Advanced JavaScript

dev-mag.com

Table of Contents

Chapter 1. Closures, Scope & Context

1. Overview

Scope is a word used to describe the visibility of a variable. Programming languages implement this concept in independent ways, however most modern languages include local, global, and statically scoped variables. Each type of scope mentioned is resolved in unique ways, providing benefits to various aspects in a program. For example global variables are often used to expose environment specific data, where as local variables in contrast are used to aid sub-routines.

Many programming languages limit the scope of local variables within the “block” or “routine” in which it is defined, however this is not the case with JavaScript as any function may become what is called a “closure”. A closure is a sub-routine which encloses or “inherits” variables available to it's parent scope. This feature provides a vast amount of flexibility for the developer to produce highly dynamic programs.

2. The Var Keyword

Many languages allow us to define local variables, usually without a prefix of any kind. For example in Ruby we are not required to tell the interpreter that we are working with a local variable using a specific type of identifier, however a unique format is used to represent a global variable which must be prefixed by the *$* character. JavaScript implements a different policy, where all variables are global *unless* specifically declared using the `var` keyword.

JavaScript's `var` keyword declares a bound variable in the local scope, which is then available only to child closures within the current scope chain. In the example below we have used the `var` keyword to define the `message` variable to a string “hello”, which is then printed when we invoke **`sayHello()`**. The following **`print(message)`** call will throw the error ReferenceError: message is not defined since message is declared within the **`sayHello()`** function, and is not available within the global scope.

```
function sayHello(){
  var message = 'hello';
  print(message);
}
sayHello();
print(message);
```

If we were to omit the `var` keyword in the example above, the `message` variable then becomes a global variable which can potentially be overriden, or will override any previously defined variable of the same name.

```
function sayHello() {
  message = 'hello'
  print(message)
}
sayHello()
print(message)
```

The example below further illustrates the issue of omitting `var`, in which an entire sub-routine is overridden when **`splitWords()`** is called below. The call to **`each()`** below it fails and throws the error TypeError: each is not a function.

```
function each(array, callback){
  for (var i = 0; i < array.length; i++)
    callback(array[i])
}

function splitWords(string) {
  each = string.split(/\s+/)
  return each
}

words = splitWords('foo bar baz')
each(words, function(val){
  print(val)
```

```
})
```

As you may have noticed, even global variables and functions are properties of the global object. This global object is often aliased as the `window` variable when JavaScript is run within a windowed environment such as a browser. The examples below reference the same variable `foo` and function **`helloWorld()`**.

```
foo = 'bar'

foo
// => 'bar'

window.foo
// => 'bar'

this.foo
// => 'bar'

function helloWorld() {
  print('Hello')
}

helloWorld()
// => 'Hello'

this.helloWorld()
// => 'Hello'

window.helloWorld()
// => 'Hello'
```

As we have seen, having a thorough understanding of scope is crucial to developing stable applications.

3. Functions

JavaScript functions are first-class objects just like any other, containing both properties and methods of their own. This is unlike several languages which imply specific use of functions and methods, considering them a unique entity within the language's design. This means functions may be assigned to variables, passed as an argument, destroyed, or dynamically created during runtime, providing a great deal of flexibilty to the developer.

3.1. Function Literal

Below we create a function named `helloWorld()`, which when called prints the text “Hello World”. The function literal in this context is quite similar to those of many other languages.

```
function helloWorld() {
  print('Hello World')
}
```

To illustrate that a function is an object, we assign the variable `helloWorld2` to the function on the right-hand side, as we would with any other object.

```
helloWorld2 = function() {
  print('Hello World')
}
```

Our last example looks very strange, although it is a perfectly legal expression.

```
helloWorld3 = function helloWorld3() {
```

```
  print('Hello World')
}
```

So far the only difference is that both the first, and third examples have a name property, while the second example is what we call an *anonymous* function as we did not provide a name after the `function` keyword.

```
print(helloWord.name)  // => "helloWorld"
print(helloWord2.name) // => ""
print(helloWord3.name) // => "helloWorld3"
```

The following anonymous function idiom is commonly used to “lock” variables and functions into an anonymous scope, preventing pollution of the global namespace. As shown in the example below, we may also pass arguments to the anonymous function, just like we would any other. Here we pass the `jQuery` variable which is then assigned to a parameter named `$` so we can now use `$` locally to reference `jQuery`.

```
;(function($){
  // Never available to the public
  function bar() {
  }
  // Available to the public via myLibrary.foo()
  function foo() {
    bar()
  }
  // Available to the public since "var" is not used
  myLibrary = {
    foo: foo
  }
})(jQuery)
```

jQuery users will also notice that anonymous functions are used in conjunction with iteration, as shown below.


```
$('ul li').each(function(){
  // ... work with each element
})
```

3.2. Self Reference With Arguments.callee

Anonymous functions may also reference themselves, using the `arguments.callee` property. The function below is called immediately and assigns the `retry` variable to itself, which then proceeds to calling another anonymous function after 1000 milliseconds. This calls **`retry()`**, repeating the entire routine causing an infinite loop.

```
;(function(){
  var retry = arguments.callee
  setTimeout(function() {
    // Do something
    retry()
  }, 1000)
})()
```

Now although this is already very handy, there is a cleaner way to approach the previous example. The function below is no longer anonymous; we have named it “retry”, which allows us to reference it without using the `arguments.callee` property. As we are beginning to see, treating functions as objects can lead to some very concise, powerful code.

```
;(function retry(){
  setTimeout(function() {
    // Do something
    retry()
  }, 1000)
```

```
})()
```

3.3. Function Arity

A useful example of when we might use the “arity”, or length of a function, is in the case of iterators. In order to provide a user friendly API for our library, we will pass the element index only when several parameters are expected, allowing it to be elegantly ignored.

```
each(['foo', 'bar'], function(value){
  // Do something
})
each(['foo', 'bar'], function(i, value){
  // Do something
})
```

To implement this concept we first we loop through the array, and call the function, then we check the length of the callback function, and pass the index only when two or more parameters are present using the *ternary* operator.

```
function each(array, callback) {
  for (var i = 0, len = array.length; i < len; ++i)
    callback.length > 1 ?
      callback(i, array[i]) :
        callback(array[i])
}
```

Note

The ternary operator takes the form EXPR ? TRUE-EXPR : FALSE-EXPR and is the only operator with 3 operands in the language.

3.4. Function Bodies

Since a function's `toString()` method returns itself as a string, we can create a new `Function` method which will return only the body of the function. By assigning an anonymous function to `Function.prototype.contents`, all functions will now have a method named `contents` available.

```
Function.prototype.contents = function() {
  return this.toString().match(/^[^\{]*{((.*\n*)*)}/m)[1]
}
```

Now lets see what happens when we try it on the function below:

```
function User(name, email){
  return {
    name: name,
    email: email
  }
}
User.contents()
// "return {name:name, email:email};"
```

The whitespace and formatting returned by the function's `toString()` function varies from one JavaScript implementation to another, however the code itself is now available for us to parse, analyze, or store.

JSpec uses this functionality to provide helper functions to a block of code, without polluting the global namespace. Implemented below is a similar concept, in which we use the `with` statement to add the properties of `helpers` to the scope

within its braces. This allows us to utilize the helper functions without referencing `helpers.foo` etc.

```
;(function(){
  var helpers = {
    foo: function() { return 'foo' },
    bar: function() { return 'bar' }
  }

  it = function(description, callback) {
    with (helpers) {
      eval(callback.contents())
    }
  }
})()

it('should do something', function(){
  foo()
  bar()
})
```

Note

Use of the `with` statement is often consider ill-practice due to it being difficult to disambiguate between global variables and properties on the object passed to the statement. I would recommend usage to experienced JavaScript developers in order to prevent unintentional results.

3.5. Function Parameters

The parameter names of a function can also be returned as an array using the `Function`'s `toString()` method, and the `params()` function shown below, which simply matches the first set of parenthesis, and returns an array using `/\w+/g` to match words.

```
```

```
Function.prototype.params = function(){
  return this.toString().match(/\((.*?)\)/)[1].match(/\w+/
g) || []
}
(function(foo, bar){}).params()
// ['foo', 'bar']
```

Now suppose we want to alter our **each()** iterator function to pass the index, and value based on the order callback parameters. We can do this by first checking the name of the first value returned by the **params()** method, and seeing if it is "value". Then when we loop through, we can simply alter the callback invocation based on the order.

```
function each(array, callback) {
  var valueFirst = callback.params()[0] == 'value'
  for (var i = 0, len = array.length; i < len; ++i)
    valueFirst ?
      callback(array[i], i) :
        callback(i, array[i])
}
```

Now when we iterate, our index and value parameters will have the expected contents, regardless of the order how cool is that!

```
each(['foo', 'bar'], function(value, i){
  print(i, value)
})
// 0 "foo"
// 1 "bar"

each(['foo', 'bar'], function(i, value){
  print(i, value)
})
// 0 "foo"
// 1 "bar"
```


4. Methods & Receivers

A JavaScript “method” is simply a function with a receiver. A function without a receiver takes the following form `name(arg, ...)`, where as a function invoked with a receiver looks like this `object.name(arg, ...)`. JavaScript's prototype model allows us to elegantly share methods between object prototypes.

4.1. Method Portability & Gotchas

The following examples demonstrate method portability, as well as potential gotchas when using a prototype based language. We will start by looking at a real-world example. Below we call **`push()`** with `array` as the receiver, which then becomes the value of `this` while the method is executed. The method then finishes execution and we can see that we have successfully appended several strings to our array.

```
array = []
array.push('a', 'b', 'c')
array[0] // => 'a'
array[1] // => 'b'
```

Our second example illustrates that these methods are simply functions like any other. When we call `Array.prototype.push()` directly it is now executed in context to `Array.prototype`, pushing the values directly to the prototype object, which is not very productive, however it is valid JavaScript.

```
Array.prototype.push('a', 'b', 'c')
Array.prototype[0] // => 'a'
Array.prototype[1] // => 'b'
```

```
```

Our final example shows how a regular object can become array-like by simply inheriting one or more methods from `Array.prototype`. Since functions hold no reference to their original owner, we can simply assign it to `letters`` singleton prototype, which when invoked simply assigns each letter to the corresponding index. jQuery and many other libraries utilize this trick to provide library-specific "collections" without extending JavaScript's core **`Array`** prototype.

```
letters = {}
letters.push = Array.prototype.push
letters.push('a', 'b', 'c')
letters[0] // => 'a'
letters[1] // => 'b'
```

Tip

When using the subscript notation arbitrary strings or objects may be used as property names. Example: `this['anything you want'] = function(){}`

4.2. Altering Execution Context

`Function`'s prototype contains two methods for altering the context in which a given function is invoked. Both **`call()`** and the **`apply()`** methods accept an object as the first argument, which becomes the value of `this` while the function is executed. These methods differ only in how they accept the remaining arguments.

Before we begin our examples lets create some code to work with. Below we have a person object, and a function named **`example()`**, which simply prints the value of `this`, as well as each argument passed.

```
```

```
var person = {
  name: 'TJ',
  age: 22,
  toString: function(){ return this.name }
}

function example() {
  print(this)
  for (var i = 0, len = arguments.length; i < len; ++i)
    print(arguments[i])
}
```

With our example code above, if we call `example()` directly we can see that the value of `this` as previously mentioned is simply the global object. We then alter the execution context by using the `call()` method, passing our `person` object and an arbitrary list of arguments. Finally we use the `apply()` method to once again change the context, however this method accepts an array of arguments, which are "applied" to the function.

```
example('foo', 'bar')
example.call(person, 'foo', 'bar')
example.apply(person, ['foo', 'bar'])
```

Which would output the following:

```
[object global]
foo
bar

TJ
foo
bar

TJ
foo
bar
```


Tip

Ruby developers may consider `apply()` analogous to Ruby's splat operator `*`.

Chapter 2. Prototypal Inheritance

1. Overview

In contrast with relatively static languages such as C and PHP, JavaScript employes a highly dynamic inheritance model known as “prototypal inheritance”. This flexible model allows methods and properties to be dynamically added, removed, or inspected during runtime. We will be taking a look at the same concept of an “Animal” class implemented in both PHP and JavaScript.

2. Class Based

In class-based languages, classes and objects are usually very distinct entities; objects are instance of a class, and classes contain methods. When we create an instance of our PHP `Animal` class below and call the `__toString()` method, our instance must first lookup the method within the `Animal` class, then execute it in context to the current instance.

```php
<?php

class Animal {
 public $name;

 public function Animal($name) {
  $this->name = $name;
 }

 public function __toString() {
  return "Im $this->name\n";
 }
}

echo new Animal('Niko');
// => 'Im Niko'
```

To inherit functionality we would simply subclass our initial class with **`HugeAnimal`** as shown below.

```
class HugeAnimal extends Animal {
 public function __toString() {
  return "Im $this->name, a ******* huge animal\n";
 }
}

echo new HugeAnimal('Simon');
// => 'Im Simon, a ******* huge animal'
```

3. Prototype Based

In JavaScript all objects are instances, and our methods live in a property aptly named prototype. When a function call is preceded by the `new` keyword a new empty object is created, and **`Animal`** is executed in context to the new object (`this` becomes the object). When the **`Animal`** function invoked in this manor we call it a “constructor”.

```
function Animal(name) {
  this.name = name
}

Animal.prototype.toString = function() {
  return 'Im ' + this.name
}

print(new Animal('Niko'))
// => 'Im Niko'
```


3.1. Implementing Inheritance

To inherit from an object in JavaScript requires several steps. First we create a new constructor, in which we apply all of our `arguments` to the parent constructor **`Animal`**. Next we create a clone of **`Animal`**'s prototype by creating a new instance, which is added to the prototype chain. We proceed to restoring the constructor to HugeAnimal, and then implement our new **`toString()`** method.

```
function HugeAnimal(name) {
  Animal.apply(this, arguments)
}

HugeAnimal.prototype = new Animal
HugeAnimal.prototype.constructor = HugeAnimal
HugeAnimal.prototype.toString = function() {
 return 'im ' + this.name + ' a ******* huge animal'
}

print(new HugeAnimal('Simon'))
// => 'Im Simon, a ******* huge animal'
```

Note

We clone rather than assiging the two prototypes as shown here `HugeAnimal.prototype = Animal.prototype` to prevent pollution of the parent prototype's members.

3.2. Asserting Object Relationships

To assert the type of object based on it's prototype chain, we can use the **`instanceof`** operator, which traverses the prototype chain looking for a matching constructor. The final example of

`simon.__proto__.__proto__.constructor.name` illustrates how the operator traverses.

```
niko = new Animal('Niko')

print(niko.constructor.name)
// => Animal
print(niko instanceof Animal)
// => true
print(niko instanceof HugeAnimal)
// => false

simon = new HugeAnimal('Simon')

print(simon.constructor.name)
// => HugeAnimal
print(simon instanceof Animal)
// => true
print(simon instanceof HugeAnimal)
// => true
print(simon.__proto__.__proto__.constructor.name)
// => Animal
```

Warning

The __proto__ property may not be available in all JavaScript engines

3.3. Constructor Gotchas

Any function may be a constructor, although unusual side-effects may occur when attempting to construct an object from an arbitrary function as shown below. Unlike classes in many languages, JavaScript does not require constructor names to be capitalized, however this convention is used to disambiguate between regular functions and those designed to construct objects.

```
new print // => [object Object]

Animal('Niko')
name // => "Niko"

Array.prototype.push('a', 'b', 'c')
Array.prototype[0] // => 'a'
Array.prototype[1] // => 'b'

push = Array.prototype.push
push('a', 'b', 'c')
this[0] // => 'a'
this[1] // => 'b'
```

4. Creating JavaScript Classes

Many full-fledged JavaScript libraries such as Prototype.js and MooTools provide inheritance implementations, making the process both easier on the eyes, and the brain. Smaller stand-alone implementations can also be found, which can be paired up with libraries such as jQuery which do not currently support prototypal inheritance utilities.

4.1. Implementation

In this section we will be looking at how class-like functionality can be created using JavaScript. Below is the source code for a light-weight `Class` implementation based on the Base2 and Prototype libraries. This project is currently available at `http://github.com/visionmedia/js-oo`.

```
;(function(){
  var global = this, initialize = true
  var referencesSuper = /xyz/.test(function(){ xyz }) ? /
\b__super__\b/ : /.*/

  Class = function(props){
    if (this == global)
      return Class.extend(props)
```

```
    }

    Class.extend = function(props) {
      var __super__ = this.prototype

      initialize = false
      var prototype = new this
      initialize = true

      for (var name in props)
        prototype[name] =
          typeof props[name] == 'function' &&
          typeof __super__[name] == 'function' &&
          referencesSuper.test(props[name]) ?
            (function(name, fn){
              return function() {
                this.__super__ = __super__[name]
                return fn.apply(this, arguments)
              }
            })(name, props[name])
          : props[name]

      function Class() {
        if (initialize && this.init)
          this.init.apply(this, arguments)
      }

      Class.prototype = prototype
      Class.constructor = Class
      Class.extend = arguments.callee

      return Class
    }
  })()
```

Initially we set up a localized scope like our previous chapters, pulling in the global object as `global` for faster reference. Next we use the following regular expression to check if function bodies may be converted to a string via **`toString()`**. This will later be used to "spy" on a method to see if it references the **`__super__()`** method, which we may optimize for.

```
...
var referencesSuper = /xyz/.test(function(){ xyz }) ? /
\b__super__\b/ : /.*/
...
```

Next we create the `Class` constructor, which will act as a shortcut for `Class.extend(props)` which will construct our class instance. Since we use `new Class` internally we first check if the value of `this` is the `global` object, meaning we have called the function via `Class()`. Otherwise we do nothing, allowing the `new` keyword to function properly.

```
...
Class = function(props){
  if (this == global)
    return Class.extend(props)
}
...
```

By design we will be using `Class.extend()` to both create new classes, and extend others. With this in mind we first store a reference to the current prototype object in `__super__` which will be used later to reference superclass methods.

We now need a clone of our “super” class's prototype so that we may use the `instanceof` operator. By wrapping `var prototype = new this` with the boolean `initialize` assignments allows us to ignore the `init()` method when cloning the prototype of a subclass. Failure to implement this mechanism would mean that `init()` would be called several times when creating an instance, yielding poor performance.

```
...
```

```
  Class.extend = function(props) {
    var __super__ = this.prototype

    initialize = false
    var prototype = new this
    initialize = true
    ...
```

Now we must iterate our `props` object which contains methods and properties for the subclass. If a property is a function and our superclass contains a method of the same name, then we continue to using our `referencesSuper` regular expression to see if the text "__super__" is found. When found we create an anonymous function, passing in the method `name` as well as the function `props[name]`. If we were to reference these variables without this anonymous scope, we would always reference the *last* method in the `props` object due to their assignment during iteration.

In order to simiulate calling of superclass methods, we return an anonymous function which dynamically assigns the **`__super__()`** method before calling the original `fn` method.

```
...
for (var name in props)
  prototype[name] =
    typeof props[name] == 'function' &&
    typeof __super__[name] == 'function' &&
    referencesSuper.test(props[name]) ?
      (function(name, fn){
        return function() {
          this.__super__ = __super__[name]
          return fn.apply(this, arguments)
        }
      })(name, props[name])
    : props[name]
...
```


Following our previous for loop we have a locally scoped function named **`Class`** acting as our constructor. The previously mentioned `initialize` variable is true only when an instance is created, which then calls the **`init()`** method when present.

Finally we assign **`Class`**'s prototype property to our custom `prototype` object which supports **`__super__()`**. We then assign the new classes's **`extend()`** method to `arguments.callee` which references the current **`Class.extend()`** method being called.

```
...
  function Class() {
    if (initialize && this.init)
      this.init.apply(this, arguments)
  }

  Class.prototype = prototype
  Class.extend = arguments.callee

  return Class
}
```

There you have it, a robust, light-weight class implementation for JavaScript. In the next section we will be taking a look at how to use this library.

4.2. Usage Examples

Create a class, with no properties or methods:

```
User = Class()
// or Class.extend()
```

Create a class containing both properties and methods:

```
User = Class({
  type: 'user',
  init: function(name) {
    this.name = name
  }
})
```

Create an instance of the **User** class shown above:

```
tj = new User('tj')
tj.type
// => "user"
tj.name
// => "tj"
```

Subclass the **User** class we created earlier:

```
Admin = User.extend({
  type: 'admin'
})

tj = new Admin('tj')
tj.type
// => "admin"
tj.name
// => "tj"
```

We may also reference superclass methods using **__super__**:

```
Admin = User.extend({
  init: function(name, age) {
    this.__super__(name)
```

```
    this.age = age
  },

  toString: function() {
    return '<Admin name="' + this.name + '" age='
 + this.age + '>'
  }
})

tj = new Admin('tj', 22)
tj.toString()
// => "<Admin name=\"tj\" age=22>"
```

Chapter 3. Advanced Meta-programming Techniques

1. Overview

In this chapter we will be talking about meta-programming techniques used by JavaScript professionals in order to create clean, concise, DRY libraries. In this chapter, we will be creating a DSL similar to that of Ruby's Sinatra, for handling RESTful routes. Our goal is support the `GET`, `POST`, `PUT`, and `DELETE` methods, which will look like the example below:

```
get('/user/:id',function(id){
  return 'Viewing user' + id
})

del('/user/:id',function(){
  return 'User deleted'
})
```

2. Getting Started

First we need to create an anonymous scope that we can build our library in. Within the anonymous function we create a routes array using the `var` keyword, which will prevent any code outside of the anonymous function from accessing it.

```
;(function(){
  var routes = []
})()
```


Now that we have an array to store our routes, we can begin by creating the `get()` function.

3. Defining Route Functions

Our `get()` function accepts a path, and a callback function; which are then pushed onto the routes array so we may loop through them later. We also include a method property to the object we are pushing, allowing us to match routes based on their HTTP verb.

The first thing we will notice, is that we now have a variable named `main` assigned to `this`. We will be using this later to generate our functions dynamically, however it simply references the global object (window). The other irregularity we may see is that when we are generating the `del()` function we pass the object `{ as: 'delete' }`. This is because `delete` is a reserved word in JavaScript, so we must use a different name; `del` in this case.

Note

We do not use the `function get()` syntax, as this works much like the `var` keyword, however `get = function` defines a global function, so it may be accessed outside of our anonymous function.

```
;(function(){
  var routes = []

  get = function(path, callback) {
    routes.push({
      method: 'get',
      path: path,
      callback: callback
    })
  }
})()
```

3.1. Route Function Generators

Although we could simply copy and paste the `get()` function above to start creating our `del()`, `put()` and `post()` functions, we will take a meta-programming approach to solve the problem. We will create a locally scoped function named `def()` which will create our route functions for us. This is called a generator function.

In our `def()` function below, we have the expression `main[method]` which is equivalent to `window.get`, `window.put`, etc, however we are assigning a function dynamically using the `[]=` operator.

Since we allow the option `{ as: 'delete' }` we assign the method as `options.as` or `method` by default. This feature would allow us to create alternative functions such as `def('update', { as: 'put' })` or `def('destroy', { as: 'delete' })`.

```
;(function(){
  var main = this, routes = []

  function def(method, options) {
    options = options || {}
    main[method] = function(path, callback){
      routes.push({
        method: options.as || method,
        path: path,
        callback: callback
      })
    }
  }

  def('get')
  def('post')
  def('put')
  def('del', { as: 'delete' })
})()
```

To illustrate this further, the following functions are essentially the same. The one exception is that the bottom two do not have the name property since they are anonymous.

```
function foo(){}
foo = function foo(){}
foo = function(){}
this['foo'] = function(){}
```

3.2. Calling Routes

For testing purposes we will need an API to call our routes, for this we will extend our **`def()`** generator function to allow calling each method function without a callback like below:

```
get('/user/1')
// "Viewing user 1"
```

Below is the extended version of **`def()`** which now uses the `typeof` operator to check if callback is a function, in which case the route is added like usual, otherwise we invoke the **`call()`** function with the given path and HTTP method.

Since we will allow placeholders in our paths, we must use a regular expression to match routes. We will pass our `path` variable to the **`prep()`** function, which will generate a regular expression for that path. This is done at the time of route creation because it will be more efficient than creating them while looping through all the routes.

```
function def(method, options) {
  options = options || {}
  main[method] = function(path, callback){
    if (typeof callback == 'function')
      routes.push({
        method: options.as || method,
        path: path,
        pattern: prep(path),
        callback: callback
      })
    else
      return call(path, options.as || method)
  }
}
```

4. Dynamic Route Regular Expressions

Our `prep()` function simply accepts a path such as `/user/:id/:operation` and returns a regular expression similar to `/^\/user\/(\w+)\/(\w+)$/`. The `RegExp` constructor can be used to create regular expression dynamically as shown below.

```
function prep(path) {
  return new RegExp('^' + path.replace(/:\w+/, '(\\w+)')
 + '$')
}
```

We now must define the `call()` function which will locate the appropriate route to handle the request. First we loop through all of our routes, passing each one to our `match()` function which will check if the current route is the correct one for the request.

Remember how we allowed the syntax of `/path/:a/:b` in our paths? Our `match()` function will return the associated placeholder values as an array. So `/`

`user/:id/:operation`, requested with `/user/3/edit`, will return `[3, 'edit']`. If the route matches we apply our captures array to the matching route's callback function. Finally we throw an error if no routes matched our request.

```
function call(path, method) {
  for (var i = 0; i < routes.length; ++i)
    if (captures = match(routes[i], path, method))
      return routes[i].callback.apply(this, captures)
  throw "Route for path: `" + path + "' method: " + method
 + " cannot be found"
}
```

5. Matching Routes

Next we must define the **`match()`** function, used to determine if the route passed matches the given request. Since our API is RESTful, only routes matching the same HTTP method are acceptable. This can be simply implemented as the expression `route.method == method`, we then continue to use String's **`match()`** method against our previously generated `route.pattern` which will return an array of captures if the string matches. Finally we return the captures.

The first array element returned by **`String`**'s **`match()`** method is always the original string such as `['/user/2/edit', '2', 'edit']` which is not useful to pass to our route callback. What we really want is simply `['2', 'edit']`, which is why we use **`Array`**'s **`slice()`** method to return every element after the first.

```
function match(route, path, method) {
  if (route.method == method)
    if (captures = path.match(route.pattern))
      return captures.slice(1)
}
```

6. Complete Source

The entire source for what we have created is shown below for reference:

```
;(function(){
  var main = this, routes = []

  function match(route, path, method) {
    if (route.method == method)
      if (captures = path.match(route.pattern))
        return captures.slice(1)
  }

  function prep(path) {
    return new RegExp('^' + path.replace(/:\w+/, '(\\w+)')
 + '$')
  }

  function call(path, method) {
    for (var i = 0; i < routes.length; ++i)
      if (captures = match(routes[i], path, method))
        return routes[i].callback.apply(this, captures)
    throw "Route for path: `" + path + "' method: " + method
 + " cannot be found"
  }

  function def(method, options) {
    options = options || {}
    main[method] = function(path, callback){
      if (typeof callback == 'function')
        routes.push({
          method: options.as || method,
          path: path,
          pattern: prep(path),
          callback: callback
        })
      else
        return call(path, options.as || method)
    }
  }
```

```
  def('get')
  def('post')
  def('put')
  def('del', { as: 'delete' })
})()

get('/user/:id', function(id) {
  return 'Viewing user ' + id
})

del('/user/:id', function() {
  return 'User deleted'
})

print(get('/user/1'))
// “Viewing user 1”

print(del('/user/4'))
// “User deleted”

print(del('/user'))
// uncaught exception: Route for path: `/user' method: delete
 cannot be found
```

Chapter 4. Behavior Driven Development with JSpec

1. Overview

Now that we have looked at some interesting features of JavaScript, we will now be looking into a relatively new style of developing called Behavior Driven Development or BDD. Testing source code is nothing new, however the readability and simplicity of this task is becoming increasingly greater.

Although there are several testing frameworks for JavaScript, arguably the most the readable, extensive of those frameworks is “JSpec”. JSpec is a new framework in the JavaScript BDD scene, however it is highly innovative, actively developed and vastly increasing in popularity day by day.

JSpec sports hundreds of features, including the following:

- Framework agnostic (jQuery, MooTools, Prototype, etc)
- Custom (optional) grammar for highly readable specs
- Async support
- Mock Ajax/Timers
- Fixture support
- Rhino/Node.js/Browser support
- Over 50 core matchers
- Project templates
- Drupal/Ruby on Rails integration
- Modular extensions

2. Installation

Although JSpec may be manually downloaded from `http://jspec.info`, we will be using the Ruby gem, which can be installed by running the following command:


```
$ gem sources -a http://gems.github.com (if you have not pre-
viously done so)

$ sudo gem install visionmedia-jspec
```

3. Creating A Project Template

Now that the JSpec Ruby gem is installed, creating a project template for testing our RESTful library from the last chapter is as simple as executing the following command:

```
$ jspec init restful
```

We can then execute the command below to display the template structure JSpec has created for us:

```
$ ch restful && find . -print
```

```
.
./History.rdoc
./lib
./lib/yourlib.core.js
./README.rdoc
./spec
./spec/server.rb
./spec/spec.core.js
./spec/spec.dom.html
./spec/spec.rhino.js
./spec/spec.server.html
```

Although this may very version to version, JSpec creates us a nice basic project with a history (change log) file, readme, a `./lib` directory, as well as a `./spec` directory to place our test suites. All of the files in the `./spec` directory may be configured to run

each environment as you like, however `./spec/spec.core.js` is where we write our tests.

Before we begin with editing `./spec/spec.core.js`, the source from our RESTful library chapter should be pasted in `./lib/yourlib.core.js` so we have the library loaded to test against.

Note

All filenames may be renamed, however we will keep them as-is for simplicity

3.1. Writing Specifications

If we now open up `./spec/spec.core.js` in our favorite text editor, what we will see is some example JSpec code:

```
describe 'YourLib'
  describe '.someMethod()'
    it 'should do something'
      true.should.be true
    end
  end
end
```

This code is parsed by JSpec's custom grammar, and converted to the literal JavaScript shown below. It is note-worthy to mention that JSpec's grammar is not for everyone, if you dislike it and prefer to use the literal JavaScript there is no problem with doing so.

```
describe('YourLib', function() {
  describe('.someMethod()', function() {
    it('should do something', function(){
      expect(true).to(be, true)
```

```
    })
  })
})
```

We will begin by creating a `describe` block named “My Restful Library”, and then begin to describe each method we have provided as our public API. All assertions are enclosed with in an `it` block, which should describe a specific aspect of functionality. Each line containing “should” or “should.not” are what we call an *assertion* which is simply an expected value of `true` or `false`. Finally the right-hand of the “should” / “should.not” keywords is called the *matcher*, which is a specific function used to perform the assertion.

In the example below, we are asserting that each HTTP method has an associated object, with a constructor of Function.

```
describe 'My Restful Library'
  describe 'get()'
    it 'should be a function'
      get.should.be_a Function
    end
  end

  describe 'put()'
    it 'should be a function'
      put.should.be_a Function
    end
  end

  describe 'post()'
    it 'should be a function'
      post.should.be_a Function
    end
  end

  describe 'del()'
    it 'should be a function'
      del.should.be_a Function
    end
```

```
  end
end
```

3.2. Running Specifications

Since our library does not rely on the DOM, we do not need a browser to run the tests. If Mozilla's Rhino JavaScript interpreter is installed, we may simply execute the following command within our restful project's root directory:

```
$ jspec run --rhino
```

The results will output directly to your terminal, and will look similar to below:

```
Passes: 4 Failures: 0

 My Restful Library get()
  should be a function.

 My Restful Library put()
  should be a function.

 My Restful Library post()
  should be a function.

 My Restful Library del()
  should be a function.
```

To run the specifications in Safari execute the following command:

```
$ jspec run
```

Or we may also specify one or more browsers using the –browsers switch:

```
$ jspec run –browsers safari,firefox,chrome
```

To run and output the results every time a file is modified, use the following command:

```
$ jspec run --bind
```

This may be used in conjunction with other switches as well:

```
$ jspec run --bind --browsers safari,firefox,chrome
```

```
$ jspec run --bind --rhino
```

The following are equivalent, as jspec defaults to running with --bind

```
$ jspec
```

```
$ jspec run --bind
```

Finally we may use JSpec's Ruby server which will load `./spec/server.rb` and then start a server, opening the specified –browsers to `./spec/spec.server.html` which output results of each browser to the terminal. This is the quickest way to ensure your code is working across multiple browsers.

```
$ jspec run --server
```

```
$ jspec run --server --browers ff,opera,safari,chrome
```

3.3. Full Test Coverage

So we wrote a few tests, but its time to give our library full test coverage. A few more specifications have been added below, each asserting that routes can be matched, and placeholders are passed to the callback functions.

```
describe 'My Restful Library'
  describe 'get()'
    it 'should be a function'
      get.should.be_a Function
    end

    it 'should support simple routes'
      get('/user', function(){ return true })
      get('/user').should.be_true
    end

    it 'should support placeholders'
      get('/user/:id', function(id){ return id })
      get('/user/13').should.eql '13'
    end

    it 'should support several placeholders'
      get('/user/:id/:operation', function(id, operation)
{
        return {
          id: id,
          operation: operation
        }
      })
      get('/user/1/edit').id.should.eql '1'
      get('/user/1/edit').operation.should.eql 'edit'
    end
  end
end
```

Now that we have some more specifications, lets run them again. Oh no! We have an error; did you catch it when we wrote the library? This is why we write specifications.

```
Passes: 3 Failures: 1

 My Restful Library get()
  should be a function.
  should support simple routes.
  should support placeholders.
```

```
  should support several placeholders.
  Route for path: `/user/1/edit' method: get cannot be
found
```

Lets see if we can fix this bug. Open up the library code, and view the **prep()** function, if you look closely, we forgot to add the “g” or “global” modifier on the regular expression used to convert our placeholders to regular expressions. Once fixed, run the specifications again to watch your suite pass.

```
path.replace(/:\w+/, '(\\w+)')
path.replace(/:\w+/g, '(\\w+)')
```

So far we have been optimistic, assuming everything will work properly. Do not forget to test to prevent unexpected results. For example we have not yet written any specifications for what happens when a route cannot be found, so lets do so.

The **throw_error** matcher below may be used in several ways. Below we are asserting that a specific error message is thrown, however we may use it without arguments, supply an exception constructor, regular expression, or a combination of both.

The odd looking `-{` at the beginning of the line is provided by JSpec’s grammar, and converts to `function(){` to increase readability.

```
it 'should throw an error when no routes match'
    -{ get('/foo/bar') }.should.throw_error "Route for
 path: `/foo/bar' method: get cannot be found"
end
```

3.4. Additional Information

- JSpec documentation http://jspec.info
- Google Group http://groups.google.com/group/jspec
- IRC channel irc://irc.freenode.net#jspec
- TextMate bundle http://github.com/visionmedia/jspec.tmbundle/tree/master
- Source http://github.com/visionmedia/jspec
- Rails integration http://visionmedia.github.com/jspec/rails.html
- Drupal integration https://github.com/visionmedia/jspec-drupal
- Slide Show http://slidechop.com/presentations/68/slides

Chapter 5. Creating a jQuery Clone

1. Overview

In this chapter we are going to be creating a small, light-weight jQuery-like clone. This time we are going to approach the project with our new Behavior Driven Development knowledge, and write specifications before the implementation.

Note

> The goal of this chapter is not to create a full-blown cross browser compatible clone, it is meant to present concepts and techniques.

2. Laying The Foundation

To start the project, create a JSpec project named “mini”, as this will be the name of our small jQuery clone.

```
$ jspec init mini && cd mini
```

Then open the project up in your editor of choice. Open up `spec/spec.core.js` and write the specification below. This spec ensures that our `$` variable will be equivalent to accessing the `Mini` variable.

```
describe 'Mini'
  describe '$'
    it 'should alias Mini'
      $.should.eql Mini
    end
  end
end
```

Run the specifications using the command below:

```
$ jspec run
```

You should see the failure message below, due to us not defining `$`, or `Mini`.

```
should alias Mini ReferenceError: Can't find variable: $ near
 line 2
```

We can now start our library. Open up `lib/yourlib.core.js`, and write the following code. Here we are simply defining both the `$` variable, an `Mini` variable to the anonymous function on the right. If we run the specs again, we will see that it now passes.

```
;(function(){
  $ = Mini = function() {
  }
})()
```

3. Implementing $(“selector”)

The next step is to allow jQuery’s `$(“selector”)` and `$(“selector”, context)` syntax. Before we begin we should write the specification. First we have a `before_each` block, which will execute before each `it` block is run, allowing us to test against a fresh <ul> list. Then we assert that when a selector and context is given, that we should receive a collection of elements, ’s in this case.

```
describe '$("selector", context)'
  before_each
    ul = document.createElement('ul')
    ul.appendChild(document.createElement('li'))
```

```
    ul.appendChild(document.createElement('li'))
  end

  it 'should return a collection of elements'
    $('li', ul).should.have_length 2
  end
end
```

Running our specs would surely lead to a failure, so lets begin our implementation. Back in our library we now need to accept the selector and context parameters, which are then passed to the `Mini.init` constructor. The reason **`Mini`** itself is not the constructor, is because we would then have to use `new $('ul li')` instead of `$('ul li')`.

```
;(function(){
  $ = Mini = function(selector, context) {
    return new Mini.init(selector, context)
  }

  Mini.init = function(selector, context) {
  }
})()
```

We now initialize an array `elements`, and assign a context to the given context or document. The method **`querySelectorAll()`** then allows us to query the context element using css selectors.

We then return the call to `$.arrayLike(this, elements)`, which will transform our instance into an array-like object. Our collection of elements will contain the length property, indexed elements, as well as methods.

Before implementing `$.arrayLike()`, we need to define `$.toArray()` which accepts an array-like object, converting it to an array. An array-like object is simply an object with a length property, and indexed values. This may be a list of nodes, the `arguments` variable, an Array instance, etc. We call `Array.prototype.slice.call`

passing the array-like object as the receiver, which while internally iterating the object returns a **`Array`** instance. The slice method always returns a new array, so we also alias `$.toArray()` as `$.clone()`.

```
;(function(){

  $ = Mini = function(selector, context) {
    return new Mini.init(selector, context)
  }

  Mini.init = function(selector, context) {
    var elements = [], context = context || document
    elements = context.querySelectorAll(selector)
    return $.arrayLike(this, elements)
  }

  $.toArray = $.clone = function(array) {
    return Array.prototype.slice.call(array)
  }

})()
```

Our `$.arrayLike()` function may now accept the `self` argument, which is the object to become an array-like object, as well as the **`Array`** instance. You will see the odd looking `Array.prototype.splice.apply(self, clone)` call below, this approach is similar to the solution we used in `$.toArray()` however now we use the **`splice()`** method, which is essentially the same as iterating our cloned array, and assigning `this[0] = array[0]`, `this[1] = array[1]`, etc.

```
...
$.arrayLike = function(self, array) {
  var clone = $.clone(array)
  self.length = clone.length
  clone.unshift(0, clone.length)
  Array.prototype.splice.apply(self, clone)
  return self
}
```

```
...
```

If we were now to execute the following code, we would see that our collection now looks very much like an array, however its constructor is not `Array`, and it does not inherit any methods provided by `Array.prototype`.

```
console.log($('li', ul));
Object
0: HTMLLIElement
1: HTMLLIElement
length: 2
```

4. Defining Collection Methods

4.1. $.fn.get()

We can now begin creating methods for our collections. The first method we will create is `$("selector").get(0)`, which will return the element at the given index. First we must alter our specs slightly, moving the `before_each` block back a level, so that our other `describe` blocks will have access to the `ul` variable.

```
describe 'Mini'
  before_each
    ul = document.createElement('ul')
    ul.appendChild(document.createElement('li'))
    ul.appendChild(document.createElement('li'))
  end

  describe '$'
    it 'should alias Mini'
      $.should.eql Mini
    end
```

```
  end

  describe '$("selector", context)'
    it 'should return a collection of elements'
      $('li', ul).should.have_length 2
    end
  end

  describe '$.fn'
    describe '.get(i)'
      it 'should return the elment at the given index'
        $('li', ul).get(0).should.have_property 'nodeType'
      end

      it 'should return null otherwise'
        $('li', ul).get(3).should.be_null
      end
    end
  end
end
```

In our library we can define the `get()` method quite simply as below, which is really an alias for `$("selector")[0]`, since our object has array-like indices. We then alias `Mini.init.prototype` as `$.fn` since this is much less verbose, and easier to remember.

```
...
$.fn = Mini.init.prototype = {
  get: function(i) {
    return this[i]
  }
}
...
```


4.2. $.fn.each()

The next method we will be defining is `each()` which will iterate the collection using a callback function. Our specifications below state that both the index and the element itself should be passed to the callback function, as well as returning the collection so that it may be chain-able.

```
...
describe '$.fn'
  ...
  describe '.each(function(){})'
    it 'should iterate a collection'
      var indices = []
      var elements = []
      $('li', ul).each(function(i, element){
       indices.push(i)
       elements.push(element)
      })
      indices.should.eql [0, 1]
      elements.should.eql [$('li', ul).get(0), $('li',
 ul).get(1)]
    end

   it 'should return the collection for chaining'
      $('li', ul).each(function()
{}).should.be_an_instance_of Mini.init
   end
  end
end
...
```

Below is our implementation, as you can see the process is quite simple. We first iterate using a cached for loop to increase performance, invoke the callback with both the index and element a the current index, then return the collection.

```
```

```
  ...
  $.fn = Mini.init.prototype = {
    ...
   each: function(callback) {
    for (var i = 0, len = this.length; i < len; ++i)
      callback(i, this[i])
    return this
   }
  }
  ...
```

5. Implementing $(element), $([element, …]), and $(Mini)

A current flaw with our API is that we always assume a selector string is passed to `$()`, however this is not the case with jQuery. We must also accept a single element, arrays of elements, and a Mini collection object itself. To do this lets define some specifications first:

```
describe '$(element)'
    it 'should return a collection with the element passed'
      $(ul).should.have_length 1
    end
  end

  describe '$([element, ...])'
    it 'should return a collection with the elements passed'
      $($.toArray(ul.children)).should.have_length 2
    end
  end

  describe '$(Mini)'
    it 'should return itself'
      $($('li', ul)).should.have_length 2
    end
  end
end
```

Since we initially created our `Mini.init()` constructor to simply start with an empty array, and then finish with converting the Mini object into an array-like object, we can simply assign the `elements` variable at any time along the execution of the constructor.

As shown by the comments below, you can see we populate the `elements` array in various ways, depending on what has been passed as our first argument (`selector`). We first check if `selector` is a string, if so we use our **`querySelectorAll()`** call to get a node list of elements. Then we move on to check if a single element is passed, if so we simply push it to the elements array. When an array of elements is passed we can simply assign it to the `elements` variable, finally if the `selector` is an **`instanceof`** `Mini.init`, then we simply return it.

```
Mini.init = function(selector, context) {
  var elements = [], context = context || document

  // $('selector')

  if (typeof selector == 'string')
    elements = context.querySelectorAll(selector)

  // $(element)

  else if (selector.nodeType)
    elements.push(selector)

  // $([element, ...])

  else if (selector.constructor == Array)
    elements = selector

  // $(Mini)

  else if (selector instanceof Mini.init)
    return selector

  return $.arrayLike(this, elements)
}
```

```
$.fn = Mini.init.prototype = {
  mini: '0.0.1',
  get : function(i) {
    return this[i]
  }
}
```

6. Implementing $(function(){})

jQuery allows a function to be passed as a shortcut for the idiom `$(document).ready(function(){})`, however we will support the shortcut only. First we create a locally scoped array named `observers`, then check if our `selector` is a function, if so push it onto our stack of observers.

```
;(function(){

 var observers = []

  $ = Mini = function(selector, context) {
   return new Mini.init(selector, context)
  }

  Mini.init = function(selector, context) {
    var elements = [], context = context || document

    // $(function(){})

    if (typeof selector == 'function')
      return observers.push(selector)

  ...
```

At the bottom of our library we then need to add an event listener for the “DOMContentLoaded” event, which will be called once the DOM is available. Then we proceed to iterating our `observers`, calling each one.

```
  ...

 if (document.addEventListener)
  document.addEventListener("DOMContentLoaded", function(){
   document.removeEventListener("DOMContentLoaded",
 arguments.callee, false)
   for (var i = 0, len = observers.length; i < len; ++i)
     observers[i]()
  }, false)

})()
```

7. Test Drive

Now that we have a functioning library, lets try things out. Create an html document with the following content, open it in a browser such as Safari or FireFox, and open up your console.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://
www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Light-weight jQuery Clone</title>
    <script src="jquery_clone.js" type="text/javascript"></
script>
    <script type="text/javascript">
      $(function(){
        console.log($('ul li'));
        $('ul li:last-child').get(0).style.opacity = 0.3
      })
```



```
      </script>
    </head>
    <body>
      <ul>
        <li>One</li>
        <li>Two</li>
        <li>Three</li>
        <li>Four</li>
        <li>Five</li>
      </ul>
    </body>
  </html>
```

With this foundation, and the techniques learnt throughout the chapters, you will now have the knowledge to create a robust, lightweight jQuery clone.

Chapter 6. Tools Of The Trade

1. Overview

Every issue we bring you the hottest development tools and libraries in a chapter we call *Tools Of The Trade*.

2. Lispy JavaScript With Jasper

Jasper is a lisp interpreter written in JavaScript created by Defunkt (Chris Wanstrath). A hand-coded JavaScript recursive descent parser interprets sequences of lisp-like expressions, utilizing JavaScript internals to create a small yet elegant language.

2.1. Example

```
"Comments are just strings (for now)"
(defmacro defun (name args &rest body)
  (cons '= (cons name (list (append (list 'lambda args)
 body)))))

"Lispy aliases."
(= setq =)
(setq eq? ==)

"t and f and nil"
(= t (js "true"))
(= f (js "false"))
(= nil (js "null"))

(defun not (condition)
  (empty? condition))

(defmacro when (condition &rest body)
  (list 'if condition (append (list 'progn) body) f))

(defmacro unless (condition &rest body)
```

```
    (list 'if condition f (append (list 'progn) body)))

(defun each (items iterator)
  (unless (empty? items)
    (iterator (car items))
    (each (cdr items) iterator)))
```

2.2. More Information

- Jasper source `http://github.com/defunkt/jasper`

3. Mojo Mustache Templates

Mojo is a JavaScript implementation of the language *Mustache*, a relatively logic free templating system. Through the use of a C parser Mojo templates are statically compiled to literal JavaScript for use in your web or server based application.

Mustache is an abstract language which does not depend on the syntax of a host language. This means that the language can be ported to various host languages without change to it's own syntax, allowing the templates themselves to become portable as well.

3.1. Example

```
<h1>{{ title }}</h1>
{{# hasItems }}
  <ul>
    {{# items }}
      <li>{{ title }}</li>
    {{/ items }}
  </ul>
{{/ hasItems }}
```

3.2. More Information

- Mojo source code `http://github.com/visionmedia/mojo`
- Ruby implementation `http://github.com/defunkt/mustache`
- Python implementation `http://github.com/defunkt/pystache`
- Lua implementation `http://github.com/nrk/hige`

4. Vector Graphics With Raphaël

Raphaël is a stand-alone light-weight vector graphics library written by Dmitry Baranovskiy. A charting library based on Raphaël named gRaphaël supports several common graph formats such as line, pie, bar, and more.

4.1. More Information

- Documentation `http://raphaeljs.com`
- Charting `http://g.raphaeljs.com`
- Source code `http://github.com/DmitryBaranovskiy/raphael`

5. Underscore.js JavaScript Utilities

Underscore.js is a stand-alone JavaScript library providing object-oriented, and functional utilities created by Jeremy Ashkenas of DocumentCloud. These utilities range from enumerating array-like objects, assertions, to templating.

5.1. Example

Nearly every method in Underscore has a contrasting functional alternative. The example below demonstates how we may use Underscore's `_.map()` method in both a functional, and object-oriented manor, while yielding the same results.


```
_.map([1, 2, 3], function(num){ return num * 3 })
// => [3, 6, 9]

_([1, 2, 3]).map(function(num){ return num * 3 })
// => [3, 6, 9]
```

5.2. More Information

- Documentation `http://documentcloud.github.com/underscore/`
- Source code `http://github.com/documentcloud/underscore/`

6. RightJS Framework

RightJS is a new JavaScript framework authored by Nikolay V. Nemshilov, whom set out to create high performance, user-friendly, light-weight framework. Built on idioms founded by popular frameworks such as jQuery and Prototype, a solid UI library, and loads of documentation this framework is certainly one worth watching.

6.1. Example

RightJS has a lot to explore, however thankfully a lot of documentation is provided as well. Below is are a few concise examples.

```
// Add class when clicked
$(element).on('click', 'addClass', 'clicked')

// Fade in and handle callback
$(element).fade('in', { onFinish: function(){ }})

// Basic request
new Xhr('/foo/bar').send()
```



```
// JSON request
Xhr.load('/some.json', {
  onSuccess: function(request) {
    var json = request.responseJSON;
    // ....
  }
})
```

6.2. More Information

- Documentation `http://rightjs.org/`
- Source code `http://github.com/rightjs/rightjs-core`

7. Express Web Framework

Express is a Sinatra inspired web development framework powered by Node.js. Due to the asynchronous nature of node, early benchmarks of Express show performance levels nearing *400%* greater than Ruby's Sinatra.

7.1. Example

Express is packed with features ranging from routing and environments to sessions. The beginnings of an application shown below illustrates the simplicity of the RESTful routing DSL provided to make web applications intuitive to develop.

```
configure(function(){
  use(MethodOverride)
  use(ContentLength)
  use(CommonLogger)
  set('root', __dirname)
})

configure('production', function(){
  enable('cache views')
})
```

```
get('/user', function(){
  this.redirect('/user/' + currentUser.id + '/view')
})

get('/user/:id/view?', function(id){
  this.render('user.haml.html', { locals:
 { user: User.get(id) }})
})

del('/user/:id', function(id){
  User.get(id).destroy
})

run()
```

7.2. More Information

- Documentation `http://wiki.github.com/visionmedia/express`
- Source code `http://github.com/visionmedia/express`
- Node.js `http://nodejs.org`

8. JavaScript Extensions Via Ext.js

Ext.js is a growing library of JavaScript extensions, utilizing the CommonJS module pattern used by node.js amoung other frameworks.

8.1. Example

Below is a small snippit of functionality provided by Ext.js. If you plan on using all the functionality you can simply `require('ext')`.

```
require.paths.unshift('lib')
require('ext')
process.mixin(require('sys'))
```

```
var number = require('ext/number')

p((5).hours)
// => 180000

p(number.ordinalize(4))
// => '4th'

p((2).ordinalize)
// => '2nd'

p('user-name'.camelcase)
// => 'UserName'

p('foo bar'.md5)
// => '327b6f07435811239bc47e1544353273'

p('foo bar'.base64Encode)
// => 'Zm9vIGJhcg=='

p('Zm9vIGJhcg=='.base64Decode)
// => 'foo bar'

p(number.currency(1000))
// => '1,000'

p((10000000.99).currency)
// => '10,000,000.99'

printf('%10s made $%C on the %D\n', 'tj', 30000, 14)
printf('%10s made $%C on the %D\n', 'scott', 2000, 3)
/*
        tj made $30,000.00 on the 14th
     scott made $2,000.00 on the 3rd
*/

p(sprintf('#%X%X%X', 255, 255, 0))
// => '#FFFF00'

p((new Date('May 25, 1987')).format('%Y-%m-%d'))
// => '1987-05-25'

p((new Date('May 25, 1987')).format('took place on %A the
 %nd'))
// => 'took place on Monday the 25th'
```

```
SiteGenerator = function(){}
p('site_generator'.camelcase.variable)
// => [Function]
```

8.2. More Information

- Documentation `http://wiki.github.com/visionmedia/ext.js`
- CommonJS `http://commonjs.org`