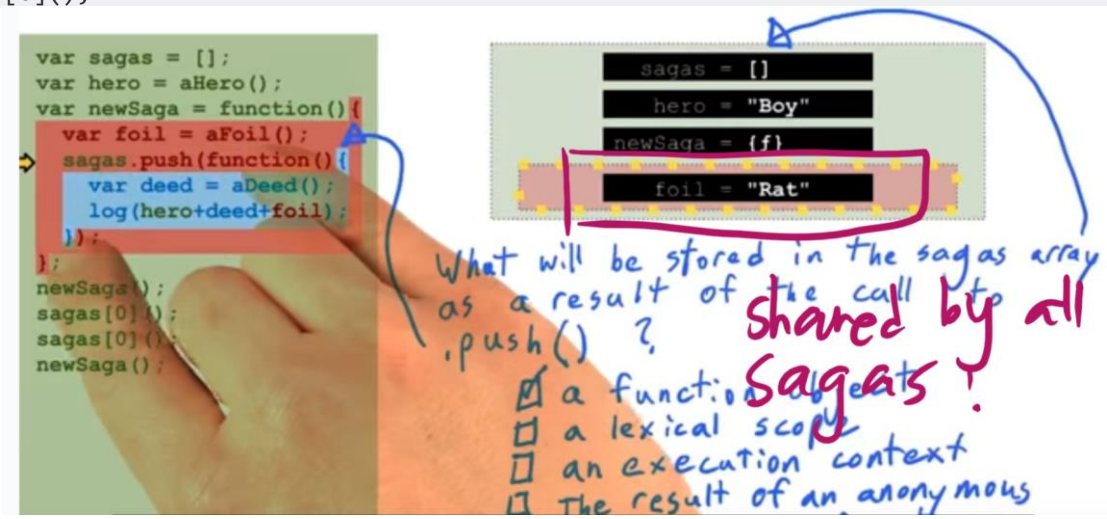
Closures: Every function should have access to all the variables from all the scioes that surround it. A closure is just any function that somehow remains available after those any outer scopes have returned.

Quiz: How to keep a reference of each of Saga() function during invocation of newSaga()?

```
var saga = function () {
  var a=1;
};

var newSaga = function () {
  saga();
}

newSaga();
```

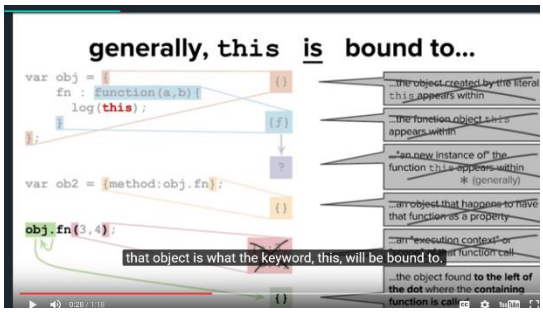
One possible solution: put each saga function to a global array

```
var sagas = [];
var hero = aHero(); // abstract function that randomly generates a HERO charachter
var newSaga = function() {
  var foil = aFoil(); // abstract function that randomly generates a FOIL charachter
  sagas.push(function() {
    var deed = aDeed(); // abstract function that randomly generates a DEED
charachter
    console.log(hero + deed + foil);
  });
};

// we only have an empty array SAGAS, HERO and a declared but not yet usen NEWSAGA
function

newSaga();
sagas[0]();
```



“This”:

There's a dot to the left of this example. When you notice a dot to the left of a function invocation, meaning it was looked up as a property of an object, you can look to the left of that dot and see what object it was looked up upon. The object that a function is looked up upon when it's being invoked, **that object is what the keyword, this, will be bound to.**




```
var fn = function(one, two){
  log(this, one, two);
};
var r={}, g={}, b={};

fn(g,b);
```

What will be logged?

Answer: this = Global

How to specify the value that you’d like the parameter this to get bound to?

fn.**call**(ObjectWantedToBeReferredAsThis, fnParameter1, fnParameter2)

```
var fn = function(one, two){
  log(this, one, two);
};
var r={}, g={}, b={}, y={};
r.method = fn;

r.method(g,b);
fn(g,b);
fn.call(r,g,b);
r.method.call(y,g,b);
setTimeout(fn, 1000);
setTimeout(r.method, 1000);
setTimeout(function(){
            r.method(g,b)
          }, 1000);
log(one);
log(this);
new r.method(g,b);
```

Take a guess! what will be logged?

A brand new object

```
// {}, {}, {}
// <global>, {}, {}
// {}, {}, {}
// {}, {}, {}
// <global>, undefined, undefined
// <global>, undefined, undefined
// {}
// th
// <gl
// {}, {}, {}
```

The keyword this will actually get bound to an entirely new object that gets created automatically in the background as a result of the keyword new having appeared.

So, the keyword this makes it possible for it to build just one function object and use that as a method on a bunch of other objects. And every time we call the method, it'll have access to whatever object it's being called on. This can be really useful for conserving memory, and it's only possible because we have access to the parameter this.

Delegation: Delegation is a technique that promotes code reuse by allowing runtime function invocation in the context of a specific instance – regardless of the hierarchical lineage of instance and function. JavaScript has excellent support for Delegation in the form of call and apply which lets us inject an object into the this value of any function. This permits unfeterred code sharing, free from the constraints of unwieldy, unnatural and overly complex hierarchies.

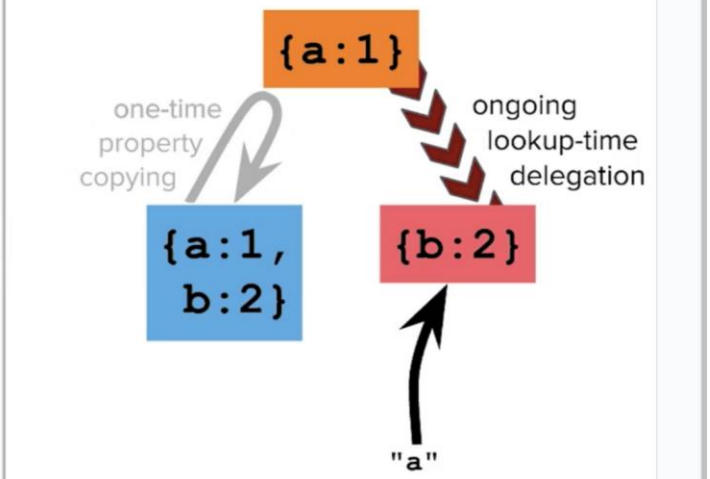
Prototype chains: Prototype chains are a mechanism for making objects that resemble other objects. When you want two objects to have all the same properties, either to save memory or to avoid code duplication, you might decide to copy every property over from one object to another. But as an alternative, JavaScript provides the option of prototype chains. This makes one object behave as if it has all the properties of the other object, by delegating the field lookups from the first object to the second one.

Var objB = **Object.created** (objA);
objB.propertyFromA; → Fall Back through sources of properties, up the chain to the prototype object. In another word, "delegation lookup". In this case, there is a prototype relationship between b and a.

What's a's prototype? ➔ Object Prototype e.g. a.toString();

```
1  var gold = {a:1};
2  log(gold.a); // 1
3  log(gold.z); // undefined
4
5  var blue = extend({}, gold);
6  blue.b = 2;
7  log(blue.a); // 1
8  log(blue.b); // 2
9  log(blue.z); // undefined
10
11 var rose = Object.create(gold);
12 rose.b = 2;
13 log(rose.a);
```

What do you think will be logged in this situation?



A number of available helper functions:
Object.prototype.hasOwnProperty()

Object.prototype.constructor() - Returns a reference to the Object constructor function that created the instance object.

```
var o = {};
o.constructor === Object; // true

var o = new Object;
o.constructor === Object; //true

var a = [];
a.constructor === Array; // true

var a = new Array;
a.constructor === Array //true

var n = new Number(3);
n.constructor === Number; // true
```

The following example creates a prototype, `Tree`, and an object of that type, `theTree`. The example then displays the `constructor` property for the object `theTree`.

```
function Tree(name) {
  this.name = name;
}

var theTree = new Tree('Redwood');

console.log('theTree.constructor is ' + theTree.constructor);
```

This example displays the following output:

```
theTree.constructor is function Tree(name) {
  this.name = name;
}
```

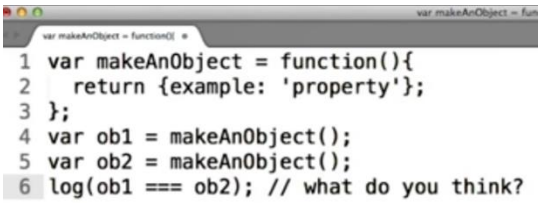
The `instanceof` operator

The `instanceof` operator tests presence of `constructor.prototype` in `object`'s prototype chain.

```
// defining constructors
function C() {}
function D() {}

var o = new C();

// true, because: Object.getPrototypeOf(o) === C.prototype
o instanceof C;

// false, because D.prototype is nowhere in o's prototype chain
o instanceof D;
```

```
o instanceof Object; // true, because:
C.prototype instanceof Object // true

C.prototype = {};
var o2 = new C();

o2 instanceof C; // true

// false, because C.prototype is nowhere in
// o's prototype chain anymore
o instanceof C;

D.prototype = new C(); // use inheritance
var o3 = new D();
o3 instanceof D; // true
o3 instanceof C; // true
```

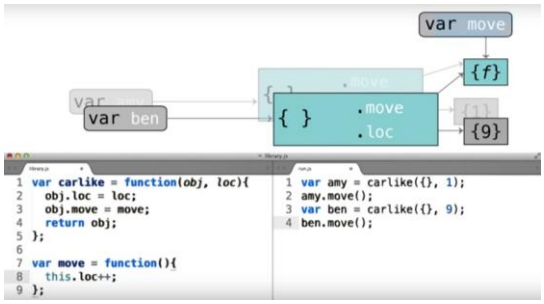
Strict Comparison of Objects

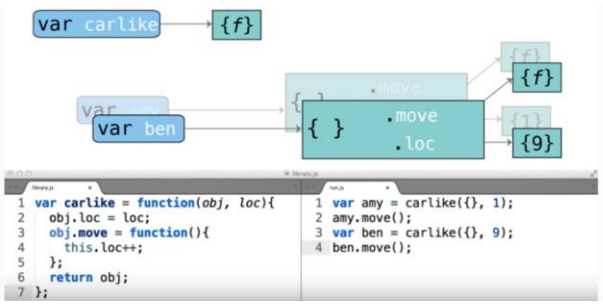


➔ **False**

Chapter 2: Prototype Classes

Object Decorator Pattern





Decorator Functions: add properties to objects

Obviously, we prefer putting move function outside carlike, so that all carlike instances could share one function.

Functional Classes

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

var move = function(){
  this.loc++;
};
var on  = function(){ /*...*/ };
var off = function(){ /*...*/ };
var ben = Car(9);
ben.move();
```

Decorator Code Vs Classes: The class builds the object that it's going to augment. The decorator accepts the object that it's going to augment as an input.

A class is a construct that is capable of building a fleet of similar objects that all conform to roughly the same interface.

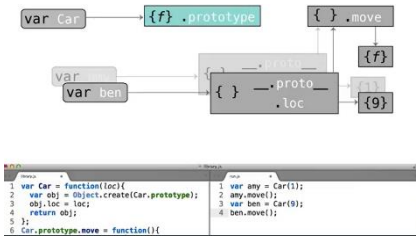
<< An example of a “functional class”.

However, methods in the functional class is not shared by all instances

Prototype Classes

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

The steps for making the class in this prototypal pattern are pretty clear. All you need is a function that allows you to make instances, a line in that function that generates the new instance object, a delegation from the new object to some prototype object, and some logic for augmenting the object with properties that make it unique from all the other objects of the same class.



Since this pattern is so common, the language designer decided to add official conventions to support it. JavaScript uses keyword “prototype” instead of “methods”

Ambiguity of Dot Prototype

Historically, the fact that this property is called dot prototype, has been very confusing for people learning JavaScript. By using this word to describe the methods container object, that's available on every function. JavaScript introduces an ambiguous 2nd meaning for the word prototype. So, if someone says, object one’s prototype is object two, reasonable interpretation would be to think that field lookups on this first object should fall through to the second one. So, you might **say Amy's prototype is car dot prototype**, but this is not the relationship that car has with car dot prototype.

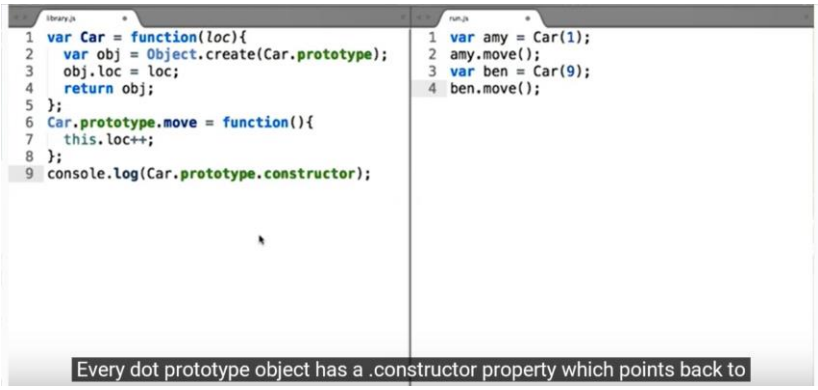
In that case, car is a function object and field lookups on it will fall through to some function prototype, where all function objects their field lookups. The car functions relationship with car-dot-prototype is very different from the one that Amy has with car-dot-prototype.

This other relationship reflects the second interpretation of the statement, "object one's prototype is object two". The relationship is that when a car function runs, it will create objects that delegate their

field lookups to car dot prototype. So, in this sense, you might say, cars prototype is car dot prototype.

So, to review**, saying Amy's prototype is car dot prototype means something very different from saying car's prototype is car dot prototype even though those sentences look so similar.**

We can leave this discussion behind, but be aware that it's extremely easy to conflate these two uses of the word prototype, and it's well worth investing extra thought in ironing out exactly what the differences between the two uses are.

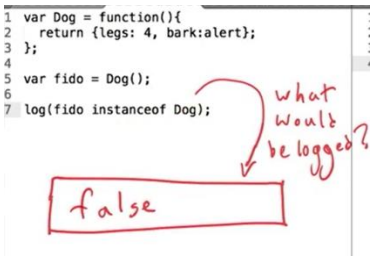


Every dot prototype object has a .constructor property which points back to the function it came attached to. Thus, Car.prototype links to Car's prototype. Car.prototype constructor is Car itself.

Quiz: What is the output of console.log(amy.constructor)?

In this case, Amy delegates the field-lookup for a .constructor property to Car.prototype which does has that property. So Amy's .constructor property is reported as being Car. Furthermore, the instanceof operator works by checking to see if the right operand's .prototype object can be found anywhere in the left operand's prototype chain. Here, Car.prototype can be found somewhere in amy's prototype chain. Thus (amy instanceof Car) == true.

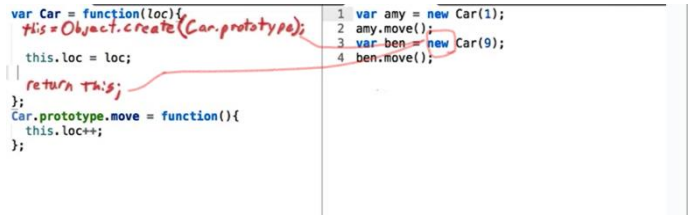
Quiz: What is the output of console.log(fido instanceof Dog)?



The answer is if we're writing in the functional style, the instanceof operator won't work. In this case, fido is a simple object that was created with an object literal, So it just delegates to object.prototype. Dog.prototype can't be found anywhere in fido's prototype chain.

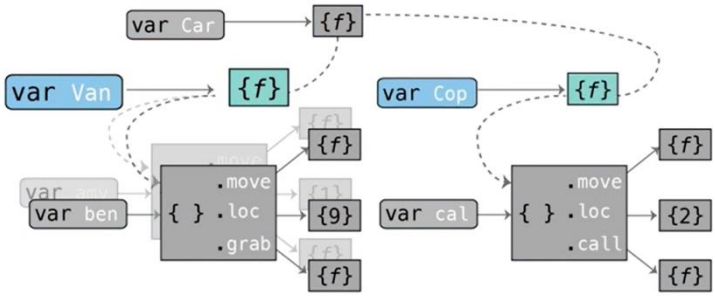
Chapter 3: Super-Classes and Sub-Classes

The new operator



The new operator creates an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

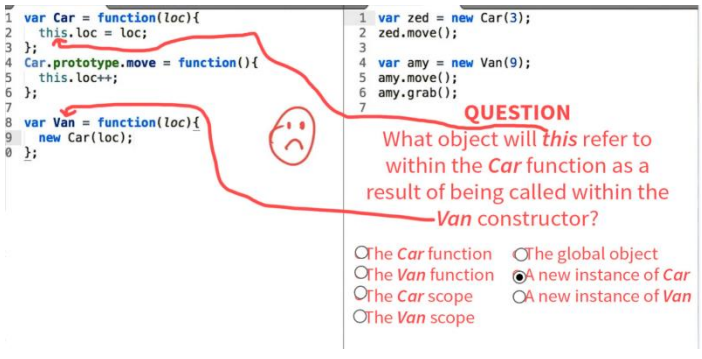
Super-Classes and Sub-Classes in Functional Pattern

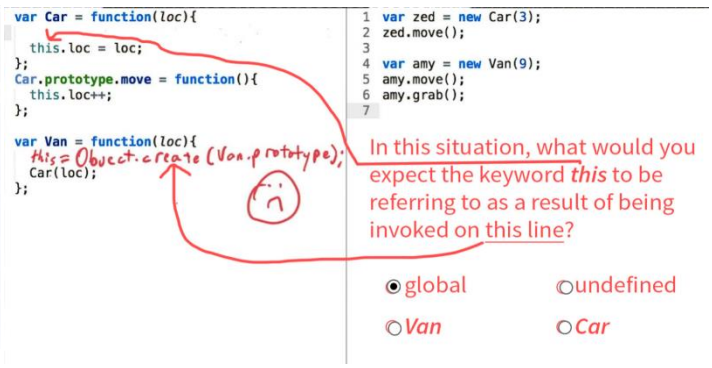


Looking at the diagram, you can see that we're achieving the exact same thing as before but with less repeated code. The Van and the Cop functions produce the same kinds of objects they always did, but they're allowing Car function to do most of the work for them first. But in addition to providing a basis for the Van and the Cop constructor functions, the Car function is also a full-fledged class itself. If my code ever needed to make simple Car objects that lack the more specific Van and Cop methods, then we could just call car directly. And with that we have completed our refractor to include a super class and two sub classes. Interestingly, the car class now looks exactly the way the van class did before we introduced the whole idea of cops into our program.

Pseudo-classical Super-Classes and Sub-Classes

Some common errors:





This is also not the correct version for a sub-class, because amy.move() will be null:

```
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);
};
```

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

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

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

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

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

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

Would you expect the lookup for amy.loc to work correctly now?

○ yes ○ no

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

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

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

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

What object does Van.prototype delegate to?

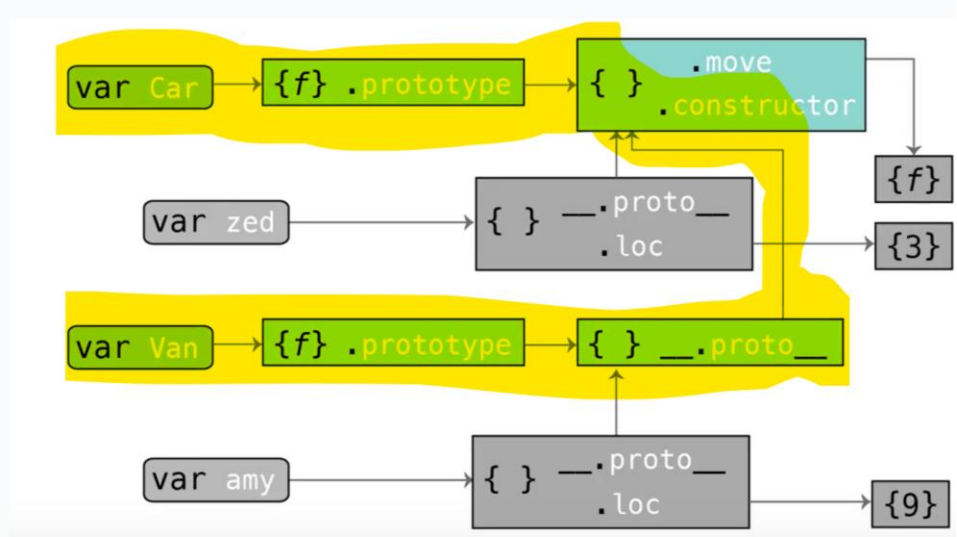
Object.prototype

The example below is also not correct, because there is no way for Van to override a function from Car

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

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

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

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

The example below is the correct version:




```
1  var Car = function(loc){
2    this.loc = loc;
3  };
4  Car.prototype.move = function(){
5    this.loc++;
6  };
7
8  var Van = function(loc){
9    Car.call(this, loc);
10 };
11 Van.prototype = Object.create(Car.prototype);
12 Van.prototype.constructor = ????
13 Van.prototype.grab = function(){ /*...*/ };
14
```

```
1  var zed = new Car(3);
2  zed.move();
3
4  var amy = new Van(9);
5  console.log(amy.loc)
6  amy.move();
7  amy.grab();
8
```

We don't want Van.prototype to be the exact same obj as for Car.prototype. If we want Van.prototype to be an object that delegates to Car.prototype we only need to pass it into the Object.create function and generate a new object that delegates there.

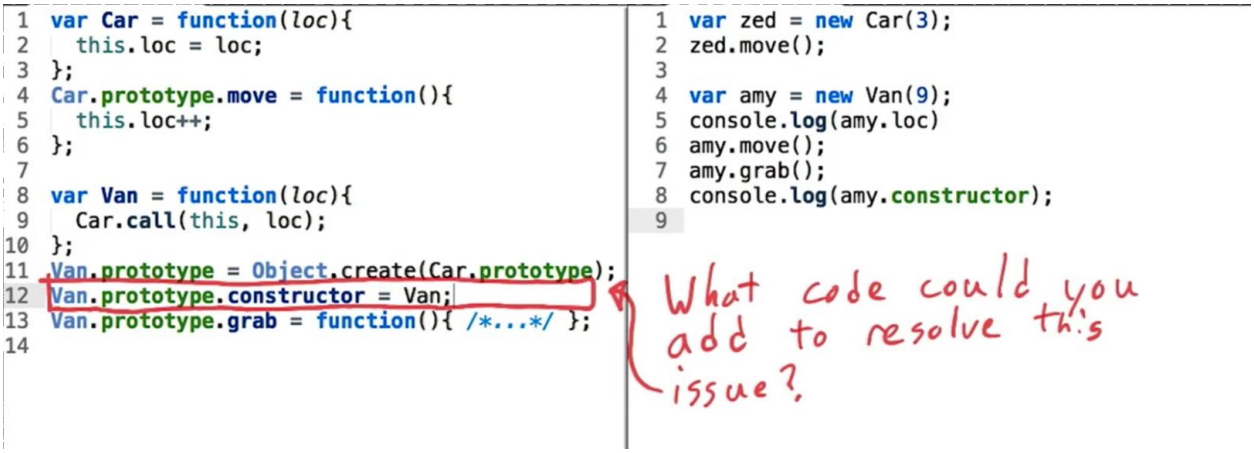
Lol, however, we need to add one line for **Subclass Constructor Delegation** to make the Van perfect.

```
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.grab = function(){ /*...*/ };
```

```
1  var zed = new Car(3);
2  zed.move();
3
4  var amy = new Van(9);
5  console.log(amy.loc)
6  amy.move();
7  amy.grab();
8  console.log(amy.constructor);
9
```

What would be logged by this line?

- amy
- Van
- Van.prototype
- zed
- Car
- car.prototype



Thanks. That's the End.

Congratulations.

At this point you've learned about the most important and difficult language features in JavaScript, including scopes, closures, the keyword this and prototype chains.

You also learned a lot of different code reuse patterns that are built on top of those features, such as function decorators and all of the major classing and subclassing patterns that are available within JavaScript.