

CS 252:

Advanced Programming Language Principles



Taming the Dark, Scary Corners of JavaScript

Prof. Tom Austin

San José State University

JavaScript has first-class functions.

```
function makeAdder(x) {  
  return function (y) {  
    return x + y;  
  }  
}  
  
var addOne = makeAdder(1);  
console.log(addOne(10));
```

Warm up exercise: Create a `makeListOfAdders` function.

input: a list of numbers

returns: a list of adders

```
a = makeListOfAdders([1, 5]);
```

```
a[0](42); // 43
```

```
a[1](42); // 47
```

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

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

Prints:

NaN

NaN

NaN

NaN

What is going on in this wacky language???!!!



JavaScript does *not* have block scope.

So while you see:

```
for (var i=0; i<lst.length; i++)  
    var n = lst[i];
```

the interpreter sees:

```
var i, n;  
for (i=0; i<lst.length; i++)  
    n = lst[i];
```

In JavaScript, this is known as *variable hoisting*.

Faking block scope

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



Function creates
new scope

A JavaScript constructor

```
name = "Monty";  
function Rabbit(name) {  
  this.name = name;  
}  
var r = Rabbit("Python");  
console.log(r.name);  
// ERROR!!!  
console.log(name);  
// Prints "Python"
```

Forgot new

A 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);
```

Execution Contexts

Comprised of:

- A variable object
 - Container for variables & functions
- A scope chain
 - The variable object plus parent scopes
- A context object (`this`)

Global context

- Top level context.
- Variable object is known as the *global object*.
- `this` refers to global object

Function contexts

- Variable objects (aka *activation objects*) include
 - Arguments passed to the function
 - A special arguments object
 - Local variables
- What is `this`? It's complicated...

What does `this` refer to?

- Normal function calls: the global object
- Object methods: the object
- Constructors (functions called w/ `new`):
 - the new object being created.
- Special cases:
 - `call`, `apply`, `bind`
 - in-line event handlers on DOM elements

apply, call, and bind

```
x = 3;
```

```
function foo(y) {  
  console.log(this.x + y);  
}
```

```
foo(100);
```

```
foo.apply(null, [100]); // Array passed for args
```

```
foo.apply({x:4}, [100]);
```

```
foo.call({x:4}, 100); // No array needed
```

```
var bf = foo.bind({x:5}); // Create a new function  
bf(100);
```

Additional challenges ...

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
```

Semicolon insertion does strange things

```
function makeObject () {  
  return  
  {  
    madeBy: 'Austin Tech. Sys.'  
  }  
}  
  
var o = makeObject();  
console.log(o.madeBy); // error
```

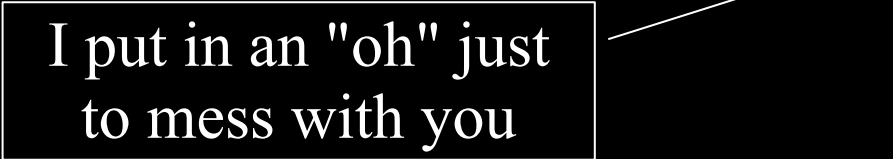
parseInt won't warn you of problems

```
console.log(parseInt("42"));
```

```
console.log("what do you get? "  
           + parseInt("16 tons"));
```

```
console.log(parseInt("101"));
```

I put in an "oh" just
to mess with you



NaN does not help matters

```
function productOf(arr) {
  var prod = 1;
  for (var i in arr) {
    var n = parseInt(arr[i])
    prod = prod * n;
  }
  return prod;
}
console.log(
  productOf(["9", "42", "1"])); // 378
console.log(productOf(
  ["9", "forty-two", "1"])); // NaN
```

We might try to fix our code ...

```
function productOf(arr) {  
  var prod = 1;  
  for (var i in arr) {  
    var n = parseInt(arr[i])  
    if (typeof n === "number")  
      prod = prod * n;  
  }  
  return prod;  
}
```

... but `typeof` does not help us.

```
> typeof NaN  
'number'
```

Nor does it help us check for `null`.

```
> typeof null  
'object'
```

The == operator is not transitive

```
' ' == '0'           // false
0 == ' '            // true
0 == '0'           // true

false == 'false'    // false
false == '0'        // true

false == undefined  // false
false == null        // true
null == undefined   // true

' \t\r\n ' == 0     // true
```

```
function typeOfChar(ch) {
  var sType = 'Other character';
  switch (ch) {
    case 'A':
    case 'B':
      ...
      sType = "Capital letter"
    case 'a':
      ...
      sType = "Lowercase letter"
    case '0':
      ...
      sType = "Digit"
  }
  return sType;
}
```

```
var str = "Hello 42";
for (var i=0; i<str.length; i++) {
    console.log(
        typeofChar(str.charAt(i)));
}
```

Output:

```
Digit
Digit
Digit
Digit
Digit
Other character
Digit
Digit
```

How can we tame the ugliness?

Tools to write cleaner/safer JavaScript:

- JSLint (<http://www.jshint.com/>)
- TypeScript– Static typechecker for JS

JSLint: *The JavaScript Code Quality Tool*

Source clear

```
function makeListOfAdders(lst) {  
  var arr = [];  
  for (var i=0; i<lst.length; i++)  
    arr[i]=function(x) {return x + lst[i];}  
  return arr;  
}  
  
var adders =  
  makeListOfAdders([1,3,99,21]);  
adders.forEach(function(adder) {  
  console.log(adder(100));  
});
```

JSLint

Options clear options

<input type="checkbox"/> <small>default</small> a browser	<input type="checkbox"/> <small>default</small> assignment expressions	<input type="checkbox"/> <small>default</small> eval	<input checked="" type="checkbox"/> <small>true</small> missing 'use strict' pragma	<input type="checkbox"/> Indentation
<input type="checkbox"/> <small>default</small> CouchDB	<input type="checkbox"/> <small>default</small> bitwise operators	<input type="checkbox"/> <small>default</small> unfiltered for in	<input type="checkbox"/> <small>default</small> stupidity	<input type="checkbox"/> Maximum line length
<input type="checkbox"/> <small>default</small> console, alert, ...	<input type="checkbox"/> <small>default</small> Google Closure	<input type="checkbox"/> <small>default</small> uncapitalized constructors	<input type="checkbox"/> <small>default</small> inefficient subscribing	<input type="checkbox"/> Maximum number of errors
<input type="checkbox"/> <small>default</small> Node.js	<input type="checkbox"/> <small>default</small> continue	<input type="checkbox"/> <small>default</small> dangling _ in identifiers	<input type="checkbox"/> <small>default</small> TODO comments	
<input type="checkbox"/> <small>default</small> Rhino	<input type="checkbox"/> <small>default</small> debugger statements	<input type="checkbox"/> <small>default</small> ++ and --	<input type="checkbox"/> <small>default</small> many var statements per function	
<input type="checkbox"/> <small>default</small> Stop on first error	<input type="checkbox"/> <small>default</small> == and !=	<input type="checkbox"/> <small>default</small> . and [^...] in /RegExp/	<input type="checkbox"/> <small>default</small> messy white space	
		<input type="checkbox"/> <small>default</small> unused parameters		

JSLint

- Static code analysis tool
- Developed by Douglas Crockford.
- Inspired by lint tool
 - catch common programming errors.

JSLint Expectations

- Variables declared before use
- Semicolons required
- Double equals not used
- (And getting more opinionated)

makeListOfAdders source

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

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

Debug makeListOfAdders
(in class)

TypeScript



What do type systems give us?

- Tips for compilers
- Hints for IDEs
- Enforced documentation
- But most importantly...

Type systems prevent us from running code with errors.

TypeScript

- Developed by Microsoft
- A new language (sort-of)
 - Type annotations
 - Classes
 - A superset of JavaScript
 - or it tries to be
- Compiles to JavaScript

TypeScript file

greeter.ts

```
function greeter(person) {  
    return "Hello, " + person;  
}  
  
var user = "Vlad the Impaler";  
console.log(greeter(user));
```

Compiled TypeScript

greeter.js

```
function greeter(person) {  
    return "Hello, " + person;  
}  
  
var user = "Vlad the Impaler";  
console.log(greeter(user));
```

TypeScript file, with annotations

greeter.ts

```
function greeter(person: string) {  
    return "Hello, " + person;  
}  
  
var user = "Vlad the Impaler";  
console.log(greeter(user));
```

Basic Types

- **number** (`var pi: number = 3.14`)
- **boolean** (`var b: boolean = true`)
- **string** (`var greet: string = "hi"`)
- **array** (`var lst: number[] = [1, 3]`)
- **enum**
- **any** (`var a: any = 3;`
`var b: any = "hi";`)
- **void**

Functions

```
function add(x: number,  
            y: number) : number {  
    return x + y;  
}
```

```
add(3, 4)
```

Classes

```
class Employee {  
  name: string;  
  salary: number;  
  constructor(name: string, salary: number) {  
    this.name = name;  
    this.salary = salary;  
  }  
  display() { console.log(this.name); }  
}
```

```
var emp = new Employee("Jon", 87321);  
console.log(emp.salary);
```

Translated code

```
var Employee = (function () {
    function Employee(name, salary) {
        this.name = name;
        this.salary = salary;
    }
    Employee.prototype.display =
        function () {console.log(this.name);};
    return Employee;
})();
var emp = new Employee("Jon", 87321);
console.log(emp.salary);
```

Lab

Today's lab will contrast JSLint and TypeScript.

Details are available in Canvas.