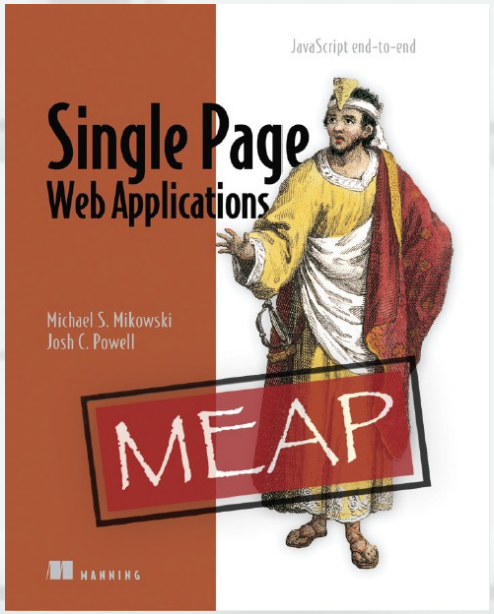
What Every JS Developer Should Know
In 30 Minutes

Michael Mikowski
For The Women's JavaScript Study Group

2014-12-10 San Francisco, CA
http://manning.com/mikowski

Who is Michael Mikowski?

- Co-Author Single Page Web Applications – JavaScript end-to-end
- http://manning.com/mikowski
- UI Architect at various SF companies
- Architect on 5 production SPAs since 2007, Primary developer on a 6th.
- Previous back-end developer on HP/HA clusters (2B wt/week)
- First SPA: European and US AMD “where to buy” shopping site before Backbone, TaffyDB, Node, or IE7



Linkedin for more...

Why JS: Market

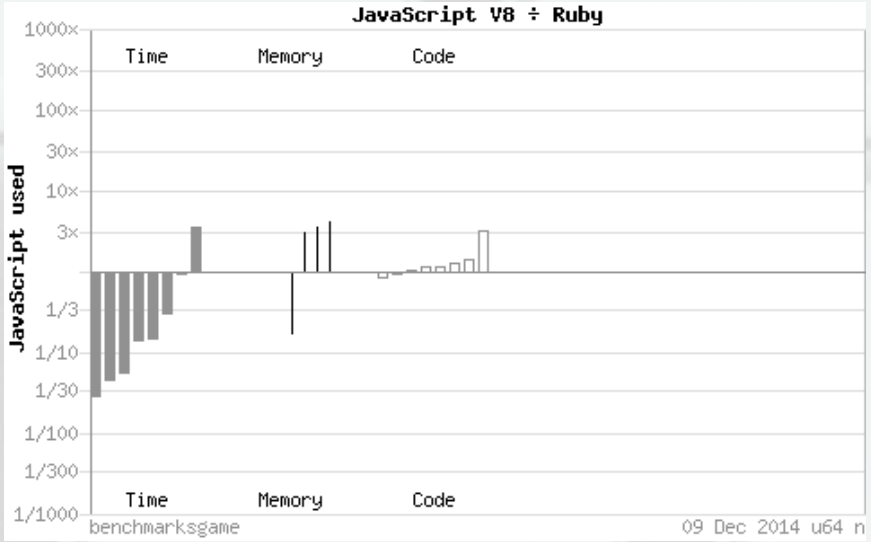
- JavaScript JIT Compilers on ~3-4 billion mobile devices
- Instead of switching JavaScript for a better language, Browser makers are making JavaScript a better language (more later)

Why JS: Capabilities

- Standards are converging (IE9+ are much less awful)
- High-performance rendering options are now standard WebGL (ask me about VRML), SVG, Canvas, CSS3 (“HTML5”)
- Comprehensive conventions and tools are now available (see JSLint, jQuery)
- IDEs have really improved – see WebStorm

Why JS: Performance

- JS now compiles to machine code and is by composite benchmarks around 10x faster than Ruby or Python per core
- New features like load balancing and SMID instructions are bringing greater improvements

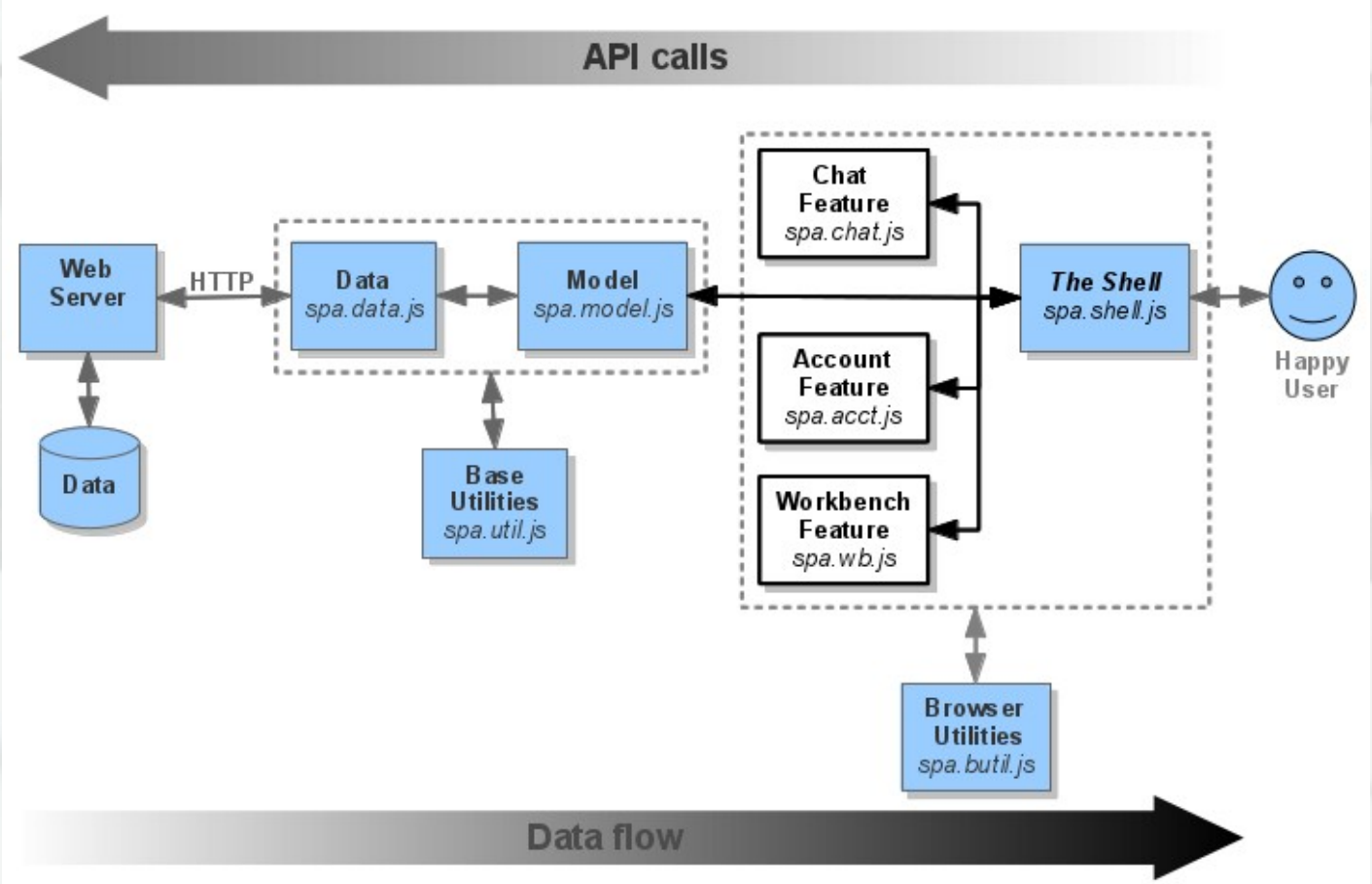


JavaScript advances: Pick a career...

Overview

- Architecture
- Data types and names
- Variable scope, Scope chains
- Variable hoisting
- Prototype inheritance and chains
- Self-executing anonymous functions
- Execution context
- Exploiting closures for fun and profit
- The module pattern

Architecture



Data types and Names

- Local variables use **under_scores**
- Module scope use **camelCase**
- **There should be only one global variable in our spa** – our root namespace

Data types and Names

DataType	Indicator	Examples	Notes
Boolean	sw, is, has, do	is_used	***true*** or ***false***
String	name, text, title, type, key, str	user_name	Type and key indicate enum
Integer	int, count, i,j,k, index, length	list_length	Indicates intent; only Firefox uses type-inference
Number	num,n,ratio	scale_ratio	Always signed double fp
Regex	regex	regex_match	Technically an object
Array	list	user_list	Ordered list
Hash (Map)	map	user_map	Technically an object
Object	obj, or <noun>	house_obj	Traditional object with methods
JQuery Obj	$	$tabs	Technically an object
Function	<verb>noun	make_dog	First class artifact
unknown	data	http_data	Unknown or polymorphic

Variable scope - basic

```
var
  visitPrison,
  freemanText = 'I am global!';

visitPrison = function () {
  var prisoner_text = 'I am local!';
}
visitPrison();

console.log( freemanText  );  // 'I am global!'
console.log( prisoner_text ); // error - not declared
```

- Scope provided by JS is functional only
- **freeman_text** is available within containing function
- **prisoner_text** is not

Variable scope – unintentional global

```
// bad
var visitPrison = function () {
  for( i = 0; i < 10; i++ ) {
    // ...
  }
}
visitPrison();
console.log(i); // whoops, we declared a global
delete window.i;

// good
visitPrison = function () {
  var i;
  for ( i = 0; i < 10; i++ ) {
    // ...
  }
}
visit_prison();
console.log(i); // error - not declared - good!
```

Variable scope – declare and assign

```
var visitPrison = function () {
  var
    i, j, k, shoe_count, sheet_count,  // declare
    prisoner_text = 'I am local!',     // declare and assign
    warden_text   = 'I am local too!',
    guard_text    = 'I am local three!'
    ;
};
```

- Use a single var statement per functional scope
- Multiple **declarations** per line
- One **assignment** per line
- Prefer to declare first, assign later
- Combination of declare *and* assign can confuse!

Scope chains A

```
var freeman_text, visitSupermax;
freeman_text = 'here!'; // window

visitSupermax = function () {
  var visit_prison;
  visit_prison = function () {
    console.log( freeman_text ); // window, logs 'here!'
  }

  visit_prison(); // logs 'here!'
  console.log(freeman_text); // window, logs 'here!'
};

visitSupermax(); // logs 'here!'.  Twice.

console.log(freeman_text); // window, logs 'here!'
```

Scope chains B

```
var freeman_text, visitSupermax;
freeman_text = 'here!'; // window

visitSupermax = function () {
  var
    visit_prison,
    freeman_text = 'assigned'; // window.visitSupermax

  visit_prison = function () {
    var freeman_text; // window.visitSupermax.visit_prision
    // window.visitSupermax.visit_prison.freeman_text
    console.log( freeman_text ); // logs undefined
  };

  visit_prison(); // logs undefined
  // window.visitSupermax
  console.log(freeman_text); // logs 'assigned'
};
visitSupermax(); // logs 'undefined' and 'assigned'
console.log(freeman_text); // window, logs 'here!'
```

Hoisting - Basic

```
var visitPrison = function () {
  console.log( prisoner_text );
  var prisoner_text = 'Now I am defined!';
  console.log( prisoner_text );
};
visitPrison(); // logs 'undefined', 'Now I am defined'
```

- Variable declarations are always “hoisted” to top of function scope, but not assignments
- **First log:** prisoner_text is declared but unassigned, and therefore logs **`undefined`**
- **Second log:** variable declared *and* assigned, logs provided value, **`Now I am defined`**

Hoisting – in scope

```
var
  visitPrison,
  freemanText = 'Regular Joe'
  ;

visitPrison = function () {
  console.log( freemanText ); // logs 'Regular Joe'
};

visitPrison();
```

- **`freemanText`** is in window scope, and therefore is defined and assigned when its value is logged.

Hoisting – surprise!

```
var
  visitPrison,
  freemanText = 'freemanText is assigned';

VisitPrison = function () {
  console.log( freemanText );
  var freemanText;
}
VisitPrison(); // logs undefined
```

- **`freemanText`** used in **`console.log()`** uses closest scope chain – **`visitPrison.freemanText`**, which is hoisted and undefined
- Placing declarations anywhere but the top only buys you confusion – there is no performance benefit!

Class vs. Prototype – Importance

- Most OO languages: Class-based objects
- JavaScript: Prototype-based objects

Class vs. Prototype – Simple Object
construction steps

- Class
  - Define class
  - Create constructor
  - Instantiate
- Prototype
  - Explicitly declare object properties

Class vs. Prototype – Simple Object
example

```
/* simple object */
public class Prisoner {
  public int sentence  = 4;
  public int probation = 2;
  public string name   = "Joe";
  public int id        = 1234;
  public Prisoner(string name, int id){
    this.name = name;
    this.id   = id;
  }
}
Prisoner prisoner = new Prisoner();
```

```
// simple object
var prisoner = {
  id   : 1234,
  name : 'Joe',
  probation_length : 2,
  sentence_length  : 4
};
```

Class vs. Prototype – Inheritance
construction steps

- Class
  - Define class
  - Create constructor
  - Instantiate
- Prototype
  - Define prototype
  - Create constructor
  - Explicitly inherit from prototype(s)
  - Instantiate

Class vs. Prototype – Inheritance
example

```
/* step 1 */
public class Prisoner {
  public int sentence  = 4;
  public int probation = 2;
  public string name;
  public string id;
  /* step 2 */
  public Prisoner( string name,
    string id ) {
    this.name = name;
    this.id   = id;
  }
}
/* step 3 */
Prisoner firstPrisoner
  = new Prisoner("Joe","12A");
Prisoner secondPrisoner
  = new Prisoner("Sam","2BC");
```

```
// * step 1 *
var prisonerProto = {
  probation_length : 2,
  sentence_length  : 4
};
// * step 2 *
var makePrisoner = function(
  name, id ) {
  // * step 3 *
  var prisoner
    = Object.create(prisonerProto);
  prisoner.id   = id;
  prisoner.name = name;
  return prisoner;
};
// * step 4 *
var firstPrisoner =
  makePrisoner( 'Joe','12A' );
var secondPrisoner =
  makePrisoner( 'Sam', '2BC' );
```

Prototype chain

Requested property	Prototype chain
`prisoner1.name`	`{ name : 'Joe' }`

`prisoner1.sentence_length`	`{ name : 'Joe',`
	`  __proto__ : { sentence_length : 4 }`
	`}`

`prisoner1.toString`	`{ name : 'Joe',`
	`  __proto__ : { sentence_length : 4,`
	`  __proto : {`
	`  toString :`
	`  [ Function: toString ],`
	`  …`
	`  }`
	`  }`
	`}`

`prisoner.is_hopeless`	*undefined* → no value in prototype chain

Class vs. Prototype – Summary

- Prototypical inheritance can be very fast
- Remember the prototype chain – changing a prototype *will* change all existent objects that use it
- See **`hasOwnProperty`** to get not-inherited properties
- Other keys may be “hidden” depending on environment, e.g. **`__proto__`** is not shown in node.js
- There are libraries that create pseudo-classical object patterns. If you are into living a lie, you can use them. They are not recommended.
- Object factories can provide class-like capabilities

Execution context

```
var run_outer = function ( int1, int2 ) {
  var
    local_text1 = 'foo',
    local_text2 = 'bar',
    run_inner
    ;

  run_inner = function () {
    console.log('inner');
  };
};
run_outer(1,2);
```

every time you call a function, an **Execution Context Object** is created!

Code	execution object - pass 1
`run_outer( 1, 2 );`	`{}`

`var run_outer`
`  = function ( int1, int2 ) {`
	`{ int1 : 1,`
	`  int2 : 2`
	`}`

`  var`
`  local_text1 = 'foo',`
`  local_text2 = 'bar',`
`  run_inner`
`  ;`
	`{ int1 : 1,`
	`  int2 : 2,`
	`  local_text1 : undefined,`
	`  local_text2 : undefined,`
	`  run_inner  : undefined`
	`}`

`  run_inner = function () {`
`  console.log('inner');`
`  };`
	`{ int1 : 1,`
	`  int2 : 2,`
	`  local_text1 : undefined,`
	`  local_text2 : undefined,`
	`  run_inner  : function (){`
	`  console.log('inner');`
	`  }`
	`}`

Code	execution object – pass 2
`var run_outer = function ( int1, int2 ) {`	`{ int1 : 1, int2 : 2, local_text1 : undefined, local_text2 : undefined, run_inner : function (){ console.log('inner'); } }`
`var local_text1 = 'foo', local_text2 = 'bar', run_inner ;`	`{ int1 : 1, int2 : 2, local_text1 : 'foo', local_text2 : 'bar', run_inner : function (){ console.log('inner'); } }`
`run_inner = function () { console.log('inner'); };`	`{ int1 : 1, int2 : 2, local_text1 : 'foo', local_text2 : 'bar', run_inner : function (){ console.log('inner'); } }`

Exploiting closures

```
var curryLogMsg, logMsg;
function curryLogMsg ( arg_text ) {  // outer function
  var log_msg;
  log_msg = function () {            // inner function
    console.log( arg_text );
  };
  return log_msg;
}

logMsg = curryLogMsg('yeah');  // closure results from assignment

LogMsg(); // logs 'yeah'

curryLogMsg('Bob Newhart rules')(); // null context - no closure
```

Closure and reference counts

```
var curryLogMsg, logHello, logStaynAlive;
curryLogMsg = function ( arg_text ){
  var log_msg = function (){ console.log( arg_text ); };
  return log_msg;
};

logHello      = curryLogMsg('hello');
LogCopy       = logHello;
logStaynAlive = curryLogMsg('stayn alive!');

logHello();         // logs 'hello'
logCopy();          // logs 'hello' again
logStaynAlive();    // logs 'stayn alive!'

// reference count to logHello is 2
delete window.logCopy;  // reference count now 1
delete window.logHello; // reference count now 0

// execution object for logHello may now be garbage collect
```

Closures and deep thoughts

An experienced developer will conclude:

- It is very easy create deep closures
- Closure **very good** when we want it (think 'object inheritance')
- It is **very bad** when we don't (think 'memory leak')
- There are rules and tools we apply to avoid unintended closures
    - e.g. do not declare functions in a loop
- ***Execution object are maintained until reference counts to a closure drops to zero***

Self-executing anonymous functions

```
var warningMsg = 'Tornado!';

(function ( $ ) {
  var warningMsg = 'hear me now '
    + 'and believe me later'
    ;
  // logs 'hear me now and … '
  console.log( warningMsg );
}( jQuery ));

// logs 'Tornado!'
console.log(warningMsg);
```

- Function executes immediately
- **`warningMsg`** is local to function scope
- “Turkey sandwich” jQuery reference implementation

The module pattern

```
var spa = (function () {
  var
    warnUser,
    alert_msg = 'Hear me now '
      + 'and believe me later'
    ;

  warnUser = function ( warn_msg ){
    console.warn(
      [ alert_msg, warn_msg ]
      .join(', ')
    );
  };
  return { warnUser : warnUser };
}());

spa.warnUser( 'JS Denzians!' );
```

- Defines namespace
- Returns public methods
- In this example, **`warnUser`** is a public method. All other variables are private to spa namespace.

Module pattern best practice

- Pick only one namespace for your app, e.g. **`spa`**
- Subdivide namespace, one division per file
  - Must be declared in-order e.g. spa → spa.wb → spa.wb.render
- Name JS file per namespace provided, e.g. **`spa.wb.js`**
- Use parallel namespaces for CSS files and classes
- Buy the book for more (daddy needs a new car)

Summary

- Understanding types, scope, and hoisting demystify otherwise odd-seeming variable behavior
- Execution context is at the heart of much of head-scratching JS behavior
- Prototype inheritance is can be easy and useful
- Self-executing anonymous functions replace block scope and are basis for module pattern
- Closures are all about reference counts

What Every JS Developer Should Know
In 30 Minutes

Michael Mikowski
For The Women's JavaScript Study Group

2014-12-10 San Francisco, CA
http://manning.com/mikowski