

Modern JavaScript

Kevin Jones

kevin@rocksolidknowledge.com

@kevinrjones

<http://github.com/kevinrjones>



Functions

Objectives

- **Define and use JavaScript functions**
- **Understand context**
- **Understand parameters**



Overview

- **Fundamental to understanding JavaScript**
- **'First class' objects**



Functions as first class objects

- **Functions**
 - Can be assigned to variables
 - Passed to other functions
 - Returned from functions



Assigning functions

- One of the first things we'll do as a JavaScript programmer

```
function onStart(){  
}  
  
window.onload = onStart;  
  
// or  
  
window.onload = function(){  
};
```



Declaring functions

- **Declared using a function literal**
 - function keyword
 - optional name
 - comma separated parameter list
 - body



Examples of functions

```
function foo(){return true;}  
assert(typeof foo === "function", "defined");  
assert(foo.name === "foo", "named");
```

```
var bar = function(){return true;}  
assert(typeof bar === "function", "defined");  
assert(bar.name === "", "no name");
```

```
window.baz = function(){return true;}  
assert(typeof baz === "function", "defined");  
assert(baz.name === "", "no name");
```



Non global functions

- **Previous slide showed global functions**
 - Scoped to 'window'
- **Functions can be scoped**

```
function outer(){
  assert(typeof outer === "function", "function");
  assert(typeof inner === "function", "nested");
  function inner(){}
}
outer();
assert(typeof inner === "undefined", "nested");
```



JavaScript scoping

- **Scopes created by functions not blocks**
 - i.e. {} does not create a scope
 - functions are hoisted to the top of the declaring block

```
function outer(){
  assert(typeof outer === "function", "function");
  assert(typeof inner === "function", "nested");
  function inner(){}
}
outer();
assert(typeof inner === "undefined", "nested");
```



Calling functions

- **Four ways to call**
 - As a function
 - As a 'method'
 - As a constructor
 - Using `.call/.apply`



Function parameters

- **A list of arguments can be provided when calling a function**
 - these are assigned to the function parameters
 - numbers of arguments and parameters do not have to match
- **If fewer arguments than parameters**
 - extra parameters are set to undefined
- **If more arguments than parameters**
 - excess arguments are not assigned



Implicit argument parameter

- **'this' and 'arguments' are also available inside the function**
 - 'arguments' is collection of all arguments passed
 - has .length property
 - access using array syntax



'arguments' parameter

- **Not an array**
 - is 'array like'
- **Often see this**

```
function func() {  
    var args = Array.prototype.slice.call(arguments);  
}
```



Implicit 'this' parameter

- **Reference to the *invoker* of the function**
 - Also known as the *function context*
 - 'this' can vary depending on how the function is invoked



'function' invocation

- This is invocation as you would think of it

```
function createUser(){  
  createUser();  
  
var updateUser = function(){  
  updateUser();  
}
```



'method' invocation

- **Function added as a property on an object**
 - and called through that object
 - Inside the function 'this' is a reference to the calling object

```
var user = {};  
user.createUser = function(){}  
user.createUser();
```

```
function createUser(){ return this;}  
createUser(); // this === window
```

```
var user = {};  
user.createUser = createUser;  
user.createUser(); // this === user
```



Functions as constructors

- **Declared like other functions**
 - invoked differently

```
function User(){  
};  
  
var user = new User();
```



Constructor invocation

- **A new empty object is created**
- **New object is passed to the function as the 'this'**
- **New object is returned implicitly from the function**
 - don't return anything else!
- **Constructor functions start with uppercase first letter**
 - By convention



Using constructors

```
function User(){  
    this.create = function(){  
        return this;  
    };  
    this.update = function(){};  
};  
  
var user1 = new User();  
var user2 = new User();  
  
// user1.create() != user2.create()
```



Invoking functions with 'call' and 'apply'

- **Used to set caller's context (this) explicitly**
- **All functions have 'call' and 'apply' methods**
 - functions are just objects
 - created with the Function() constructor
- **'apply'**
 - two parameters, the context and array of args
- **'call'**
 - similar but args passed individually



Call and Apply

- **Useful for**
 - changing the context of the function
 - split up parameters to one function to pass to another



Using call and apply

```
function createUser(count){  
    this.count = count;  
}  
  
var user1 = {};  
var user2 = {};  
  
createUser.call(user1, 10);  
createUser.apply(user2, [20]);  
  
// user1.count == 10  
// user2.count == 20
```



Useful for callbacks

```
function forEach(collection, fn){  
    for(var n = 0; n < collection.length; n++){  
        fn.call(collection[n], collection[n], n);  
    }  
};  
  
var items = ['user', 'meeting', 'clock'];  
  
forEach(items, function(){  
    console.log(this.toString())  
});
```



Anonymous functions

- Previous slide is an example of an anonymous function

```
window.onload = function(){};

var user = {
  create: function(){}
}

setInterval(function() {}, 500);
```



Storing functions

- **Sometimes want to store related functions**
 - e.g. event management

```
var store = {  
  nextId: 1,  
  cache: {},  
  add: function(fn) {  
    if (!fn.id) {  
      fn.id = store.nextId++;  
      return !(store.cache[fn.id] = fn);  
    }  
  }  
};  
function create(){}  
store.add(create);  
store.add(create); // returns false - already stored
```



Memoizing

- **Functions that remember the result of a previous called**
 - Cache the result of the previous call
 - i.e. create a 'memo' of it
 - Makes calls more efficient



Self-memoizing

- **Functions are objects**
 - Can add properties to them
- **Add a hash to the function to cache previous results**
 - refer to this hash before executing function



Self memoizing functions

```
function fibonacci(value) {  
  if (!fibonacci.answers) fibonacci.answers = {};  
  if (fibonacci.answers[value] != null) {  
    return fibonacci.answers[value];  
  }  
  var val = 0;  
  var next = 0;  
  var nextnext = 1;  
  
  if(value == 1){  
    return 1;  
  }  
  
  for (var i = 1; i < value; i++) {  
    val = next + nextnext;  
    next = nextnext;  
    nextnext = val;  
  }  
  return fibonacci.answers[value] = val;  
}  
assert(fibonacci(6) == 8, fibonacci(6) + " == 8");  
assert(fibonacci.answers[6] == 8, "fibonacci[6] is cached");
```



Summary

- **Functions can be declared with or without a name**
- **Functions can be used as constructors**
- **Functions have an implicit this**
- **'this' can be set by 'calling' or 'applying' functions**
- **Functions can be stored**
- **Functions set up a scope**
- **Functions can be memoized**



Closures

Objectives

- Understand closures
- Understand how to use closures in JavaScript



Closures

- **Closure defines a scope**
 - Created when function is declared
 - Allows access to variables defined outside function
 - Variables can still be accessed when function is used
 - Even if their scope has disappeared



Example

```
var outerValue = 'outer';

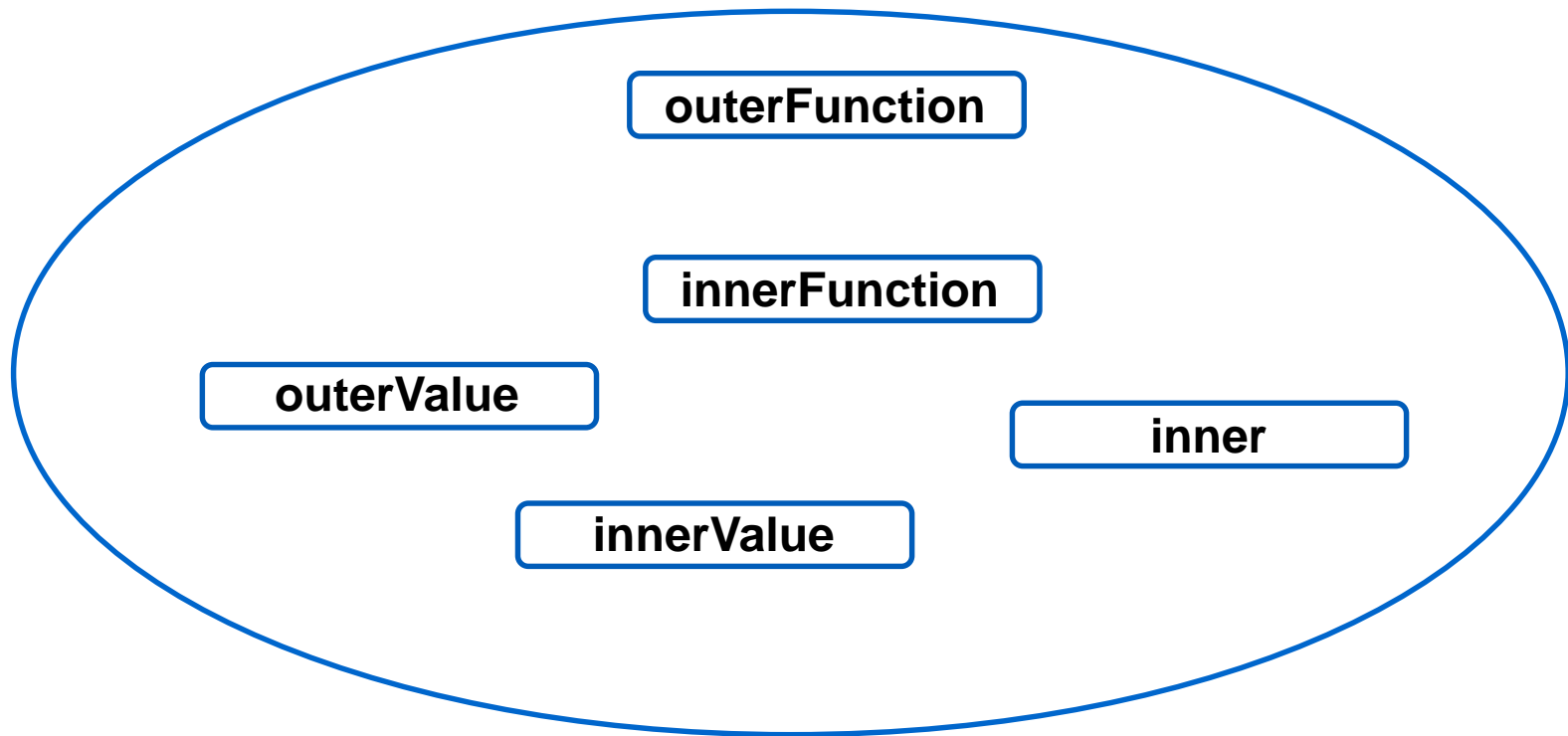
var inner;

test("closure tests", function () {
    function outerFunction() {
        var innerValue = 'inner';
        assert(outerValue == "outer", "ok");

        function innerFunction(){
            assert(innerValue == "inner", "ok");
        }
        inner = innerFunction;
    }
    outerFunction();
    inner(); // called but 'outerFunction' has
            // long gone away
});
```

How Closures work

- **innerFunction is a closure**
 - It captures the scope of where it was when it was declared
 - 'closes over' the scope



Using closures

- **Closures have many uses**
 - Private variables
 - Binding 'this'
 - Event handlers



Declaring private variables

- **Can define a constructor**
 - Variables defined within are 'private' to the constructor

```
function User(name) {  
    var _name = name;  
  
    this.getName = function () {  
        return _name;  
    };  
}  
  
var kevin = new User('kevin');  
  
assert("kevin" == kevin.getName(), "Name is kevin");
```



Callbacks

```
$(function () {  
    $.ajaxSetup({'accepts': 'text/JSON'});  
    var dataDiv$ = $('#data');  
    dataDiv$.html('Loading...');  
    $.ajax({  
        url: "http://localhost:49578/api/simple",  
        success: function (data) {  
            var html = $("<span>" + data.firstName +  
                "</span> <span>" +  
                    data.lastName + "</span>" )  
            dataDiv$.html(html);  
        }  
    });  
})
```



Event handlers can be problematic

- When called the button click method is bound to the element

```
<button id="test">Click Me!</button>
```

```
function Button() {  
    this.isClicked = false;  
    this.click = function () {  
        this.isClicked = true;  
        alert(button === this);  
    };  
}
```

```
var button = new Button();  
var elem = document.getElementById("test");  
elem.addEventListener("click", button.click, false);
```



One way to fix

```
function Button() {  
    var self = this;  
    self.isClicked = false;  
    self.click = function () {  
        self.isClicked = true;  
        alert(button === self);  
    };  
}
```



Binding contexts (this)

```
function bind(context, name) {  
    return function() {  
        return context[name].apply(context, arguments);  
    };  
}  
  
function Button() {  
    this.isClicked = false;  
    this.click = function () {  
        this.isClicked = true;  
    };  
}  
  
var button = new Button();  
$('#clickMe').click(bind(button, "click"));
```



Creating Partial Functions

- **Better known as currying**
 - Create a function with a predefined set of parameters
 - Apply other parameters to this function



Simple example

- **Function that wraps another and stores the arguments**
 - Returned function concatenates its args with stored args ...
 - ...and calls function

```
function curry(fn) {  
  // turn arguments into an array  
  var args = Array.prototype.slice.call(arguments, 1);  
  
  return function() {  
    return fn.apply(this, args.concat(  
      Array.prototype.slice.call(arguments)));  
  };  
}
```



Using curry

- **'curry' the split function so it splits on ','**
 - Create a csv function

```
test("curry tests", function () {  
  String.prototype.csv = curry(String.prototype.split, /\s*/);  
  var results = ("Harry, Sam, Alex").csv();  
  assert(results[0]=="Harry" &&  
    results[1]=="Sam" &&  
    results[2]=="Alex",  
    "The text values were split properly");  
});
```



Another way

- Apply 'curry' method to function prototype

```
Function.prototype.partial = function() {  
    var fn = this;  
    var args = Array.prototype.slice.call(arguments);  
    return function() {  
        return fn.apply(this, args.concat(  
            Array.prototype.slice.call(arguments)));  
    };  
}
```

```
test("more closure tests", function () {  
    String.prototype.csv  
        = String.prototype.split.partial(/,\s*/);  
});
```



Immediate functions

- **IIFE**
 - Immediately Invoked Function Expression
 - A way of creating closures
 - Define a function and immediately execute it
 - Use the return value

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



Scoping

- **IIFE used to enforce scope**
 - For example, use of \$ when using jQuery

```
(function($){  
    // do something with jQuery object here  
})(jQuery);
```



Module pattern

- Often used as part of the 'revealing module pattern'



Summary

- **Closures are used as a scoping tool in JavaScript**
 - Enclose over the variables it uses
 - Has many uses
 - Partial functions/currying
 - Data hiding
 - Organisation of code



"Object Oriented" JavaScript

Objectives

- **How to define objects in JavaScript**
- **Prototypes**
- **Inheritance**
- **Instantiation**
- **Sub-classing**



Prototypes

- **JavaScript supports prototypical inheritance**
 - Each JavaScript functions have a 'prototype' property
 - Shared across instances
 - Can use to share functions



Constructor functions

- **Can create objects many ways in JavaScript**
 - Can use the object syntax
 - `var k = {name: 'Kevin'};`
 - Can use the constructor syntax

```
function User(){  
};  
  
var user = new User();
```



Issues with instances

- **Creating instances creates multiple copies**
 - Each instance gets its own copy of functions

```
function User(){  
    this.name = function(){...}  
};  
  
var kevin = new User();  
kevin.name("Kevin");  
var terry = new User();  
terry.name("Terry");
```



Sharing with prototype

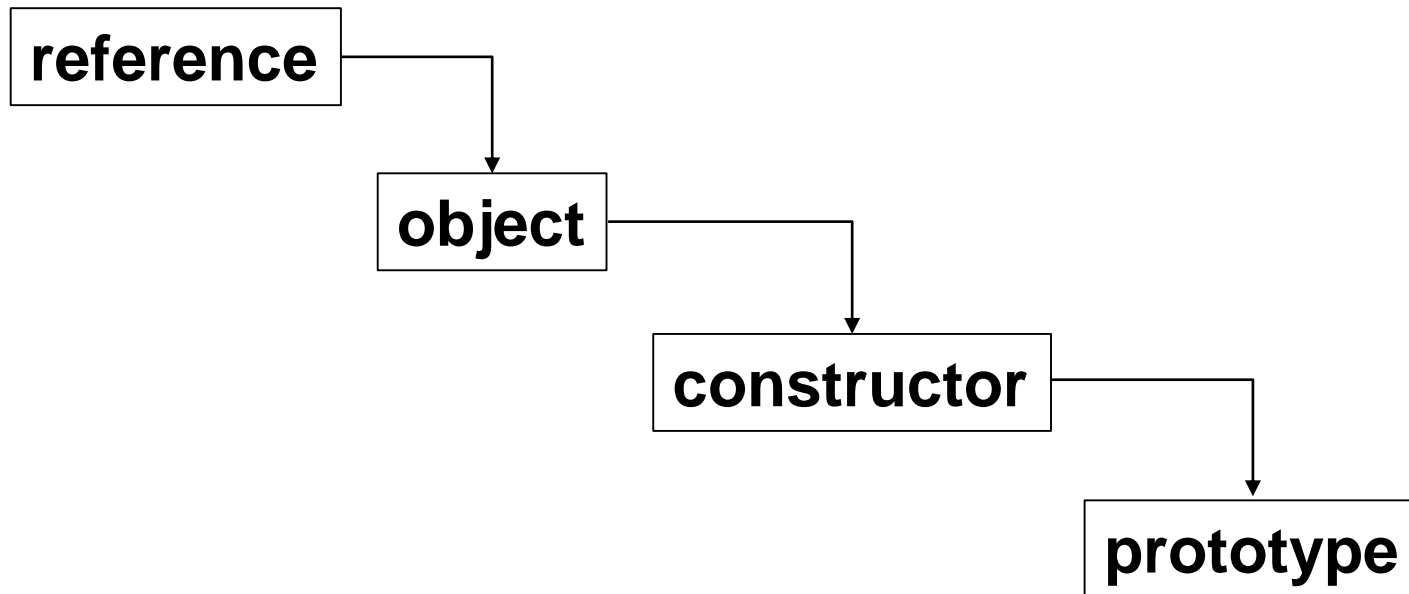
- **Prototype is shared across instances**

```
function User(){  
};  
  
User.prototype.name = function(){  
};  
  
var kevin = new User();  
kevin.name("Kevin");  
var terry = new User();  
terry.name("Terry");
```



Resolving the Prototype

- **Prototype properties are not copied into instance**
 - Sort of defeats the purpose
 - Each object has a constructor property
 - This references a prototype property



Chrome Debugger

```
Elements  Resources  Network  Sources  Timeline  Profiles  Audits  Console
> kevin
▼ User {name: function, fullName: function} ⓘ
  ▶ name: function (name){
    ▼ __proto__: User
      ▼ constructor: function User(){
        arguments: null
        caller: null
        length: 0
        name: "User"
      }
      ▼ prototype: User
        ▶ constructor: function User(){
          ▶ fullName: function (firstName, lastName){
            ▶ __proto__: Object
          }
          ▶ __proto__: function Empty() {}
          ▶ <function scope>
        }
        ▶ fullName: function (firstName, lastName){
          ▶ __proto__: Object
        }
      }
    }
  }
> |
```

```
Elements  Resources  Network  Sources  Timeline  Profiles  Audits  Console
> kevin.constructor
function User(){
  this.name = function(name){
    return name;
  }
}
> kevin.constructor.prototype
▶ User {fullName: function}
> kevin.constructor.prototype.fullName
function (firstName, lastName){
  return firstName + " " + lastName;
}
> |
```



Live updates

- **Prototype is 'live'**
 - Can attach to the prototype after object construction



Order of preference

- Object instance looked at before prototype

```
function User(){  
};  
  
User.prototype.name = function(){  
};  
  
var kevin = new User();  
kevin.name = function(){// use this one}
```



Prototypes for identity

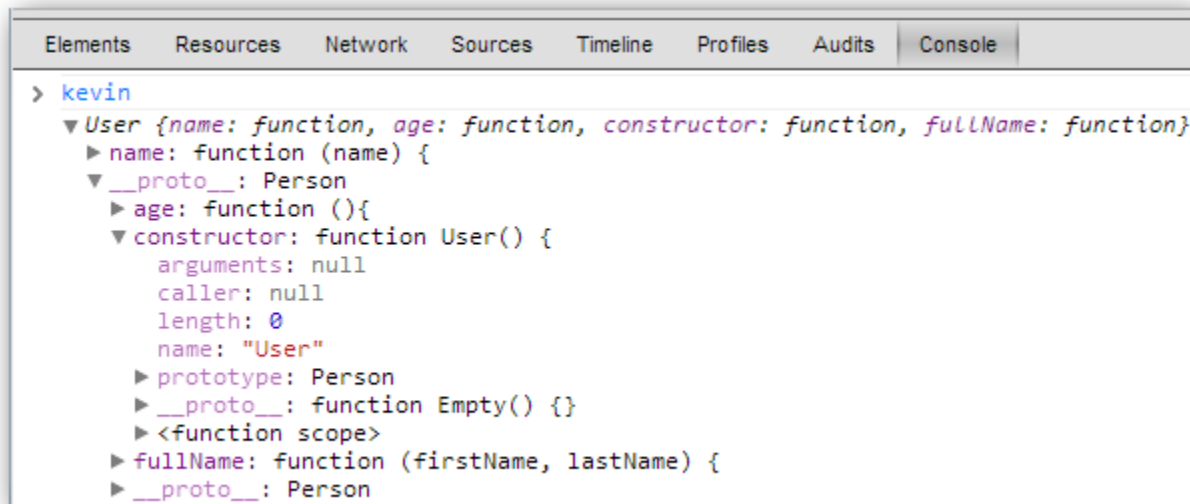
- Can use constructor to identify type

```
test("identity", function(){  
    var kevin = new User();  
  
    assert(kevin instanceof User);  
    assert(kevin.constructor === User);  
});
```



Inheritance

- **Prototypes also enable inheritance**
 - Via the prototype chain
 - Can use instance of one object as prototype of another



```
> kevin
▼ User {name: function, age: function, constructor: function, fullName: function}
  ► name: function (name) {
    ▼ __proto__: Person
    ► age: function () {
      ▼ constructor: function User() {
        arguments: null
        caller: null
        length: 0
        name: "User"
        ► prototype: Person
        ► __proto__: function Empty() {}
        ► <function scope>
        ► fullName: function (firstName, lastName) {
          ► __proto__: Person
```



Inheritance

- **Set 'derived' prototype as 'base class'**
 - Sometimes see `User.prototype = new Person()`
 - Now all derived instances share single base's properties

```
function Dummy () {}  
function Person() {  
}  
Person.prototype.age = function () {...}  
function User() {  
    this.name = function (name) {...}  
}  
Dummy.prototype = Person.prototype;  
User.prototype = new Dummy();  
// otherwise User's ctor == Person  
User.prototype.constructor = User;
```



Base class constructor

- **Call the base class through the constructor function**
 - remember constructors are functions

```
function Dummy () {}  
function Person(name) {  
    this.name = name;  
}  
  
function User(name, age) {  
    Person.call(this, name);  
    this.age = age;  
}
```



Calling super methods

- **Some libraries allow you to set up calling chains**
 - e.g. Crockford and John Resig
- **Considered not to be particularly worthwhile**



Summary

- **JavaScript supports prototypical 'inheritance'**
- **Add functions to the prototype**
- **Set the prototype as another class**
- **Can take this further, calling superclass etc.**
 - See Crockford, Resig and others



Modules

Objectives

- Understand the module pattern in JavaScript



Constructing objects

- Many ways to create JavaScript objects

```
var user = {  
    name: 'Kevin'  
};  
  
function User(){}  
  
var kevin = new User();
```



Constructing objects

- **Using 'new' can have issues**
 - Each instance has own copies of functions
- **Use prototype instead for shared functions**

```
function User(){  
};  
  
User.prototype.setName = function(){  
};  
  
var kevin = new User();  
kevin.setName("Kevin");  
var terry = new User();  
terry.setName("Terry");
```



Module Patterns

- **Modules are a common way of managing JavaScript code**
 - Allow for encapsulation
 - Good tool support (require, CJS, AMD)
 - Often created using Immediate Functions

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



Why the Module Pattern?

- **The major benefit is encapsulation**
 - Can pass needed dependencies to the module ...
 - ... scoped within the module
 - Variables are scoped within the module
- **Other benefit is what you return ...**



(Revealing) Module Pattern

- **Return what you need from the module**
 - Object
 - Constructor



Returning an object

- **Pass in jQuery reference**
 - scoped to module
- **Return an object that exposes functionality**

```
var authn = (function ($) {  
    var email = "";  
  
    var vm = {  
        email: email,  
        signIn: signIn  
    };  
  
    function initialize(params) {}  
    function signIn() {  
        $.post("")...  
    };  
  
    return vm;  
})(jQuery);
```



Returning a constructor

- Can now create instances

```
function blogPost () {  
    var shared;  
    function BlogPost(item) {  
        var title;  
        if (item != null) {  
            this = item.title;  
        }  
        this.getTitle = function () {  
            return title;  
        };  
    }  
    return BlogPost;  
};  
  
var ctor = blogPost();  
var post = new ctor({title: 'Title'});
```



Augmenting module

- Add methods to an existing module

```
var MODULE = (function (my) {  
    my.anotherMethod = function(){}  
    return my;  
})(MODULE);
```



Loose Augmenting module

- **Add methods to an existing module**
 - weird `MODULE || {}` checks if module exists in global namespace

```
var MODULE = (function (my) {  
    my.anotherMethod = function(){}  
    return my;  
})(MODULE || {});
```



Tight Augmenting module

- Add methods to an existing module

```
var MODULE = (function (my) {  
  var old_moduleMethod = my.moduleMethod;  
  
  my.moduleMethod = function () {  
    // method override, has access to old through  
    // old_moduleMethod...  
    return my;  
  })(MODULE || {});
```



Summary

- **Module pattern is powerful**
- **Used for encapsulation**
- **Can be used as an extension mechanism**



Require.js

Objectives

- Understand module loading in JavaScript
- Understand why asynchronous loading is necessary
- Understand how to use require
- Use require with jQuery



Why modules?

- **As we build 'apps' not 'sites' code complexity grows**
 - Need to manage the complexity
 - Fewer globals
 - Assembly of sites gets more complex
 - Would like to be able to optimize the code



Solution

- **Use some for of modules**
 - cf 'import', 'include' in other languages



CommonJS

- **Defined an API for loading modules**
- **Synchronous**
 - Great on the server
 - Not so great in the browser

```
var User = require("types/User");

function UserManager () {
}

//Error if require call is async
UserManager.prototype = new User();
```



Attempts to provide async loading

- **XHR**
 - uses eval
 - eval is evil
- **Web Workers**
 - Not all browsers (IE < 10)
- **document.write**
 - Need to know all the required scripts ahead of time
 - Does not work after pageload (perceived performance is bad)



Enter AMD

- **Asynchronous Module Definition**
 - Provides a mechanism to encapsulate modules
 - and a way to specify dependencies



AMD Example

- **define** – define the module
- **'jquery'** – specify dependencies
- **\$** - reference to the loaded dependency
- **function(\$)** – factory, executed after dependencies load

```
define(['jquery'] , function ($) {  
    return function () {};  
});
```



AMD - dependencies

- **Dependencies are string values**
 - 'utils/helper'
 - 'jquery'
- **Follows CommonJS practice**



AMD Modules

- **Modules wrapped in a 'define' call**
 - Allows AMD to resolve dependencies
 - Execute inner (factory) function after dependencies loaded
 - Does not litter global namespace



Naming modules

- **Can also name modules**
 - May have multiple modules per file
 - This is discouraged



AMD Loaders

- **AMD needs a loader to load the JavaScript**
 - Dojo (1.7+)
 - curl
 - lsjs
 - require



require.js

- **AMD loader**
 - Provides a standard way to load modules
 - Also provides an optimizer (uses node.js or Java)



Require Examples

- **Set up the page to use required**
 - Include `require.js`
 - Use data-main to reference `initial JavaScript`

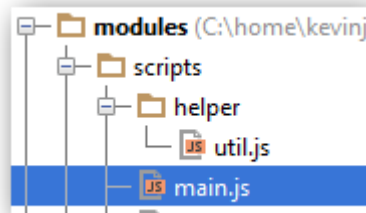
```
<html>
<head>
  <title>Test Suite</title>
  <script data-main="scripts/main"
          src="scripts/require.js"></script>
</head>
<body>
</body>
</html>
```



Initial JavaScript

- **This is the page JavaScript**
 - Can reference other Javascript to be used in this
 - The 'references' are relative to the load location

```
// main.js
require(["helper/util"], function(util) {
    util.doSomething();
});
```



```
// util.js
define(function() {
});
```

References

- **Code is loaded relative to base url**
 - set in data-main (**scripts/main**)
 - set via config (later)
 - if neither of these then url of loaded html is used



Loading Modules

- **RequireJs assumes that all dependencies are scripts**
 - No need to specify .js at end of module ids
- **Can override this behavior and use regular URL**
 - End module with .js
 - Start module with '/'
 - Start module with url protocol (e.g. 'http:' or 'https:')



Define a module

- **With no dependencies**

```
define(function() {  
    return {  
        name: 'Kevin',  
        id: 1  
    }  
});
```



Define a module

- **With dependencies**
 - function is not called until module is loaded

```
define(['resources'], function(res) {  
    return {  
        name: res.resource('name'),  
        id: 1  
    }  
});
```



Define a module

- **Modules can return 'anything'**

```
define(['...'], function(a) {  
    function BlogPost(item, parent) {  
    };  
  
    return BlogPost;  
});
```



Configuration

- **Sometimes need to configure requires**
 - Set base path
 - Set module names
 - Set loading time



Configuration

- **Specify configuration at startup**
 - Can be used for `versioning`
 - Useful for changing `relative paths`

```
<script src="scripts/require.js"></script>
<script>
    require.config({
        baseUrl: "/someUrl",
        paths: {
            "services": "app/services",
            "viewmodels": "app/viewmodels",
            "knockout": "knockout-2.2.1"
        },
        waitSeconds: 15
    });
</script>
```

Testing

- **Configure require**
 - Test code and 'real' code may not be on the same paths
 - Use requirejs configuration
- **Load the test using requires**
- **Don't start QUnit until tests are loaded**



Testing

- Using QUnit

- 'Require' the test
- Setup QUnit code

```
<script type="text/javascript"
      data-main="main" src="require.js">
</script>
<script>
  require.config({
    paths: {
      "services": "app/services",
      "views": "app/views"
    }
  });
</script>
```



Testing

- **Using Qunit**
 - 'Require' the test

```
define(['app/viewmodels/blogPost.js'], function (BlogPost) {  
    module("blogPost Tests", {  
        setup: function () {  
        }  
    });  
  
    test("initialize blogpost", function () {  
        var post = new BlogPost();  
        ok(post != null, "Have a BlogPost");  
    });  
});
```

Testing

- **Using Qunit**
 - 'Require' the test

```
//main.js

QUnit.config.autostart = false;

require(['./blogPostTests'], function () {
    QUnit.start(); //Tests loaded, run tests
});
```


Summary

- **Very useful to be able to modularize applications**
- **requirejs provides a way to load modules**
- **Can test**
 - different test libraries may require different setup



Testing Java Script

kevin@rocksolidknowledge.com

@kevinrjones

github.com/kevinrjones

Objectives

- **Why Unit Test?**
- **How do I Unit Test in JavaScript**
- **Several frameworks and tools**
 - QUnit
 - Jasmine
 - Sinon
 - Karma



Test Driven Development

- **For each new feature write a test first**
 - Define code requirements before writing code
 - Test based on use case
 - Acceptance test to test the big picture
 - Unit test to test the implementation
- **Paradigm shift**
 - From “what code must I write to solve this problem?”
 - To “how will I know when I’ve solved this problem”



Red, Green, Refactor

- **Write test to express how code is used and what it must do**
 - This test will fail
- **Write enough code to pass the test, no more**
- **Re-factor**
 - Clean code to remove redundancy & improve design
 - Re-run tests to make sure you didn't break anything
- **Repeat until done**
 - Write another test, push functionality forward



Benefits

- **Code without tests risks being defective**
- **TDD guarantees high degree of test coverage**
 - Test written as each feature added
- **Reduces uncertainty**
 - You can ***prove*** your code works successfully
- **Reduces cost of bugs**
 - Prevents bugs – less debugging later
 - Bugs prevented or found early save time later



What is a UnitTest?

- **Michael Feathers wrote “A set of Unit Testing Rules”**

A test is not a unit test if:

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as other tests
- You have to do special things in your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes



Tools

- **Many tools available**
 - qUnit
 - Buster
 - Testacular/Karma
 - TestSwarm
 - JSTestDriver
 - YUI Yeti
 - Jasmine
 - Sinon



QUnit

- **Originally written as the jQuery test library**
 - Now spun off as its own project
 - no dependencies on jQuery



Initial Setup

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet" href="resources/qunit.css">
</head>
<body>
<div id="qunit"></div>
<div id="qunit-fixture"></div>
<script src="resources/qunit.js"></script>
<script src="tests/tests.js"></script>
</body>
</html>
```

First Test

```
test( "hello test", function() {  
    ok( 1 == "1", "Passed!" );  
});
```



First result

QUnit Example

☐ Hide passed tests ☐ Check for Globals ☐ No try-catch

Mozilla/5.0 (Windows NT 6.1; WOW64; rv:19.0) Gecko/20100101 Firefox/19.0

Tests completed in 40 milliseconds.
1 assertions of 1 passed, 0 failed.

1. hello test (0, 1, 1) [Rerun](#)

JUnit Methods

- **ok(*truthy* [, *message*])**
 - Evaluate the first argument
- **equal(*actual*, *expected* [, *message*])**
 - Use `==` to compare actual and expected
- **deepEqual(*actual*, *expected* [, *message*])**
 - Use `===` to compare actual and expected



Grouping tests

- **Want to logically organize tests**
 - Can add a module to the tests

```
module( "group a" );

test( "first test", function() {
    ok( 1 == "1", "Passed!" );
});

test( "second test", function() {
    ok( 2 == "2", "Passed!" );
});
```

Common Code

- **Want to logically organize tests**
 - Can add a module to the tests

```
module( "group a" ,{
    setup: function(){
    }
});

test( "first test", function() {
    ok( 1 == "1", "Passed!" );
});

test( "second test", function() {
    ok( 2 == "2", "Passed!" );
});
```



Using test doubles

- **Replacements for part of your code**
 - Depended on Components (DOC) of SUT
 - Double provides same API as DOC

Why doubles?

- **Return values of components may not be repeatable**
 - Date/time values
- **Calls may be 'risky' or may be charged for**
 - Calling live web services during test
- **Parts of the application are 'slow'**
 - Database access
 - File access
- **Unit tests should not rely on external resources**
 - Databases
 - Web Services



Mocks, fakes, spies, stubs

- **Fake Object**
 - Provides same implementation as DOC but is much simpler
- **Test Stub**
 - Used to specify control options for SUT
 - e.g. different return values force SUT down different paths
- **Test Spy**
 - Like stub but also captures outputs of SUT
- **Mock Object**
 - Provides behaviour verification
 - e.g. correct methods called in correct order

JavaScript doubles

- **Can simply replace methods on an instance**
 - There's no need for a library
- **However a library can help**
 - Eases the repetitive tasks



Sinon

- **Sinon is a JavaScript mocking library**

Standalone test spies, stubs and mocks for JavaScript.
No dependencies, works with any unit testing framework.



With Sinon you can

- Write spies
- Write stubs
- Test Ajax
- Test XMLHttpRequest
- Test timers



Sinon Spies

- **Spies used to show that a callback executes**

```
var blogPosts = function() {  
  getPosts = function (callback) {  
    if(callback !== null){  
      callback();  
    }  
  }  
  return {  
    getPosts: getPosts  
  }  
}();
```

```
test("callback is called", function () {  
  var callback = sinon.spy();  
  blogPosts.getPosts(callback);  
  ok(callback.called, "Callback called")  
});
```

Testing Ajax

```
getPosts = function (callback) {  
    $.ajax({  
        url: "/blog/posts",  
        success: function (data) {  
            callback(data);  
        }  
    });  
}
```

```
test("callback is called", function () {  
    var callback = sinon.spy();  
    sinon.stub(jQuery, "ajax").yieldsTo("success", [1, 2, 3]);  
    blogPosts.getPosts(callback);  
    ok(callback.called, "Call the callback ");  
    ok(jQuery.ajax.calledWithMatch({url: "/blog/posts"}))  
    ok(callback.calledWith([1,2,3]))  
    //sinon.restore(jQuery.ajax);  
});
```

Replacing XHR

```
var xhr, requests;
module("Test with XHR", {
  setup: function () {
    xhr = sinon.useFakeXMLHttpRequest();
    requests = [];
    xhr.onCreate = function(req) { requests.push(req); };
  },
  teardown: function(){
    xhr.restore();
  }
})

test("xhr is called", function () {
  var callback = sinon.spy();
  blogPosts.getPosts(callback);
  equal(requests.length, 1);
  equal(requests[0].url, "/blog/posts");
});
```


Replacing XHR – Sending Response

```
var xhr, requests;
module("Test with XHR", {
  setup: function () {
    xhr = sinon.useFakeXMLHttpRequest();
    requests = [];
    xhr.onCreate = function(req) { requests.push(req); };
  },teardown: function(){
    xhr.restore();
  }
})

test("callback is called", function () {
  var callback = sinon.spy();
  blogPosts.getPosts(callback);
  requests[0].respond(200,
    {"Content-Type": "application/json"},
    JSON.stringify({foo: "bar"}));
  ok(callback.called, "Call the callback")
});
```

Jasmine

- **A BDD tool for JavaScript**
 - Behaviour Driven Development
 - 'describe' the 'expectations' of the code
 - defined in English



Setup Jasmine

- Include the correct libraries

```
<script type="text/javascript" src="spec/SpecHelper.js"></script>
<script type="text/javascript" src="spec/SlipsServiceSpec.js"></script>
<script type="text/javascript" src="spec/ControllersSpec.js"></script>

<script type="text/javascript">
  (function () {
    var jasmineEnv = jasmine.getEnv();
    ...
```

Test suite

- Describe the test

```
describe("RSK.Services.Entries", function () {  
  
});
```

Specifications

- **Specifies what the code should do**
 - Use the Jasmine 'it' function

```
describe("RSK.Services.Entries", function () {  
  it("is defined", function () {  
    expect(RSK.Services.Entries).toBeDefined();  
  });  
});
```



Expectations

- **If these are met the test has passed**
 - Jasmine 'expect' function

```
describe("RSK.Services.Entries", function () {  
  it("is defined", function () {  
    expect(RSK.Services.Entries).toBeDefined();  
  });  
  
  it("returns data", function () {  
    var slipsService = new RSK.Services.Entries.slipsService(http);  
    var data = slipsService.get(new Date());  
    expect(data.length).toBe(2);  
  });  
});
```

Setup and teardown

- **beforeEach and afterEach**

```
describe("RSK.Services.Entries", function () {
  beforeEach(function() {
    localize = {};
    scope = {};
    location = {};
    authenticate = {};
  });

  it("is defined", function () {
    expect(RSK.Services.Entries).toBeDefined();
  });

  it("returns data", function () {
    var slipsService = new RSK.Services.Entries.slipsService(http);
    var data = slipsService.get(new Date());
    expect(data.length).toBe(2);
  });
});
```

Nested describes

- **Allows better descriptions of the test output**

```
describe("RSK.Services.Entries", function () {  
  it("is defined", function () {  
    expect(RSK.Services.Entries).toBeDefined();  
  });  
  
  describe("slipsService", function () {  
    it("is defined", function () {  
      expect(RSK.Services.Entries.slipsService).toBeDefined();  
    });  
  });  
});
```


Jasmine spies

- **Jasmine offers its own 'spy' syntax**
 - spyOn
 - createSpy
 - andReturn
 - andCallThrough

Karma

- **Test runner**
 - Install in Node.js
 - Monitors file system and runs test continually
 - Works with any test framework
 - requires an adapter
 - Works with multiple browsers



Installing Karma

- **Karma is a Node application**
 - Install node first (<http://nodejs.org/download/>)
 - the use npm to install Karma as a global module

```
$ npm install -g karma
```



Karma configuration

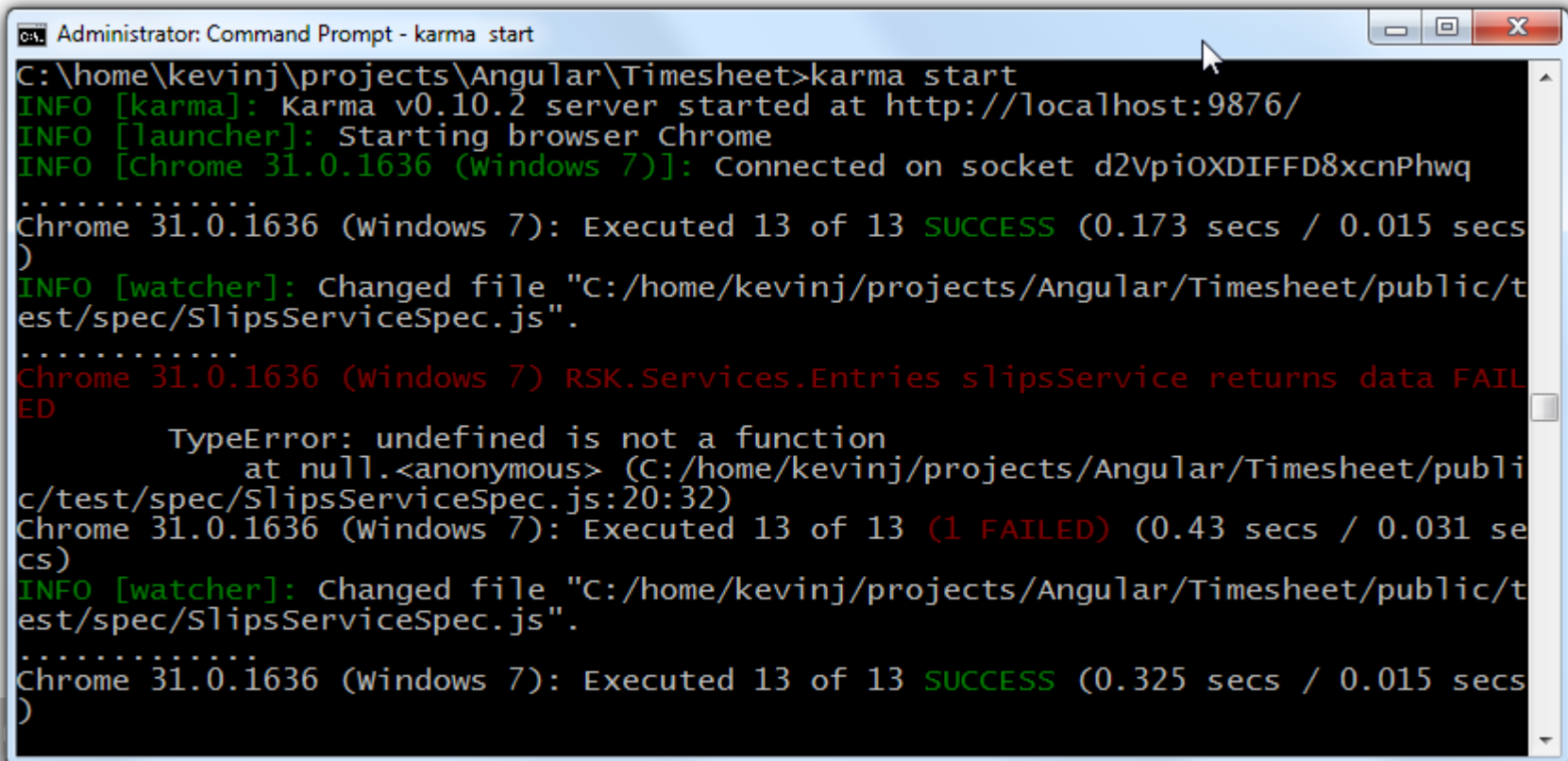
- **Create a configuration file to run Karma**
 - Which browsers
 - Which test framework
 - Which tests

```
$ karma init myconfig.js
```

Running Karma

- **karma start**
 - Looks for karma.conf.js by default

```
$karma start mykarmaconf.js
```



```
Administrator: Command Prompt - karma start
C:\home\kevinj\projects\Angular\Timesheet>karma start
INFO [karma]: Karma v0.10.2 server started at http://localhost:9876/
INFO [launcher]: Starting browser Chrome
INFO [Chrome 31.0.1636 (Windows 7)]: Connected on socket d2VpiOXDIFFD8xcnPhwq
Chrome 31.0.1636 (Windows 7): Executed 13 of 13 SUCCESS (0.173 secs / 0.015 secs)
INFO [watcher]: Changed file "C:/home/kevinj/projects/Angular/Timesheet/public/test/spec/SlipsServiceSpec.js".
Chrome 31.0.1636 (Windows 7) RSK.Services.Entries slipsService returns data FAILED
TypeError: undefined is not a function
    at null.<anonymous> (C:/home/kevinj/projects/Angular/Timesheet/public/test/spec/SlipsServiceSpec.js:20:32)
Chrome 31.0.1636 (Windows 7): Executed 13 of 13 (1 FAILED) (0.43 secs / 0.031 secs)
INFO [watcher]: Changed file "C:/home/kevinj/projects/Angular/Timesheet/public/test/spec/SlipsServiceSpec.js".
Chrome 31.0.1636 (Windows 7): Executed 13 of 13 SUCCESS (0.325 secs / 0.015 secs)
```

Summary

- **Unit testing is necessary to help write defect free code**
- **JUnit is an easy to use testing tool**
- **Can use Sinon to fake calls**
- **Can use Sinon to spy on calls**
- **Jasmine lets you use a BDD style syntax**
- **Karma to run tests continually**

