CS 152: *Programming Language Paradigm*

ES6 JavaScript, Metaprogramming, & Object Proxies

Prof. Tom Austin

San José State University

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

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

```javascript
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

```javascript
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.

```javascript
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

```javascript
function Rabbit(name, favFoods) {
  this.name = name;
  this.myFoods = [];
  favFoods.forEach(function(food) {
    this.myFoods.push(food);
  });
}
var bugs = new Rabbit("Bugs",
    ["carrots", "lettuce", "souls"]);
console.log(bugs.myFoods);
```

this refers to the global scope

this bound lexically with arrows

```
function Rabbit(name, favFoods) {
  this.name = name;
  this.myFoods = [];
  favFoods.forEach((food) =>
    this.myFoods.push(food);
  );
}
var bugs = new Rabbit("Bugs",
    ["carrots", "lettuce", "souls"]);
console.log(bugs.myFoods);
```

Now this refers to the new object

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

```javascript
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!"));
```

```javascript
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.

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

WE NEVER SLEEP

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

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

```javascript
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);
```

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

- Use proxies to log all actions taken on an object

- Avoids having complexity of logging framework

- **No starter code for this assignment**