

CSC 347 - Concepts of Programming Languages

Delegation-based OOP in JavaScript

Instructor: Stefan Mitsch



Learning Objectives

- ❓ Does object-oriented programming require classes and inheritance?
 - Describe delegation-based object-oriented programming in JavaScript
 - Evaluate delegation chains



JavaScript OOP

- Object literals `{}` are syntactic primitives
- `class` declarations are not syntactic primitives
- Delegation replaces class-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
2 'Alice'
3 > person.name.length
4 5
```

- Update properties

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

- Access undefined property

```
1 > person.occupation
2 undefined
```

- Access property of undefined

```
1 > person.occupation.length
2 Uncaught TypeError: Cannot read
3 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] ]
```



Access Properties from Functions

! functions are values of properties (not methods): `this` is mandatory to access other properties of an object

```
1 var counter1 = { n: 0, get: function () { return this.n++; } };  
2 var counter2 = { n: 0, get: function () { return n++; } };
```

- Increment `counter1`

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

- Increment `counter2`

```
1 > counter2.get()  
2 Uncaught ReferenceError: n is not defined at Object.get (repl:1:43)
```



Encapsulation

- All properties are public

```
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 // create array of counters, 1-value uses again  
8 // array notation to decompose into elements  
9 var [c1,c2] = [createCounter(), createCounter()]
```

- `createCounter` returns an object literal with closure for function `get`
- `c1` and `c2` each have their own `n`

- Increment `c1`

```
1 > c1.get()  
2 0
```

- Increment both `c1` and `c2`

```
1 > [c1,c2].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       console.log("n=" + n);  
7       return this.n++;  
8     }  
9   };  
10 };  
11 var c1 = createCounter ();
```

- Increment `c1`

```
1 > c1.get ();  
2 this.n=undefined  
3 n=0  
4 NaN // from undefined++
```



Binding `this`

? What is `this` if there is no class?

Method context

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

Function context

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

- JS binds `this` based on calling context, see [Mozilla Docs: this](#)
 - `o.m()` is method context, `this===o`
 - `f()` is function context, `this===global/window`
 - `new C()` is constructor context, `this` is new object



Binding **this**

JavaScript

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

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

Scala

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

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



Binding `this`

❗ What is wrong below?

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

- Inside `map` is function context, so `this` refers to `global/window`

- 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 ]
```

- Outside `map` but inside `add` is method context, so `this` refers to `o`
- Unlike `this`, ordinary variable `me` is closed



Binding `this`

- EcmaScript5 fix: manual binding

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

- Explicitly bind `this` in `map` (function context) to `this` in `add` (method context)

- 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 ]
```

- In JS, lambda expressions are not ordinary functions
- Unlike in functions, `this` is closed in lambda expressions



Creating Objects: Factory Function

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

```
1 function createCounter () {  
2   return {  
3     n: -1,  
4     next: function () { return ++this.n; }  
5   };  
6 }  
7 var c1 = createCounter (); // Function context  
8 var c2 = createCounter ();  
9 [c1.next(), c1.next(), c2.next(), c1.next()] // [ 0, 1, 0, 2 ]
```



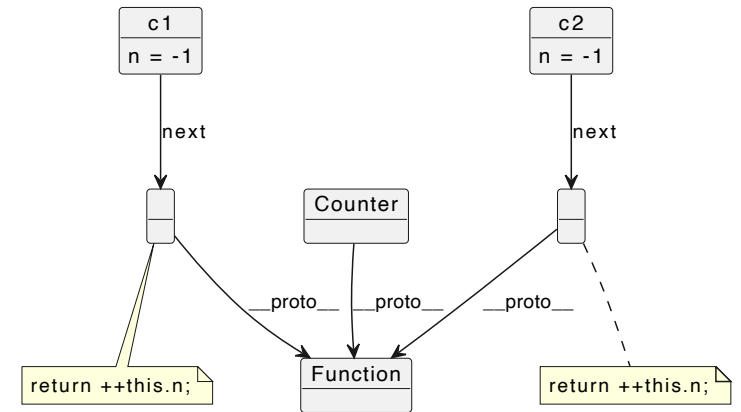
Creating Objects: Constructor Function

- Call function using `new` : constructor context
- Binds `this` to newly created object inside function

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

```
1 var c1 = new Counter (); // Constructor context  
2 var c2 = new Counter ();  
3 [c1.next(), c1.next(), c2.next(), c1.next()] // [ 0, 1, 0, 2 ]
```

```
1 var cx = Counter (); // Function context!  
2 cx.next() // Uncaught TypeError:  
3           // cx is undefined (no return in Counter)  
4 // instead, window now has properties n and next  
5 [window.next(), window.next()] // [ 0, 1 ]
```



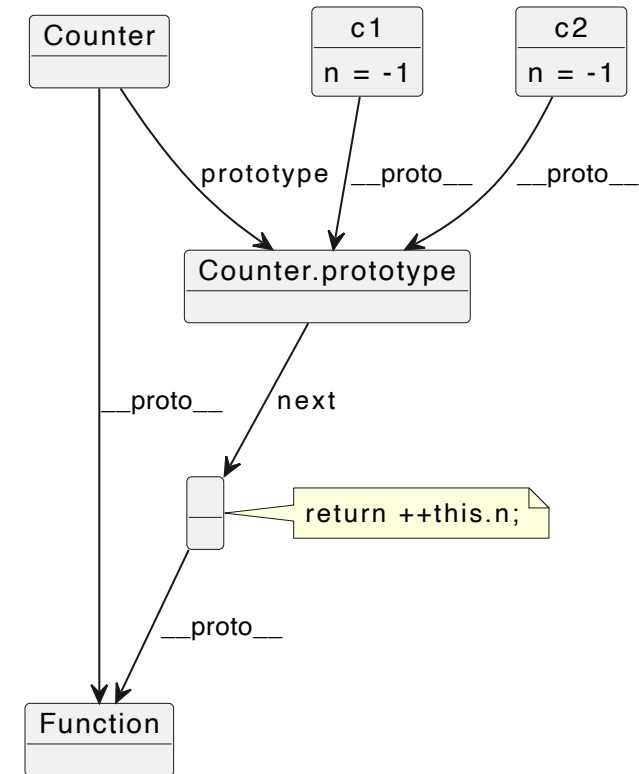
! Every object has own set of methods `c1.next` `!=` `c2.next`



Sharing Properties

- JS functions are objects, so can have properties
- Define properties on `prototype` object of function
- Names looked up along delegation chain `__proto__`

```
1 function Counter () { this.n = -1; }  
2 Counter.prototype.next = function () {  
3   return ++this.n;  
4 }  
5 var c1 = new Counter ();  
6 var c2 = new Counter ();  
7 [c1.next(), c1.next(),  
8  c2.next(), c1.next()] // [0, 1, 0, 2]
```



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



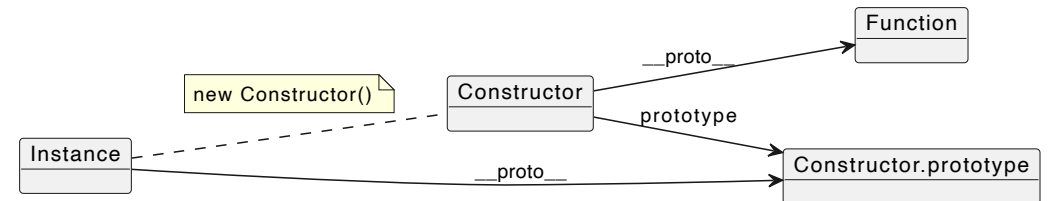
Delegation-based Inheritance

Delegation chain `__proto__`

- All objects have `__proto__`: points to another object
- Resolve names by searching `__proto__` recursively until found
 - Mozilla,
 - JavaScript Core,
 - JavaScript Prototype

Function companion `prototype`

- Only functions have `prototype`
- In constructor context, new object's `__proto__` is set to the function's `prototype`





Dynamically Update Methods at Runtime

```
1 function Counter () { this.n = -1; }
2 Counter.prototype.next = function() {
3   return this.n += 1;
4 }
5 var c1 = new Counter();
6 [c1.next(), c1.next(), c1.next()] // [0, 1, 2]
```

Change the behavior of `next`

```
1 Counter.prototype.next = function () {
2   return this.n += 2;
3 }
4 [c1.next(), c1.next(), c1.next()] // [4,6,8]
```



Dynamically Change `__proto__` at Runtime

```
1 function UpCounter () { this.n = -1; }
2 UpCounter.prototype.next = function () { return ++this.n; }
3
4 var c = new UpCounter ();
5 c.next ();    // 0
6 c.next ();    // 1
```

Change delegation chain `__proto__`

```
1 function DownCounter () { this.n = 1; }
2 DownCounter.prototype.next = function () { return --this.n; }
3
4 c.__proto__ = DownCounter.prototype
5 c.next ();    // 0
6 c.next ();    // -1
```



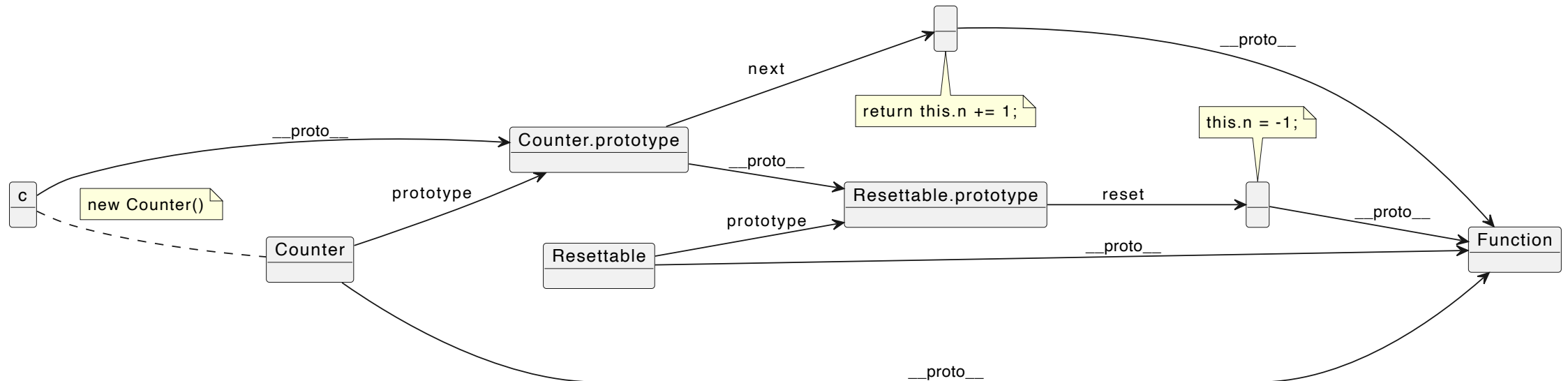
Delegation Chain

Create object

```
1 function Counter () { this.n = -1; }  
2 Counter.prototype.next = function () {  
3   return this.n += 1; }  
4 var c = new Counter ();  
5 c.next(); // 0  
6 c.next(); // 1  
7 c.reset(); // TypeError: c.reset is undefined
```

Extend delegation chain

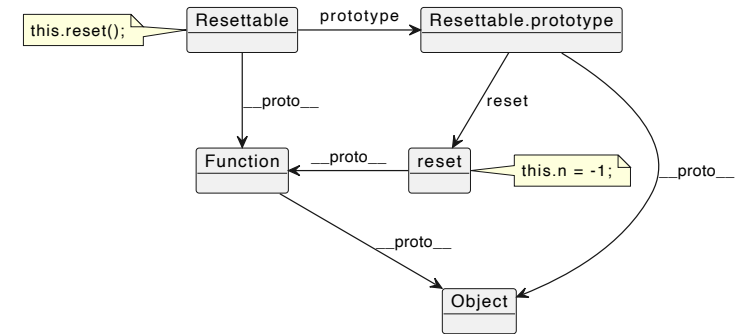
```
1 function Resettable () { this.reset(); }  
2 Resettable.prototype.reset = function () {  
3   this.n = -1; }  
4 // Counter "extends" Resettable  
5 Counter.prototype.__proto__ = Resettable.prototype  
6  
7 c.reset(); // 0  
8 c.next (); // 1
```



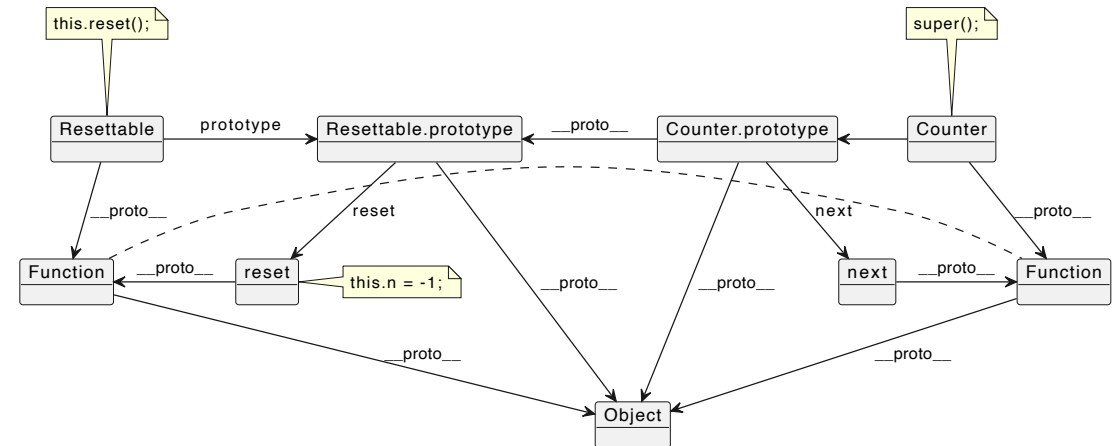


Classes: Create Delegation Chains

```
1 // sets Resettable.prototype.__proto__ to Object.prototype
2 class Resettable {
3   // function "Resettable"
4   constructor() { this.reset(); }
5   // method "reset" created on Resettable.prototype
6   reset() { this.n = -1; }
7 }
```



```
1 // sets Counter.prototype.__proto__ to Resettable.prototype
2 class Counter extends Resettable {
3   // must call super constructor before doing other work
4   constructor() { super(); }
5   // method "next" created on Counter.prototype
6   next() { return this.n += 1; }
7 }
```





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 helps setting up delegation chains