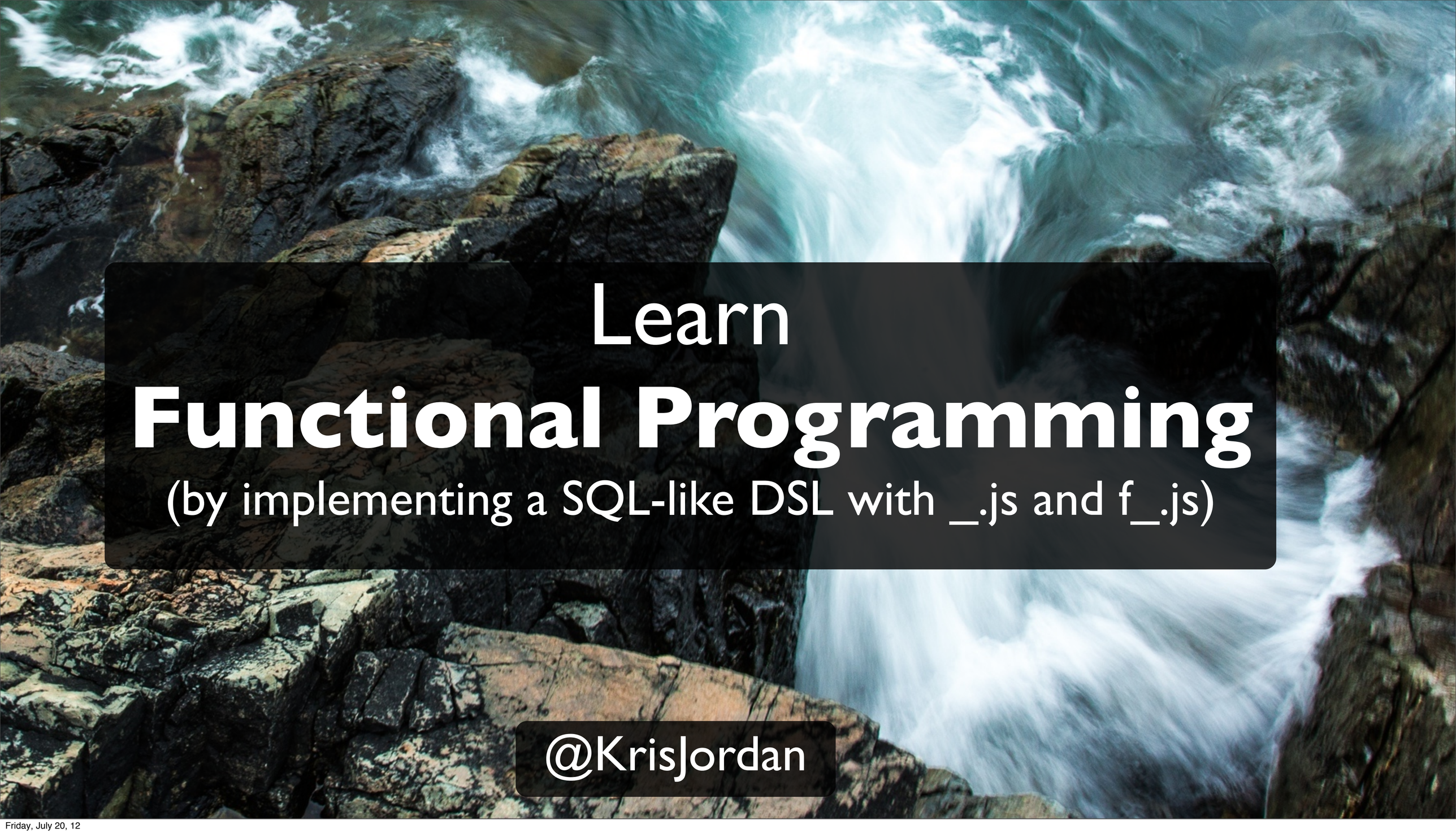


Friday, July 20, 12

This is an introductory functional programming talk aimed at programmers who are comfortable with imperative style, perhaps object-oriented style, programming and want to learn functional programming.

Objectives

1. **Grok functional programming fundamentals**
2. Transfer SQL familiarity to Functional Programming
3. Exposure to underscore.js & f_underscore.js

@KrisJordan

Why learn functional programming?

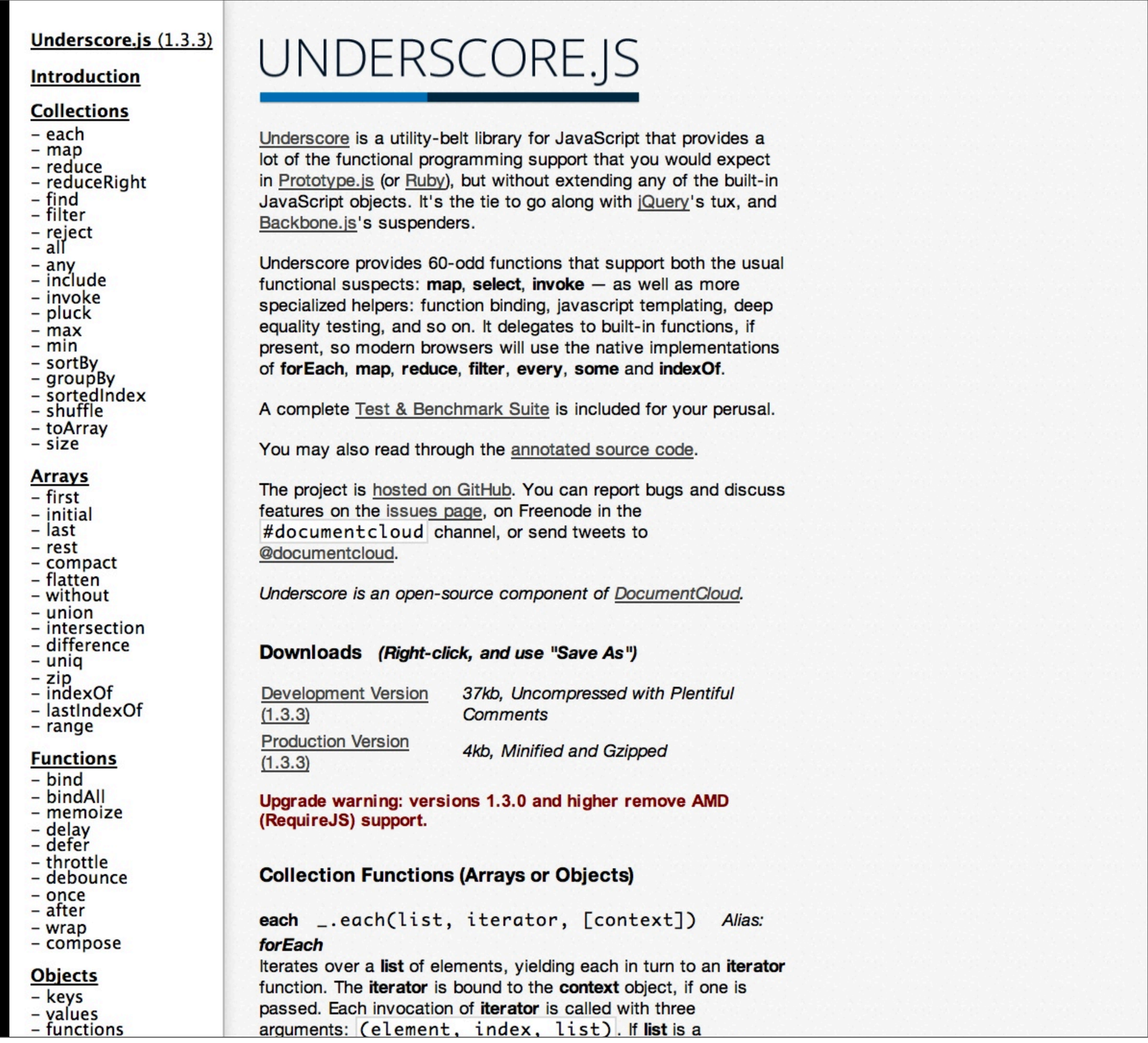
It will change your life.

Learning
functional programming
will change the way
you design programs.

It will change your life.

Why is it
~~hard~~ *frustrating*
to learn functional
programming?

Lists of simple functions aren't exciting.



Underscore.js (1.3.3)

Introduction

Collections
- each
- map
- reduce
- reduceRight
- find
- filter
- reject
- all
- any
- include
- invoke
- pluck
- max
- min
- sortBy
- groupBy
- sortedIndex
- shuffle
- toArray
- size

Arrays
- first
- initial
- last
- rest
- compact
- flatten
- without
- union
- intersection
- difference
- uniq
- zip
- indexOf
- lastIndexOf
- range

Functions
- bind
- bindAll
- memoize
- delay
- defer
- throttle
- debounce
- once
- after
- wrap
- compose

Objects
- keys
- values
- functions

UNDERSCORE.JS

Underscore is a utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Prototype.js (or Ruby), but without extending any of the built-in JavaScript objects. It's the tie to go along with jQuery's tux, and Backbone.js's suspenders.

Underscore provides 60-odd functions that support both the usual functional suspects: **map**, **select**, **invoke** — as well as more specialized helpers: function binding, javascript templating, deep equality testing, and so on. It delegates to built-in functions, if present, so modern browsers will use the native implementations of **forEach**, **map**, **reduce**, **filter**, **every**, **some** and **indexOf**.

A complete Test & Benchmark Suite is included for your perusal.

You may also read through the annotated source code.

The project is hosted on GitHub. You can report bugs and discuss features on the issues page, on Freenode in the `#documentcloud` channel, or send tweets to @documentcloud.

Underscore is an open-source component of DocumentCloud.

Downloads ***(Right-click, and use "Save As")***

Development Version (1.3.3)	*37kb, Uncompressed with Plentiful Comments*
Production Version (1.3.3)	*4kb, Minified and Gzipped*

Upgrade warning: versions 1.3.0 and higher remove AMD (RequireJS) support.

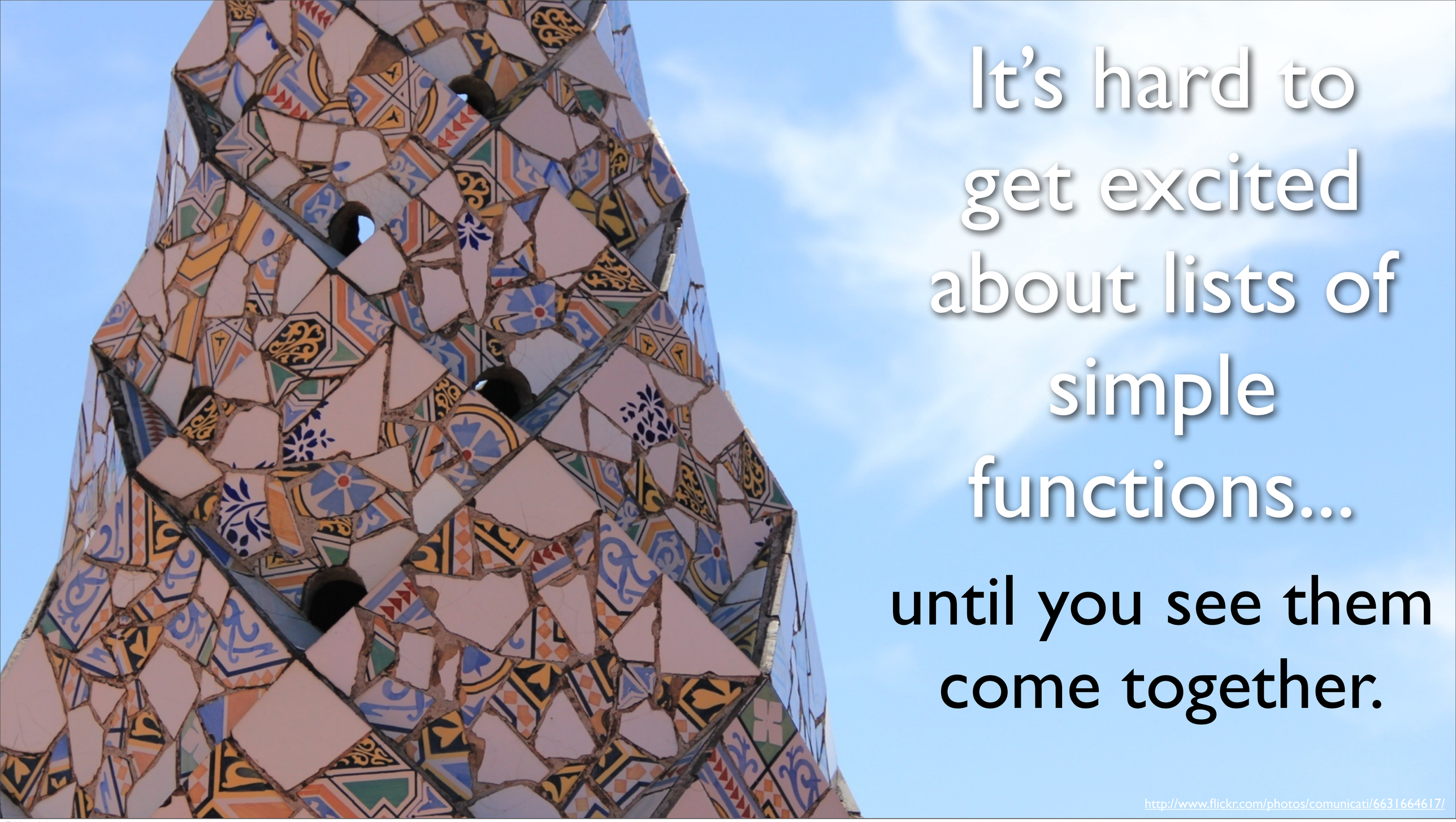
Collection Functions (Arrays or Objects)

each `_.each(list, iterator, [context])` *Alias:* ***forEach***

Iterates over a **list** of elements, yielding each in turn to an **iterator** function. The **iterator** is bound to the **context** object, if one is passed. Each invocation of **iterator** is called with three arguments: `(element, index, list)`. If **list** is a

Friday, July 20, 12

Dude is telling you functional programming has changed my life. Changed the way I program. Underscore.js is amazing. So you open up the website and you see a list of simple functions. "MAX". So you're thinking: "What is this guy smoking?"



Unix reference. Functional belief that there's more value in having 100 functions that can operate on 1 data type, than 10 functions that can operate on 10 data types.

Our Goal

```
select([“author”,
        “changes”,
        “sha”],
orderBy(“author”,
where(greaterThan(get(“changes”),5),
from(commits))));
```

@KrisJordan

6 function calls that are composed together and would be super simple to tweak and play around with.

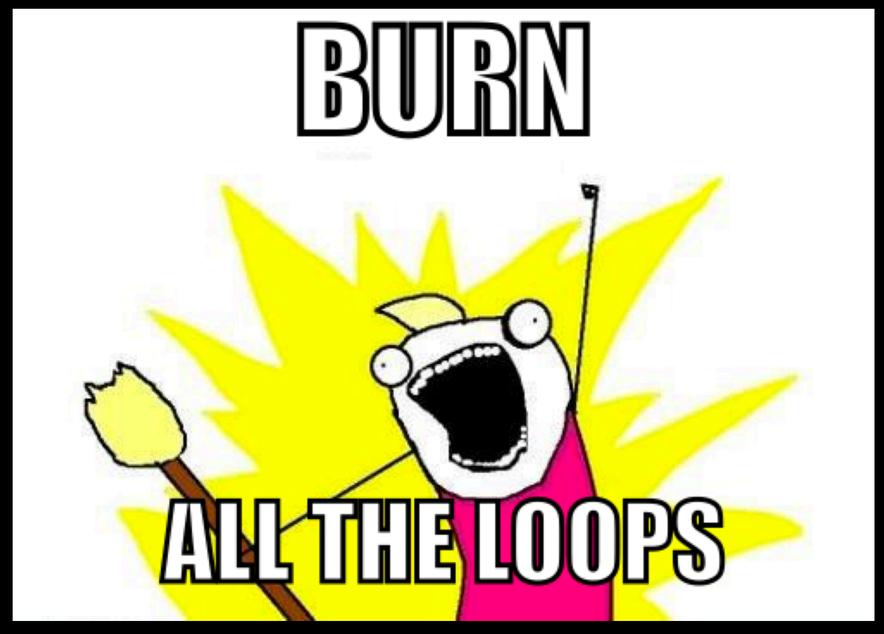
Very declarative like SQL.

Why is it ~~hard~~ *frustrating* to learn functional programming?

Academic, Mathematical Roots

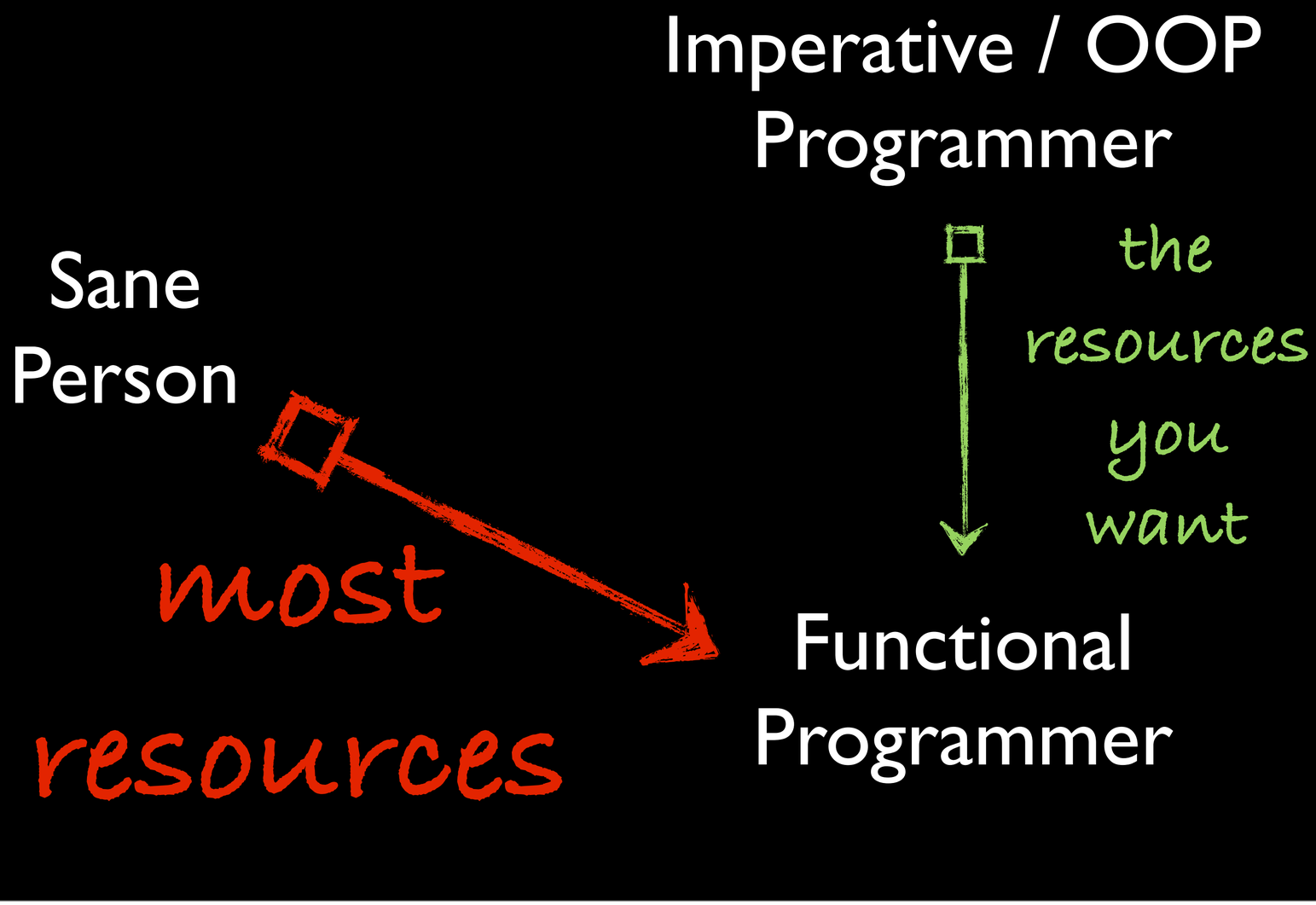
can lead to

Pedantic Resources



Alonzo Church - 30s - Lambda Calculus
John McCarthy - 50s - Lisp

Why is it ~~hard~~ *frustrating* to learn functional programming?



Once you've learned imperative, object-oriented programming there's no going back to being a sane person.

http://www.flickr.com/photos/jramspott/7355480358/

You will see...

- Imperative Code
- Loops
- State

So, what is
“Functional Programming”
?

@KrisJordan

Functions are values.

@KrisJordan

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- ...and one more important thing we'll get to

Variables can be assigned functions.

```
var sayHello = function() {
  console.log(“Hello, world.”);
};
sayHello();
```

```
=> Hello, world.
```

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- ...and one more important thing we'll get to

jQuery's API is awesome thanks to this.

```
$("h1").click(function(){
  console.log("Hello, world");
});


$("p").each(function(){
  $(this).wrap("<marquee/>");
});
```

@KrisJordan

Functions can be *passed* to functions.

```
var sayHello = function() {
  console.log("Hello, world.");
};
var executor = function(aFunction){
  aFunction();
};
executor(sayHello);
=> Hello, world.
```

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- ...and one more important thing we'll get to

Functions can be *return* values.

```
var say = function(message) {
  return function() {
    console.log(message);
  };
};
var sayHello = say("Hello, world.");
sayHello();
```

```
=> Hello, world.
```

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- ...and one more important thing we'll get to

Functions
are the smallest complete unit of
functionality.

Let's Implement our SQL-like DSL

@KrisJordan

Why? Because it's a familiar problem. It's useful to be able to work with data pulled from APIs without shoving it in a DB. Clients are fast, APIs don't always give you what you want.

Input data pulled from GitHub API

```
var commits = [
  {
    "author":"jashkenas",
    "sha":"f6f9d37",
    "message":"Merge branch 'master'",
    "changes":38,
    "additions":19,
    "deletions":19
  },
  ...
];
```

@KrisJordan

Our SQL Target

```
SELECT author, sha, changes
FROM commits
WHERE changes > 5
ORDER BY author
```

@KrisJordan

Which Step Should We Choose First?

1. SELECT author, sha, changes
2. FROM commits
3. WHERE changes > 5
4. ORDER BY author

@KrisJordan

We're talking about a pipeline of work, so the order in which we choose to execute matters. We're playing the role of SQL query optimizer.

Let's Start Here

```
SELECT author, sha, changes
FROM commits
WHERE changes > 5
ORDER BY author
```

@KrisJordan

How would we implement this step imperatively?

We've got an array of objects. We can write a function that returns a new array of objects where its changes property is greater than 5. Easy.

Imperative Where

```
var tables = { commits: [...] };

var whereGt5Changes = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(commit.changes > 5) {
            result.push(commit);
        }
    }
    return result;
};

console.log(whereGt5Changes(tables.commits));
```

Friday, July 20, 12

Be sure to take time walking through code.

Imperative Where

```
FROM commits
WHERE changes > 5
```

```
FROM commits
WHERE deletions = 0
```

```
var whereGt5Changes = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(commit.changes > 5) {
            result.push(commit);
        }
    }
    return result;
};
```

```
var whereEq0Deletions = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(commit.deletions === 0) {
            result.push(commit);
        }
    }
    return result;
};
```

Imperative Where

WHERE changes > 5

```
var whereGt5Changes = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(commit.changes > 5) {
            result.push(commit);
        }
    }
    return result;
};
```

WHERE deletions = 0

```
var whereEq0Deletions = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(commit.deletions === 0) {
            result.push(commit);
        }
    }
    return result;
};
```


This smells.
- Code duplication.
- Conflating generic algorithm with specific application-level request

How can we refactor this? Let's turn back to our functional bag of tricks.

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- ...and one more important thing we'll get to

Let's refactor to a higher-order function.

```
var whereGt5Changes = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(                              ) {
            result.push(commit);
        }
    }
    return result;
};

console.log(whereGt5Changes(tables.commits));
```

```
commit.changes > 5
```


So we can start by pulling our singular concern out from our where algorithm. We've broken things. Where do we get the commit object from in the comparison?

Let's refactor to a higher-order function.

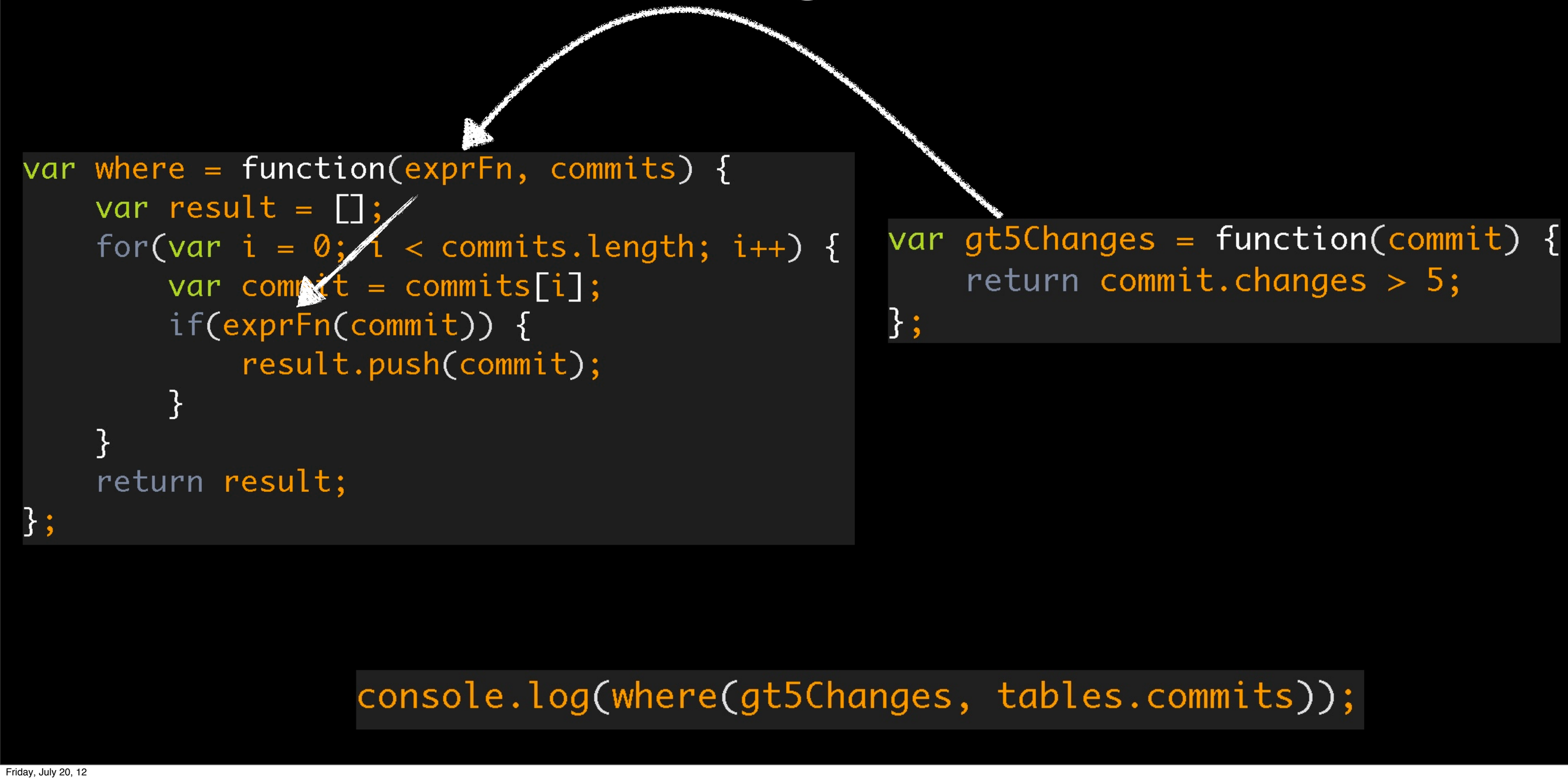
```
var whereGt5Changes = function(commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(                              ) {
            result.push(commit);
        }
    }
    return result;
};

console.log(whereGt5Changes(tables.commits));
```

```
var gt5Changes = function(commit) {
    return commit.changes > 5;
};
```


We wrap it in a function whose argument will provide it. We need to be sure to return the evaluation, too, and that it returns a boolean. This is called a predicate function.

So how do we plug our individual concern back in to our where algorithm? Remember, functions are just values. We can pass this gt5Changes function into our where.

Let's refactor to a higher-order function.



```
var where = function(exprFn, commits) {
    var result = [];
    for(var i = 0; i < commits.length; i++) {
        var commit = commits[i];
        if(exprFn(commit)) {
            result.push(commit);
        }
    }
    return result;
};
```

```
var gt5Changes = function(commit) {
    return commit.changes > 5;
};
```

```
console.log(where(gt5Changes, tables.commits));
```

Friday, July 20, 12

Take time explaining this. Walk through.

Everything is happy again.

We've abstracted a generalized filtering algorithm for a collection from the criteria we filter with for a single item by making it possible to plugin logic.

Abstract generic algorithms
from singular concerns.

Pass singular concerns
in as “iterator” functions.

@KrisJordan

underscore.js

60-some classic functions

Friday, July 20, 12

Underscore is a library of 60-some useful functions just like this. Reject (inverse of filter), all, any, map, reduce. These collection functions are higher order functions that abstract out a generic algorithm and allow you to plugin

Jeremy Ashkenas describes it as the tie to jQuery's tux.

Why underscore.js?

- MIT License
- Simple, high quality, lightweight library from Jeremy Ashkenas
- Most depended upon node.js package manager (npm) package
  - *9% of npm published packages depend on _*
- Packaged with backbone.js so all backbone collections have underscore.js functions built-in

https://github.com/documentcloud/underscore/

Now let's refactor to use underscore.js

filter `_.filter(list, iterator, [context])` *Alias:* ***select***

Looks through each value in the **list**, returning an array of all the values that pass a truth test (**iterator**). Delegates to the native **filter** method, if it exists.

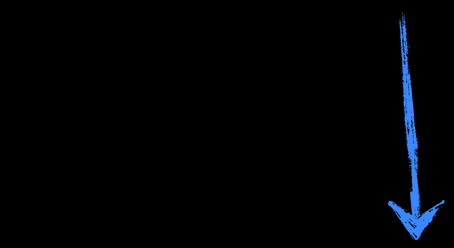
```
var evens = _.filter([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });
=> [2, 4, 6]
```

```
var where = function(expr, data) {
    return _.filter(data, expr);
};
```


There's a function for that. It's called filter.

Demo

There's something unsatisfying about this.



```
var gt5Changes = function(commit) {
    return commit.changes > 5;
};
```


How can we do a better job making the singular item logic easier to specify?

This is lame because the property we're reading and the value 5 are hard coded.

Why can't we change the signature? Because the signature is what the algorithm expects.

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- ...and one more important thing we'll get to

Something smells fishy, doesn't `message` pop off the stack?

```
var say = function(message) {
  return function() {
    console.log(message);
  };
};

var sayHello = say("Hello, world.");

sayHello();
```

Functions are values.

- Variables can assigned functions
- Functions can be passed as arguments
- Functions can be return values
- Functions can be “closures”

The one more important thing about functional programming in JavaScript is that functions can be closures.

Functions can be *closures*.
Closures "trap" var refs into a function's scope.

Closures in OOP terms: native, minimal Command pattern

OOP

```
class MsgCommand {
  private _msg;
  function MsgCommand(msg) {
    this._msg = msg;
  }
  function exec() {
    print this._msg;
  }

cmd = new MsgCommand(“OOP”);
cmd.exec();
```

FP

```
var msgCommand = function(msg) {
  return function() {
    console.log(msg);
  };
};

cmd = msgCommand(“FP”);
cmd();
```

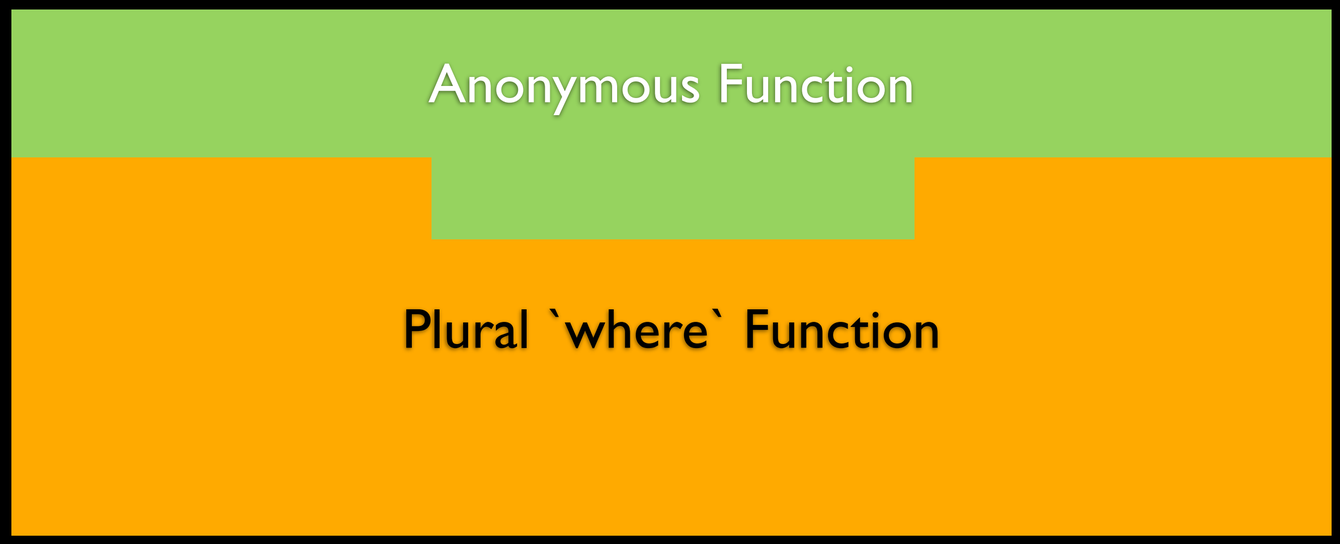
@KrisJordan

Let’s refactor to generate a closure.

Plural `where` Function

```
where(                    , commits);
```

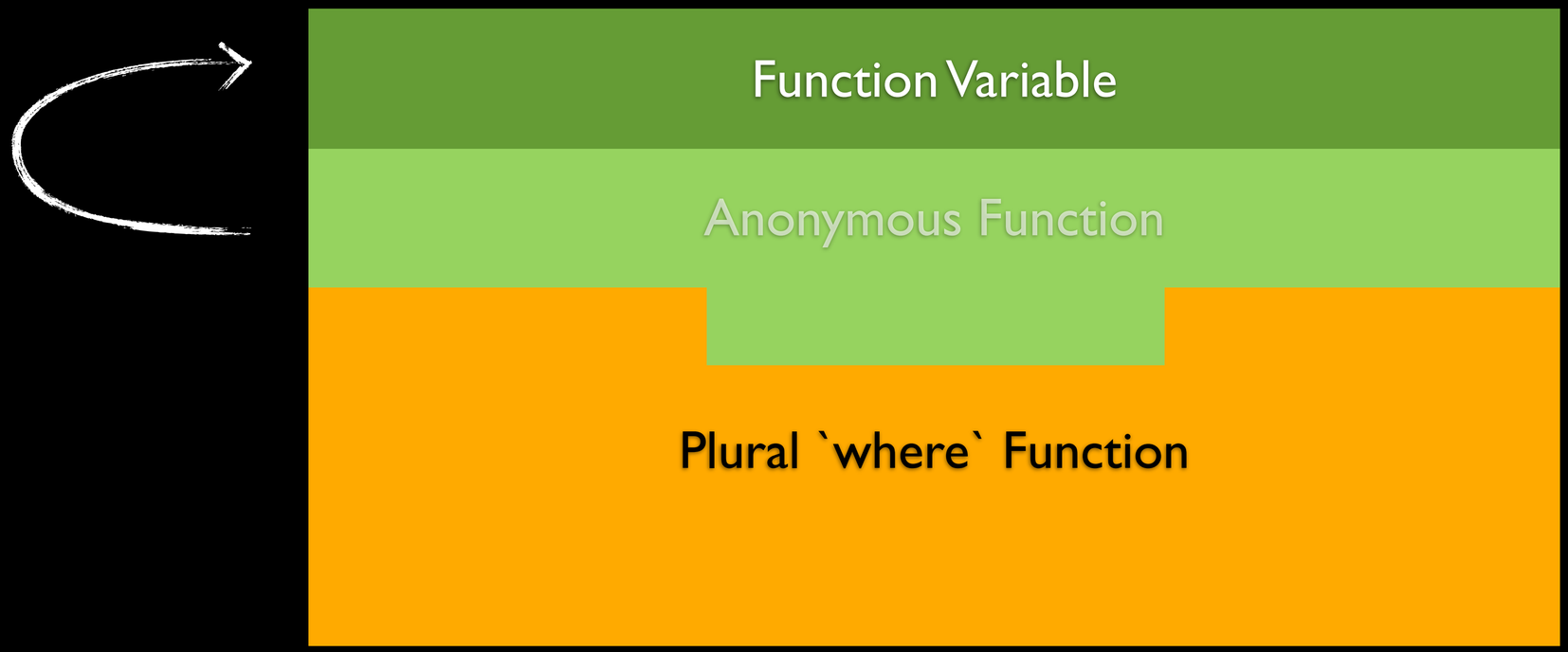
@KrisJordan




```
where(function(commit){
  return commit.changes > 5;
},commits);
```

@KrisJordan

We skipped over this before, but you can write functions anonymously in JavaScript and pass them as values. Just like you can hard code constants. In general it's not a good idea. It's hard to read and hard to test. jQuery apps tend to abuse the simplicity of this.

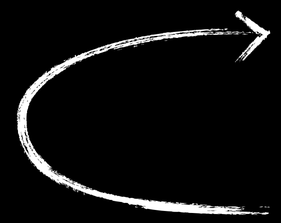


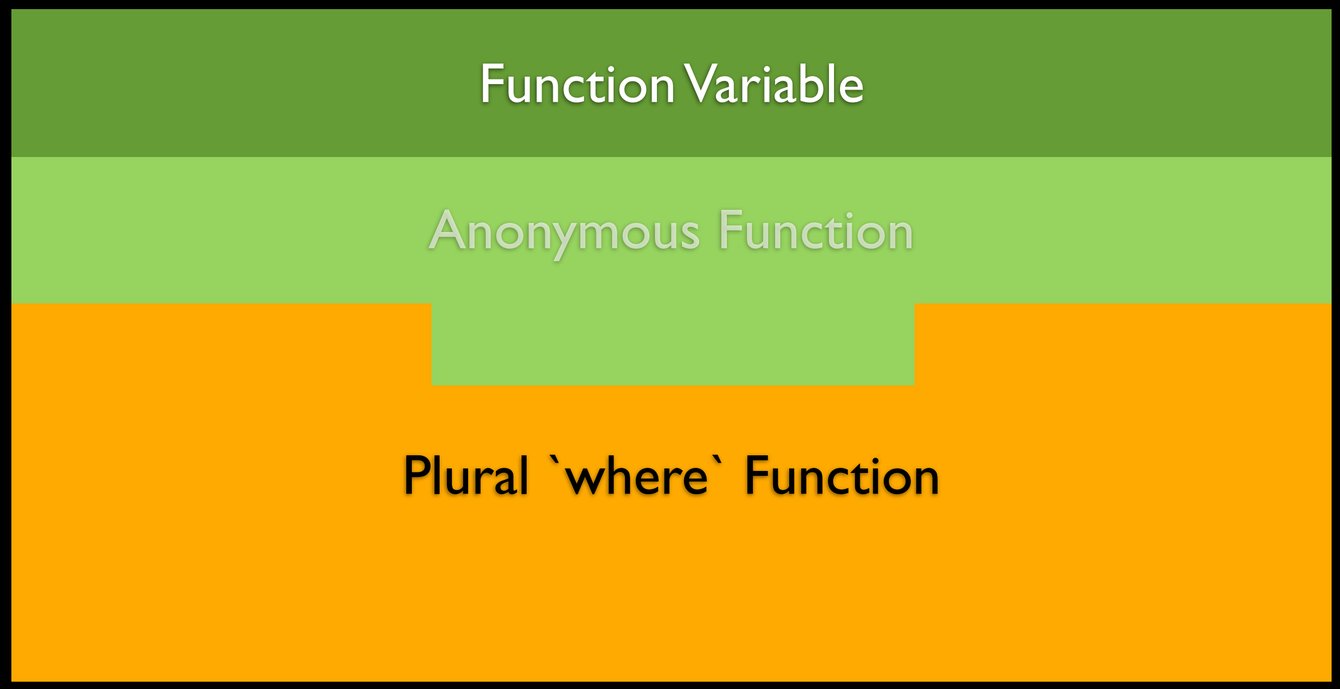

```
var gt5Changes = function(commit){
  return commit.changes > 5;
};
where(gt5Changes, commits);
```

@KrisJordan

Friday, July 20, 12

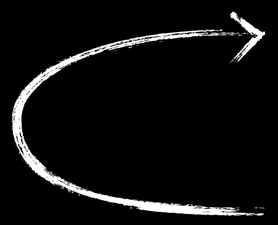
We had already been using a function variable but this is an interesting refactoring to make note of.

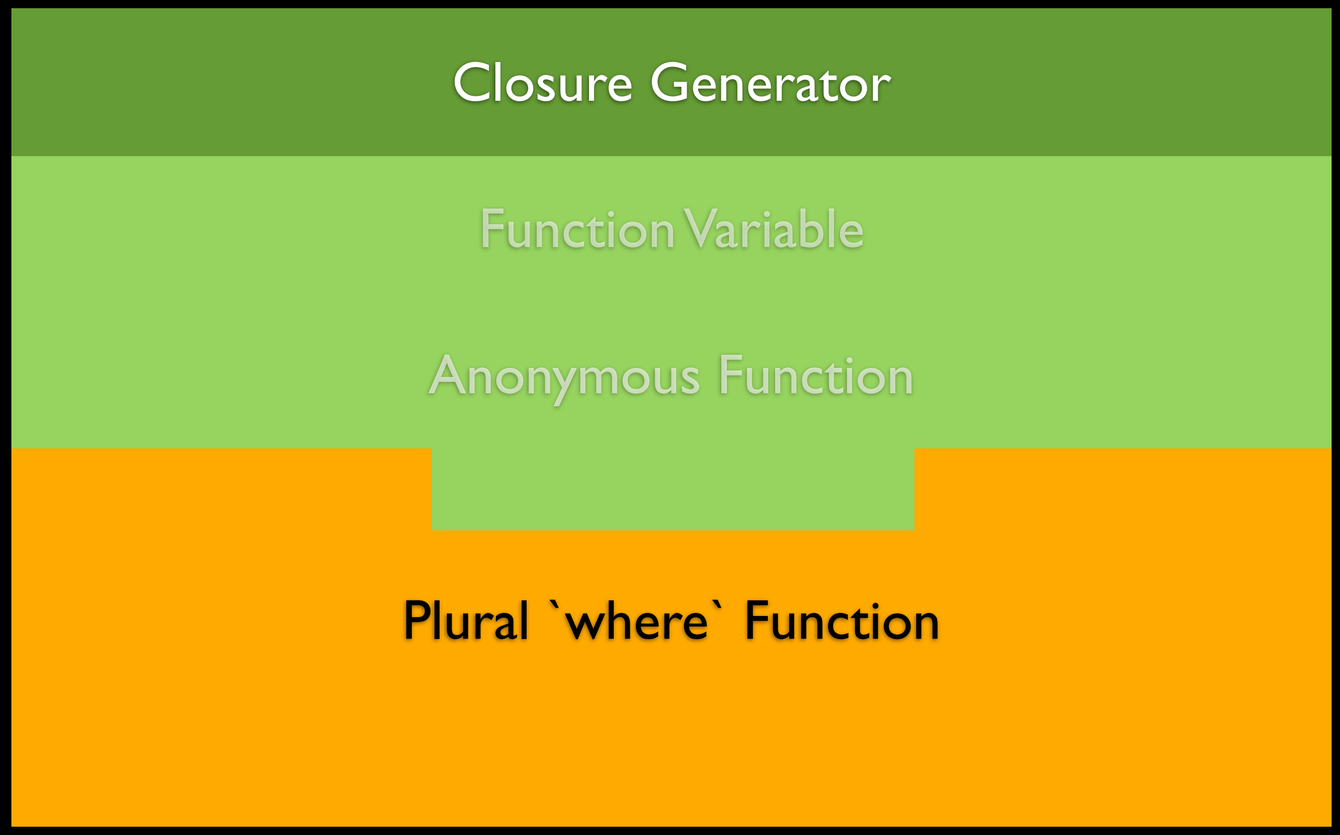




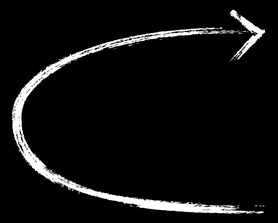

```
where(greaterThan("changes", 5), commits);
```

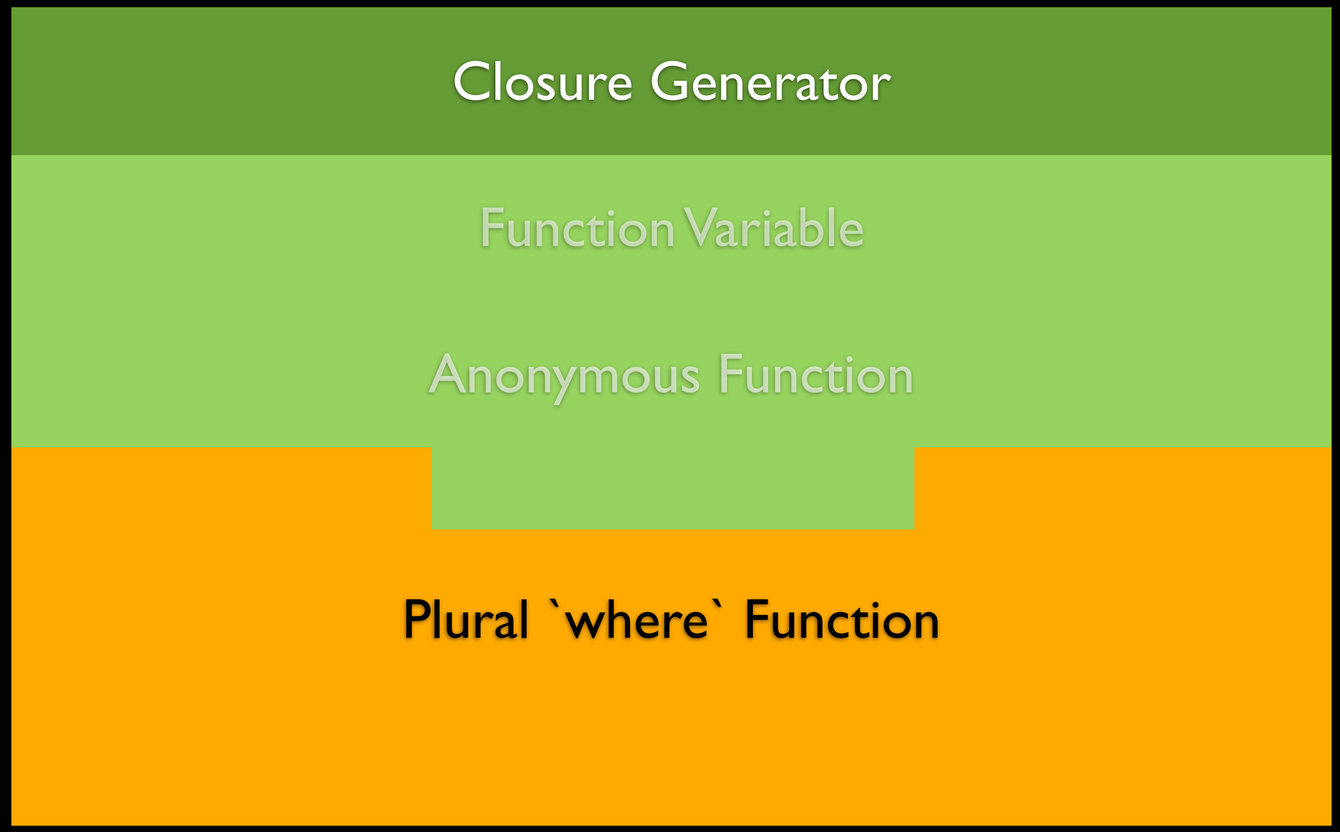
@KrisJordan






```
var greaterThan = function(prop, value){
  return function(item){
    return item[prop] > value;
  };
};
where(greaterThan(“changes”, 5), commits);
```






```
var greaterThan = function(prop, value){
  return function(item){
    return item[prop] > value;
  };
};
where(greaterThan("changes", 5), commits);
```

"Trapped" in Scope by **Closure**

For OOP folks closures are like the command pattern.

Demo

Show this refactoring.

f_underscore.js

- Companion library to underscore.js for welding and weilding iterator functions
- Dynamically generate expressions and predicate closures
- MIT licensed
- http://krisjordan.github.com/f_underscore/

Turns out you can do this for complete expressions.

Now let's refactor to use f_underscore.js

```
var greaterThan = f_.greaterThan,
    get = f_.get;

where(greaterThan(get("changes"),5), commits);
```

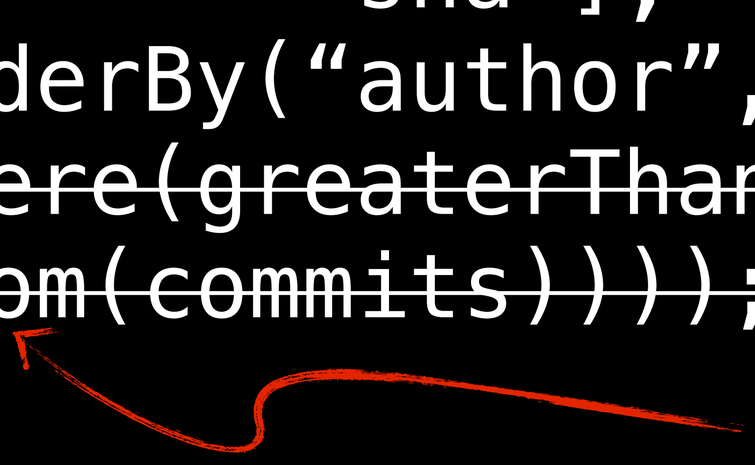

Note: we'll come back to `get`

Why `get`?

```
var get = function(prop) {
        return function(obj) {
          return obj[prop];
        };
    };

greaterThan(get("additions"),get("deletions")), commits);
```

What's left?

```
select(["author",
        "changes",
        "sha"],
orderBy("author",
~~where(greaterThan(get("changes"),5),~~
~~from(commits))));~~
```



`from` is _.identity

What's left?

```
orderBy("author", commits);
```

sortBy `_.sortBy(list, iterator, [context])`

Returns a sorted copy of **list**, ranked in ascending order by the results of running each value through **iterator**. Iterator may also be the string name of the property to sort by (eg. `length`).

```
_.sortBy([1, 2, 3, 4, 5, 6], function(num){ return Math.sin(num); });
=> [5, 4, 6, 3, 1, 2]
```

What's left?

```
orderBy("author", commits);
```

```
var orderBy = function(field, commits) {
  return _.sortBy(commits, f_.get(field));
};
```

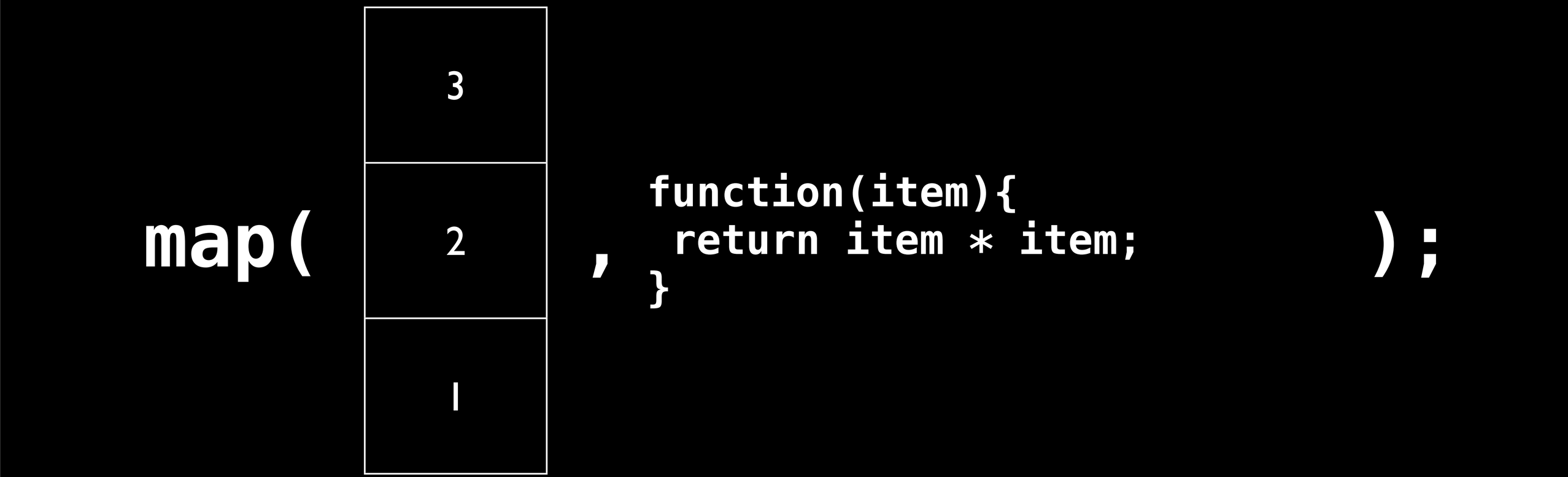
What's left?

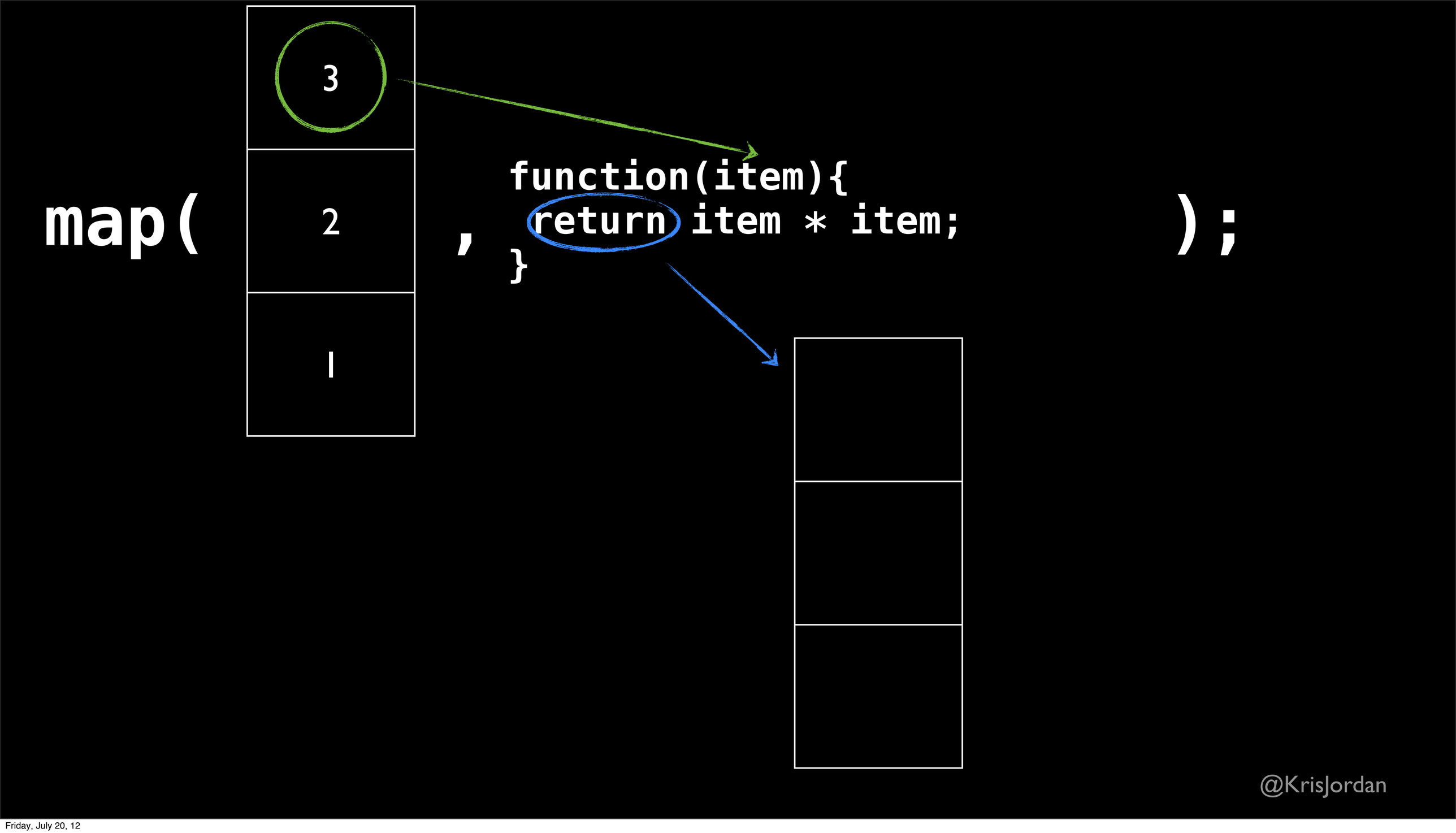
```
select(["author","changes","sha"], commits);
```

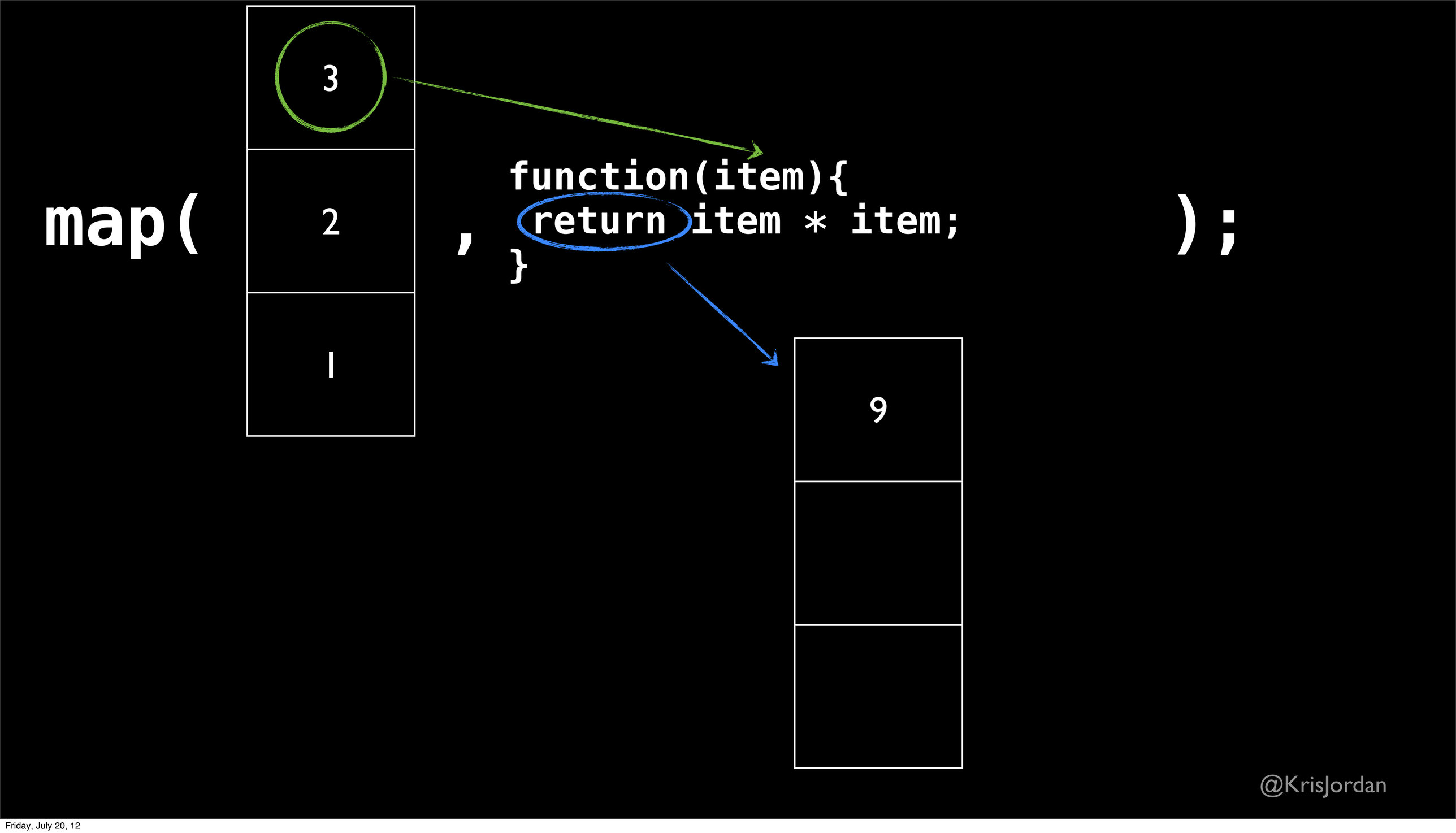
map `_.map(list, iterator, [context])` *Alias:* ***collect***

Produces a new array of values by mapping each value in **list** through a transformation function (**iterator**). If the native **map** method exists, it will be used instead. If **list** is a JavaScript object, **iterator**'s arguments will be `(value, key, list)`.

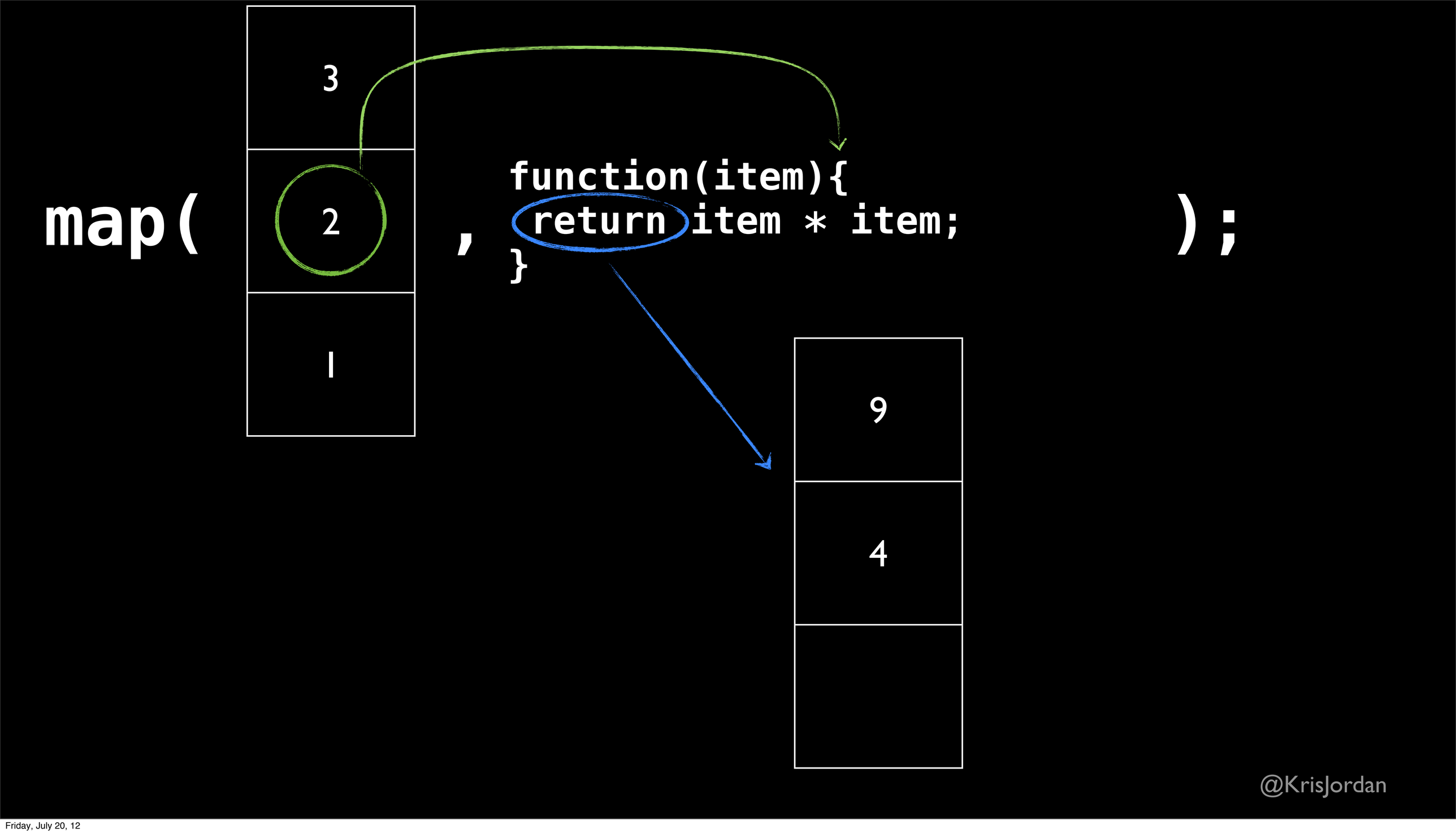
```
_.map([1, 2, 3], function(num){ return num * 3; });
=> [3, 6, 9]
_.map({one : 1, two : 2, three : 3}, function(num, key){ return num * 3; });
=> [3, 6, 9]
```


map(
3
2
1
, function(item){
return item * item;
}
);
@KrisJordan

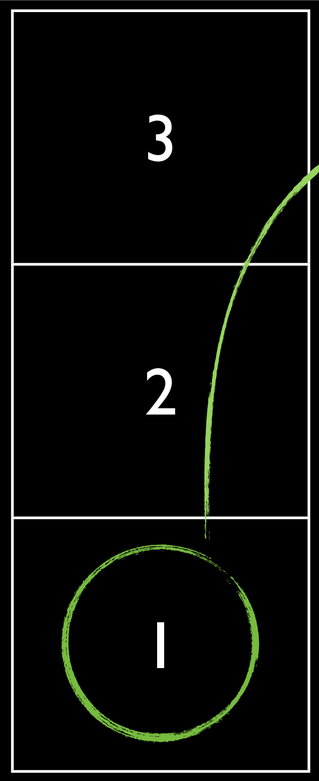

3
2
1
map(
,
function(item){
return item * item;
}
);
@KrisJordan

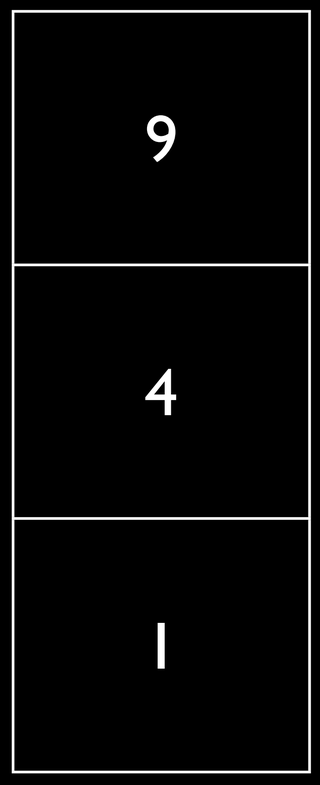


@KrisJordan


3
2
1
map(
,
function(item){
return item * item;
}
);
9
4
@KrisJordan

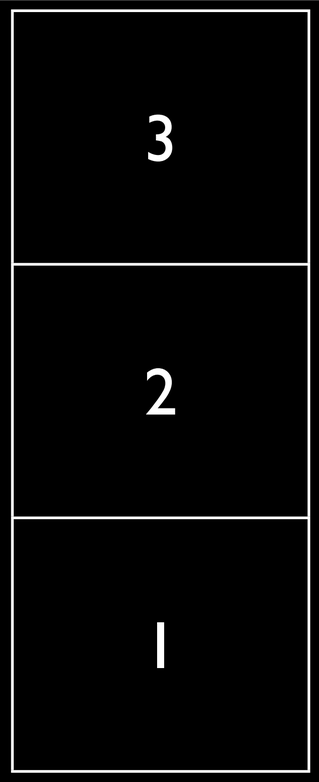
```
map(                    , function(item){
                            return item * item;
                        }                       );
```





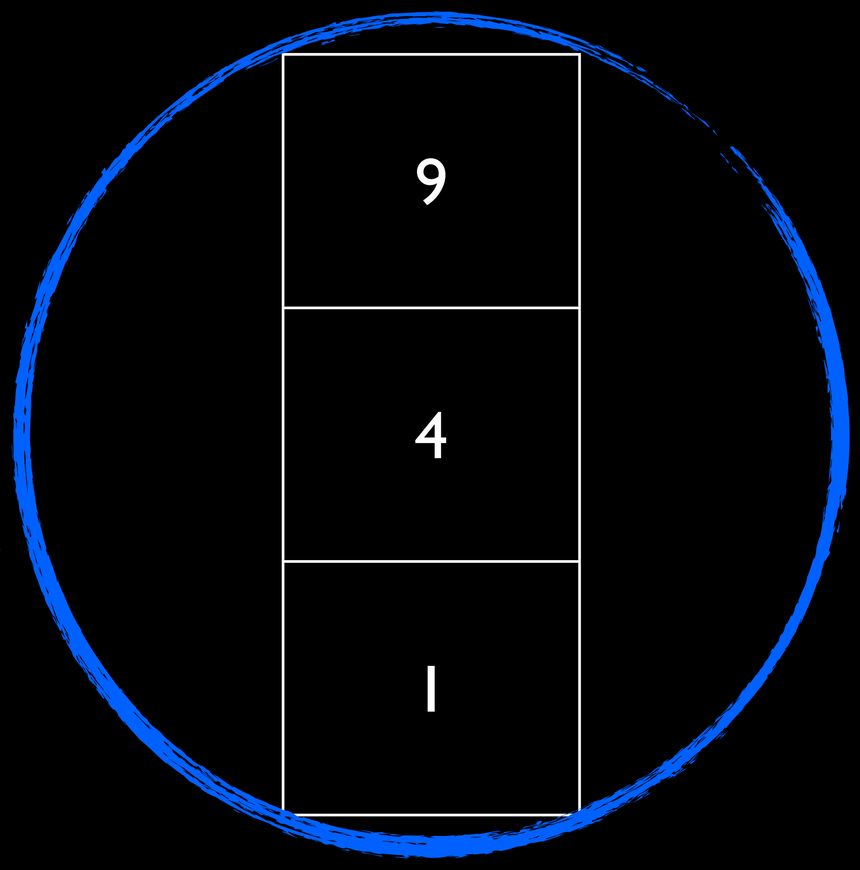
@KrisJordan

```
map(
```




```
, function(item){
    return item * item;
  }
);
```

return to caller



What's left?

```
select(["author","changes","sha"], commits);
```

project `f_.project(propertyNameArray) => iterator(object)`

Returns an **iterator** function that will create and return a new object with properties in the **propertyNameArray** whose values are copied from **object**.

```
var iceCream = {flavor:'Vanilla', size:'Large', cone:'Waffle'};
var projectFn = f_.project(['cone', 'flavor']);
projectFn(iceCream);
=> {cone:'Waffle', flavor:'Vanilla'}
```

```
var select = function(fields, commits) {
  return _.map(commits, f_.project(fields));
};
```

Complete Implementation of DSL

```
var greaterThan = f_.greaterThan,
    get = f_.get,
    select = function(fields, commits) {
      return _.map(commits, f_.project(fields));
    },
    orderBy = function(field, commits) {
      return _.sortBy(commits, f_.get(field));
    },
    where = function(predicate, data) {
      return _.filter(data, predicate);
    },
    from = _.identity
    ;
```


If you’re looking at this thinking, but wait, isn’t all we did here alias and change the order of parameters? You’re right.

Demo

If you’re looking at this thinking, but wait, isn’t all we did here alias and change the order of parameters? You’re right.

Our “DSL”

```
select([“author”,
        “changes”,
        “sha”],
orderBy(“author”,
where(
  greaterThan(
   get(“changes”),5
  ),
  from(commits))));
```

Idiomatic

```
_(commits)
 .chain()
 .filter(f_(“changes”).gt(5))
 .sortBy(f_(“author”))
 .map(f_.project(
   [“author”,
    ”sha”,
    ”changes”]))
 .value();
```

Recap

- Functions are data
- Refactor to functions as arguments
  - Abstract “plural” algorithm away from “singular” logic
  - Pass “singular” functions in
- Refactor to function/closure generators
  - Poor man’s macro
  - Parameterize iterator functions
- Check out underscore.js and f_underscore.js

@KrisJordan

- Functions are data
- Closures “trap” variable references in scope, useful for generating functions
- Write singular function generators, leverage with plural executor functions
- Underscore provides classic functions that are widely useful

Questions?

Friday, July 20, 12