Object-Oriented JavaScript Notes

https://www.udacity.com/course/viewer#!/c-ud015

This document summarizes content and augments some screenshots for reference.

Code examples for OOJS

Code examples with extra comments from the course:
https://github.com/batmanimal/object-oriented-js

Scopes

Lexical Scoping in JavaScript:

- 'The region in your source code where you can refer to variables by name without getting access errors.'
- New lexical scopes are created every time you make a new *function* definition (NOT in *if* statements).
- Variables defined *within* a lexical scope **cannot** be accessed *outside* that scope.
- Remember to use the `var` keyword when you make a new variable in a scope (it is not required by the language, but you should *always* do it! If you do not use `var`, the variable will be placed in the global namespace - don't use this 'feature' to accomplish that, it's bad).

The global scope is shared between (.js) files.

Execution Contexts ('in-memory scopes') in JavaScript:

- A new execution context is created each time you run a function.

Closures

Closures in JavaScript

- 'A closure is any function that remains available after any outer scopes have returned.'
- The most useful feature of closures is variable access, functions within an inner scope have access to variables defined in an outer scope:

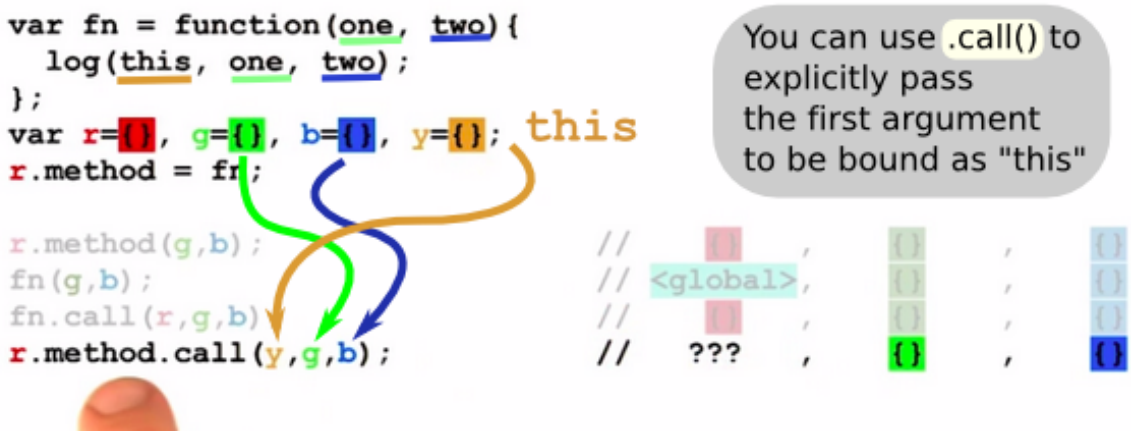
```
function foo() {
 var fooVar = 'A Variable!';

 function bar() {
   console.log(fooVar); // fooVar is available here
 }
}
```

'this' keyword

Passing a reference to 'this' with .call()

Either pass in an object and invoke the function or override by using `.call()` to literally pass in the reference you want to use for `this`

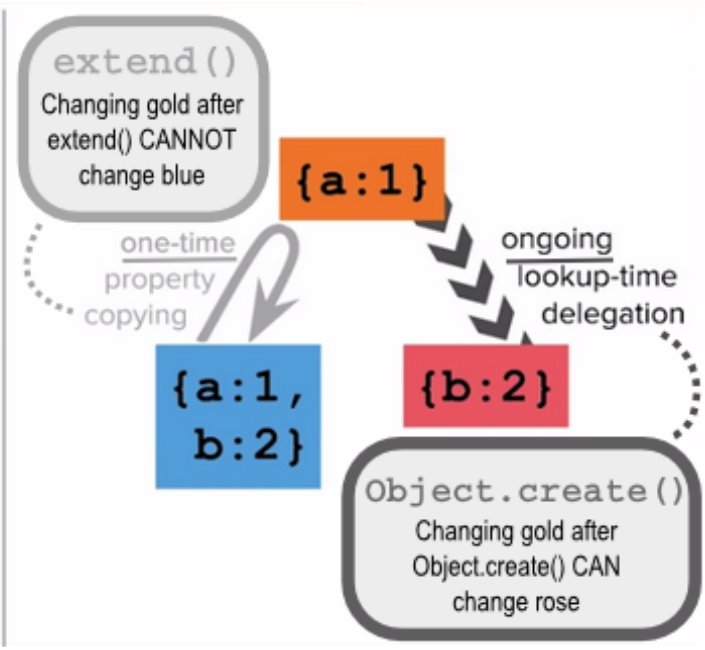


'extend()' vs. 'Object.create()'

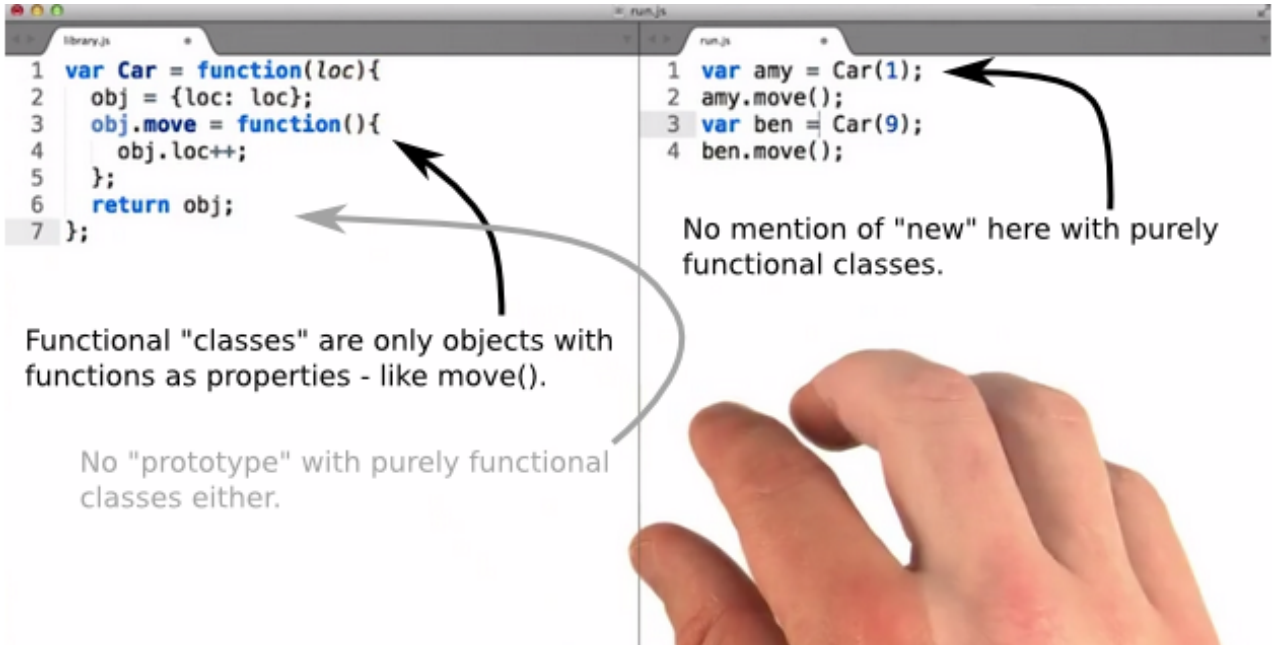
`extend()`	is an example function that copies all properties *one-time*
`Object.create()`	creates an *ongoing* lookup to the parent object

Note: `extend()` doesn't exist in vanilla JS. Many libraries include this functionality because it's so useful (jQuery for instance, http://api.jquery.com/jquery.extend).

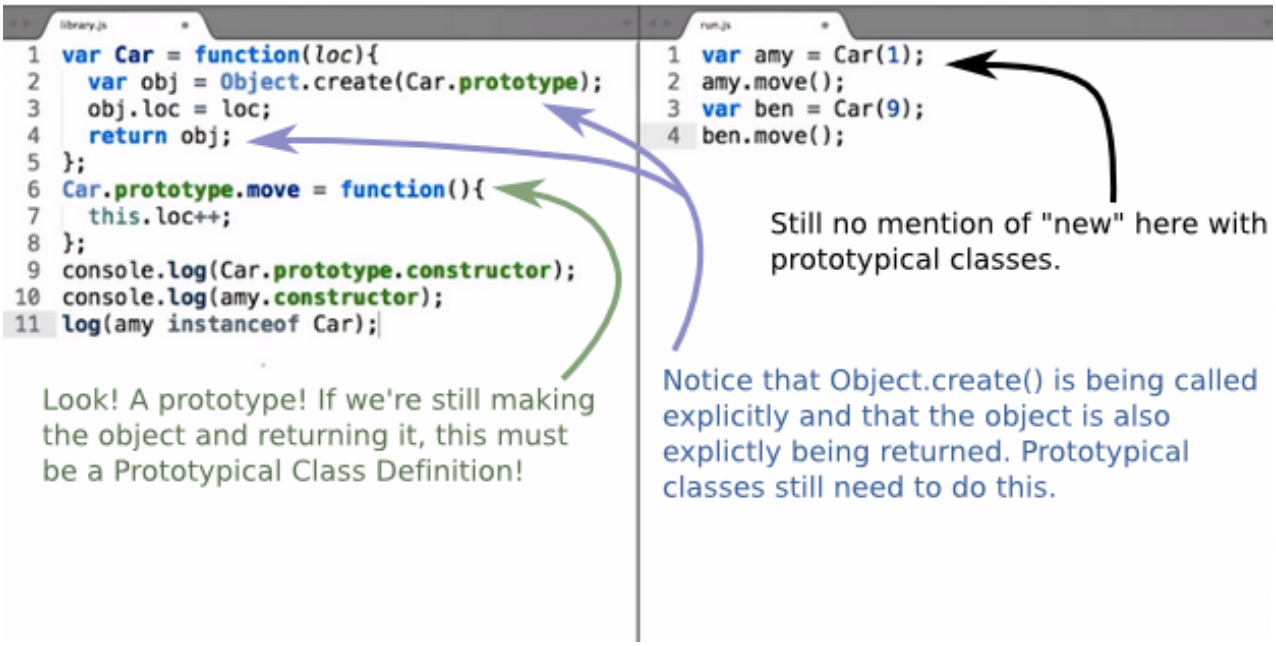
```
1  var gold = {a:1};
2  log(gold.a); // 1
3  log(gold.z); // undefined
4
5  var blue = extend({}, gold);
6  blue.b = 2;
7  log(blue.a); // 1
8  log(blue.b); // 2
9  log(blue.z); // undefined
10
11 var rose = Object.create(gold);
12 rose.b = 2;
13 log(rose.a); // 1
14 log(rose.b); // 2
15 log(rose.z); // undefined
16
17 gold.z = 3;
18 log(blue.z); // undefined
19 log(rose.z); // 3
20 rose.toString();
```



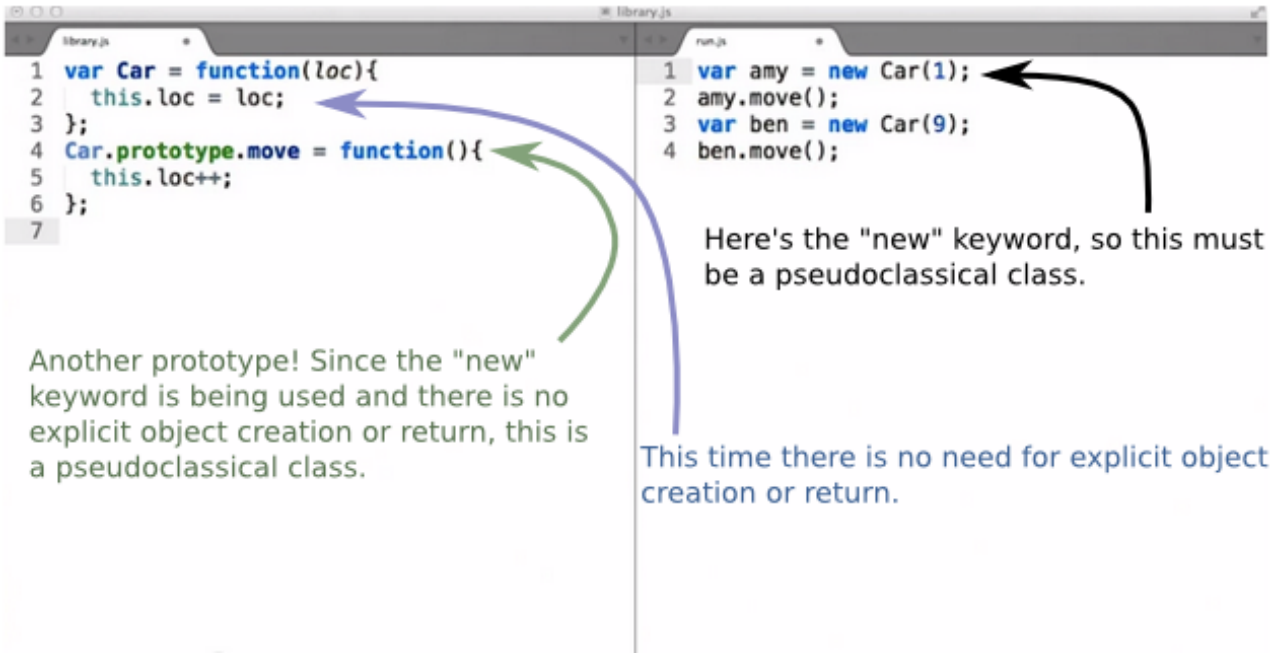
Functional Class Definitions



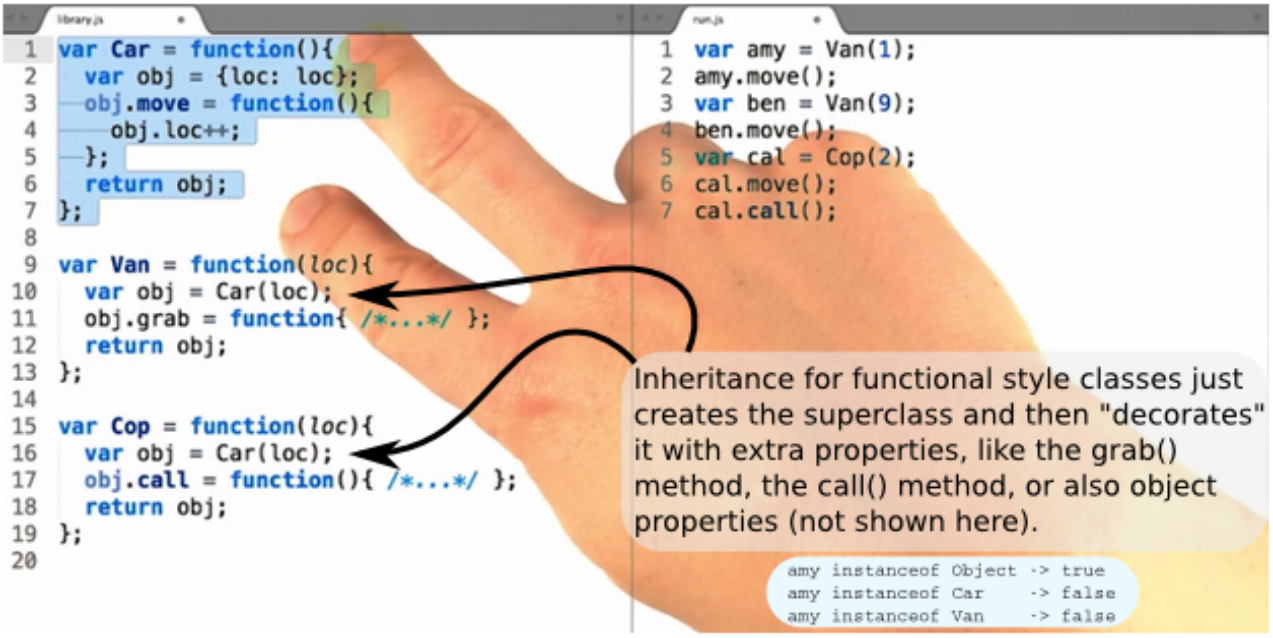
Prototypical Class Definitions



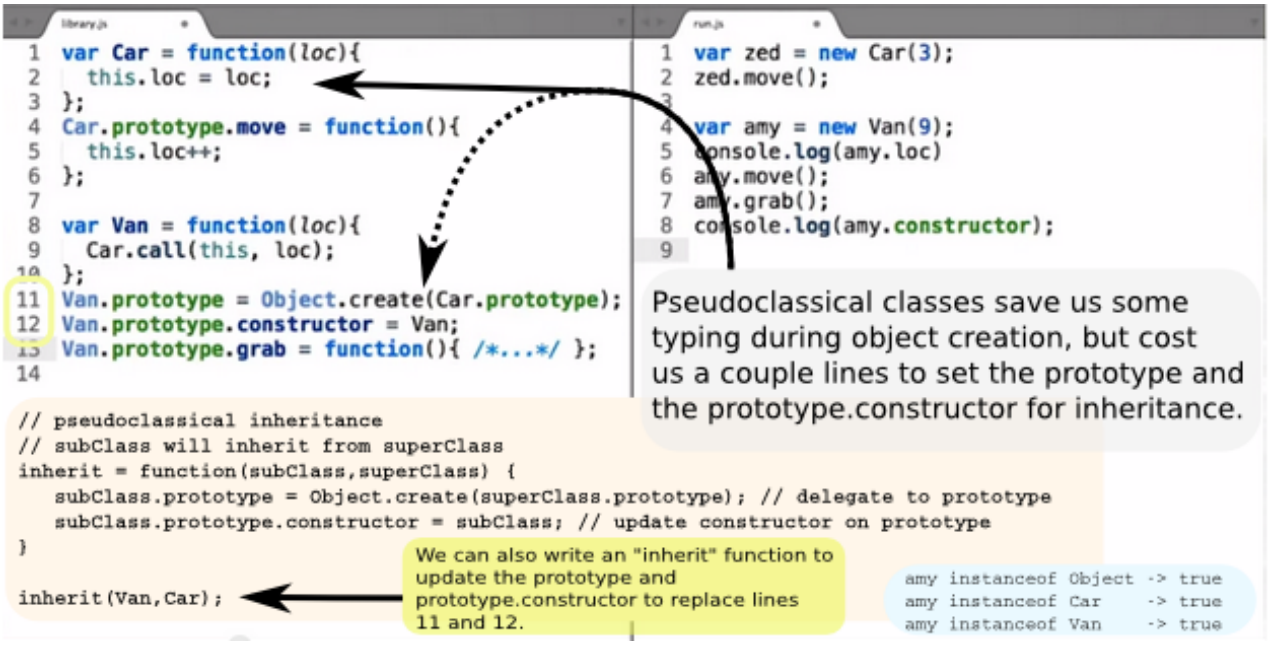
Pseudoclassical Class Definitions



Superclass and Subclass Definitions (Functional Class stye)



Superclass and Subclass Definitions (Pseudoclassical Class style)



Here is the code for `inherit` if you want to copy it:

```
// pseudoclassical inheritance
// subClass will inherit from superClass
inherit = function(subClass,superClass) {
   subClass.prototype = Object.create(superClass.prototype); // delegate to
prototype
   subClass.prototype.constructor = subClass; // set constructor on prototype
}

inherit(Van,Car);
```

As further guidance, you can find more elaborate schemes for making pseudoclassical inheritance in JavaScript 'easier'. A Google search offers up this link:

http://phrogz.net/js/classes/OOPinJS2.html

which would allow you to write something like this if you follow the example:

```
Van.inheritsFrom( Car );
```

Bonus (advanced) example of closures

Regarding the `fooVar` example above, we might ask: is `fooVar` frozen in time? That is, if `fooVar` were changed and accessed in a later call of `bar()`, would `fooVar` hold the original value or would it have the new value because `fooVar` is an ongoing lookup? The answer is that it *will change* because the lookup is *ongoing*. We can test this in the console of a browser with the use of `setTimeout`, printing the time and nesting some scopes in an extended version of the `fooVar` example above. Don't worry if you don't understand this, but if you're curious, this will show you that the variables (eg. `fooVar`) in the execution context remain 'live', ie. they can be modified and will affect any scopes that have them in their closures. The intended way to understand this is to read the output first and then see what the code was doing to produce it, matching the printed statements to the code.

```
// zero padding by profitehlolz found on stack overflow:
// http://stackoverflow.com/questions/1267283/how-can-i-create-a-zerofilled-value-using-javascript
function zeropad(n, p, c) {
   var pad_char = typeof c !== 'undefined' ? c : '0';
   var pad = new Array(1 + p).join(pad_char);
   return (pad + n).slice(-pad.length);
}

function getTime(currentdate) {
   return currentdate.getHours() + ':'
       + zeropad(currentdate.getMinutes(),2) + ':'
       + zeropad(currentdate.getSeconds(),2);
}
console.log('TEST ' + getTime(new Date()));

var globalVar = 'globalVar ' + getTime(new Date());

function foo() {
   var fooVar = 'Foo Variable!', fooDate = new Date();

   console.log('Time in foo = ' + getTime(fooDate));

   function modGlobalVar() {
        globalVar = 'globalVar ' + getTime(new Date());
   }

   function bar() {
       console.log('----------');
       console.log('Time in bar = ' + getTime(fooDate));
       console.log('fooVar in bar = ' + fooVar); // fooVar is available here
       console.log('globalVar in bar = ' + globalVar);
       console.log('==========');
   }
   bar();
   console.log('\nThis test shows that fooVar can be changed after bar is defined and run.\n');
   fooVar = 'Foo Variable has been changed!';
   setTimeout(modGlobalVar,         2500); // modify globalVar before bar is called again
   setTimeout(bar,                 5000);
}

foo();
```

Which produces this output (in Chrome's JavaScript console):

```
TEST 10:48:04
Time in foo = 10:48:04
----------
Time in bar = 10:48:04
fooVar in bar = Foo Variable!
globalVar in bar = globalVar 10:48:04
==========

This test shows that fooVar can be changed after bar is defined and run.

----------
Time in bar = 10:48:04
```

```
fooVar in bar = Foo Variable has been changed!
globalVar in bar = globalVar 10:48:07
==========
```