

# CSC 347 - Concepts of Programming Languages

## JavaScript OOP

Instructor: Stefan Mitsch



## Learning Objectives

- Understand delegation-based object-oriented programming in JavaScript



# JavaScript OOP

- Object literals `{}` are syntactic primitives
- `class` declarations are not syntactic primitives, only syntactic sugar
- Delegation-based inheritance



# Object Literals

- Objects have properties

```
1 var empty = {}; /* Object literal with no contents */
2 var person = { /* Object literal with three properties */
3   name: "Alice",
4   age: 50,
5   addr: "243 S Wabash Ave"
6 };
```

- Access properties

```
1 > [ person.name, person.name.length ]
2 [ 'Alice', 5 ]
```

- Update properties

```
1 > person.age = person.age + 1
2 51
```

- Access `undefined` property

```
1 > person.occupation
2 undefined /* huh???? */
```

- Access property of `undefined`

```
1 > person.occupation.length
2 Uncaught TypeError: Cannot read property 'length' of undefined
```



# Object Properties

- Object is map from strings to values
- Dot notation when string has no spaces

```
1 var person = { name: "Alice", age: 50, addr: "243 S Wabash Ave" };
```

- Add a property `occupation`

```
1 person.occupation = "Developer";
```

- Add a property `happy place`

```
1 person["happy place"] = "Home";
```

- List all properties

```
1 > for (p in person) { console.log (p + ": " + person[p]); }  
2 name: Alice  
3 age: 50  
4 addr: 243 S Wabash Ave  
5 occupation: Developer  
6 happy place: Home
```

- Turn object into a JSON string

```
1 > JSON.stringify(person)  
2 '{"name":"Alice","age":50,"addr":"243 S Wabash Ave",  
3   "occupation":"Developer","happy place":"Home"}'
```



# Function Properties

- Functions can be named or anonymous

```
1 // objects with number-property `n` and function-property `get`  
2 var counter1 = { n: 0, get: function () { return this.n++; } };  
3 var counter2 = { n: 0, get () { return this.n++; } };
```

- Invoke the functions

```
1 > [counter1.get(), counter2.get()]  
2 [0, 0]
```

- Access the functions

```
1 > [counter1.get, counter2.get]  
2 [ [Function: get],  
3   [Function: get] ]
```

- Serialize functions to string

```
1 > [counter1.get.toString(), counter2.get.toString()]  
2 [ 'function () { return this.n++; }',  
3   'get () { return this.n++; }' ]
```



# Access Properties from Functions

! `this` is mandatory

```
1 var counter1 = { n: 0, get: function () { return this.n++; } };  
2 var counter3 = { n: 0, get: function () { return n++; } };
```

- Increment `counter1`

```
1 > counter1.get()  
2 0
```

- Increment `counter3`

```
1 > counter3.get()  
2 Uncaught ReferenceError: n is not defined  
3   at Object.get (repl:1:43)
```



# Encapsulation

- No private properties

```
1 var counter1 = { n: 0, get: function () { return this.n++; } };
```

- Increment counter1

```
1 > counter1.n = 30  
2 > counter1.get()  
3 30
```



# Encapsulation

- Emulate encapsulation using closures

```
1 function createCounter () {  
2   var n = 0;  
3   return {  
4     get: function () { return n++; }  
5   };  
6 };  
7 var [c4a,c4b] = [createCounter(), createCounter()]
```

- `createCounter` returns an object literal with closure for function `get`
- `c4a` and `c4b` each have their own `n`

- Increment `c4a`

```
1 > c4a.get()  
2 0
```

- Increment both `c4a` and `c4b`

```
1 > [c4a,c4b].map (x => x.get())  
2 [ 1, 0 ]
```



# Encapsulation

- Closure variables are not fields

```
1 function createCounter () {  
2   var n = 0;  
3   return {  
4     get: function () {  
5       console.log ("this.n=" + this.n);  
6       return this.n++;  
7     }  
8   };  
9 };  
10 var c5 = createCounter ();
```

- Increment `c5`

```
1 > c5.get ();  
2 this.n=undefined  
3 NaN
```



# Binding `this`

- JS binds `this` based on calling context. See [here](#)
  - `o.m()` is method context, `this===o`
  - `f()` is function context, `this===global/window`
  - `new C()` is constructor context, `this` is new object
- Access `this` from method context

```
1 var o = { getThis () { return this; } };  
2 o.getThis() === o;           // true
```

- Access `this` from function context

```
1 var fThis = o.getThis; // save method as function  
2 fThis() === o           // false --- this=o not closed above  
3 fThis() === global      // true (in Node.js)  
4 fThis() === window      // true (in Browser)
```



# Binding **this**

## JavaScript

```
1 var oJS = { n: -1, next () { this.n += 1; return this.n; } };
2 var fNext = oJS.next
3 fNext() // NaN --- closure does not bind this=oJS
```

- JS methods are just properties that hold functions
- **this** is not bound in the closure

## Scala

```
1 object oScala { var n = -1; def next() = { this.n += 1; this.n } }
2 var fNext = oScala.next _
3 fNext() // 0 --- closure binds this=oScala
```

- Scala **Methods are not Functions**
- **this** is closed when converting method to function



## Binding `this`

- What is wrong in the code below?

```
1 var o = {  
2   v : 5,  
3   add : function (xs) {  
4     return xs.map(function(x){ return this.v + x; });  
5   }  
6 }  
7 o.add([10,20,30]) // [ NaN, NaN, NaN ]
```

- Before EcmaScript5 fix: alias `this`

```
1 var o = {  
2   v : 5,  
3   add : function (xs) {  
4     var me = this;  
5     return xs.map(function(x){ return me.v + x; });  
6   }  
7 }  
8 o.add([10,20,30]) // [ 15, 25, 35 ]
```



## Binding **this**

- EcmaScript5 fix: manual binding

```
1 var o = {  
2   v : 5,  
3   add : function (xs) {  
4     return xs.map(function(x){ return this.v + x; }.bind(this));  
5   }  
6 }  
7 o.add([10,20,30]) // [ 15, 25, 35 ]
```

- EcmaScript6 fix: lambda notation

```
1 var o = {  
2   v : 5,  
3   add : function (xs) {  
4     return xs.map(x => this.v + x);  
5   }  
6 }  
7 o.add([10,20,30]) // [ 15, 25, 35 ]
```



# Lambda Notation

- Lambda notation (fat arrow) uses the current value of `this`

```
1 var y = {getThese: function(xs){ return xs.map(x => this)}};  
2 var z = {getThese: function(xs){ return xs.map(function(x){ return this; })}};
```

```
1 y.getThese([10,20,30])[0] === y // true  
2 z.getThese([10,20,30])[0] === global // true (in Node.js)  
3 z.getThese([10,20,30])[0] === window // true (in Browser)
```



# Creating Objects

- Create an object with a function
- Call with *function context*
- Function builds and returns object literal

```
1 function createCounter () {  
2   return {  
3     n: -1,  
4     next: function () { return ++this.n; },  
5     reset: function () { this.n = -1; }  
6   };  
7 }  
8 var c1 = createCounter (); // Function context  
9 var c2 = createCounter ();
```

```
1 > [c1.next(), c1.next(), c1.next(), c1.reset(), c1.next(), c1.next()]  
2 [ 0, 1, 2, undefined, 0, 1 ]  
3 > [c2.next(), c2.next(), c2.next(), c2.reset(), c2.next(), c2.next()]  
4 [ 0, 1, 2, undefined, 0, 1 ]
```



# Creating Objects

- Call with *constructor context*
- Object created with `new` : implicitly passed to function as `this`

```
1 function Counter () {  
2   this.n = -1;  
3   this.next = function () { return ++this.n; },  
4   this.reset = function () { this.n = -1; }  
5 }
```

```
1 var c1 = new Counter (); // Constructor context  
2 var c2 = new Counter ();
```

```
1 var cx = Counter (); // Function context! `this` points to global/window  
2 // cx is undefined (no return statement in Counter)
```

```
1 > [c1.next(), c1.next(), c1.next(), c1.reset(), c1.next(), c1.next()]  
2 [ 0, 1, 2, undefined, 0, 1 ]  
3 > [c2.next(), c2.next(), c2.next(), c2.reset(), c2.next(), c2.next()]  
4 [ 0, 1, 2, undefined, 0, 1 ]
```

! Every object has own set of methods > `c1.next === c2.next // false!!?`



# Sharing Properties

- JS functions are objects, so can have properties
- Define properties on `prototype` object of function

```
1 function Counter () { this.n = -1; }
2 Counter.prototype.next = function () { return ++this.n; }
3 Counter.prototype.reset = function () { this.n = -1; }
4
5 var c1 = new Counter ();
6 var c2 = new Counter ();
```

```
1 > [c1.next(), c1.next(), c1.reset()]
2 [ 0, 1, 2, undefined]
3 > [c2.next(), c2.next(), c2.reset(), c2.next(), c2.next()]
4 [ 0, 1, undefined, 0, 1 ]
```

! `c1` and `c2` share `Counter.prototype`: `c1.next === c2.next // true`



# Dynamically Update Methods at Runtime

```
1 function Counter () { this.n = -1; }  
2 Counter.prototype.next = function () { return this.n += 1; }  
3 Counter.prototype.reset = function () { this.n = -1; }  
4  
5 var c1 = new Counter ();
```

- Output

```
1 > [c1.next(), c1.next(), c1.next(), c1.reset(), c1.next(), c1.next()]  
2 [ 0, 1, 2, undefined, 0, 1 ]
```

- Change the behavior of `next` and `reset`

```
1 > Counter.prototype.next = function () { return this.n += 2; }  
2 > Counter.prototype.reset = function () { this.n = -2; }
```

- Output

```
1 > [c1.next(), c1.next(), c1.next(), c1.reset(), c1.next(), c1.next()]  
2 [ 3, 5, 7, undefined, 0, 2 ]
```



## Delegation-based OOP

- `__proto__` *internal* property of object: points to another object
- If property does not exist, search `__proto__` *recursively* until found
  - [Mozilla](#),
  - [JavaScript Core](#),
  - [JavaScript Prototype](#)
- Terminology
  - Textbook: *delegation-based* inheritance
  - ECMAScript: *prototype-based* inheritance

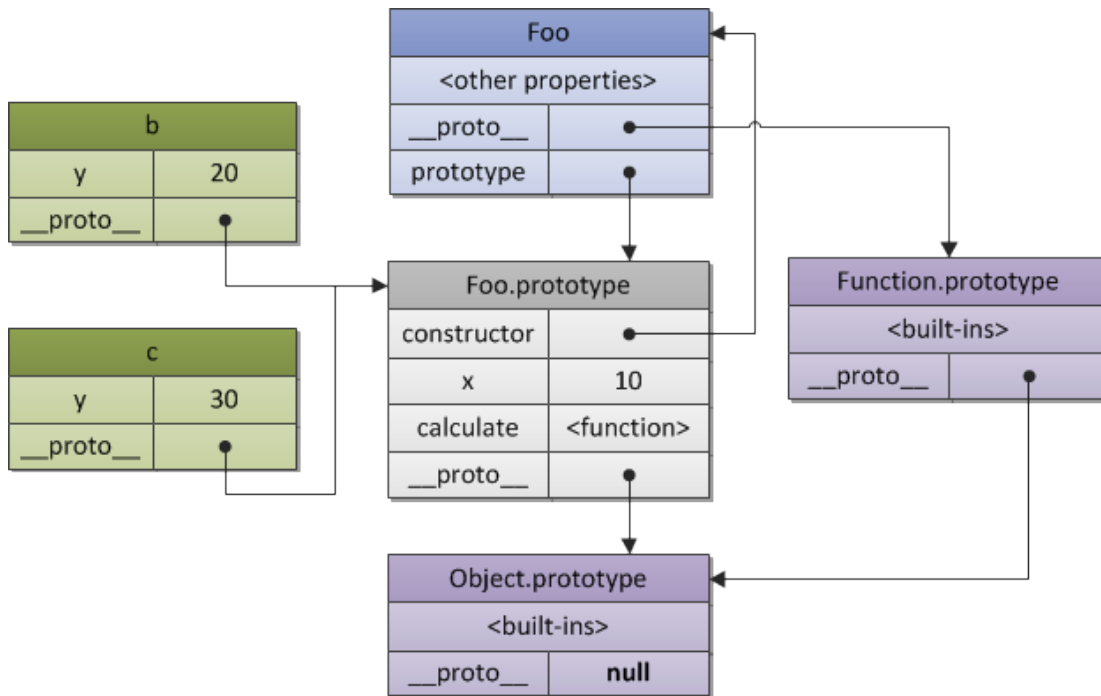


## Prototype vs. `__proto__`

- All objects have `__proto__`
- Functions have `prototype`
- In constructor context, new object's `__proto__` is set to the function's `prototype`



# Prototype Implementation



```
1 function Foo(y) { this.y = y; }
2 Foo.prototype.x = 10;
3 Foo.prototype.calculate = function (z) {
4   return this.x + this.y + z;
5 };
6 var b = new Foo(20);
7 var c = new Foo(30);
```



## Dynamically Change `__proto__` at Runtime

```
1 function UpCounter () { this.n = -1; }
2 UpCounter.prototype.next = function () { return ++this.n; }
3
4 function DownCounter () { this.n = 1; }
5 DownCounter.prototype.next = function () { return --this.n; }
6
7 var c = new UpCounter ();
8 c.next ();    // Returns 0
9 c.next ();    // Returns 1
10
11 c.__proto__ = DownCounter.prototype
12 c.next ();    // Returns 0
13 c.next ();    // Returns -1
```



# Delegation Chain

- The delegate object can also delegate

```
1 function ResetCounter () { this.reset(); }
2 ResetCounter.prototype.reset = function () { this.n = -1; }
3 var c = new ResetCounter ();
4 c.next ();    // TypeError: c.next is not a function
5
6 function Counter () { this.n = -1; }
7 Counter.prototype.next = function () { return ++this.n; }
8
9 ResetCounter.prototype.__proto__ = Counter.prototype
10 c.next ();    // Returns 0
11 c.next ();    // Returns 1
```



# Classes

- JavaScript supports classes as syntactic sugar to create prototype chains

```
1 // implicit __proto__ = Object.prototype
2 class Counter {
3   // constructor function
4   constructor() { this.n = -1; }
5   // method "next" created on Counter.prototype
6   next() { return ++this.n; }
7 }
8 // extends sets __proto__
9 class ResetCounter extends Counter {
10   // must call super constructor before doing other work
11   constructor() { super(); this.reset(); }
12   // method "reset" created on ResetCounter.prototype
13   reset() { this.n = -1; }
14 }
15 class EvenCounter extends ResetCounter {
16   next() { this.n += 2; return this.n; }
17 }
```



## Summary

- Object literals are syntactic elements of JavaScript
- Delegation-based OOP: objects delegate responsibility to other objects (instead of inherit from other objects)
- Java-like `class` syntax is syntactic sugar for setting up delegation chains