2 Sessions

JavaScript Primer

Node Training

© nearForm

A New Look at JavaScript

- Embracing JavaScript
- JavaScript Data Structures
- JavaScript Functions
- Functional JavaScript
- JavaScript Objects

The Truth about JavaScript

- JavaScript looks like a toy language
  - It has a lot of really bad features:
  - `eval`, `with`, implicit global leaking
  - the DOM - it's terrible, but it's not JS, it's a browser API
- But it's also been misrepresented, misunderstood and misused for years.
- Any way you look at it, there's an inescapable fact:
  - JavaScript is the universal language of the web

nearForm

The Truth about JavaScript

- It has some really great features:
  - standardized under the name ECMAScript, latest version 5.1 (version 6 by 2015)
  - functional, you get callbacks and closures
  - prototype-based, hash maps are baked-in
  - has a literal data syntax (of which JSON is a subset)
  - has low code volume, while remaining readable

nearForm

JavaScript Data Structures

- The fundamental data structure is the "object"
  - essentially a hash table of string properties referencing values of any type
  - there are no classes, but each object has a prototype object that it can inherit properties from
  - arrays are just objects with numbers for properties, a special length property accessor plus some extra methods
- The best thing about objects is that they can be written as "literals", that is, a superset of the JSON format.

nearForm

Object Literals

- Objects are a set of key:value pairs defining properties
  - The keys (property names) are strings (always)
  - and the values can be anything, including other objects (this allows nested data structures)
- The literal syntax for defining an object is:
  - `{<key1>:<value1>, <key2>:<value2>, ...}`
  - Example:
    - `var objname = {firstName: 'Richard', lastName: 'Rodger'}`

nearForm

Object Literals

- You can then access the values using the following notations:
  - **dot notation**, where the property name must be a valid identifier:
    ```
    objname.firstName
    ```
  - **square bracket notation**, where the property name can be specified using a literal or a variable:
    ```
    objname['firstName']
    ```
- ```
  var propertyName = 'firstName';
  objname[propertyName];
  ```

nearForm
  ```

Array Literals

- Arrays are an ordered list of values (of any type)
- The literal syntax is:
  - `[<value1>,<value2>,...]`Example:
  - `['foo', true, 11, {a: 1}]`You can access elements using standard square bracket notation:
  - `myArray[0] // is 'foo'`

nearForm

Array Literals

- Arrays are just special objects, that can be initialised with a literal syntax
  - the array indexes are not numbers, but properties - the index number is converted into a string:
    - `myArray['0'] // is also 'foo'` With a special `length` accessory property,
  - and some utility methods:
    - `push`, `pop`, `shift`, `unshift`, `join`, `split`, `forEach`, `map`, `filter`, `reduce`

nearForm

JSON format

- Combine object and array literals, enforce a rule subset, and you get JSON syntax - JavaScript Object Notation
- See http://json.org for a formal specification
- The ECMA standard (5th ed) defines a utility object for JSON:
  - `JSON.parse()` returns a JavaScript object defined by literal string in JSON format
    - use this for input
  - `JSON.stringify(obj)` renders the JavaScript object as a string in JSON format
    - use this for output

nearForm

JavaScript Functions

- Fundamental unit of code
  - they are also objects, and can have properties
- Can be created ...
  - as declarations
  - as expressions
  - as methods
  - anonymously

nearForm

JavaScript Functions

- Can be called ...
  - as functions
  - as methods
  - as constructors
  - dynamically using the call and apply methods

nearForm

Function Declarations

```
function name( ... ) { ... }
```

- Defining a function creates a new variable scope - variables declared inside the function cannot be accessed outside the function
- Useful for top level and utility functions
- You can also define functions inside other functions
- EcmaScript 5 does not have block scope, only function scope
- EcmaScript 6 has block scope with `const` and `let`.

nearForm

Function Expressions

```
var name = function( ... ) { ... }
```

- Useful for dynamically created functions
- Example is an anonymous function expression
- Naming the anonymous expression makes more helpful stack traces
  - `var name = function name( ... ) { ... }`
- Function expressions and declarations are called in the same way
  - `name(argument1, argument2, ...)`

Variable Hoisting

```
function eg() {
  console.log(a); //undefined
  var a = 1;
  console.log(a); //1
  console.log(b); //throws
}
```

- Variables can be declared anywhere in a function
- However, it will be as if they were declared at the top of the function with a value of undefined until the point of assignment.
- This can be confusing, so it's better to declare variables at the top of functions

nearForm

Function Hoisting

```
b(); // doesn't throw
a(); // throws
var a = function () { }
function b () { }
```

- Function declarations
  - Are hoisted to the top of the current scope
  - So you can use the function before it is defined

nearForm

Function Hoisting

```
b(); // doesn't throw
a(); // throws
var a = function () { }
function b () { }
```

- Function expressions do not hoist
  - Variable hoisting takes precedent
  - The variable that references the function is `undefined` until the point in the code where the function expression is assigned to the variable (the same rule for all variables)

nearForm

Methods

```
var obj = {
  name: function(x) { return x + 1 }
}

obj.name = function(x) { return x + 2 }
obj.newFunc = function(y) { return y + 3 }
```

- Methods are function expressions
  - the function expressions are assigned to the property of an object
- You can redefine the functions of an object at any time, or add new ones:
  - They must be defined before use

nearForm

Anonymous Functions

```
(function () { }());
```

- Anonymous functions are simply unnamed function expressions
  - Can be useful for Immediately Invoked Function Expressions (IIFE's)
- The example is an IIFE - a function wrapped in brackets tells the interpreter that the function is an expression instead of a unnamed declaration (which is a syntax error)

nearForm

Anonymous Functions

```
(function () { }());
```

- Anonymous functions are often used in “callbacks” - when a function needs another function as an argument
- `setTimeout(function(){ ... }, 1000)`
  - You're often better off naming functions, it makes debugging easier

nearForm

Function Invocation

```
var foo = true;
function printFoo() {
  console.log(foo);
}
foo = false;
printFoo(); // prints false
```

- When a function is called ("invoked"), it has access to two special variables:
  - `this`
  - `arguments`
- The function also has access to a "closure"
  - all variables in scope when the function is defined

nearForm

this

- references the global object by default
- if the function is a method, `this` refers to the object on which the method exists
- if the function is a prototype method, `this` refers to the instance in front of the prototype chain
- `Function.prototype.apply` and `Function.prototype.call` can be used to set this dynamically
- `Function.prototype.bind` a partially applies a function with a new this context

nearForm

arguments

```
for (var i = 0; i < arguments.length; i++) {
  //do something with arguments[i]
}
```

- the `arguments` variable is not an array (unfortunately),
- It does have a `length` property, so you can iterate through the arguments using a `for` loop
- The `for` loop is the most performant way to deal with arguments object (by a long way).
- the `arguments` object allows you to define functions that have a variable number of arguments

nearForm

Method Invocation

```
var obj = {
  foo: 11,
  print: function() {
    console.log(this.foo)
  }
}
obj.print() // prints 11
```

- When a function is a property of an object, it is known as a “method”
- The special `this` variable references the object

nearForm

Losing "this" across scopes

```
var obj = {
  foo: 11,
  print: function() {
    setTimeout(function() {
      console.log(this.foo)
   }, 1)
  }
}
obj.print() //undefined
```

- The `this` context is does not carry into nested function scopes
- Common is case callbacks, the this variable may no longer refer to your object.

nearForm

Preserving "this" across scopes

EcmaScript 3

```
var obj = {
  foo: 11,
  print: function() {
    var self = this;
    setTimeout(function() {
      console.log(self.foo)
    }, 1);
  }
};
```

nearForm

Preserving "this" across scopes

EcmaScript 5

```
var obj = {
  foo: 11,
  print: function() {
   setTimeout(function() {
       console.log(this.foo)
    }.bind(this), 1)
  }
}
```

nearForm

Preserving "this" across scopes

EcmaScript 6

```
var obj = {
  foo: 11,
  print: function() {
    setTimeout(() => {
      console.log(this.foo)
    }, 1)
  }
}
```

nearForm

Constructor Invocation

```
function Foo() { ... }

var foo = new Foo()
```

- The `new` operator declares an alternative form of invocation
- When the function executes, a new object is created
- The `this` variable points at the new object, and is the default return value of the function
- Functions intended to be used in this way are given a capital letter by convention, as they can be thought of as traditional classes (they aren't)
- This "feature" will be examined in more detail shortly...

nearForm

Invocation using call

```
function printFoo(bar) {
  console.log(this.foo + bar)
}

var obj = {foo: 1}
printFoo.call(obj, 2) // prints 3
```

- Functions inherit methods from the prototype of the native `Function` constructor
- The `call` method is particularly interesting, and is used like so:
- `myFunction.call(context, arg1, ..argN)`
  - This calls the `myFunction` function, setting the `this` variable to the context object, passing all subsequent `arguments` as parameters of the target function

nearForm

Invocation using call

```
function printFoo(bar) {
  console.log(this.foo + bar)
}
var obj = {foo: 1}
printFoo.call(obj, 2) // prints 3
```

- You can dynamically call any function with any list of arguments, and you can make the function act like a method of any object you like
- Useful trick - turn the arguments object into a real array with
  - `var args = Array.prototype.slice.call(arguments)`
    - Unfortunately JIT compilers are terrible at optimising this scenario

nearForm

Invocation using apply

```
function printFoo(bar, baz) {
  console.log(this.foo + bar + baz)
}
var obj = {foo: 1}
printFoo.apply(obj,[2, 4]) // prints 7
```

- Exactly like `call` invocation, except an array of arguments can be passed as the second argument

nearForm

Invocation using apply

In certain cases, `apply` keeps code DRYer

```
var o = {
  count: 0,
  total: 0,
  totaller: function (a, b, c) {
    this.total = a + b + c;
  }
}
function totallerProxy() {
  this.count += 1;
  this.totaller.apply(this,
    Array.prototype.slice.call(arguments));
}

totallerProxy.call(o, 1, 2, 3);
```

nearForm

Partial Invocation

A.K.A Partial Application

```
var o = {
  count: 0,
  total: 0,
  totaller: function (a, b, c) {
    this.total = a + b + c;
  }
}
function totallerProxy() {
  this.count += 1;
  this.totaller.apply(this,
    Array.prototype.slice.call(arguments));
}.bind(o);

totallerProxy(1, 2, 3);
```

nearForm

Functional Programming

- Mantra: everything is a Function
  - versus: everything is an Object
- Based on the Lambda Calculus...
  - which is a way of talking about "computable" things:
    - Calculate PI: computable? Yes.
    - Will this random program ever stop? No.
- Frequently uses anonymous functions
  - often as event handlers, like anonymous inner classes in Java

nearForm

Functional JavaScript

- As with object-oriented patterns, there are functional patterns:
  - Callbacks
  - Dynamic functions
  - Recursion
  - EcmaScript 5 functional extensions
  - The underscore library

nearForm

Callback Functions

```
fs.readFile('mine.txt', function(err, data) {
  if (err) {
    return console.log('error:' + err)
  }
  doStuff(data)
)
```

- Pass a function into another function as an argument.
- The function passed in is “called back” later by the other function.
- The passed-in function tends to be anonymous and defined in-place
- The most common API pattern for Node.js modules

nearForm

Dynamic Functions

```
function err (win) {
  return function (err, data) {
    if (err) { return console.log('error:' + err); }
    win(data)
  }
}
fs.readFile( 'mine.txt', err(function (data) {
  doStuff(data)
}))
```

- Dynamically create functions when you need them
  - Useful for wrapping other functions
  - handle common cases – such as error handling
  - extend functionality – add a throttle to database requests to reduce load

nearForm

Recursive Functions

A function that calls itself

```
function printObject(obj,indent) {
  for (property in obj) {
    var value = obj[property]
    if ('object' === typeof(value)) {
      console.log(indent + property + ':')
      printObject(value, '  ' + indent)
    } else {
      console.log(indent + property + ':' + value)
    }
  }
}
```

nearForm

Recursive Functions

- Often provides a more concise solution than iterating
  - Beware stack overflow however!
  - Can be solved with the trampoline pattern (essentially flattens the call stack)
  - However it's slow - ideally code should be rewritten
  - EcmaScript 6 has Tail Call Optimization
- Stick to mutually exclusive recursion,
  - mutually dependant recursion becomes very difficult to follow and debug

nearForm

EcmaScript 5 functional additions

- Provides a range of utility function for functional programming
  - See http://mzl.la/1kML43T
- Examples:
  - `Array.prototype.forEach`
    - applies a function to each element of an array
  - `Array.prototype.map`
    - returns a new array of values based a supplied transform function

nearForm

EcmaScript 5 functional additions

- `Array.prototype.reduce`
  - reduces an array down to one value - e.g summing
  - can also be used to build an object from an array
- `Object.keys`
  - Returns an array of enumerable keys (use Object.getOwnPropertyNames for all keys, including non-enumerable)

nearForm

The underscore library

- Provides a range of utility function for functional programming
- See - underscorejs.org
- Examples:
  - `_.each`, `_.map`, `_.reduce`
  - as the `Array.prototype` methods, but the array is passed as first argument, e.g
    - ```
      _.map(arr, function(item) {
        return ...;
      })
      ```
  - `_.object`
    - convert two arrays into key value pairs

nearForm
      ```

The underscore library

- `_.isEqual`
  - object comparison by value rather than reference
- `_.compose`
  - combine multiple functions into one function
  - There's also lodash - lodash.com

nearForm

JavaScript: Objects

- Object-oriented, but not Class-oriented
- Everything is an Object
  - `(function(){}) instanceOf Object //true`
  - `[] instanceof Object //true`
- Even string and number primitives are facades for special objects
  - `1..constructor // function Number() { }`
  - `1..toString() // '1'`
  - `'s'.constructor // function String () { }`

nearForm

JavaScript: Objects

- All objects have a prototype
  - This is `Object.prototype` by default
  - gives you `.toString()`, `.hasOwnProperty()`, etc.
- When you ask an object for a property, it will
  - return the value of the property if the object has that property
  - OR: look for the property in its prototype object
  - and so on, up the chain of prototypes

nearForm

Defining Object Properties

- Within a literal (most performant)
- Using dot notation (most terse)
- Using square brackets (only if it can't be dotted)
- In ES5 set advanced properties using descriptor objects

nearForm

Defining Object Properties

Data Descriptors

```
{
  value: Any,
  configurable: Boolean[false],
  writable: Boolean[false],
  enumerable: Boolean[false],
}
```

nearForm

Defining Object Properties

Accessor Descriptors

```
{
  get: Function,
  set: Function,
  configurable: Boolean[false],
  writable: Boolean[false],
  enumerable: Boolean[false],
}
```

nearForm

Defining Object Properties

- Descriptors work with `Object.create`,`Object.defineProperty` and `Object.defineProperties`
- All boolean properties on descriptors default to `false`

nearForm

Defining Object Properties

Default Behaviour

```
var o = {};
Object.defineProperty(o, 'ro', {
  value: 'look but do not touch'
})
o.ro = 'changed';
console.log(o.ro); //..didnt update - non writable
delete o.ro;
console.log(o.ro); //..nope non - writable
console.log(Object.keys(o)) //empty array non enum

//throws typeerror, non-configurable
Object.defineProperty(o, 'ro', { value: 'redef' })
```

nearForm

Defining Object Properties

Accessors and re-configuring properties:

```
var o = Object.defineProperties({}, {
  ro: {
    configurable: true, value: 'look but do not touch'
  },
   magic: {
     set: function (v) {
        Object.defineProperty(this, 'ro', {
          configurable: true, writable: true,
        })
        console.log('setting ro to ', v);
        this.ro = v;
        Object.defineProperty(this, 'ro', {
          configurable: true, writable: false,
        })
      },
     get: function () { return this.ro; }
   }
});
o.ro = 10; console.log(o.ro); //unchanged
o.magic = 10; console.log(o.ro); //OMGOSH!!!11!
```

nearForm

Locking Objects

`Object.preventExtensions` - no new properties, property values can be updated.

```
var o = {foo: 1};

Object.preventExtensions(o);
console.log(Object.isExtensible(o)); //false
o.foo = 2; //yep, changes
o.moo = 3; //doesn't get added
Object.defineProperty(o, 'foo', {
  get: function () {
    return 'alright'; }
  });

console.log(o.foo); //alright
delete o.foo;
console.log('foo' in o, o.foo);//false undefined
```

nearForm

Locking Objects

`Object.seal` - no new props, no deleting props, no redefining props. Property values can be updated

```
var o = {foo: 1};

Object.seal(o);
console.log(Object.isSealed(o)); //true

o.foo = 2; //yep, changes
o.moo = 3; //doesn't get added
delete o.foo;
console.log('foo' in o, o.foo); //true 2

Object.defineProperty(o, 'foo', {
  get: function () { return 'alright'; }
}); //throws TypeError
```

nearForm

Locking Objects

`Object.freeze` - full lock down, no changing/adding/configuring or deleting properties

```
var o = {foo: 1};
Object.freeze(o);

console.log(Object.isFrozen(o)); //true

o.foo = 2; //doesn't change
o.moo = 3; //doesn't get added

delete o.foo;
console.log('foo' in o, o.foo); //true 1

Object.defineProperty(o, 'foo', {
  get: function () { return 'alright'; }
}); //throws TypeError
```

nearForm

How Prototypes Work

- The prototype of an instance (known as the `__proto__`) is determined by the prototype object of its constructor.
- A constructor is just a function, called with `new`:
- `var F = function(){}`
  - Use the new operator to instantiate.
- ```
  F.prototype = {red: 1} //don't do this IRL
  var o = new F()
  o.red // === 1
  ```

nearForm
  ```

How Prototypes Work

- The Chain:
  - ```
    o.__proto__ === F.prototype //true
    o.__proto__.__proto__ === Object.prototype// true
    o.__proto__.__proto__.__proto__ === null // true
    ```
- JavaScript objects:
  - inherit properties from their prototypes, but the properties belong to the prototype object, not to the instance object
  - in the case of our `F` instance the `hasOwnProperty` method returns `false`:
  - `o.hasOwnProperty('red') // false`

nearForm
    ```

How Prototypes Work

- But a property of the same name can be overlaid on the instance
  - ```
    o.red = 2
    o.hasOwnProperty('red') // true
    delete o.red;
    o.red === 1 //true
    ```

nearForm
    ```

Creating a custom prototype chain

```
function F() { this.red = 1; }
function G() { this.blue = 2; }
G.prototype.color = function (col) {
  if (col in this) { return this[col]; }
  return 'nope';
}
F.prototype = new G;

var instance = new F;
instance.color('red'); //1
instance.color('blue'); //2

instance.__proto__ === F.prototype
instance.__proto__.__proto__ === G.prototype
instance.__proto__
  .__proto__.__proto__ === Object.prototype
instance.__proto__
  .__proto__.__proto__.__proto__ === null
```

nearForm

The new Operator

- JavaScript glosses over its prototypical nature using the `new` operator
- It's just setting up the prototype chain:
- `var me = new Person('My Name')`
  - step 1: create a new function (me), and set it's prototype to be `Person.prototype
  - step 2: call the Person function, setting the this variable to be the new function (me), with any arguments passed in ('My Name')
  - step 3: if the new function (me) returns an object, return that object, otherwise return the new function (me)
- The purpose of this logic is provide syntax sugar that gives the appearance of a Class-based language

Object.create

- The ECMAScript 5 standard defines a new method `Object.create`, for creating objects
- `Object.create` sets up the prototype chain in a sane manner - you pass in the object that should be the prototype, and you get back a new object with this prototype:
- ```
  var a = {red: 1}
  var b = Object.create(a)
  b.red === 1
  ```
- `Object.create` can also be used to inherit from `null`
  - ```
    var pureObject = Object.create(null);
    pureObject.toString //undefined
    ```

nearForm

Object.create polyfill

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

- `Object.create` can be (partially) polyfilled in ES3
- This polyfill is partial, because
  - It only covers the first parameter of `Object.create`
  - It cannot set a `null` prototype
- The second parameter (`propertiesObject`) can't be implemented in ES3.

nearForm

Inheritance

- Unlike class-based languages, inheritance is not "baked-in"
- You use prototypes to implement different styles of inheritance
- There are trade-offs:
  - variable conflict safety
  - memory usage / performance
  - different conceptual models
- Different models are used by different libraries

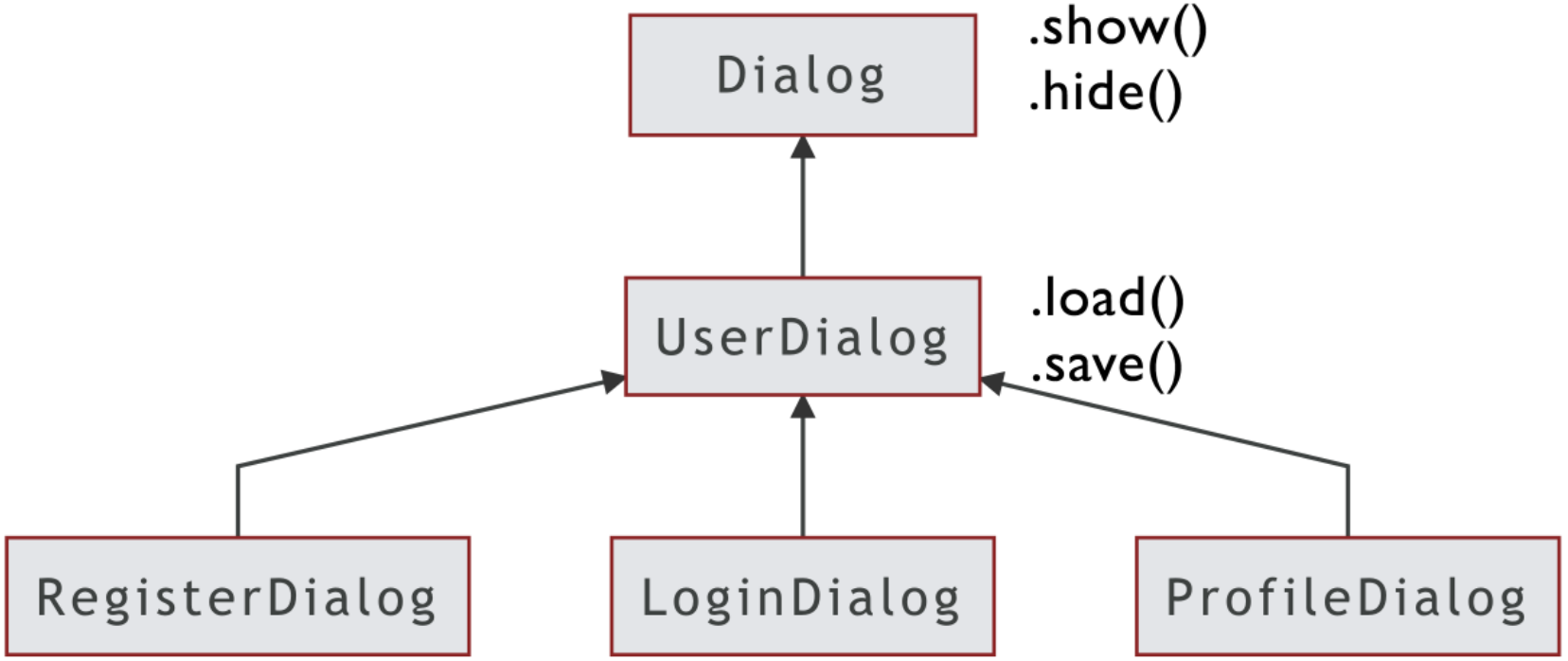
nearForm

Code Reuse in JavaScript

- Inheritance is useful in class-based languages
  - creating an inheritance hierarchy enables different behaviors
- Inheritance is not as useful in JavaScript
  - There are no classes
  - You can use “duck” typing
  - You can modify objects dynamically
  - You can use functional techniques to achieve code reuse
  - Deep inheritance hierarchies in JavaScript are a code smell

nearForm

An Example Structure



nearForm

Inheritance Patterns

- Pseudoclassical
  - `new Dialog` and `Dialog.prototype`
- Prototypical
  - `Object.create`
- Functional
  - private members
- Composition
  - mixing in functionality

nearForm

Pseudoclassical

```
function Class() { ... }
Class.prototype.method = function() { ... }
var instance = new Class()
```

- Traditional form often found in older material on the web
- An attempt to simulate classes
- Surprisingly unsafe - if you forget the `new` operator you pollute the global namespace
  - the default value of `this` is the global object
  - forgotten use of `new` can be mitigated

nearForm

Pseudoclassical

```
function Class() { ... }
Class.prototype.method = function() { ... }
var instance = new Class()
```

- Rather ironically requires you to use the `prototype` property to define methods
  - The imitation breaks down when you specify super classes
- Calling overridden “super class” methods is difficult
  - This pattern is memory efficient - may be suitable for large numbers of low complexity “data” objects

nearForm

Pseudoclassical Example

```
function Dialog(titleIn) {
  this.title = titleIn
}
Dialog.prototype.show = function() {
  log(this.title,'show')
}
var dialog = new Dialog('dialog')
dialog.show()
function UserDialog(titleIn) {
  this.title = titleIn
}
UserDialog.prototype = new Dialog()
UserDialog.prototype.load = function() {
  log(this.title,'load')
}
var userdialog = new UserDialog('userdialog')
```

nearForm

Prototypical

- Uses `Object.create` to generate new objects on demand from a sample object (an example of the "class")
  - a more natural fit with JavaScript
  - does require a different conceptual model
  - any object can be a "class"
  - does not provide private members
- The `this` object may not always be safe to use, especially within callbacks inside methods

nearForm

Prototypical Example

```
var dialog = Object.create({
  show: function() {
    log(this.title,'show')
  }
})
dialog.title = 'dialog'

dialog.show()

var userdialog = Object.create(dialog)
userdialog.title = 'userdialog'
userdialog.load = function() {
  log(this.title,'load')
}

userdialog.show()
userdialog.load()
```

nearForm

Functional

- Use a function to create a closure
  - any variables you declare inside the function are “private”
  - this is also the basis for the module pattern
- Return a custom object
  - means you can use the “class” with or without the `new` operator

nearForm

Functional

- Traditional constructors are easier to use
  - You can call the “super class” constructor
  - Create your custom object by create a `new` instance of the super class
- Less memory efficient
  - All functions are copied to each instance
  - You can’t put them in the prototype if you also want access to the closure
  - More suitable for larger business logic or single instance management objects like views or controllers.

Functional Example

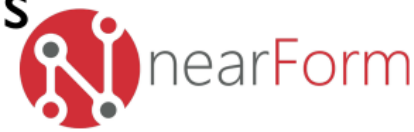
```
function Dialog(titleIn) {
  var self = {}
  var title = titleIn
  self.show = function() { log(title, 'show')}
  self.title = function() { return title }
  return self
}
var dialog = Dialog('dialog')
dialog.show()
function UserDialog(titleIn) {
  var self = Dialog(titleIn)
  self.load = function() {
    log(self.title(), 'load')
  }
  return self
}

var userdialog = UserDialog('userdialog')
userdialog.show()
userdialog.load()
```

nearForm

Composition

- JavaScript is dynamic
- Inject methods and properties into any object at any time
- Instead of a “super class”, just provide a set of methods and properties that deliver a given behaviour - this is known as a “mixin”
- Suitable for generic behaviors such as event handling
- Implementation is relatively easy - just set properties
- Using a library such as underscore makes it even easier, and handles override conflicts:
- `_.extend(A, B)` overrides A with B’s properties



Basic Composition Example

```
var _ = require('underscore')
var dialog = Object.create({
  show: function() {
    log(this.title,'show')
  }
})
dialog.title = 'dialog'

dialog.show()
var userdialog = _.extend(dialog, {
    title:'userdialog',
    load: function() {
      log(this.title,'load')
    }
})

userdialog.show()
userdialog.load()
```

nearForm

Constructor Composition

We can mix constructors by mixing their prototypes, and invoking them with the new constructors context.

```
var _ = require('underscore')

function Enablement() { this.enabled = false; }
Enablement.prototype.enable = function () { ... };
Enablement.prototype.disable = function () { ... };

function Visibility() { this.visible = true; }

Visibility.prototype.show = function () { ... };
Visibility.prototype.hide = function () { ... };

function Dialog() {
  Enablement.call(this);
  Visibility.call(this);
}

_.extend(Dialog.prototype, Enablement.prototype, Visibility.prototype)
```

nearForm

Mixin Objects

We can mix constructors by mixing their prototypes, and invoking them with the new constructors context.

```
var enablementMixin = {
  enable: function () { ... },
  disable: function () { ... }
}

var visibilityMixin = {
  show: function () { ... },
  hide: function () { ... }
}

var actionsMixin = _.extend(enablementMixin, visibilityMixin);

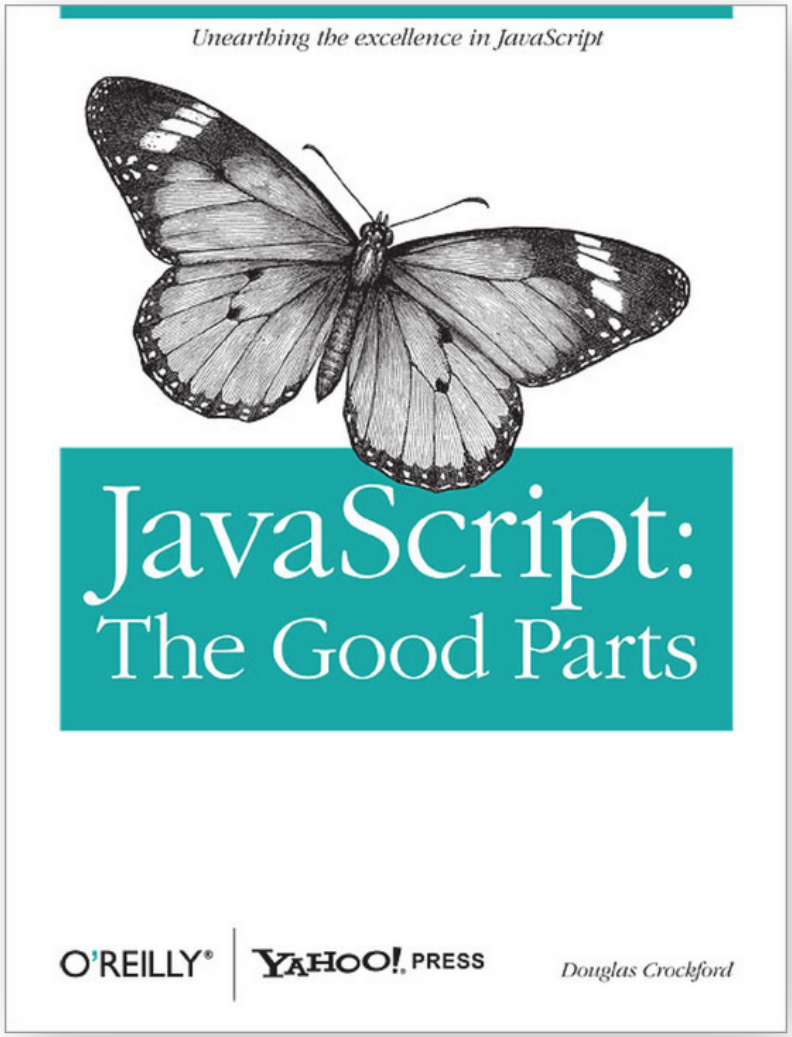
function Dialog() {
  this.enabled = true;
  this.visible = false;
}

_.extend(Dialog.prototype, actionsMixin)
```

nearForm

Where Next?

- Acquire and read this book:



nearForm