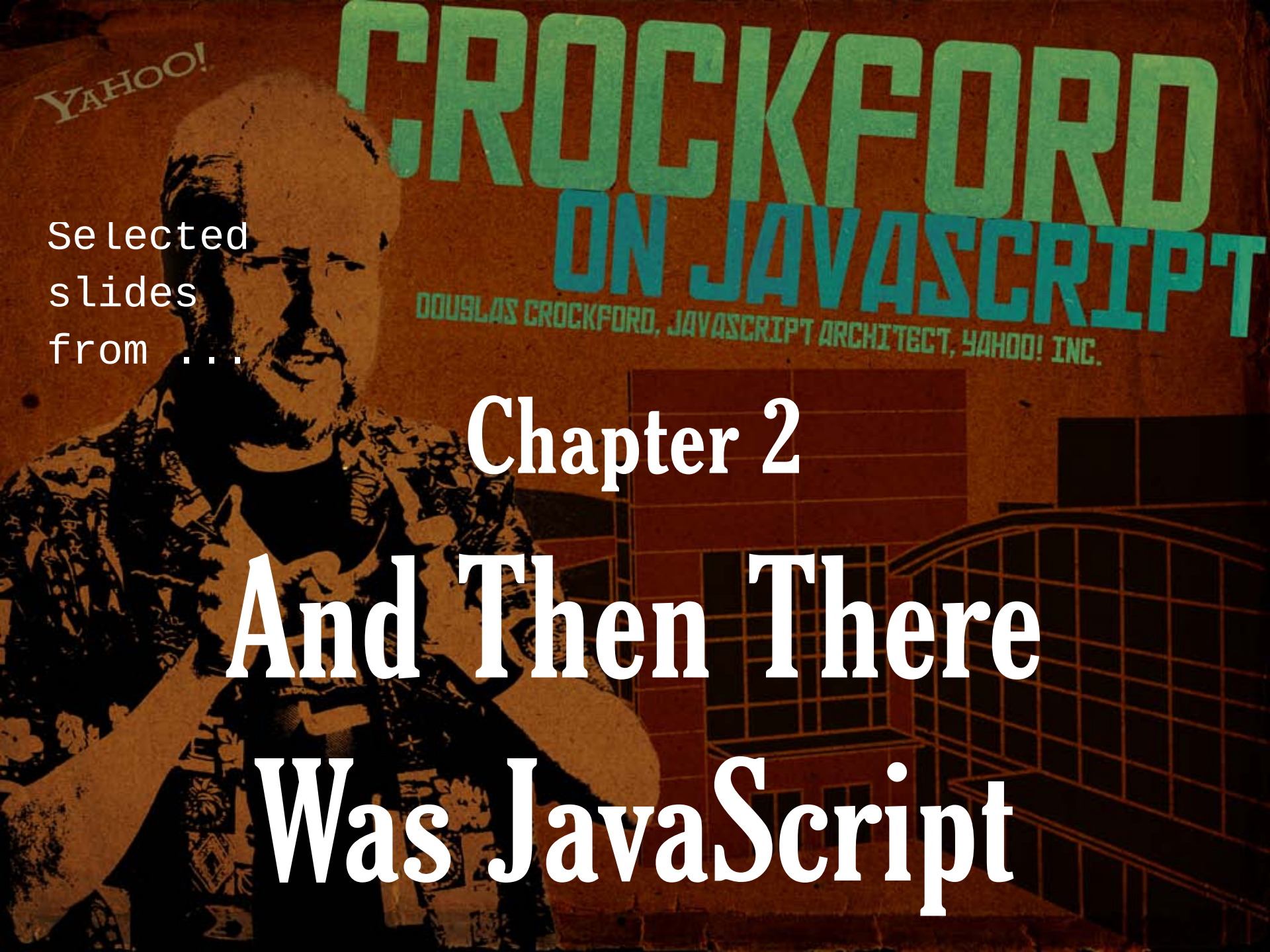


Selected slides from ...

Chapter 2

And Then There Was JavaScript

JavaScript has good parts.

... we'll get to them later.

Where do Bad Parts come from?

- Legacy
- Good Intentions
- Haste

- For the most part, the bad parts can be avoided.
- The problem with the bad parts isn't that they are useless.

Numbers

- Only one number type
  - No integer types
- 64-bit floating point
- IEEE-754 (aka “`Double`”)

Associative Law does not hold

```
(a + b) + c === a + (b + c)
```

- Produces `false` for some values of `a`, `b`, `c`.

- Integers under 9007199254740992 (9 quadrillion) are ok.

```
9007199254740992 ===
  9007199254740992 + 1
```

```
(a + 1) - 1 === a
```

Can be false.

Decimal fractions are approximate

```
a = 0.1;
b = 0.2;
c = 0.3;

(a + b) + c === a + (b + c)
false
```

Math object

- abs
- acos
- asin
- atan
- atan2
- ceil
- cos
- exp
- floor
- log
- max
- min
- pow
- random
- round
- sin
- sqrt
- tan

Math object

- E
- LN10
- LN2
- LOG10E
- LOG2E
- PI
- SQRT1_2
- SQRT2

```
function log2(x) {
    return Math.LOG2E *
        Math.log(x);
}
```

`NaN`

- Special number: Not a Number
- Result of undefined or erroneous operations
- Toxic: any arithmetic operation with `NaN` as an input will have `NaN` as a result
- `NaN` is not equal to anything, including `NaN`
- `NaN === NaN` is `false`
- `NaN !== NaN` is `true`

String

Strings

- A sequence of 0 or more 16-bit Unicode characters
  - UCS-2, not quite UTF-16
  - No awareness of surrogate pairs
- No separate character type
  - Characters are represented as strings with length of 1
- Strings are immutable
- Similar strings are equal (===)
- String literals can use single or double quotes with \ escapement.
- Use " for external strings.
- Use ' for internal strings and characters.

+

+ can concatenate or add.

```
'$' + '1' + '2' === '$12'
```

```
'$'.concat('1').concat('2')
```

Convert a number to a string

- Use number method (toString)
- Use String function

```
str = num.toString();
str = String(num);
```

Convert a string to a number

- Use the Number function.
- Use the + prefix operator.
- Use the parseInt function.

```
num = Number(str);
num = +str;
```

parseInt function

```
parseInt(str, 10)
```

- Converts the value into a number.
- It stops at the first non-digit character.

```
parseInt("12em") === 12
```

- The radix (10) should always be used.

```
parseInt("08") === 0
parseInt("08", 10) === 8
```

String `length`

- `string.length`

- The `length` property determines the number of 16-bit characters in a string.
- Extended characters are counted as 2.

Array

Arrays

- `Array` inherits from `Object`.
- Indexes are converted to strings and used as names for retrieving values.
- Very efficient for sparse arrays.
- Not very efficient in most other cases.
- One advantage: No need to provide a length or type when creating an array.

length

- Arrays, unlike objects, have a special `length` property.
- It is always 1 larger than the highest integer subscript.
- It allows use of the traditional `for` statement.

```
for (i = 0; i < a.length; i += 1) {
    …
}
```

- Do not use `for in` with arrays

Array Literals

- An array literal uses `[]`
- It can contain any number of expressions, separated by commas

```
myList = ['oats', 'peas', 'beans'];
```

- New items can be appended

```
myList[myList.length] = 'barley';
```

- The dot notation should not be used with arrays.

Array methods

- concat
- every
- filter
- forEach
- indexOf
- join
- lastIndexOf
- map
- pop
- push
- reduce
- reduceRight
- reverse
- shift
- slice
- some
- splice
- toLocaleString
- toString
- unshift

sort

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// n is [15, 16, 23, 4, 42, 8]
```

default sort is alphabetical.
beware. for numerically ascending:

```
var points = [40,100,1,5,25,10];
points.sort(function(a,b){return a-b});
```

Deleting Elements

```
delete array[number]
```

- Removes the element, but leaves a hole in the numbering.

```
array.splice(number, 1)
```

- Removes the element and renumbers all the following elements.

Deleting Elements

```
myArray = ['a', 'b', 'c', 'd'];

delete myArray[1];

// ['a', undefined, 'c', 'd']

myArray.splice(1, 1);

// ['a', 'c', 'd']
```

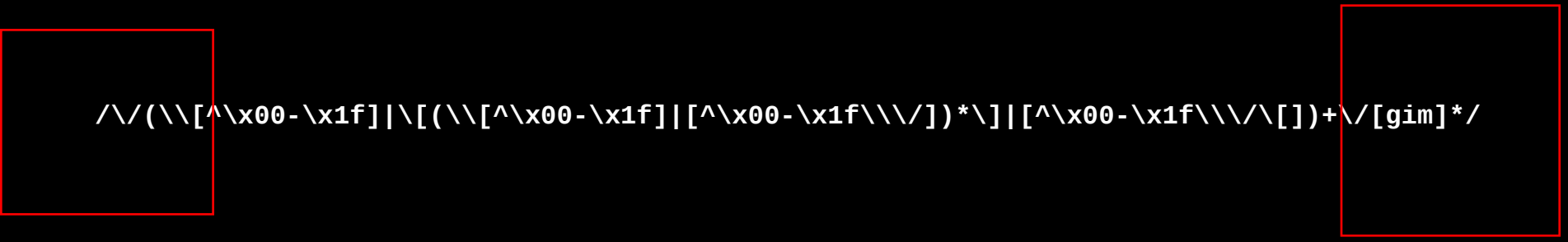
Arrays v Objects

- Use objects when the names are arbitrary strings.
- Use arrays when the names are sequential integers.
- Don't get confused by the term Associative Array.

forward slashes on either end!

weird - not a string - a different type.

RegExp



Falsy values

- `false`
- `null`
- `undefined`
- `""` (empty string)
- `0`
- `NaN`

- All other values (including all objects) are truthy.

`"0"` `"false"`

== !=

- Equal and not equal
- These operators can do type coercion
- It is always better to use === and !==, which do not do type coercion.

Evils of type coercion

- `'' == '0'` // false
- `0 == ''` // true
- `0 == '0'` // true

- `false == 'false'` // false
- `false == '0'` // true

- `false == undefined` // false
- `false == null` // false
- `null == undefined` // true

- `' \t\r\n ' == 0` // true

Function

All values are objects

Except `null` and `undefined`.

`null`

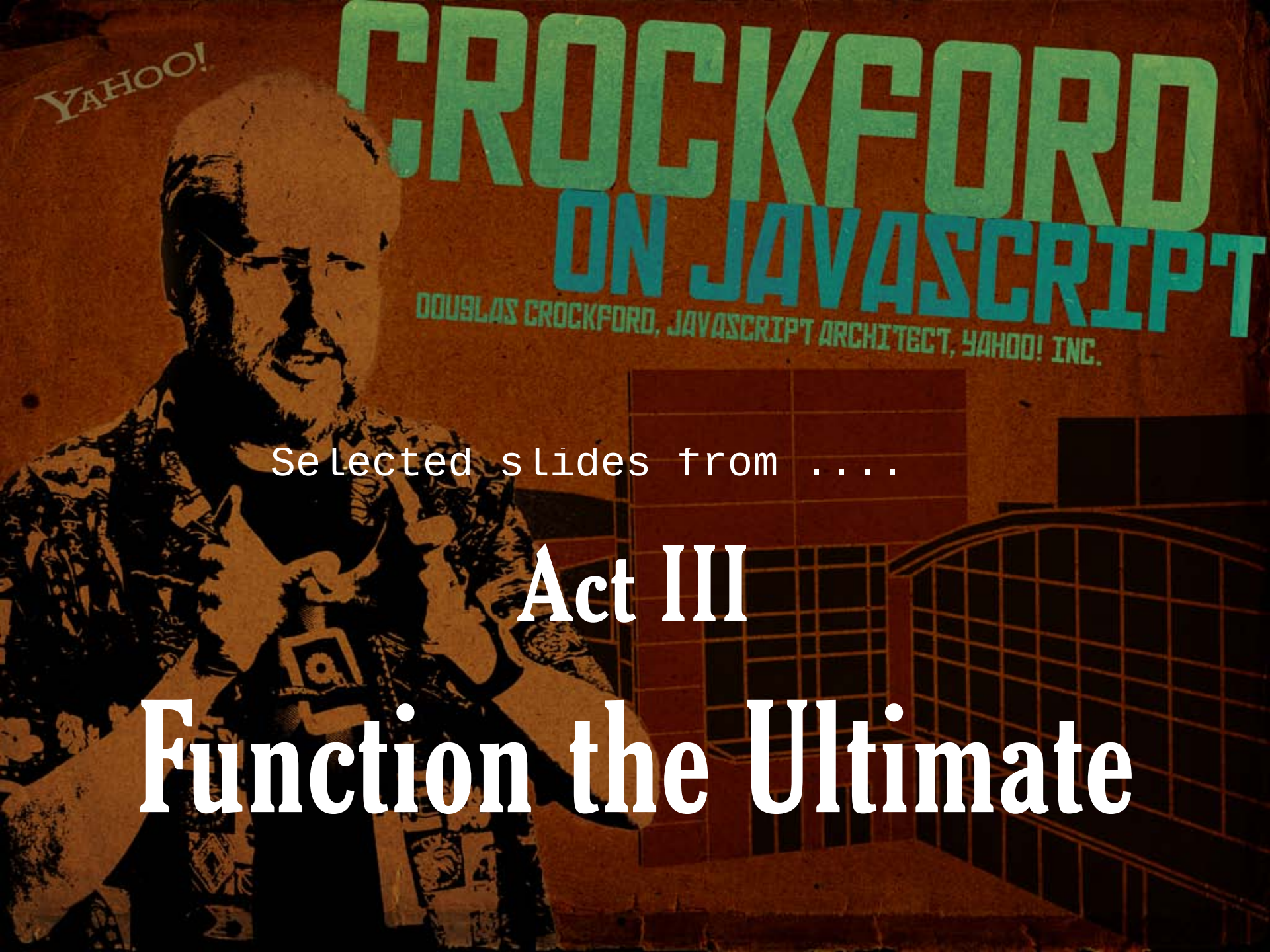
A value that isn't anything

`undefined`

- A value that isn't even that.
- The default value for variables and parameters.
- The value of missing members in objects.

Switch statement

```
switch (expression) {
case ';':
case ',':
case '.':
    punctuation();
    break;
default:
    noneOfTheAbove();
}
```



Selected slides from ....

Act III

Function the Ultimate

`function` expression

- `function`
- optional name
- parameters
  - Wrapped in parens
  - Zero or more names
  - Separated by `,` (comma)
- body
  - Wrapped in curly braces
  - Zero or more statements

```
var addOne = function(x)
{
   return x+1;
};
```

`function` expression

- Produces an instance of a function object.
- Function objects are first class.
  - May be passed as an argument to a function
  - May be returned from a function
  - May assigned to a variable
  - May be stored in an object or array
- Function objects inherit from `Function.prototype`.

function statement

- `function`
- mandatory name
- parameters
  - Wrapped in parens
  - Zero or more names
  - Separated by `,` (comma)
- body
  - Wrapped in curly braces
  - Zero or more statements

```
function addOne (x) {
   return x+1;
}
```

`function` statement

- The `function` statement is just a short-hand for a `var` statement with a function value.

```
function foo() {}
```

expands to

```
var foo = function foo() {};
```

expands to

```
var foo = undefined;
foo = function foo() {};
```

The assignment of the function is also hoisted.

```
// function statement
function addOne(x) {
   return x+1;
}
```

```
// function expression
var addOne =
function(x) {
   return x+1; }
```

`function` expression

v

`function` statement

If the first token in a statement is `function`, then it is a function statement.

`var` statement

- Declares and initializes variables within a function.
- Types are not specified.
- A variable declared anywhere within a function is visible everywhere within the function.

`var` statement

- It gets split into two parts:
  - The declaration part gets hoisted to the top of the function, initializing with `undefined`.
  - The initialization part turns into an ordinary assignment.

```
var myVar = 0, myOtherVar;
```

- Expands into

```
var myVar = undefined,
    myOtherVar = undefined;
    …
myVar = 0;
```

Scope

Block scope v function scope

Scope

- In JavaScript, {blocks} do not have scope.
- Only functions have scope.
- Variables defined in a function are not visible outside of the function.

```
function assure_positive(matrix, n) {
    for (var i = 0; i < n; i += 1) {
        var row = matrix[i];
        for (var i = 0; i < row.length;
                i += 1) {
            if (row[i] < 0) {
                throw new Error('Negative');
}   }   }   }
```

Declare all variables at the top of the function.

Declare all functions before you call them.

The language provides mechanisms that allow you to ignore this advice, but they are problematic.

Return statement

```
return expression;
```

or

```
return;
```

- If there is no *expression*, then the return value is `undefined`.
- Except for constructors, whose default return value is `this`.

`this`

- The `this` parameter contains a reference to the object of invocation.
- `this` allows a method to know what object it is concerned with.
- `this` allows a single function object to service many functions.
- `this` is key to prototypal inheritance.

Invocation

- The ( ) suffix operator surrounding zero or more comma separated arguments.
- The arguments will be bound to parameters.

Invocation

- If a function is called with too many arguments, the extra arguments are ignored.
- If a function is called with too few arguments, the missing values will be `undefined`.
- There is no implicit type checking on the arguments.

Invocation

- There are four ways to call a function:
  - Function form
    - *functionObject*(*arguments*)
  - Method form
    - *thisObject*.*methodName*(*arguments*)
    - *thisObject*["*methodName*"](*arguments*)
  - Constructor form
    - new *FunctionObject*(*arguments*)
  - Apply form
    - *functionObject*.apply(*thisObject*, [*arguments*])

Method form

thisObject.*methodName*(*arguments*)
thisObject[*methodName*](*arguments*)

- When a function is called in the method form, `this` is set to *thisObject*, the object containing the function.
- This allows methods to have a reference to the object of interest.

Function form

functionObject(*arguments*)

- When a function is called in the function form, `this` is set to the global object.
  - That is not very useful. (Fixed in ES5/Strict)
  - An inner function does not get access to the outer `this`.

```
var that = this;
```

Constructor form

`new` *FunctionValue*(*arguments*)

- When a function is called with the `new` operator, a new object is created and assigned to `this`.
- If there is not an explicit return value, then `this` will be returned.
- Used in the Pseudoclassical style.

this

- `this` is an bonus parameter. Its value depends on the calling form.
- `this` gives methods access to their objects.
- `this` is bound at invocation time.

Invocation form	`this`
function	the global object `undefined`
method	the object
constructor	the new object
apply	argument

Closure

Lexical Scoping
Static Scoping

Closure

- The context of an inner function includes the scope of the outer function.
- An inner function enjoys that context even after the parent functions have returned.

Global

```
var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

var digit_name = function (n) {
    return names[n];
};



alert(digit_name(3));    // 'three'
```

Slow

```
var digit_name = function (n) {
    var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];


    return names[n];
};



alert(digit_name(3));    // 'three'
```

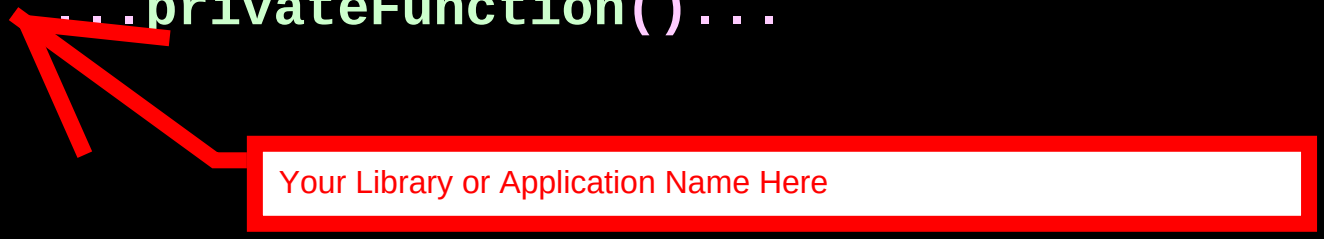
Closure

```
var digit_name = (function () {
    var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

    return function (n) {
        return names[n];
    };

}());

alert(digit_name(3));    // 'three'
```

```
function fade(id) {
    var dom = document.getElementById(id),
        level = 1;
    function step() {
        var h = level.toString(16);
        dom.style.backgroundColor =
            '#FFFF' + h + h;
        if (level < 15) {
            level += 1;
            setTimeout(step, 100);
        }
    }
    setTimeout(step, 100);
}
```

A Module Pattern

```
(function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }
    GLOBAL.firstMethod = function (a, b) {
        ...privateVariable...
    };
    GLOBAL.secondMethod = function (c) {
        ...privateFunction()...
    };
}());
```

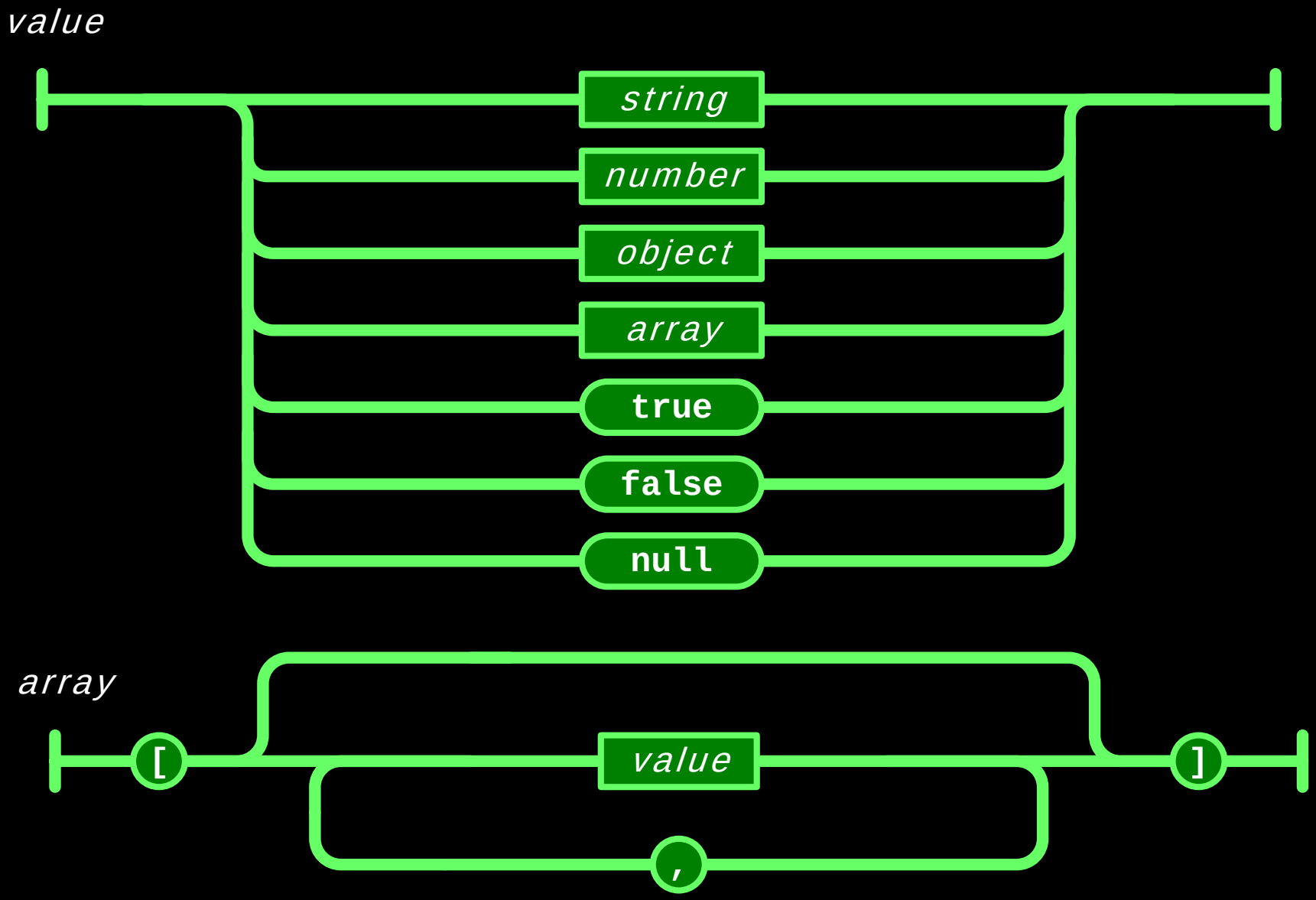


Object literals

- An expressive notation for creating objects.

```
var my_object = {foo: bar};

var my_object = Object.defineProperties(
        Object.create(Object.prototype), {
    foo: {
        value: bar,
        writeable: true,
        enumerable: true,
        configurable: true
    }
});
```

JavaScript Object Notation (JSON)



Functional Inheritance

```
function gizmo(id) {
    return {
        id: id,
        toString: function () {
            return "gizmo " + this.id;
        }
    };
}

function hoozit(id) {
    var that = gizmo(id);
    that.test = function (testid) {
        return testid === this.id;
    };
    return that;
}
```

hoozit extends gizmo here.

Privacy

```
function gizmo(id) {
    return {
        toString: function () {
            return "gizmo " + id;
        }
    };
}


function hoozit(id) {
    var that = gizmo(id);
    that.test = function (testid) {
        return testid === id;
    };
    return that;
}
```

Don’t make functions in a loop.

- It can be wasteful because a new function object is created on every iteration.
- It can be confusing because the new function closes over the loop’s variables, not over their current values.

Creating event handlers in a loop

```
for (var i ...) {
    div_id = divs[i].id;
    divs[i].onclick = function () {
        alert(div_id);
    };
}
```

<- BAD

```
var i;
function make_handler(div_id) {
    return function () {
        alert(div_id);
    }
}
for (i ...) {
    div_id = divs[i].id;
    divs[i].onclick = make_handler(div_id);
}
```

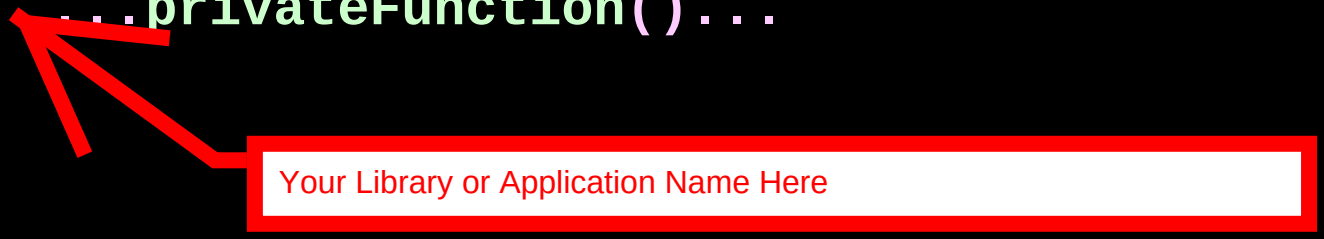
<- Better

Module pattern is easily transformed into a powerful constructor pattern.

And now (if we have time), a separate note about how to create objects with 'constructors'

A Module Pattern

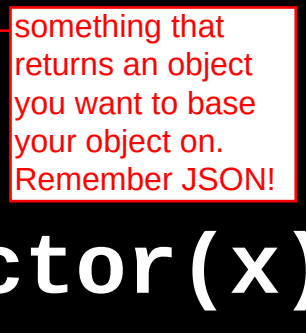
```
(function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }
    GLOBAL.firstMethod = function (a, b) {
        ...privateVariable...
    };
    GLOBAL.secondMethod = function (c) {
        ...privateFunction()...
    };
}());
```



Power Constructors

1. Make an object.
   - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
   - These become private members.
3. Augment the object with privileged methods.
4. Return the object.

Step Four




```
function myPowerConstructor(x) {
    var that = otherMaker(x);
    var secret = f(x);
    that.priv = function () {
        ... secret x that ...
    };
    return that;
}
```