

CS 152: *Programming Language Paradigm*



# ES6 JavaScript, Metaprogramming, & Object Proxies

Prof. Tom Austin

San José State University

# Fixing JavaScript

- ECMAScript committee formed to carefully evolve the language.
  - "Don't break the web."
- Involved big players in JS world:
  - Google, Microsoft, Apple, Mozilla, Adobe, and many more

## ECMAScript Schism

- ECMAScript 4 was divisive
- A group broke off to create ECMAScript 3.1
  - more minor updates
  - later became ECMAScript 5
- Adobe left the fold

# Strict Mode

- Turns anti-patterns into errors:
  - Variables must be declared.
  - Using `with` not allowed.
  - Many others.
- To use, add `"use strict";` (including quotes) to the top of your function or file.
  - Why in a string?

## Forget var, variables are global

```
function swap(arr,i,j) {
  tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
}
function sortAndGetLargest (arr) {
  tmp = arr[0]; // largest elem
  for (i=0; i<arr.length; i++) {
    if (arr[i] > tmp) tmp = arr[i];
    for (j=i+1; j<arr.length; j++)
      if (arr[i] < arr[j]) swap(arr,i,j);
  }
  return tmp;
}
var largest =
sortAndGetLargest([99,2,43,8,0,21,12]);
console.log(largest); // should be 99, but prints 0
```

But with "use strict", the error is detected.

```
"use strict";
function swap(arr,i,j) {
  tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
}
function sortAndGetLargest (arr) {
  tmp = arr[0]; // largest elem
  for (i=0; i<arr.length; i++) {
    if (arr[i] > tmp) tmp = arr[i];
    for (j=i+1; j<arr.length; j++)
      if (arr[i] < arr[j]) swap(arr,i,j);
  }
  return tmp;
}
var largest =
sortAndGetLargest([99,2,43,8,0,21,12]);
console.log(largest); // should be 99, but prints 0
```

```
$ node sort.js
```

```
/Users/taustin/temp/sort.js:6  
  tmp = arr[0]; // largest elem  
    ^
```

```
ReferenceError: tmp is not defined
```

```
  at sortAndGetLargest (/Users/taustin/temp/sort.js:6:7)  
  at Object.<anonymous> (/Users/taustin/temp/sort.js:14:15)  
  at Module._compile (module.js:652:30)  
  at Object.Module._extensions..js (module.js:663:10)  
  at Module.load (module.js:565:32)  
  at tryModuleLoad (module.js:505:12)  
  at Function.Module._load (module.js:497:3)  
  at Function.Module.runMain (module.js:693:10)  
  at startup (bootstrap_node.js:191:16)  
  at bootstrap_node.js:612:3
```

```
$
```

## ES6 Harmony: Can't we all just get along?

- ECMAScript 6 (ES6) Harmony
  - Later renamed ECMAScript 2015
- New features:
  - classes
  - block scoping
  - arrow functions (lambdas)
  - promises
  - proxies

let is the new var

```
function makeListOfAdders(lst) {  
  var arr = [];  
  for (var i=0; i<lst.length; i++) {  
    var n = lst[i];  
    arr[i] = function(x) { return x + n; }  
  }  
  return arr;  
}
```

```
var adders =  
  makeListOfAdders([1, 3, 99, 21]);  
adders.forEach(function(adder) {  
  console.log(adder(100));  
});
```

**Prints:**

121

121

121

121

```
function makeListOfAdders(lst) {  
  let arr = [];  
  for (let i=0; i<lst.length; i++) {  
    let n = lst[i];  
    arr[i] = function(x) { return x + n; }  
  }  
  return arr;  
}
```

```
var adders =  
  makeListOfAdders([1, 3, 99, 21]);  
adders.forEach(function(adder) {  
  console.log(adder(100));  
});
```

**Prints:**

101

103

199

121

## Arrow functions

- Concise function syntax
- `this` bound lexically
  - Normal functions bind `this` dynamically.

```
function sort (lst, fn) {
  for (let i=0; i<lst.length; i++) {
    for (let j=0; j<lst.length-1; j++) {
      if (fn(lst[i], lst[j])) {
        let tmp = lst[i];
        lst[i] = lst[j];
        lst[j] = tmp;
      }
    }
  }
}
```

```
let arr = [1,2,99,10,42,7,-3,88,6];
sort(arr, function(x,y) { return x<y; });
```

```
function sort (lst, fn) {
  for (let i=0; i<lst.length; i++) {
    for (let j=0; j<lst.length-1; j++) {
      if (fn(lst[i], lst[j])) {
        let tmp = lst[i];
        lst[i] = lst[j];
        lst[j] = tmp;
      }
    }
  }
}
```

```
let arr = [1, 2, 99, 10, 42, 7, -3, 88, 6];
sort(arr, (x, y) => x < y);
```

## A broken JavaScript constructor

```
function Rabbit(name, favFoods) {  
  this.name = name;  
  this.myFoods = [];  
  favFoods.forEach(function(food) {  
    this.myFoods.push(food);  
  });  
}
```

this refers to  
the global scope

```
var bugs = new Rabbit("Bugs",  
  ["carrots", "lettuce", "souls"]);  
console.log(bugs.myFoods);
```

this bound lexically with arrows

```
function Rabbit(name, favFoods) {  
  this.name = name;  
  this.myFoods = [];  
  favFoods.forEach((food) =>  
    this.myFoods.push(food);  
  );  
}
```

Now this  
refers to the  
new object

```
var bugs = new Rabbit("Bugs",  
  ["carrots", "lettuce", "souls"]);  
console.log(bugs.myFoods);
```

## Promises

- Promise: an object that *may* produce a value in the future.
- Similar to listeners, but
  - can only succeed or fail once
  - callback is called even if event took place earlier
- Simplify writing asynchronous code

## Promise states

- Fulfilled (resolved)
- Rejected
- Pending

```
let fs = require('fs');
let p = new Promise((resolve, reject) => {
  // { key: 'hello' }
  let f = fs.readFileSync('./test.json');
  resolve(f);
});

p.then(JSON.parse)
  .then((res) => res.key)
  .then((res) => console.log(res + " world!"));
```

```
let fs = require('fs');
let p = new Promise((resolve, reject) => {
  // { key: 'hello' }
  let f = fs.readFileSync('./test.json');
  resolve(f);
});

p.then(JSON.parse)
  .then((res) => res.key,
        (err) => console.error(err))
  .then((res) => console.log(res + " world!"));
```

# Proxies

What is *metaprogramming*?

Writing programs  
that manipulate  
other programs.

# JavaScript Proxies

Metaprogramming feature proposed  
for ECMAScript 6 (Harmony).

Proposed  
By:



Tom Van  
Cutsem

Mark Miller



# Proxies: Design Principles for Robust Object-oriented **Intercession** APIs

**Abstract:** Proxies are a powerful approach to implement **meta-objects** in object-oriented languages without having to resort to **metacircular interpretation**. We introduce such a **meta-level API** based on proxies for Javascript...

## Metaprogramming terms

- **Reflection**
  - **Introspection**: examine a program
  - **Self-modification**: modify a program
- **Intercession**: redefine the semantics of operations.
- Reflection is fairly common.  
Intercession is more unusual.

## Introspection

Ability to examine the structure of a program.

In JavaScript:

```
"x" in o;  
for (prop in o) { ... }
```



Property  
lookup

Property  
enumeration

## Self-modification

Ability to modify the structure of a program.

```
o["x"]; // computed property  
o.y = 42; // add new property  
delete o.x; // remove property  
o["m"].apply(o, [42]);  
// reflected method call
```

Until recently, JavaScript did not support intercession.

JavaScript proxies are intended to fix that.

But first a little history...

## Common Lisp

- Developed before object-oriented languages were popular.
- Many libraries were created with non-standard OO systems.

## Common Lisp Object System (CLOS)

- Became standard object-oriented system for Lisp
- *What could be done about pre-existing object-oriented libraries?*

# The Devil's Choice

## 1. Rewrite libraries for CLOS?

- huge # of libraries
- infeasible to rewrite them all

## 2. Make complex API?

- difficult API to understand.
- Systems had conflicting features...
- ...But were essentially doing the same things.



# Gregor Kiczales chose option 3:



- Keep API simple.
- Modify object behavior to fit different systems.

*Metaobject protocols were born...*

# JavaScript Object Proxies

## Intercession API

## Proxy and handler

The behavior of a *proxy* is determined by *traps* specified in its *handler*.



The metaobject

What kind of things do we  
want to do to an object?

## No-op forwarding proxy

No-op handler:  
All ops forwarded to  
target without change

```
var target = {};  
var p = new Proxy(target, {});  
p.a = 37; // op forwarded  
console.log(target.a); // 37.
```

## Available traps

- has
- get
- set
- deleteProperty
- apply
- construct
- getOwnPropertyDescriptor
- getPrototypeOf
- setPrototypeOf
- isExtensible
- preventExtensions
- defineProperty
- ownKeys

## Another use case for proxies

- Share a reference to an object, but do not want it to be modified.
  - Reference to the DOM, for instance
- We can modify the forwarding handler to provide this behavior:

# Read-only handler

```
let roHandler = {
  deleteProperty: function(t, prop) { return false;},
  set: function(t, prop, val, rcvr) { return false;},
  setPrototypeOf: function(t,p) { return false; } };

var constantVals = {
  pi: 3.14,
  e: 2.718,
  goldenRatio: 1.30357 };

var p = new Proxy(constantVals, roHandler);
console.log(p.pi);
delete p.pi;
console.log(p.pi);
p.pi = 3;
console.log(p.pi);
```

# Undoable object

get, set, delete trap example  
(in class)

## Safe constructor handler

```
function Cat(name) {
  this.name = name;
}
Cat = new Proxy(Cat, {
  apply: function(t, thisArg, args) {
    throw Exception("Forgot new");
  }
});
var g = new Cat("Garfield");
console.log(g.name);
var n = Cat("Nermal");
```



Forgot new:  
exception raised

# Aspect-oriented programming (AOP)

- Some code not well organized by objects
  - *Cross-cutting concern*
- Canonical example: logging statements
  - littered throughout code
  - Swap out logger = massive code changes

# Tracing API for `set`

(in class)

## Lab: Tracing API

- Use proxies to log all actions taken on an object
- Avoids having complexity of logging framework
- **No starter code for this assignment**