Advanced JavaScript

Banyan Talks

Łukasz Rajchel

Agenda

- Object-Oriented JavaScript
- Performance Improvements
- Debugging and Testing
- Distribution

What Makes Good JavaScript Code?

- It's structurized so that each component can be easily identified
- Most components can be tested automatically
- It's easily extendable and changeable
- It works and loads quickly on all modern browsers

Object-Oriented JavaScript

- References
- Function Overloading
- Type Checking
- Scopes, Closures & Context
- Object Creation
- Public, Private & Privileged Members
- Static Members
- Prototypal Inheritance
- Classical Inheritance

Object-Oriented JavaScript
References

- Variables are always assigned by reference

```
var obj = {};
var objRef = obj;
obj.myProperty = true;

console.log(obj.myProperty === objRef.myProperty); // true
```

```
var arr = [1, 2, 3];
var arrRef = arr;
arr.push(4);

console.log(arr.length);    // 4
console.log(arrRef.length); // 4
```

Object-Oriented JavaScript References

- References always point to a final referred object, not the reference itself

```
var arr = [1, 2, 3];
var arrRef = arr;
arr = [1, 2];

console.log(arr.length);    // 2
console.log(arrRef.length); // 3 - arrRef still points to original array
```

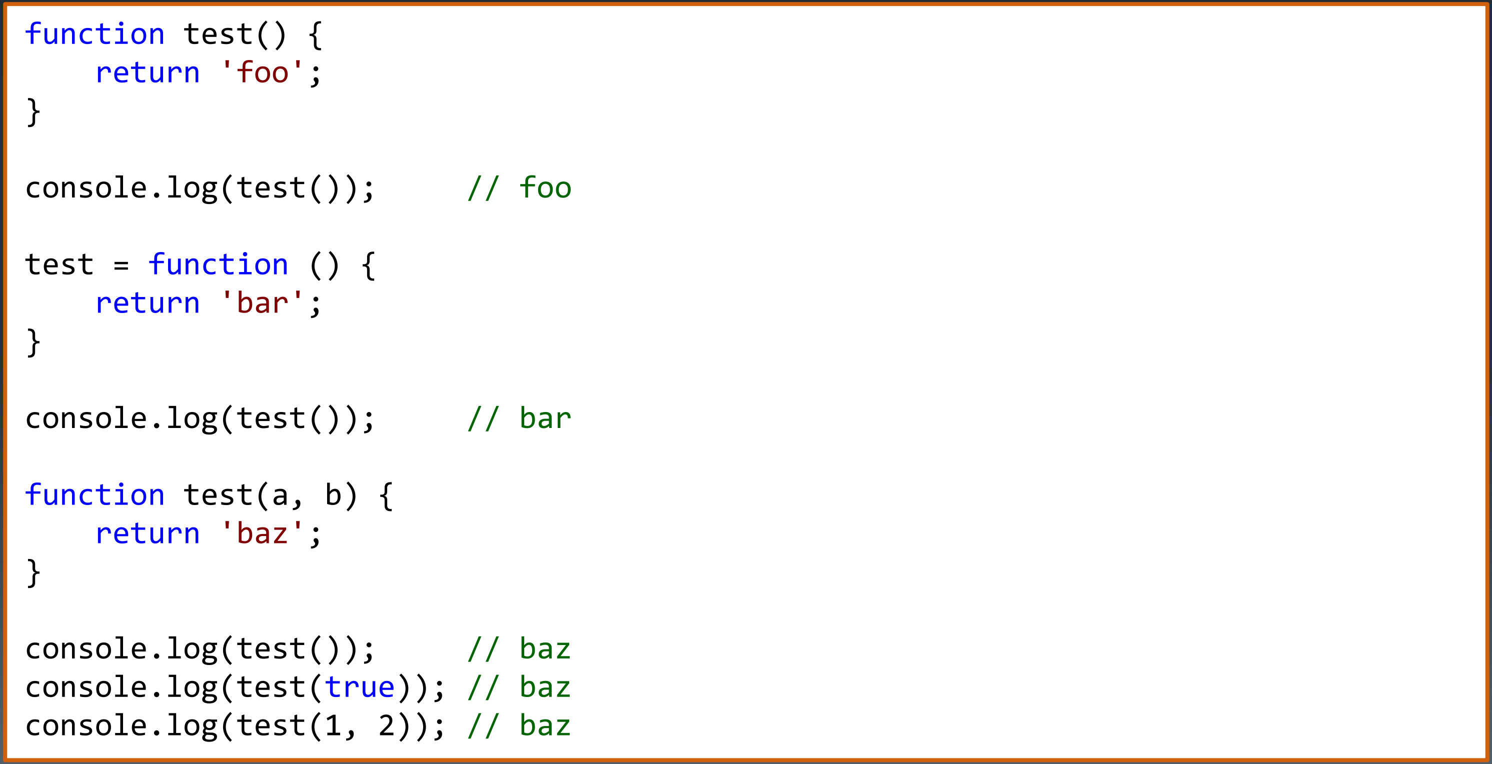
- Strings are immutable

```
var str = 'foo';
var strRef = str;
str += 'bar';               // new object instance has been created

console.log(str);           // foobar
console.log(strRef);        // foo - strRef still points to original object
```

Object-Oriented JavaScript Function Overloading

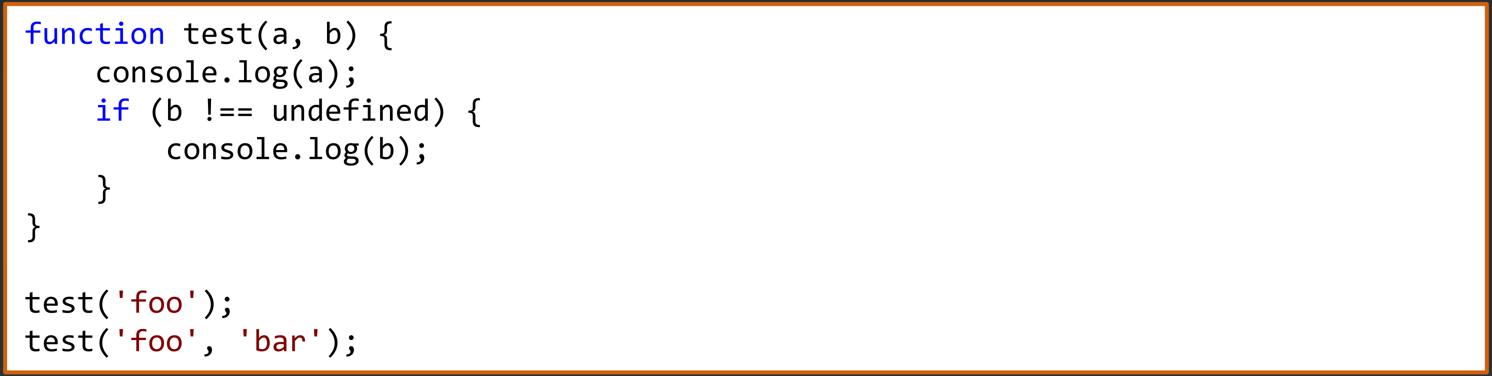
- You cannot have more then one function with the same name defined in one scope (second definition will overwrite previous one without notice)
- There are some workarounds to have function overloading:
  - optional function arguments
  - variable number of function arguments
- Remember that `function foo() { ... }` is just a shorthand for `var foo = function () { ... };`

Object-Oriented JavaScript
Function Overloading

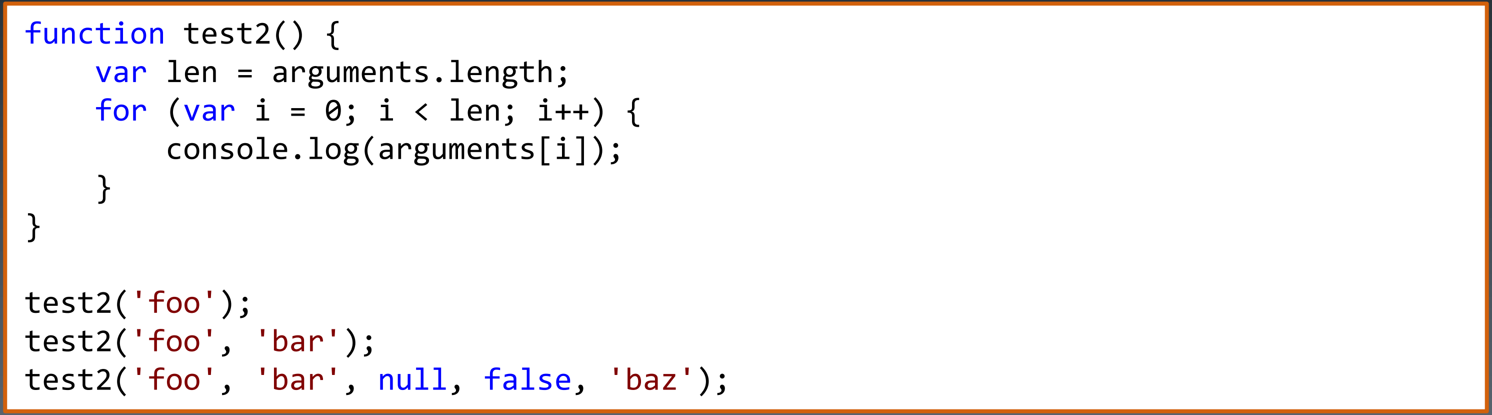


```
function test() {
    return 'foo';
}

console.log(test());     // foo

test = function () {
    return 'bar';
}

console.log(test());     // bar

function test(a, b) {
    return 'baz';
}

console.log(test());     // baz
console.log(test(true)); // baz
console.log(test(1, 2)); // baz
```

Object-Oriented JavaScript
Function Overloading



```
function test(a, b) {
    console.log(a);
    if (b !== undefined) {
        console.log(b);
    }
}

test('foo');
test('foo', 'bar');
```



```
function test2() {
    var len = arguments.length;
    for (var i = 0; i < len; i++) {
        console.log(arguments[i]);
    }
}

test2('foo');
test2('foo', 'bar');
test2('foo', 'bar', null, false, 'baz');
```

Object-Oriented JavaScript
Type Checking

- Watch out for traps when checking object types

```
typeof undefined; // undefined
typeof null;      // object
typeof {};        // object
typeof [];        // object
typeof 1;         // number
typeof 1.2;       // number
typeof true;      // bool
typeof alert;     // function

var arr = [];
if (arr !== null && arr.constructor === Array.constructor) {
    // an array
}

var obj = {};
if (obj !== null && obj.constructor === Object.constructor) {
    // an object
}
```

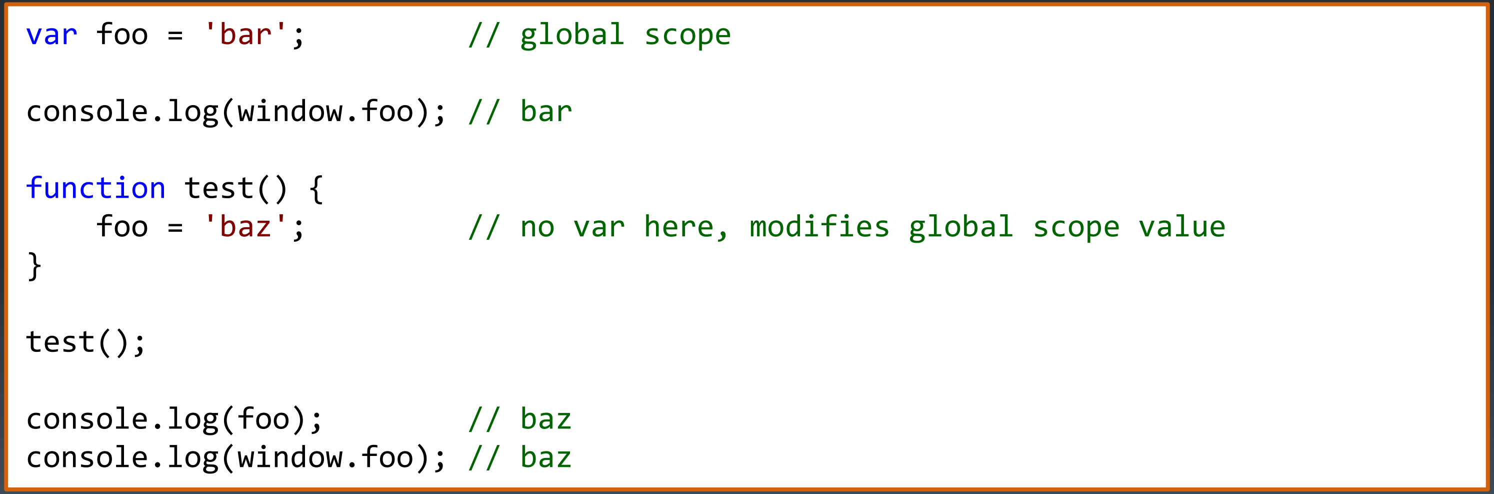
Object-Oriented JavaScript Scope

- Scope is kept within function, not within blocks (such as `while`, `if` and `for` statements)
- This may lead to some seemingly strange results if you're coming from a block-scoped language

```
var foo = 'bar';
if (true) {
    var foo = 'baz';
}

console.log(foo);     // baz

function test() {
    var foo = 'test'; // new scope
}

test();
console.log(foo);     // baz
```

Object-Oriented JavaScript Scope

- Be careful with global scope



```
var foo = 'bar';          // global scope

console.log(window.foo); // bar

function test() {
    foo = 'baz';          // no var here, modifies global scope value
}

test();

console.log(foo);         // baz
console.log(window.foo); // baz
```

Object-Oriented JavaScript
Closures

- Closures are means through which inner functions can refer to the variables present in their outer enclosing functions after their parent functions have already terminated

```
var obj = document.getElementById('someId');

setTimeout(function () {
    obj.style.display = 'none';
}, 1000);

function delayedAlert(msg, time) {
    setTimeout(function () {
        alert(msg);
    }, time);
}

delayedAlert('foo', 2000);
```

Object-Oriented JavaScript Closures

- Closures are also used to define "access modifiers" for object members

```
var Test = (function () {
    var privateMsg = 'foo';                // private members
    window.onunload = function () {
        alert(privateMsg);
    };

    return {                               // privileged members
        getMsg: function () {
            return privateMsg;
        }
    }
})();                                      // immediately execute function

console.log(Test.privateMsg === undefined); // true, private member
console.log(Test.getMsg());                // foo
```

Object-Oriented JavaScript
Closures

```
for (var i = 0; i <= 2000; i += 1000) {
    setTimeout(function () {
        console.log('i value in closure is ' + i);
    }, i);
}

// i value in closure is 2000
// i value in closure is 2000
// i value in closure is 2000
```

```
for (var i = 0; i <= 2000; i += 1000) {
    (function (i) {
        setTimeout(function () {
            console.log('i value in closure is ' + i);
        }, i);
    })(i);
}

// i value in closure is 0
// i value in closure is 1000
// i value in closure is 2000
```

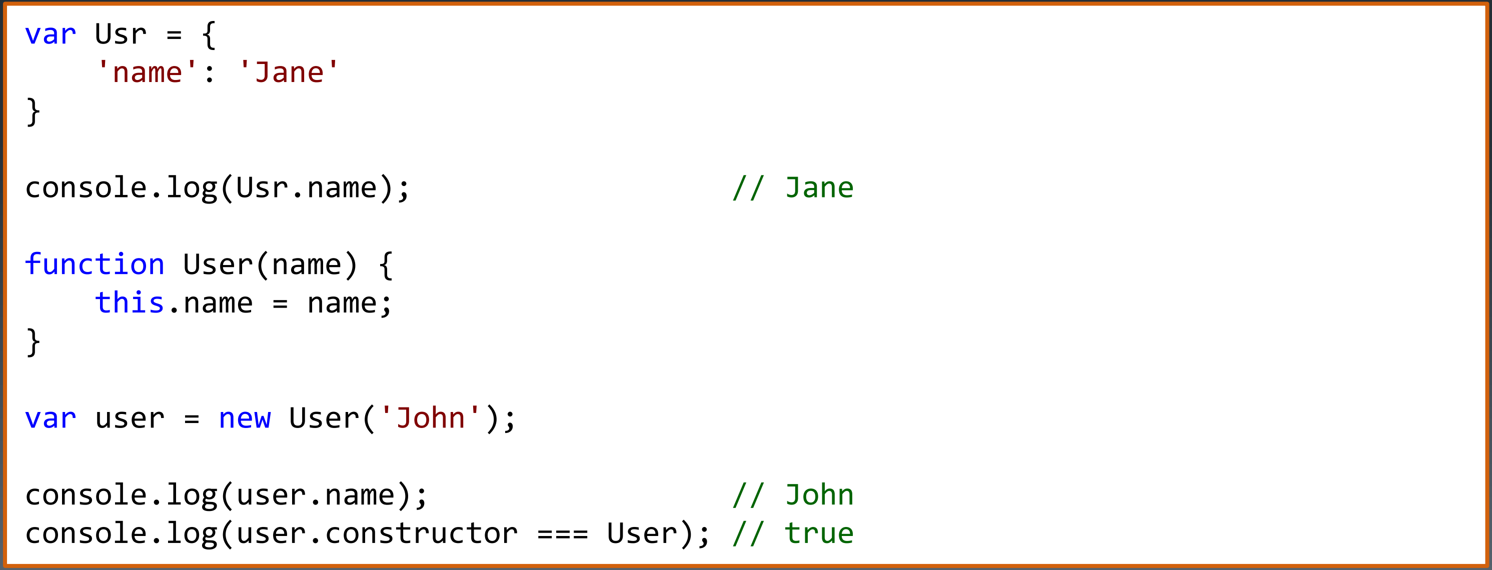
Object-Oriented JavaScript Context

- `this` always refers to the object the code is currently inside of

```
var obj = {
    'yes': function () {
        this.val = true;
    },
    'no': function () {
        this.val = false;
    }
};

console.log(obj.val === undefined); // true
obj.yes();
console.log(obj.val);               // true

window.no = obj.no;                 // switching context
window.no();

console.log(obj.val);               // true
console.log(window.val);            // false
```

Object-Oriented JavaScript Object Creation

- No classes or any kind of schemes for objects
- Objects can be created either as standard objects or as functions



```
var Usr = {
    'name': 'Jane'
}

console.log(Usr.name);                  // Jane

function User(name) {
    this.name = name;
}

var user = new User('John');

console.log(user.name);                 // John
console.log(user.constructor === User); // true
```

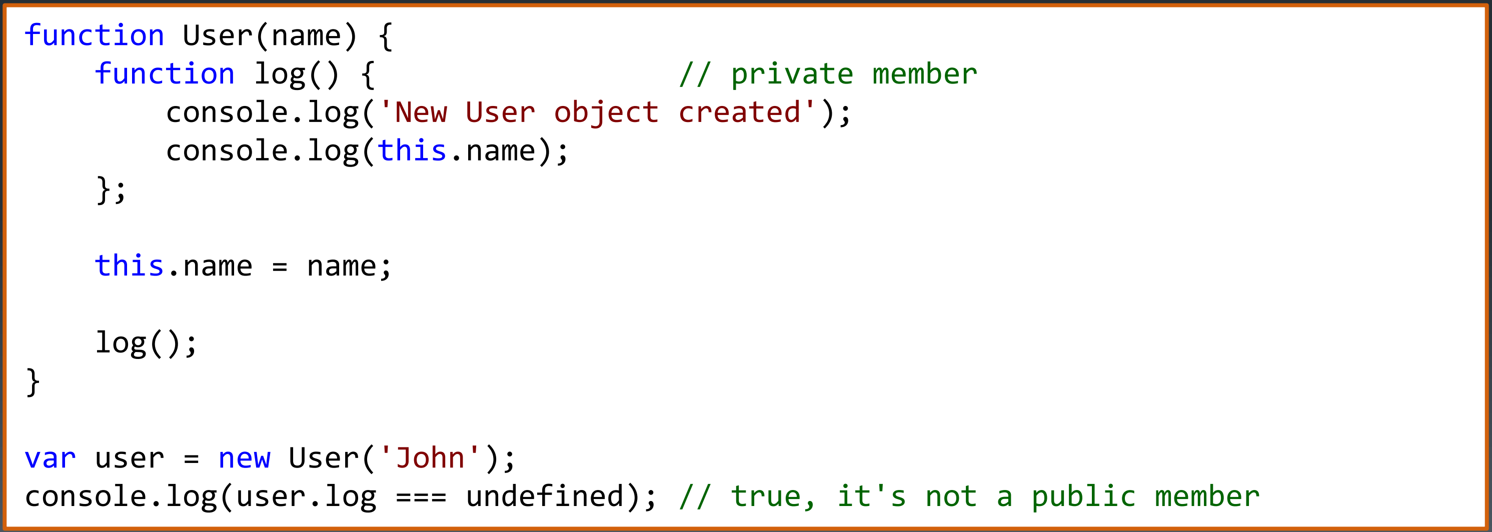
Object-Oriented JavaScript Public Members

- Added using `prorotype` property, which contains an object that will act as a base reference for all new copies of its parent object

```
function User(name) {
    this.name = name;
}

User.prototype.getName = function () { // public member
    return this.name;
};

var user = new User('John');

console.log(user.getName());           // John
```

Object-Oriented JavaScript
Private Members

- Defined as functions and variables defined inside "object-function"



```
function User(name) {
    function log() {                 // private member
        console.log('New User object created');
        console.log(this.name);
    };

    this.name = name;

    log();
}

var user = new User('John');
console.log(user.log === undefined); // true, it's not a public member
```

Object-Oriented JavaScript
Privileged Members

- Members that are able to manipulate private members while still beign accessible as public members

```
function User(name) {
    this.name = name;

    var createdDate = new Date();

    this.getCreatedDate = function () { // privileged member
        return createdDate;
    };
}

var user = new User('John');
console.log(user.createdDate());        // John
```

Object-Oriented JavaScript Static Members

- Accessible only in the same context as the main object

```
User.clone = function (user) {         // static member
    return new User(user.name);
}

var user = new User('John');
var userClone = User.clone(user);
userClone.name = 'Jack';

console.log(user.name);                // John
console.log(userClone.name);           // Jack

console.log(user.clone === undefined); // true
```

Object-Oriented JavaScript Prototypal Inheritance

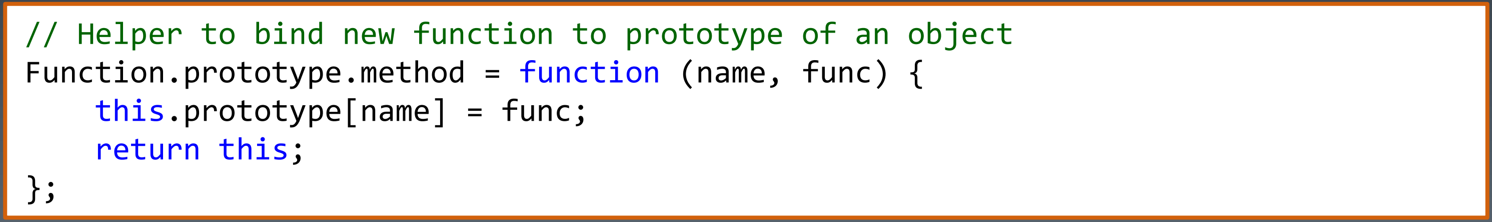
- Object constructor inherits methods from other object by creating a `prototype` object from which all other new objects are build
- Single inheritance
- Prototypes does not inherit their properties from other prototypes or constructors, they inherit them from physical objects

Object-Oriented JavaScript Prototypal Inheritance

```
function Person(name) {                  // base object
    this.name = name;
}

Person.prototype.getName = function () { // base method
    return this.name;
};

function User(name, age) {               // new object
    this.name = name;
    this.age = age;
}

User.prototype = new Person();           // inherits all base object properties
User.prototype.getAge = function () {    // derived object method
    return this.age;
};

var user = new User('John', 25);
console.log(user.getName());             // John
console.log(user.getAge());              // 25
```

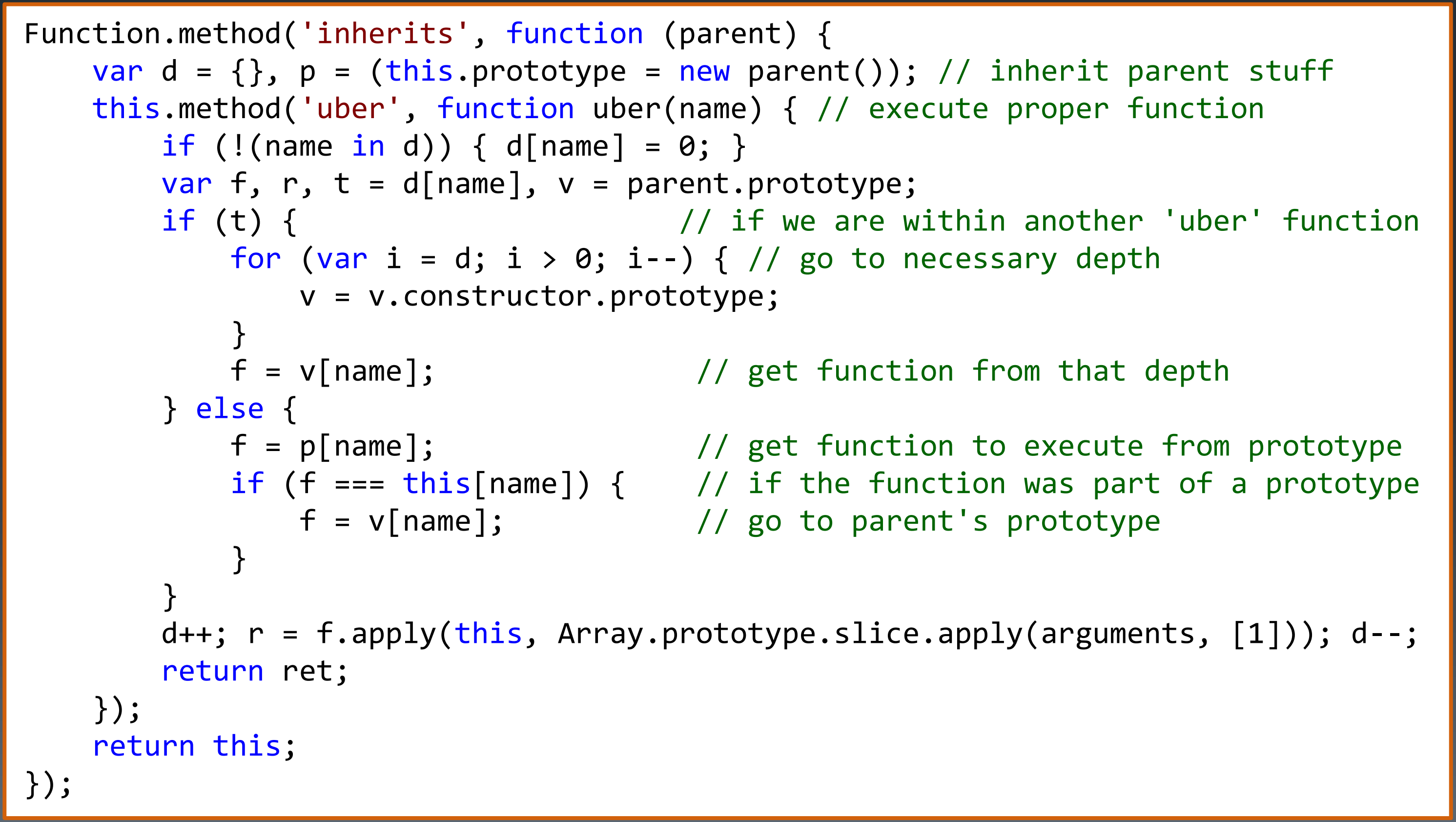
Object-Oriented JavaScript Classical Inheritance

- Based on a "classes" with methods that can be instantiated into object
- Allow the creation of objects that derive methods and still be able to call parent's object functions
- Designed by Douglas Crockford, commonly believed it's the best implementation of classical inheritance
- It starts with a small helper, `method` function attaches a function to the prototype of a constructor



```
// Helper to bind new function to prototype of an object
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

Object-Oriented JavaScript
Classical Inheritance



```
Function.method('inherits', function (parent) {
    var d = {}, p = (this.prototype = new parent()); // inherit parent stuff
    this.method('uber', function uber(name) { // execute proper function
        if (!(name in d)) { d[name] = 0; }
        var f, r, t = d[name], v = parent.prototype;
        if (t) {                      // if we are within another 'uber' function
            for (var i = d; i > 0; i--) { // go to necessary depth
                v = v.constructor.prototype;
            }
            f = v[name];               // get function from that depth
        } else {
            f = p[name];               // get function to execute from prototype
            if (f === this[name]) {    // if the function was part of a prototype
                f = v[name];           // go to parent's prototype
            }
        }
        d++; r = f.apply(this, Array.prototype.slice.apply(arguments, [1])); d--;
        return ret;
    });
    return this;
});
```

Object-Oriented JavaScript Classical Inheritance

- `inherits` is used to provide simple single-parent inheritance
- `swiss` is advanced version of method function that can grab multiple methods from a single parent object
- When `swiss` is used with multiple parent objects it acts as a form of functional, multiple inheritance

```
Function.method('swiss', function (parent) {
    for (var i = 1; i < arguments.length; i++) {
        var name = arguments[i];
        this.prototype[name] = parent.prototype[name];
    }
    return this;
});
```

Object-Oriented JavaScript Classical Inheritance

```
function Person(name) {                           // base object
    this.name = name;
}

Person.method('getName', function () {            // base method
    return this.name;
});

function User(name, age) {                        // new object
    this.name = name;
    this.age = age;
}

User.inherits(Person);                            // create derived object
User.method('getAge', function () {               // new member
    return this.age;
});

User.method('getName', function () {              // override function
    return 'My name is: ' + this.uber('getName'); // parent call still possible
});
```

Performance Improvements

- General Performance Tips
- Scope Management
- Object Caching
- Selecting DOM Nodes
- Modifying DOM Tree

Performance Improvements General Performance Tips

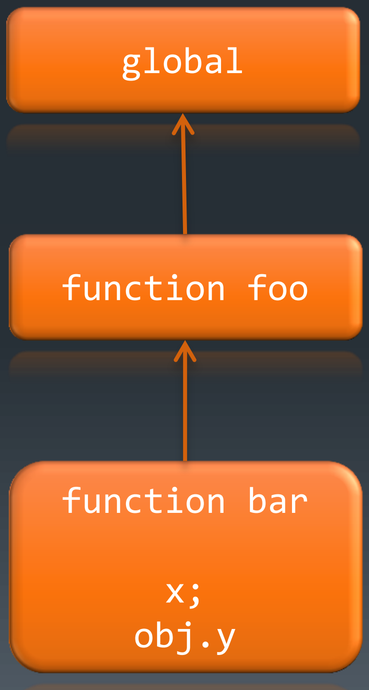
- Use fewer HTTP requests
- Use literals (`function`, `[]`, `{}` , `/regex/`) instead of constructors (`new Function()`, `new Array()`, `new Object()`, `new RegExp()`)
- Dont use `eval` or `with` (dangerous and slow)
- Avoid global variables
- Avoid `for-in` loop (JavaScript engine needs to build a list of all enumerable properties and check for duplicates prior to start)

Performance Improvements Scope Management

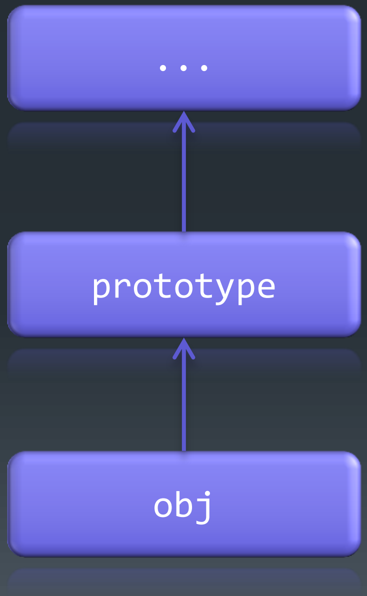
- Narrow the scope as much as possible
- All primitive or reference variables are located via their scope chain
- Once the variable is found if it's an object then it's properties are found via that object's prototype chain

Performance Improvements Scope Management

```
var obj = ...

function foo() {
    var x = 10;
    function bar() {
        alert(x);
        alert(obj.y);
    }
}
```



scope chain



prototype chain

Performance Improvements
Object Caching

- Cache every property or array item you use more then once
- Assigning variables is very fast, use it as much as possible

```
// wrong way
for (var i = 0; i < obj.prop.arr.length; i++) {
    console.log(obj.prop.array[i]);
}

// right way
var a = obj.prop.arr;
var len = a.length;
for (var i = 0; i < len; i++) {
    console.log(a[i]);
}
```

Performance Improvements Selecting DOM Nodes

- Getting element by ID is ALWAYS the fastest way
- Narrow search context as much as possible
- Try different selectors and use the one that is the fastest for your web application
- Find a compromise between speed and readiness of your code

Performance Improvements Selecting DOM Nodes

```
// How selectors are implemented
$('#i1');         // document.getElementById
$('div');         // document.getElementsByTagName
$('.c1');         // document.getElementsByClassName (searches whole DOM)
$('[value="v"]'); // has to be implemented in jQuery, very slow

// Faster selectors
$('#i1 div.c1');  // start from id (narrow scope)
$('div.c1');      // start from tag name
$('a[alt=""]');   // start from tag name

// Narrowing the scope
var c = $('#i1'); // search context, quickly found based on id
$('.c1', c);      // will be much faster then searching the whole tree
c.find('.c1');    // another way of narrowing the search scope
```

Performance Improvements Selectors Performance Test

- jQuery is one of the fastest JavaScript libraries available
- Testing id, class and attribute selectors
- DOM tree consisting of 5000 `div` elements, 5000 `span` elements, 5000 `p` elements and 5000 `small` elements
- Tested browsers: Chrome 5, FF 3.6, IE 8, Safari 4, Opera 10

`http://www.componenthouse.com/extra/jquery-analysis.html`

Performance Improvements
Selectors Performance Test

Operation	Chrome 5 (ms)	FF 3.6 (ms)	IE 8 (ms)	Safari 4 (ms)	Opera 10 (ms)
`$('#d-2642').html();`	1	12	4	2	0
`$('[id="d-2642"]').html();`	50	161	294	59	34

`$('small[id="d-2642"]')`
`.html();`
	9	13	70	10	6

Performance Improvements
Selectors Performance Test

Operation	Chrome 5 (ms)	FF 3.6 (ms)	IE 8 (ms)	Safari 4 (ms)	Opera 10 (ms)
`$('.p-4781').html();`	29	45	212	39	18
`$('p.p-4781').html();`	6	24	51	15	5

`$('p[class="p-4781"]')`
`.html();`
	14	11	69	9	6

`$('p').filter('.p-4781')`
`.html();`
	7	18	63	11	6

Performance Improvements
Selectors Performance Test

Operation	Chrome 5 (ms)	FF 3.6 (ms)	IE 8 (ms)	Safari 4 (ms)	Opera 10 (ms)
`$('[row="c-3221"]').html();`	94	208	284	104	75
`$('p[row="c-3221"]').html();`	25	58	68	28	14
`$('p').filter('[row="c-3221"]').html();`	25	59	76	25	14
`$('p[row]').filter('[row="c-3221"]').html();`	45	114	107	42	26

Performance Improvements
Modifying DOM Nodes

- Any changes in DOM tree are ALWAYS slow
- Perform as many operations as possible outside of DOM

```
var parent = document.getElementById('parentId');

for (var i = 0; i < 100; i++) {
    var item = '<div class="c' + i + '">' + i + '</div>';
    parent.innerHTML += item;      // 100 DOM tree updates
}
```

```
var parent = document.getElementById('parentId');
var items = [];

for (var i = 0; i < 100; i++) {
    var item = '<div class="c' + i + '">' + i + '</div>';
    items.push(item);
}

parent.innerHTML += items.join(''); // 1 DOM tree update
```

Performance Improvements
Modifying DOM Nodes

- Change CSS classes instead of inline styles

```
var elem = $('#item');

elem.css('display', 'block');
elem.css('color', 'red');
elem.css('border', '1px');
elem.width(100);
elem.height(100);
// 5 updates
```

```
var elem = $('#item');

elem.addClass('newClass');
// 1 update
```

Debugging and Testing

- Debugging
  - Consoles
  - DOM Inspectors
  - Profilers
- Testing
  - Unit Tests
  - Code Coverage

Debugging and Testing
Debugging

- Consoles
  - Most modern browsers have JavaScript consoles (IE8, Firefox, Chrome, Opera, Safari, ...)
  - `console.log();`
  - `debugger;`
- DOM Inspectors
  - Allows you to see current state of page after it was modified by scripts
  - Some inspectors allows content manipulation as well

Debugging and Testing
Debugging

- Profilers
  - Allows you to easily identify bottlenecks of your application
  - Analyse HTML, CSS, JavaScript, resources, compression, server configuration, caching, etc.
  - Often they offer ready-to-use solutions for some issues (YSlow, Google Chrome Audits,...)

Debugging and Testing
Unit Testing

- Test suites can save you lots of time
- Writing unit tests is easy and fast, objects are easy to mock
- There are many testing libraries to choose from
- Most test libraries work client side
- Examples
  - Can work server side: J3 Unit, DOH
  - Work server side only: Crosscheck, RhinoUnit
  - Most popular universal frameworks: JSUnit, YUI Test
  - Framework test harnesses: DOH (Dojo), UnitTesting (Prototype), QUnit (jQuery)

Debugging and Testing
Unit Testing

- Example unit tests in JSUnit

```
<script src="jsUnitCore.js"></script>
<script>
function testAssertTrue {
    var value = true;
    assertTrue('true should be true', value);
}

function testAssertEquals {
    var value = 1;
    assertEquals('value should equal 1', value, 1);
}

function testAssertNull {
    var value = null;
    assertNull('value should be null', value);
}
</script>
```

Debugging and Testing
Code Coverage

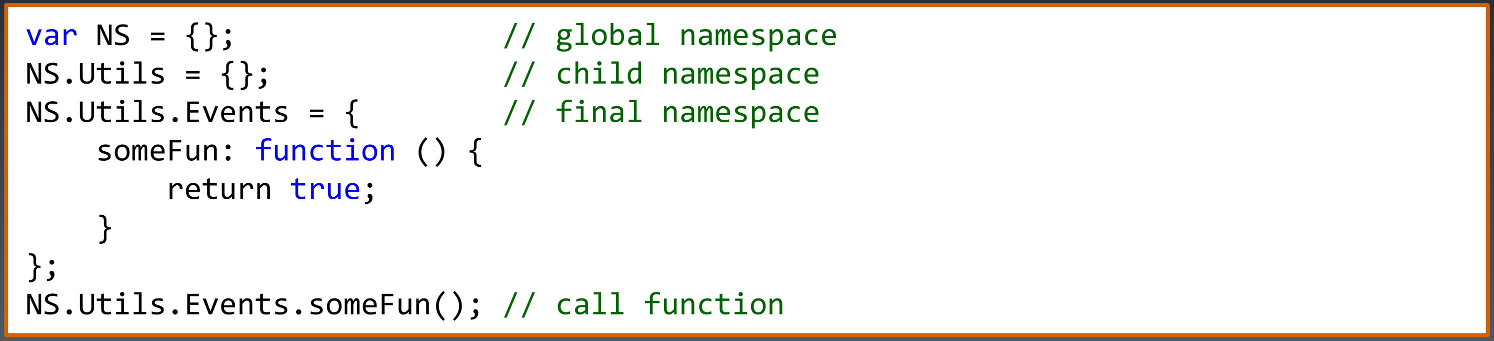
- Code Coverage Tools help you track which parts of your application hasn't been tested
- Some code coverage tools:
  - HRCov (FireFox plugin)
  - JSCoverade
  - Firebug Code Coverage (FireBug extension)
- Focus on writing the best tests for the behaviour not on the code coverage - 100% code coverage is rarely a good idea

Distribution

- Namespaces
- Code Cleanup
  - Variable Declaration
  - === and !==
  - Semicolons
  - JSLint
- Compression

Distribution Namespaces

- Namespaces prevents your code from conflicting with other code
- JavaScript doesn't support them natively but there's a simple workaround
- You can use so tools to help you out (Dojo, YUI)



```
var NS = {};               // global namespace
NS.Utils = {};             // child namespace
NS.Utils.Events = {        // final namespace
    someFun: function () {
        return true;
    }
};
NS.Utils.Events.someFun(); // call function
```

Distribution
Code Cleanup – Variables

ALWAYS declare variables before use

```
// Incorrect use
foo = 'bar';

// Correct use
var foo;
...
foo = 'bar';

var foo2 = 'baz';
```

Distribution
Code Cleanup – === and !==

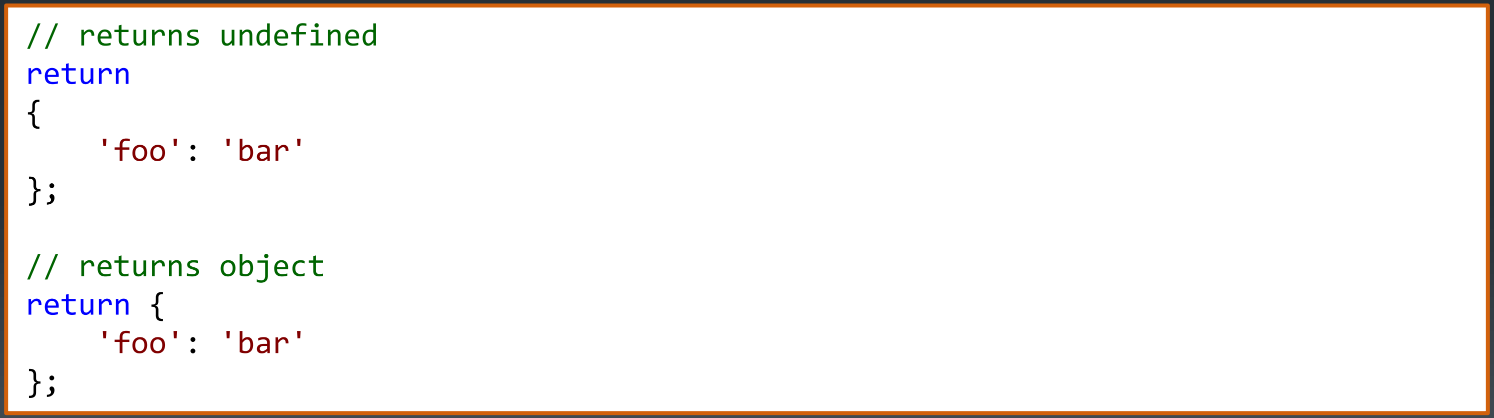
- ALWAYS use === and !==
- == and != do type coercion, which sometimes produces surprising results

```
console.log('' == 0);               // true
console.log(false == 0);            // true
console.log(' \r\n\t' == 0);        // true
console.log(null == undefined);     // true

console.log('' === 0);              // false
console.log(false === 0);           // false
console.log(' \r\n\t' === 0);       // false
console.log(null === undefined);    // false

// now the funny part
console.log(' \r\n\t' == 0);        // true
console.log(false == 0);            // true
// but
console.log(' \r\n\t' == false);    // false
```

Distribution
Code Cleanup – Semicolons

ALWAYS enter semicolons where explicetely

Lack of semicolons will break your code when you compress it



```
// returns undefined
return
{
    'foo': 'bar'
};

// returns object
return {
    'foo': 'bar'
};
```

Distribution
Code Cleanup – JSLint

- JSLink analyses JavaScript code looking for potential weakneses, like:
  - missing semicolons
  - `!=` and `==`
  - declaring variables before use
  - unsafe code detection (i.e. `eval`, `with`)
  - switch fall-through
  - assignments in conditions
- Remember/bookmark this link:

`http://jslint.com`

Distribution Compression

- Makes your app faster (less data to download)
- Different kinds of compressors
  - remove white characters and comments (JSMin)
  - as above + change variable names to be shorter
  - as above + change all words in code to be shorter (Packer)
- Check different compression tools (JSMin, YUI Compressor, Packer, ...)
- Make sure compressed code works the same as uncompressed (unit tests will be useful)
- Gzip your minified JavaScript files
- Compress production code only

Distribution Compression

File	File size (bytes)	Gzipped file size (bytes)
jQuery library	62 885	19 759
jQuery minified with JSMin	31 391	11 541
jQuery minified with Packer	21 557	11 119
jQuery minified with YUI Compressor	31 882	10 818

- jQuery library minification saves up to over 65% file size - it will download ~3 times faster then uncompressed file
- Gzipping minificated jQuery library saves up to over 82% file size - it will download ~5 times faster then uncompressed file

Sources

- John Resig "Pro JavaScript"
- Douglas Crockford "JavaScript: The Good Parts"
- http://www.crockford.com/javascript/
- Alberto Savoia "The Way of Testivus"
  http://www.agitar.com/downloads/TheWayOfTestivus.pdf
- Alberto Savoia "Testivus of Test Coverage"
  http://googletesting.blogspot.com/2010/07/code-coverage-goal-80-and-no-less.html

Questions?