Node.js Training
JavaScript

Richard Rodger
@rjrodger
richardrodger.com
richard.rodger@nearform.com

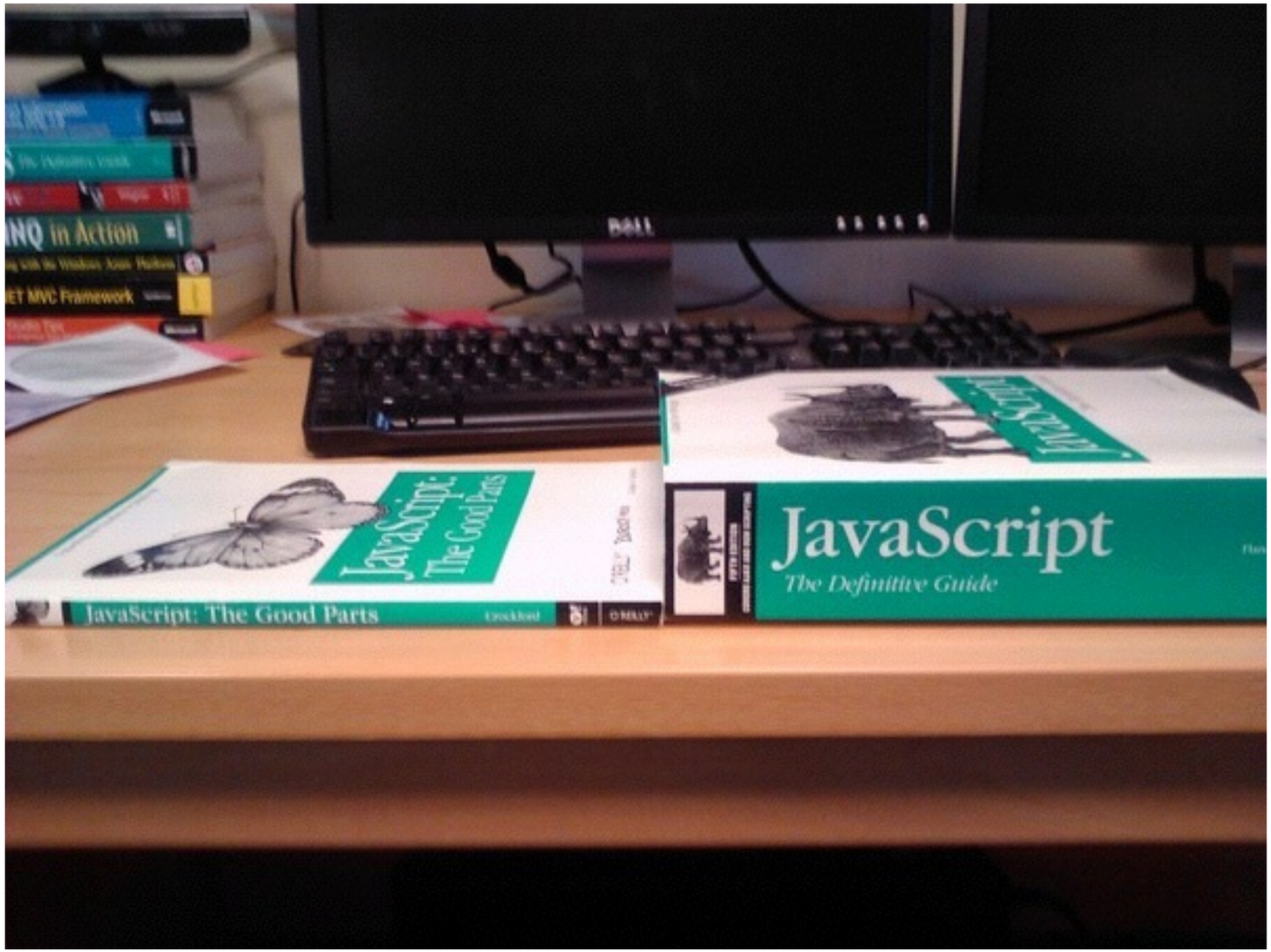
A New Look at JavaScript

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

Embracing JavaScript

- JavaScript looks like a toy language
  - It has a lot of really bad features:
    - the DOM, eval, with, obfuscated object creation
- It also has some really great features:
  - it's standardized under the name ECMAScript, latest version 5.1
  - it's functional, so you get callbacks and closures
  - it's prototype-based, so hash maps are baked-in
  - it has a literal data syntax (JSON)
  - it has low code volume, while remaining readable
- And here's the killer:
  - You're stuck with it as the universal language of the web

Embracing JavaScript



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
    - there are no arrays; they are just objects with numbers for properties, a special length property and some extra methods
- The best thing about objects is that they can be written as "literals", that is, in JSON format.

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 = { first name: 'Richard', last name: 'Rodger'
      }
      ```
- You can then access the values using the following notations:
  - **dot notation**, where the property name must be a valid identifier:
    - `objname.first_name`
  - **square bracket notation**, where the property name can be specified using a literal or a variable:
    - `objname['first_name']`
    - `var property_name = 'first_name'; objname[property_name];`


      ```

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:
  - `my_array[0] // is 'foo'`
- They are really just special objects
  - the array indexes are not numbers, but properties - the index number is converted into a string:
  - `my_array['0'] // is also 'foo'`
  - With a special length property, and some utility methods:
    - push, pop, shift, unshift, join, forEach, ...

JSON format

- Combine object and array literals and you get
  - JSON syntax - JavaScript Object Notation
  - See json.org for a formal specification
- The ECMA standard defines a utility object for JSON:
  - `JSON.parse(<literal string>)` 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

JavaScript Functions

- Fundamental unit of code
  - they are also objects, and can have properties
- Can be created ...
  - as declarations
  - as expressions
  - as methods
  - anonymously
- Can be called ...
  - as functions
  - as methods
  - as constructors
  - dynamically using the call and apply methods

Function Declarations

- Defines a function using the syntax:
  - `function name( ... ) { ... }`
- Defining a function creates a new variable scope - variables declared inside the function cannot be accessed outside the function
  - JavaScript does not have block scope, only function scope
  - All variables defined within the function are effectively "hoisted" to the start of the function, as if all the var statements were written first
    - You can use a variable inside a function before declaring it with var - not that this is a good idea
- Function definitions are themselves "hoisted" to the top of the current scope
  - So you can use the function before it is defined
- Useful for top level and utility functions
  - But you can also define functions inside other functions

Function Expressions

- Defines a function using the syntax:
  - `var name = function( ... ) { ... }`
- Unlike function declarations, there is no "hoisting"
  - You can't use the function before it is defined, because the variable referencing the function has no value yet
- Useful for dynamically created functions
- You can also use the syntax
  - `var name = function name( ... ) { ... }`
  - There is no difference, but your stack traces will now include the name of the function
- Once defined, function expressions are called in the same way as function declarations:
  - `name( argument1, argument2, ... )`

Methods

- The same pattern of function definition as function expressions
  - in this case, the functions are properties of an object:

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

  - You can redefine the functions of an object at any time, or add new ones:

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

- They must be defined before use

Anonymous Functions

- You can define a function without giving it a name:
  - `function( ... ) {}`
- You do this for convenience when you need to provide a function that will get called when an event happens later
  - DOM click handlers are a good example:
    - `element.onclick = function( ... ) { ... }`
- You also do this for "callbacks" - when a function needs another function as an argument
  - setTimeout handlers are a good example:
    - `setTimeout( function(){ ... }, 1000 )`

Function Invocation

- When a function is called ("invoked"), it has access to two special variables:
  - **this -** a reference to the global object
    - unless the function is a method, or apply is used
  - **arguments -** a list of the arguments given to the function
    - the arguments variable is not an array (unfortunately), but it does have a length property, so you can iterate through the arguments using a for loop, as if it were an array
      - `for( var i = 0; i < arguments.length; i++ )`
    - this allows you to define functions that have a variable number of arguments
- The function also has access to a "closure"
  - The closure is all variables in scope when the function is defined:

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

Method Invocation

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

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

- Watch out when using callbacks, the **this** variable may no longer refer to your object. In this case, explicitly reference your object using a new variable, self

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

Constructor Invocation

- The function call is preceded by the **new** operator
  - `function Foo() { ... }`
  - `var foo = new Foo()`
- 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...

Invocation using apply

- Functions inherit a number of utility methods from the built-in **Function** object
- The **apply** method is particularly interesting, and is used like so:
  - `myfunction.apply( anobject, argument_array )`
  - This code calls the **myfunction** function, setting the **this** variable equal to the object **anobject**, passing the **argument_array** as the arguments.
- You can dynamically call any function with any list of arguments, and you can make the function a method of any object you like

```
function print_foo(bar) {
  console.log( this.foo+bar)
}
var obj = {foo:1}
print_foo.apply(obj,[2]) // prints 3
```

- Useful trick - turn the **arguments** object in a real array with
  - `var args_array = [].slice.apply(arguments)`

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

Functional JavaScript

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

Callback Functions

- Pass a function into another function as an argument. The function you passed in is "called back" later by the other function.

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

  doStuff(data)
)
```

- The passed-in function is normally anonymous and defined in-place
- The most common API pattern for Node.js modules

Dynamic Functions

- Dynamically create functions when you need them

```
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)
}))
```

- Useful for wrapping other functions
  - handle common cases
    - such as error handling
  - extend functionality
    - add a throttle to database requests to reduce load

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)
    }
  }
}
```

- Often provides a more concise solution than iterating
- Beware stack overflow however!

The underscore library

- Provides a range of utility function for functional programming
- See underscorejs.org
- Examples:
  - `_.each( array, function( element ) { ... } )`
    - applies a function to each element of an array
  - `_.map( array, function( element ) { return ... } )`
    - updates each value of an array
  - `_.reduce( array, function(start,element) { return ...} )`
    - reduces an array down to one value - e.g summing

nearForm

JavaScript: Objects

- Object-oriented, but not Class-oriented
  - Everything is an Object
    - `var one = 1`
    - `one.toString() // == '1'`
- 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 it's prototype object
    - and so on, up the chain of prototypes

How Prototype Works

- You must create a Function object to access the prototype directly (otherwise it is hidden):
  - `var f = function(){}`
  - `f.prototype // == Object.prototype`
- Creating a prototype chain use the *new* operator:
  - `f.prototype = {red:1}`
  - `var o = new f()`
  - `o.red // == 1`

- JavaScript objects:
  - inherit properties from their prototypes
    - in this case the hasOwnProperty method returns false:
      - `o.hasOwnProperty('red') // == false`
  - but they can also redefine a property:
    - `o.red = 2`
    - `o.hasOwnProperty('red') // == true`
  - this overrides the prototype

The new Operator

- JavaScript attempts to hide it's 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 new operator is still required
  - It is the only way to gain access to the prototype chain
  - The ECMAScript 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 has a direct implementation, that hides the new operator:

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

Inheritance

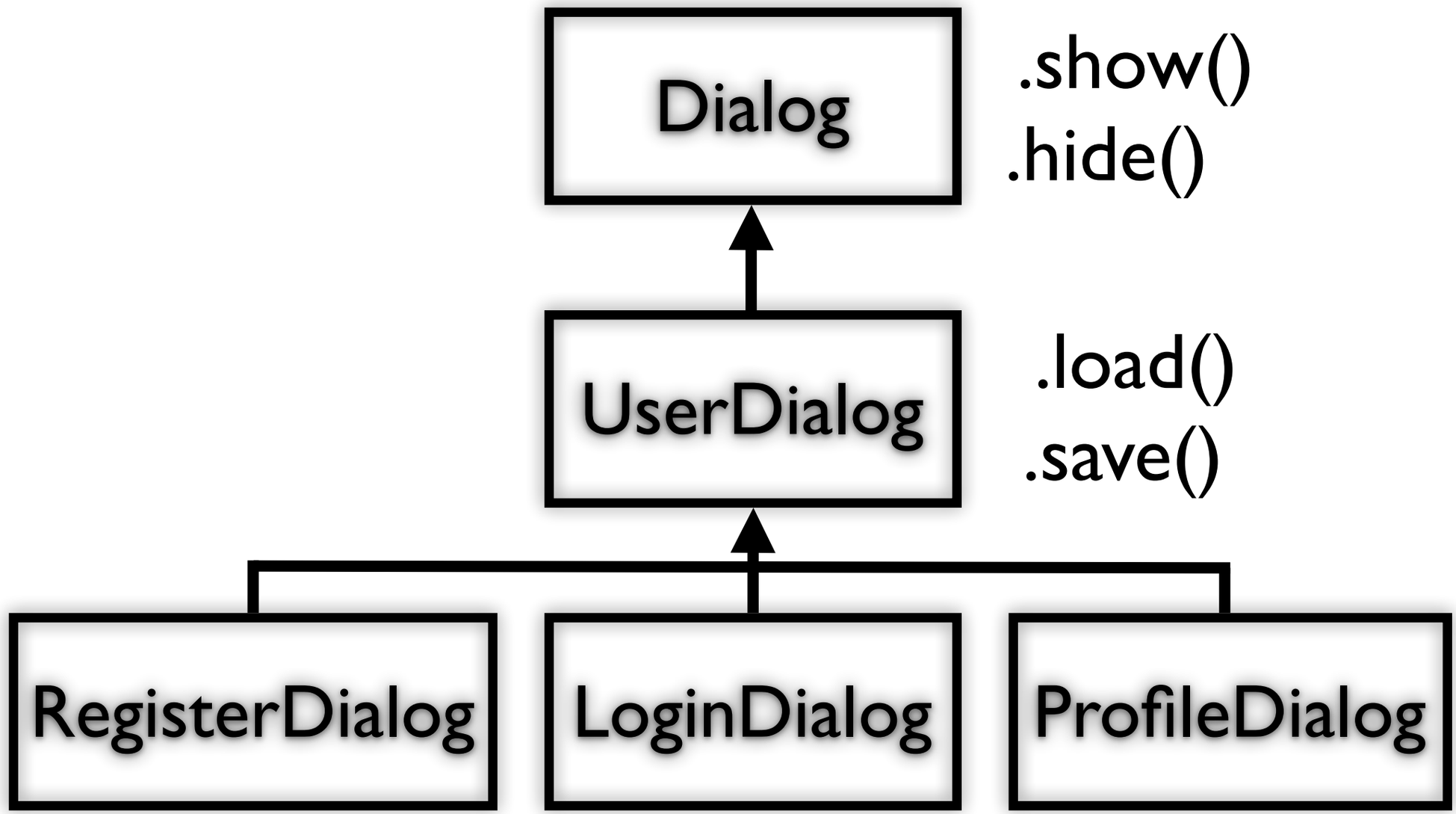
- Unlike class-based languages, inheritance is not "baked-in"
- You use prototypes to implement different styles of inheritance
  - There are trade-offs:
    - variable conflict safety
    - memory usage / performance
    - different conceptual models
- Different models are used by different libraries

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
  - You need to refactor

An Example Structure

- A set of dialog windows, with shared functions



Inheritance Patterns

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

Pseudoclassical

- Traditional form that you'll see in older material on the web
  - an attempt to simulate classes:
    - `function Class() { ... }`
    - `Class.prototype.method = function() { ... }`
    - `var instance = new Class()`
  - Surprisingly unsafe - if you forget the new operator you pollute the global namespace
    - the default value of *this* is the global object
  - 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

Pseudoclassical Example

```
function Dialog(title_in) {
  this.title = title_in
}
Dialog.prototype.show = function() {
  log(this.title,'show')
}

var dialog = new Dialog('dialog')
dialog.show()
```

```
function UserDialog(title_in) {
  this.title = title_in
}
UserDialog.prototype = new Dialog()

UserDialog.prototype.load = function() {
  log(this.title,'load')
}

var userdialog = new UserDialog('userdialog')
```

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 variable may not always be safe to use, especially within callbacks inside methods
  - leads to boilerplate code in every method:
    - `var self = this`

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()
```

Functional

- Use a function to create an closure
  - any variables you declare inside the function are “private”
- Return a custom object
  - means you can use the “class” with or without the new operator
- 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(title_in) {
  var self = {}

  var title = title_in

  self.show = function() {
    log(title,'show')
  }

  self.title = function() {
    return title
  }

  return self
}

var dialog = Dialog('dialog')
dialog.show()



function UserDialog(title_in) {
  var self = Dialog(title_in)

  self.load = function() {
    log(self.title(),'load')
  }

  return self
}

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

Composition

- JavaScript is dynamic
  - You can inject methods and properties into any object whenever you like
  - Instead of a "super class", just provide a set of methods and properties that deliver a given behavior - 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

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()
```

Tools

- Always use one of these:
  - jshint
  - jslint

Where Next?

- Buy this book: