Javascript Makes Me Happy

Joe Bowers

Codestock 2009

seriously.

Quick Poll

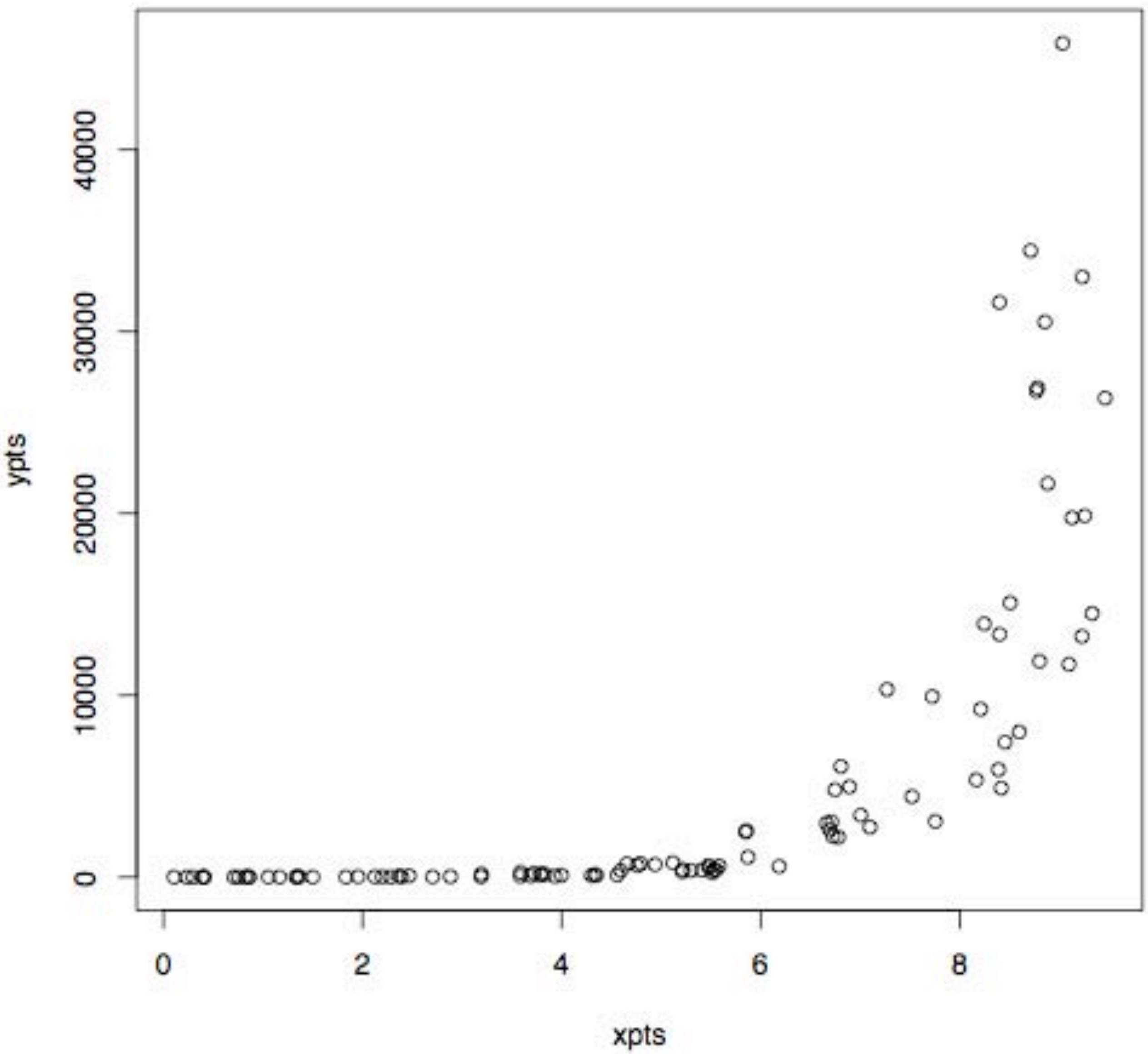
About Me

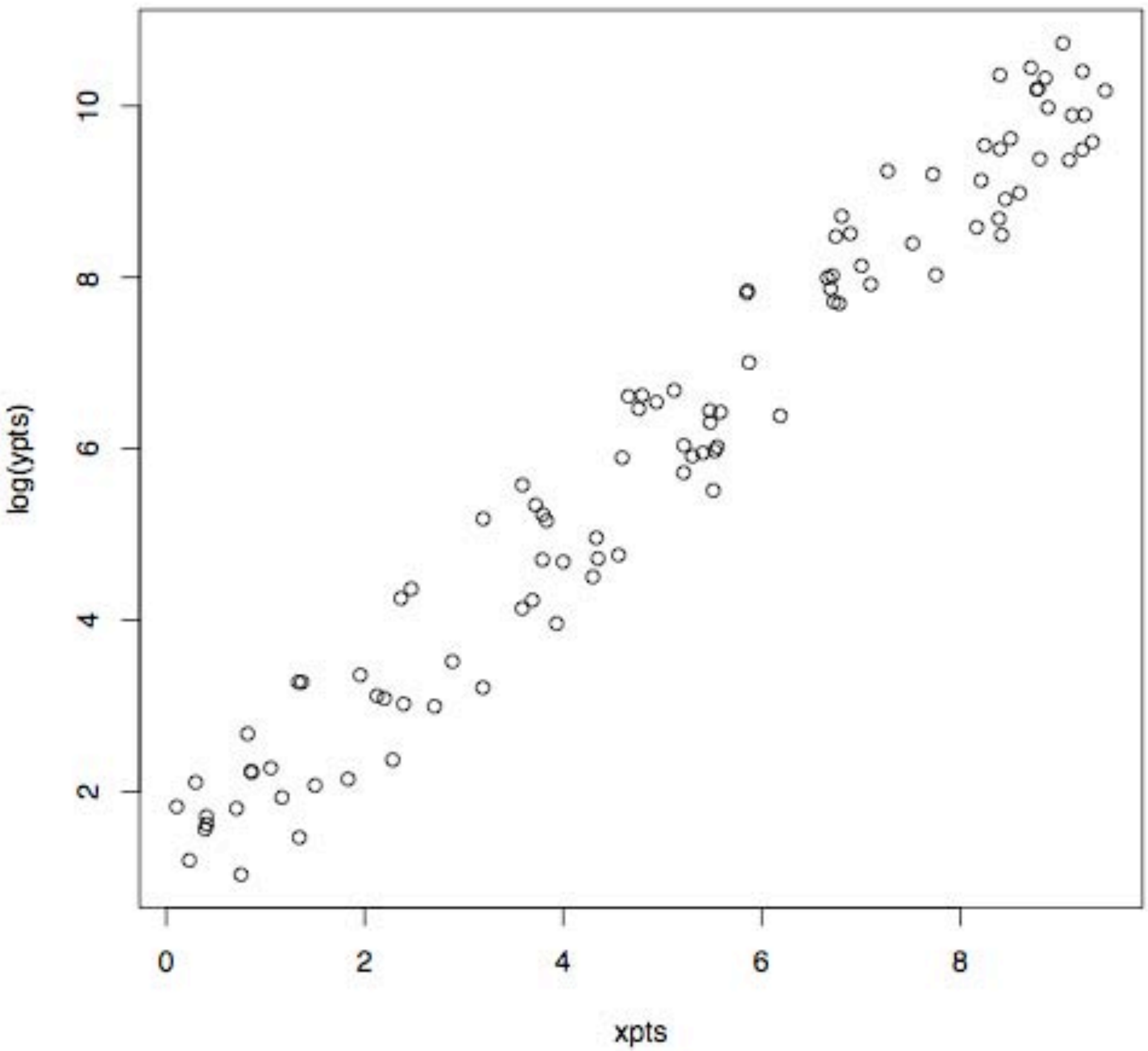
- Not a Javascript expert
- *Definitely* not a functional programming expert
- But pretty good at happy

Happy

Why Programming Computers Might Make One Happy

- It’s a good honest living
- It blows your freaking mind all the time


ypts
0
10000
20000
30000
40000
xpts
0
2
4
6
8



log(ypts)
2
4
6
8
10
xpts
0
2
4
6
8

How Javascript Can Make You Happy

- Let Javascript Be Javascript
- Let Javascript Suggest Ways to Manage Complexity

The Happy Javascript I’m going to talk about today

- Lightweight first-class functions
- Support for Closure
- Lightweight ad-hoc polymorphism

Some (Very) Basic Javascript

(Not the whole story)

Basic Javascript: Dictionaries

```
dict = {
   orange : 1,
   banana : 2,
   persimmon : 20034
};

dict["orange"]; // returns 1
dict.persimmon; // returns 20034
dict.grape = 32;
dict["pinapple"] = 3.432;
```

Basic Javascript: Iterating Through Dictionaries

```
for(var k in dict) {
    alert("The key == " + k +
          " has value == " + dict[k]);
}
```

- `k` iterates through the *key names* in `dict`
- *Not* a traditional “foreach” loop!

Basic Javascript: Arrays

```
an_empty_array = [];
another_array  = [ "a", "b", "c" ];

alert(another_array[0]); // alerts "a"
alert(another_array.length); // alerts "3"

for(ix=0; ix < another_array.length; ix++) {
  alert(another_array[ix]);
}
```

Basic Javascript: Functions

```
count_with_max = function(number) {
    var more = number + 1;
    if(more > 10) more = number;
    return more;
}

// call functions by appending parentheses
// (with arguments) to the
// end of a variable name.
count_with_max(22);
```

Functions are *Constructed*, not *Declared*

```
/* this */
function inc(number) {
  return number + 1;
}

/* is just syntactic sugar for this: */
inc = function(number) {
  return number + 1;
}
```

Event Handlers

```
complain = function() { alert("Ouch!"); }

user_button_widget.onClick = complain;
```

- Not part of the Javascript language
- But *Javascripty*, nontheless

Calling a Function Creates a Scope

- with parameters
- with the `var` keyword
- **All** other variables are global
- Function calls are the **only** scope

More Functions and Scope

```
count_with_max = function(number) {
    more = number + 1; // “more” is GLOBAL!
    if(more > 10) more = number;
    return more;
}

// NEW- "more" is in the local scope
count_with_max = function(number) {
    var more = number + 1; // local
    if(more > 10) more = number;
    return more;
}
```

Questions about Basic Javascript?

The Next Web 2.0 Sensation

```
current_count = 0;

count_with_max = function() {
    var more = current_count + 1;
    if(more <= 10) current_count = more;
    get_count_form("form1").counter_field.value =
        current_count;
}

bind_counter_event = function() {
   get_count_form("form1").counter_button.onclick =
        count_with_max;
};

execute_after_load(bind_counter_event);
```

Of Course, this Sucks

- leaving current_count in the global scope is anti-social.
- There is a better way!

```
build_counter = function(formName) {
   var current_count = 0;

   var count_with_max = function() {
       var more = current_count + 1;
       if(more <= 10) current_count = more;
       var countform = get_count_form(formName);
        countform.counter_field.value =
                current_count;
   };

   var bind_counter_event = function() {
        var countform = get_count_form(formName);
        countform.counter_button.onclick =
               count_with_max;
   };

   execute_after_load(bind_counter_event);
};// make_counter()

build_counter("form2");
```

Something Neat about build_counter()

```
build_counter("form3_1");
build_counter("form3_2");
```

The counter in these two forms *isn’t shared* and it *doesn't go away*

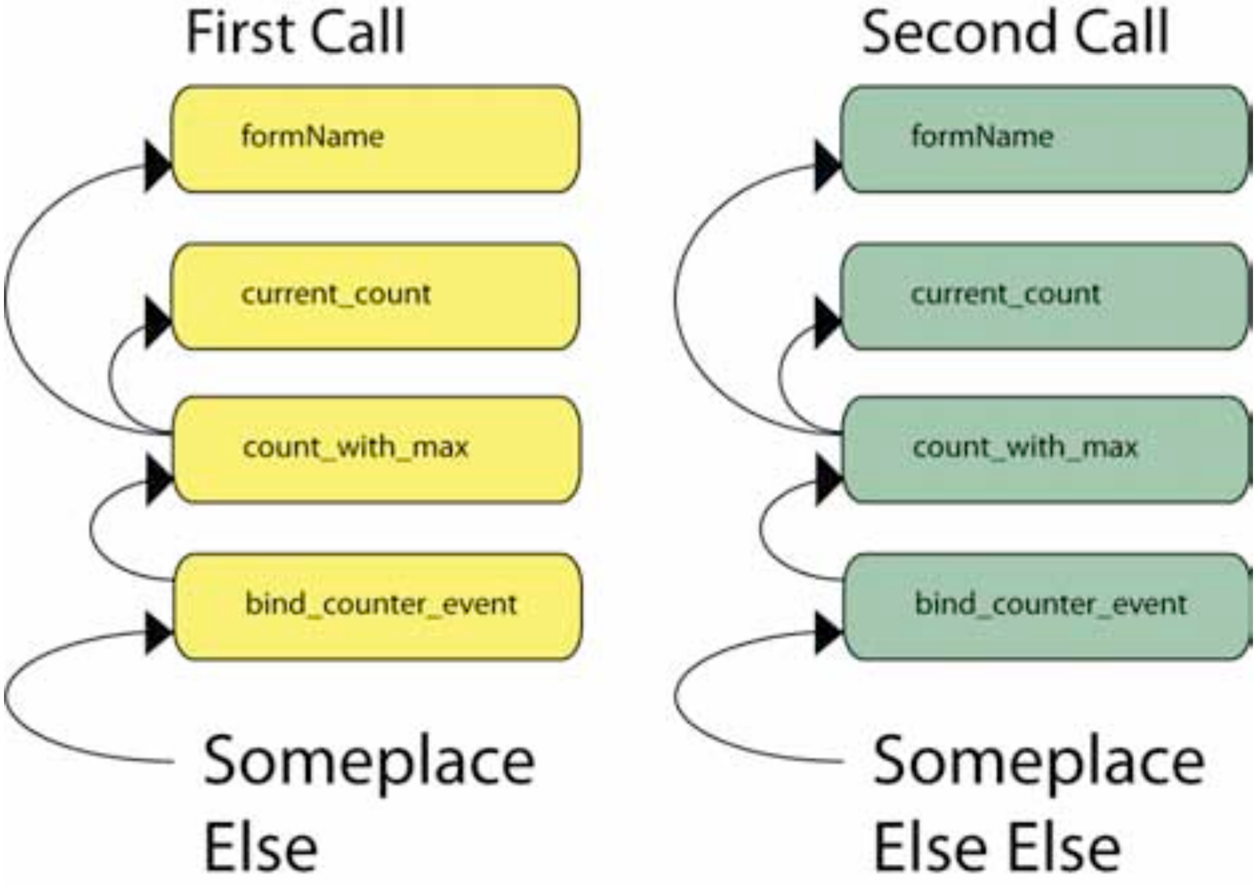
- Calling build_counter() creates a new, independent scope.

- build_counter() creates some brand new functions that refer to the new scope

- build_counter() binds those functions to global events, and then returns

- When the events occur, the handlers still refer to the scope in which they were created

- *The new scope doesn't go away when the invocation returns!*


First Call
formName
current_count
count_with_max
bind_counter_event
Someplace Else
Second Call
formName
current_count
count_with_max
bind_counter_event
Someplace Else Else

In Other Words

More generally, in Javascript:

Locally defined functions preserve references to the scope of the invocation where they are created. This phenomenon is called **Closure**

In The Same Words (1)

- *…the scope of the invocation where they are created*
  - Every time you call a function, you create a new, independent copy of its scope including all of its parameters and local variables.

In The Same Words (2)

- *Locally defined functions preserve their references…*
  - In this case, the references are to the parameters, and the variables declared with `var` in their enclosing scope or scopes
  - When the newly created functions are called later, they still have a reference to the particular execution context that existed when they were defined.

Once More (With Feeling!)

Locally defined functions preserve references to the scope of the invocation where they are created. This phenomenon is called **Closure.**

Do you feel good about this?

Why Closure Should Make You Happy

- Use Closure to Hide Information
- Use Closure to Share Information

A Bunch of Functions with Some Private, Shared State

- Gee, that sounds familiar…

```
make_counter2 = function() {
   var count = 0;
   var increase_fn = function() {
     var more = count + 1;
     if(more <= 10) count = more;
     return count;
   };

   var decrease_fn = function() {
     var less = count - 1;
     if(less >= 0) count = less;
     return count;
   };

   var current_fn = function() { return count; }

   return { increase : increase_fn,
            decrease : decrease_fn,
            current  : current_fn };
}//make_counter
```

In Action

```
counter = make_counter2();
counter.increase();
counter.increase();
counter.current(); // returns 2
counter.decrease();
counter.current(); // returns 1
```

Protocol?

```
mk_always_one = function() {
   return {
       increment: function() { },
       decrement: function() { },
       current: function() { return 1; }
   };
}//mk_always_one

mk_unbounded_up = function(){
   var val = 1;
   return {
       increment: function() { val = val + 1; },
       current: function() { return val; }
       // no decrement.
   };
}//mk_unbounded_up
```

Protocol!

- Dictionary names advertise services
- Let's call these advertisements *Protocols*

How Dictionaries Can Make You Happy

- Closures hide and share information
- Dictionaries organize information
  - by providing an with an identity
  - by conforming to a protocol

Quacks Like a Duck!

- `counter` is *encapsulated* - Closures
- `counter` has *identity* - Dictionaries
- `counter` is *abstract* - Protocols
- Inheritance? Eh.

The Happy Javascript I talked about just now

- Lightweight first-class functions
- Support for Closure
- Ultra-light ad-hoc polymorphism

The Point (so far)

- We've all seen this stuff before
- But what about the great stuff we *haven't* seen before?

Roll Your Own Control Structures

```
foreach_list = function(list, fn) {
  for(var i=0; i<list.length; i++) {
     fn(list[i]);
  }
}//each_list()

messages = ["you're","the man","now,","dogg"];

foreach_list(messages, function(msg) {
  alert(msg);
});
```

One Better: Iterators

```
eachable_list = function(list) {
   return {
     each : function(fn) { foreach_list(list, fn); }
   };
};//eachable_list

eachable_dict = function(dict) {
   return { each: function(f) {
              for(var k in dict) { f(dict[k]) }
              };
            }//each
           };
};//eachable_dict
```

Using an iterator

```
var movies = eachable_list([
 { title: 'The Blood of Dracula', stars : 3.5,
     producer: "Hammer Films" },
 { title: 'Death Race 2000 (1975)', stars : 4.0,
     producer: "Roger Corman" },
   ... // and a lot more
]);

movies.each(function(mv) {
       alert("Movie: " + mv.title);
});
```

The General Strategy: *Contextualizing Execution*

- thing.each(f) provides a context for f
  - in this case, the context is "over a collection"

- This is very similar to an event handler
  - context == "when the time is right"

- We can abstract these contexts to manage information about thing, and about f

```
map_iter = function(iter, fn) {
   var mapper = function(ifn) {
      iter.each(function (l) { ifn(fn(l)); })
   };
   return { each: mapper }
};

filter_iter = function(iter, fn) {
   var filterer = function(ifn) {
      iter.each(function (l) {
                  if(fn(l)) { ifn(l); }
                });
   };//filterer

   return { each: filterer }
};
```

The General Strategy: *Transforming Functions*

- map_iter and filter_iter take iterators as arguments
- map_iter and filter_iter return iterators
- iterators are really just *functions*

```
concat_iter = function(iter1, iter2) {
   var concat = function(ifn) {
     iter1.each(ifn); iter2.each(ifn);
   };

   return { each: concat };
};

all_iter = function(iter_of_iters) {
   return { each: function(f) {
                    var once =
                        function(iter) { iter.each(f); }
                    iter_of_iters.each(once);
                  }
          };
};


each0 = { each: function(f) { ; } };
```

And one more

```
concatmap = function(iter1, iter2, f) {
   return map_iter(concat_iter(iter1, iter2), f);
};
```

General Strategy: *Composing Functions*

- concatmap transforms it's arguments by composing other transforms

```
// Ick!
cormans_worst = function(movies) {
   return map_iter(
            filter_iter(
              filter_iter(movies,
                          function(m) {
                            return (m.stars <= 2);
                          }),
              function(m) {
                return (l.producer == "Roger Corman");
              }),
            ),
            function(l) { return l.title; }
          );
}// cormans_worst
```

What I Would Like to See

```
hammers_best =
   Query.
     FROM(movies).
     WHERE(function (m){ return (m.stars > 4) }).
     WHERE(function (m){
       return (m.producer == "Hammer Films")
     }).
     SELECT(function (m) { return m.title; });
```

```
query = function(iter) {
    return {
      SELECT : function(f) { return query(map_iter(iter, f)); },
      WHERE  : function(f) { return query(filter_iter(iter, f));},
      FROM : function(iter2) {
              return query(concat_iter(iter,iter2));
      },
      TAKE : function(n) {
        var left = n;
        var filterer = function(l) {
          var ret = (left > 0); left--; return ret;
        };
        return query(filter_iter(iter, f));
      },
      each : function(f) { iter.each(f); }
    };//return
};// query()

Query = query(each0);
```

hammers_best is an Iterator

```
hammers_best =
   Query.
     FROM(movies).
     WHERE(function (m){ return (m.stars > 4) }).
     WHERE(function (m){
       return (m.producer == "Hammer Films")
     }).
     SELECT(function (m) { return m.title; });

hammers_best.each(function(mtitle) {
    alert(mtitle + " is one of hammer's best films!");
}
```

General Strategy:
Method Chaining of Contexts

- Query is a collection of execution contexts
- Each step (.WHERE(), .SELECT(), etc) returns a context for running the next step

General Strategy: *Delayed Application*

- The results of a query are another query, *not* a collection
- We can pass partial queries around, build potentially expensive queries at low cost, etc.

Enough!

- Javascript has particular properties that suggest particular abstractions
- If you look for these abstractions, you can be happy using Javascript
- And maybe even take some of that happy back home to C++!

More Resources: Javascript

- Buy a book and read it
- Javascript is different enough to justify reading a book.
- But simple enough that it will be a quick read!
- And I don't think it matters much *which* book

More Resources: Javascriptiness

- jQuery and Prototype both make an effort to be Javascripty, it's worth learning one or the other.
- There is a "jQuery 101" session tomorrow
- Lots out there on the web!

More Resources: Higher Order Programming

- Check out some Functional programming language
  - Maybe F#?
  - Maybe Haskell?
  - Maybe Scheme?

More Resources: Happy

- The works of _why the lucky stiff, Freelance Professor
- Your friends!