

CSC 447 - Concepts of Programming Languages

Surprising Design Decisions

Instructor: Eric J. Fredericks



Learning Objectives

- Identify and revisit PL concepts in the JavaScript programming language



JavaScript

- Not related to Java
- Dynamically-typed
- First class functions and objects
- Runs in browsers and elsewhere with [Node.js](#)
- Inspired by
 - Scheme: dynamically-typed functional PL
 - Self: delegation-based object-oriented PL



Expressions and Statements

- Numbers, characters, strings: `1`, `'c'`, `"str"`
- Arrays: `[1, 2, 3]`
- Objects: `{ x: 1, text: "Hello", }`
- Functions:
`function(x,y) { return x+y; }`
- `undefined`, `NaN`, `null`
- Expressions: `1 + 2`
- Side-effecting expressions, result in `undefined`
e.g. `console.log ("Test")`
- Statements: `;` turns expressions into statements
- Function declarations need `return` statements (else implicit `return undefined`)

```
1 function f (x) {  
2   console.log ("Called with " + x);  
3   return x + 1;  
4 }
```



Example

- Builtin support for manipulating document object model (DOM, tree structure of page)
- Builtin support for functional programming (callback functions to handle events)

```
1 <input id="a"
2     type="text"
3     value="empty"/>
4
5 <div id="b">
6   <p>Hello</p>
7 </div>
8
9 <div id="c">
10  <p>World</p>
11 </div>
12
13 <script
14   type="text/javascript"
15   src="demo.js">
16 </script>
```

```
1 function main () {
2   var count = 0;
3   var a = document.querySelector('input#a');
4   var b = document.querySelector('div#b');
5   var c = document.querySelector('div#c');
6   for (var x of [b,c]) {
7     x.onclick = function (e) {
8       e.target.outerHTML += "<p>Again! " + (count++) + "</p>";
9     }
10  }
11  a.value = 1 + 2;
12 document.addEventListener("DOMContentLoaded", main);
```



Dynamically Typed

```
1 function f (x) {  
2   console.log ("1: " + x);  
3   console.log ("2: " + x.data);  
4   console.log ("3: " + x.data.length);  
5 }
```

- Output on an array

```
1 > f ({ data: [11, 21, 31] });  
2 1: [object Object]  
3 2: 11, 21, 31  
4 3: 3
```

- Output on a string

```
1 > f ("pizza");  
2 1: pizza  
3 2: undefined  
4 Uncaught TypeError: Cannot read  
5 property 'length' of undefined
```



Dynamically Typed

```
1 function f (x) {  
2   console.log ("1: " + (x / 2));  
3   console.log ("2: " + (x[0]));  
4 }
```

- Output on an array

```
1 > f ([11, 21, 31]);  
2 1: NaN  
3 2: 11
```

- Output on a number

```
1 > f (8);  
2 1: 4  
3 2: undefined
```

- Output on undefined

```
1 > f (undefined);  
2 1: NaN  
3 Uncaught TypeError:  
4 Cannot read property  
5 '0' of undefined
```



Expressions and Statements

- Object literals: `{}` , array literals: `[]`
- Unexpected type conversions

```
1 [] + [] /* '' */  
2 [] + {} /* [object Object] */  
3 {} + [] /* VS8: 0, Node.js: [object Object] */  
4 {} + {} /* VS8: NaN, Node.js: [object Object][object Object] */
```

- Dynamic types and optional arguments create surprising results

```
1 xs = ["10", "10", "10"];  
2 xs.map(parseInt) /* [10, NaN, 2] */  
3 [1, 5, 20, 10].sort() /* [1, 10, 20, 5] */
```

- `parseInt`, `map`, `sort`



Expressions and Statements

- `undefined`, `NaN`, and `Infinity`

```
1 undefined + 1 /* NaN */
2 false + 1    /* 1 (someone liked C) */
3 5/0          /* Infinity */
4 -1/0         /* -Infinity */
```



```

1  ' ' == '0'
2  0 == ' '
3  0 == '0'
4
5  10 == '10'
6  9 == '9'
7  10 > 9
8  '10' > '9'
9  1 + 9
10 1 + '9'
11 1 + '9' == 19
12 -1 + '11'
13 '11' - 1

```

[illegible]

[illegible]

11



Scope: Hoisting

Variables `var` are function-scoped

- *Hoists* variable declarations to top of nearest enclosing function
- Initialization code is not hoisted

```
1 var a = 1;
2 function f () {
3
4     console.log ("f1: a = " + a); // a=undefined
5     { var a = 2;
6       console.log ("f2: a = " + a); // a=2
7     }
8 }
9 function main() {
10    console.log ("m1: a = " + a); // a=1
11    f ();
12    console.log ("m2: a = " + a); // a=1
13 }
```

- Equivalent code: variable declared at top of function
- Initialization code remains in place

```
1 var a = 1;
2 function f () {
3     var a; /* = undefined */
4     console.log ("f1: a = " + a); // a=undefined
5     { a = 2;
6       console.log ("f2: a = " + a); // a=2
7     }
8 }
9 function main() {
10    console.log ("m1: a = " + a); // a=1
11    f ();
12    console.log ("m2: a = " + a); // a=1
13 }
```



Scope: Hoisting (Block Scope)

- ES6 introduced `let` : block-oriented scope

```
1 var a = 1;
2 function f () {
3     console.log ("f1: a = " + a);    // a=1
4     { let a = 2;
5         console.log ("f2: a = " + a); // a=2
6     }
7 }
8 function main() {
9     console.log ("m1: a = " + a);    // a=1
10    f ();
11    console.log ("m2: a = " + a);    // a=1
12 }
```



Scope: Hoisting

- Hoisting of `var` to top of function happens everywhere

```
1 var a = 1;  
2 function f (b) {  
3   a = 2;  
4   if (b) {  
5     var a;  
6     a = a + 1;  
7   }  
8   console.log (" f: a = " + a);  
9 }
```

```
1 function main() {  
2   f (true);  
3   console.log ("m1: a = " + a);  
4   f (false);  
5   console.log ("m2: a = " + a);  
6 }
```

```
1 f: a = 3  
2 m1: a = 1  
3 f: a = 2  
4 m2: a = 1
```

- ❗ Even when variable is already used "above" `var` declaration



Functional Programming

- Functions and objects are first-class citizens
 - create at runtime
 - pass as args, return, store in data structures
- Nested functions (and objects) are commonplace: callbacks, collections processing
- JS uses static (lexical) scoping, see [here](#)



Recursion: Tail-Call Optimization

- JavaScript standard includes tail-call optimization, but only Safari implements it
- Can still make sense to think about alternative implementations

```
1 function fib(n) {  
2   if (n==0) return 0;  
3   else if (n==1) return 1;  
4   else return fib(n-2) + fib(n-1);  
5 }
```

```
1 function fibtail(n) {  
2   // nested function, inner n shadows outer n  
3   function fibtail(n, r1, r2) {  
4     if (n===0) return r1;  
5     else if (n===1) return r2;  
6     else return fibtail(n-1, r2, r1+r2);  
7   }  
8   return fibtail(n, 0, 1);  
9 }
```




Argument Passing and Pattern Matching

- JavaScript uses pass-by-value
- Visible to outside
 - changes to fields of objects
 - changes to elements of arrays
- Pattern matching for arrays

```
1 function swaparray(a) {  
2   let [x,y] = a; // deconstruct array  
3   a[0] = y;  
4   a[1] = x;  
5 }
```

- Swap array

```
1 let a = [1, 2];  
2 swaparray(a);
```

- Swap in place

```
1 let x = 1;  
2 let y = 2;  
3 [y,x] = [x,y];
```



Collections Processing

- Map, filter, and fold built into modern JS
- With lambda expressions (`=>`)

```
1 var xs = [ 11, 21, 31 ];  
2  
3 xs.map (x => (2 * x));  
4 xs.filter (x => x%7===0);  
5 xs.reduce ((z,x) => z+x, 0);
```

- With functions

```
1 var xs = [ 11, 21, 31 ];  
2  
3 xs.map (function(x) { return (2 * x); });  
4 xs.filter (function(x) { return x%7===0 });  
5 xs.reduce (function(z,x) { return z+x; }, 0);
```

- Beware: `return` mandatory, outer `this` not closed



Closures

- Recall: Java requires effectively-final `i` from enclosing scope

```
1 for (int i = 0; i < 5; i++) {  
2     int x = i; /* accepted: x effectively final */  
3     new Thread (new Runnable () {  
4         public void run () {  
5             while (true) { System.out.print (x); }  
6         }  
7     }).start ();  
8 }
```

- JS allows shared `i`; unwanted effect with `var`

```
1 var funcs = [];  
2 for (var i = 0; i < 5; i++) {  
3     funcs.push (function () { return i; });  
4 }  
5 funcs.map (f => f());  
6 // [ 5, 5, 5, 5, 5 ]
```

- ❗ Closure stores reference to shared mutable variable `i`



Closures

- Fix with `var` similar to Java?

```
1 var funcs = [];  
2 for (var i = 0; i < 5; i++) {  
3   var x = i;  
4   funcs.push (function () { return x; });  
5 }  
6 funcs.map (f => f());  
7 // [ 4, 4, 4, 4, 4 ]
```

- ❗ Beware: hoisting!

- Use block-scoped `let`

```
1 var funcs = [];  
2 for (var i = 0; i < 5; i++) {  
3   let x = i; /* block scope */  
4   funcs.push (function () { return x; });  
5 }  
6 funcs.map (f => f());  
7 // [ 0, 1, 2, 3, 4 ]
```

- Use `let` directly in `for`

```
1 var funcs = [];  
2 for (let i = 0; i < 5; i++) {  
3   funcs.push (function () { return i; });  
4 }  
5 funcs.map (f => f());  
6 // [ 0, 1, 2, 3, 4 ]
```



Summary

- JavaScript standardized as ECMAScript, ECMA-262
 - [New Editions released regularly](#)
 - [Mozilla](#) has the best documentation
 - [ECMAScript Compatibility](#)
 - Bad parts eventually [fixed](#)
- Book Recommendations
 - [You Don't Know JS](#) by Kyle Simpson
 - [Axel Rauschmayer's Exploring JS](#)