Iteration in Javascript

Abram Hindle
abram.hindle@ualberta.ca
Department of Computing Science
University of Alberta
http://softwareprocess.es/

Warning

- This talk is not idiomatic

Iteration

- Programming is useful because we can repeat tasks
- Iteration is one of the fundamental building blocks of most programming languages
- Iteration usually refers to repetition
  - In mathematics iterating a functions means applying it repeatedly
  - In programming iteration can refer to any method of repetition.

Concepts

- Block
  - A chunk of code
  - Usually the part that gets repeated
  - Usually follows scope rules
- Condition
  - An expression that evaluates to true or false
  - Commonly used to determine if a loop continues

Concepts

- Strict – Everything gets done
- Lazy – Something gets done when needed.
- In Order – Execute in Sequence
- Out of Order – Execute in any order
- Dependency – One value depends on another

While Loops

- Repeats a block until a condition is met.
- Loop invariant executes first

```
var condition = 1;
while (condition) {
    // ...
    condition = !condition;
}
// condition is False

//how many OKs ?
var count = 0;
while(confirm("OK???")) {
    count++;
}
// alert(count);
```

```
var x = 10;
while (x > 0) {
    x--;
}
// x is 0

// maybe you're not sure how many
// iterations you need?
var x = 100.0;
while (x > 1) {
    x /= 3;
}
// x is 0.4115226337448604
```

For Loop

- Iterate with a condition or over a collection of elements.

```
var sum = 0;
for (var i = 0 ; i < 10; i++) {
    sum += i;
}
// sum is 45

function range(start,end) {
    var out = [];
    var j = 0;
    for (var i = start; i < end; i++) {
        out[j++] = i;
    }
    return out;
}
//alert(range(1,10).length);
```

```
for var i in range(1,10) {
    alert(i);
}

var s = "";
var v = ["a","b","c"];
var u = {"A":"a", "B":"b",
         "C":"c"};
for (var i in v) {
    s += v[i]; //  over keys
}
for (var i in u) {
    s += u[i]; //  over keys
}
alert(s);
// s starts with abc
```

Recursion

- Arbitrary flow control
- Good for iterating datastructures like trees
- Watch out for stackoverflows!

```
var tree = {"a":{"b":{"c":1,"d":2,"e":3,"f":4},"g":5},
            "h":{"i":6}};
function treesum(tree) {
    if (typeof tree === "number") {
        return tree;
    } else {
        var sum = 0;
        for (var key in tree) {
            sum += treesum(tree[key]);
        }
        return sum;
    }
}
alert(treesum(tree));
```

OO Iteration (Iterators)

- Object with a `next()` method and **has_next()**

```
// OO Iterator way
OnlyEvens = function(seq) {
    this.seq = seq;
    this.index = 0;
    self = this;
    this.hasNext = function() {
        return self.index < self.seq.length
    }
    this.next = function() {
        var v = self.seq[self.index];
        self.index += 2;
        return v;
    }
};
```

```
var oe = new OnlyEvens([0,1,
        2,3,4,5,6,7,8,9,10]);
var s = "";
while (oe.hasNext()) {
    s += oe.next();
}
alert(s);
```

Order

- Did you notice something?
- Everything iterated in order.
- But what if order doesn't really matter?

Map

- In mathematics iterating a functions means applying it repeatedly
- A map function applies 1 function to all elements in a collection and produces a new collection of the results of that function
  - Usually this is in order
  - But you don't have to do it in order
- Map f (a,b,c) => (f(a), f(b), f(c))
- Map square (1,2,3) => (1,4,9)
- Note: JS map has extra paremeters only send in 1 parameter functions!!!

Map Example

```
// add 1 to a list
var v = [1,2,3,4,5];
function inc(x) { return 1 + x; }
var u = v.map(inc);
// alert(u); 2,3,4,5,6
// alert(v); 1,2,3,4,5

function basename(path) {
    var sp = path.split("/");
    return sp[sp.length - 1];
}
var v = ["/home","/file",
         "/usr/local"];
var u = v.map(basename);
// u = ['home','file','local']
```

```
function GET(url) {
    var request = new
          XMLHttpRequest();
    request.open("GET",
          url, false);
    request.send(null);
    return request.responseText;
}
var urls = ["http://cbc.ca",
            "http://gc.ca",
            "http://alberta.ca"];
var pages = urls.map(GET);
```


Firefox Web Browser
Thu 17:37
Parallel.js: Parallel computing with Javascript - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Parallel.js: Parallel computing ...
adambom.github.io/parallel.js/
Google
Fork me on GitHub
Parallel.js
Parallel Computing with Javascript
Parallel.js is a tiny library for multi-core processing in Javascript. It was created to take full advantage of the ever-maturing web-workers API. Javascript is fast, no doubt, but lacks the parallel computing capabilites of its peer languages due to its single-threaded computing model. In a world where the numbers of cores on a CPU are increasing faster than the speed of the cores themselves, isn't it a shame that we can't take advantage of this raw parallelism?
Parallel.js solves that problem by giving you high level access to multicore processing using web workers. It runs in your browser (as long as it supports web workers). Check it out.
Download
Unminified: parallel.js
Minified (1490 Bytes gzipped): parallel.min.js
Source: github
Installation
Include parallel.js in your web projects like so:
Map
Map Example

Parallelism with Map

- Think in “map” -- Think Parallel
- Limit dependencies of a block in order to parallelize the computation!

```
// http://adambom.github.io/parallel.js/
<script src="parallel.js"></script>
<script>
function sum(l) {
    var sum = 0.0;
    for (var i in l) {
       sum += l[i];
    }
    return sum;
}
function sqr(v) { return v * v; }
alert(sum(range(1,100).map(sqr)));
var p = new Parallel(range(1,100));//this could mean 100 workers!
p.map(sqr).then(function(d) { alert("what"+sum(d)); });
```

Parallelism with Map (Continued)

- IO is slow and inherently parallelizable!

```
// You need proper URLs and permission!
var urls = ["file:///./map.html",
            "file:///./map.html","file:///./map.html"];
var urls = ["http://cbc.ca","http://gc.ca","http://alberta.ca"];
new Parallel(urls).map(GET).then(alert);
```

Reduce

- Linear, 1 at a time
- Collapse a collection in a single value via an operator or function of 2 args
  - f(e1, f(e2, f(e4, ... f(e99,e100)))...)))
  - add(e1, add(e2, add(e4, ... add(e99,e100)))...)))
- Sum is a reduce

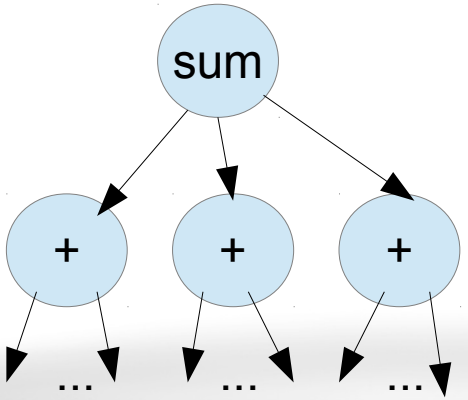
```
var v = [1,2,3,4,5,6,7,8,9];
alert(v.reduce(function(x,y) { return x + y }));
function sum(l) {
    return l.reduce(function(x,y) {return x + y});
}
alert(sum(v));
```

Reduce Can Be Parallel (Sometimes)

```
// http://adambom.github.io/parallel.js/
<script src="parallel.js"></script>
<script>
function sum(l) {
    return l.reduce(function(x,y) {return x + y});
}
// split the job up in 3 parts
var p = new Parallel([range(1,10000),range(10001,20000),
range(20001,30000)]);
p.map(sum).reduce(sum).then(alert);
```

Trees, Communtativeness and Initialization

- Can your problem be modelled as a TREE?
- Problems with commutative or associative parts can often be modelled as a tree of computation.
- Different branches may be executed in Parallel.
- One can reduce dependencies by avoid initialization (e.g. sum = 0)



Conclusions

- Main forms of perl iteration:
  - For / While / Iterators / Recursion / Map / Reduce
- Reducing dependencies in blocks allows iteration to be parallelized.
- Consider if order or strictness can be are actually needed?
- These concepts apply to other languages as well.