Boot Camp JavaScript

http://rix0r.nl/bootcamp Sioux, March 31, 2011

SIOUX

BOOT camp

Agenda

Part 1: JavaScript the Language

Short break

Part 2: JavaScript in the Browser

History

May 1995	LiveScript is written by Brendan Eich for NN2
Dec 1995	Renamed to “JavaScript” in a deal with Sun
Aug 1996	Microsoft copies JScript for IE 3.0
Nov 1996	Netscape goes shopping for a standard Ecma – ECMAScript
Dec 1998	dynamicdrive.com
Mar 1999	OWA team invents Xml.HTTP (IE 5.0)
Dec 2000	Firefox copies XmlHttpRequest
2004	GMail
Feb 2005	The term “AJAX” is invented
2005	Google Maps
Apr 2006	XmlHttpRequest standardized

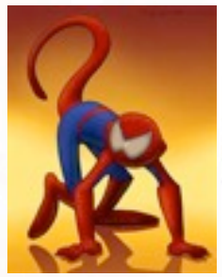
JavaScript renaissance

Where?

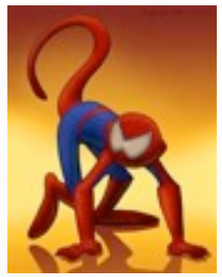
Where?

Where?

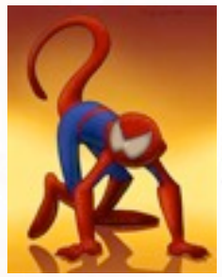
Where?



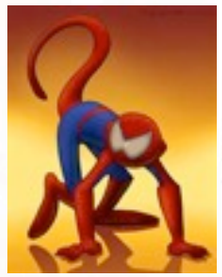
Where?



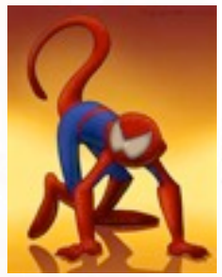
Where?



Where?



Where?

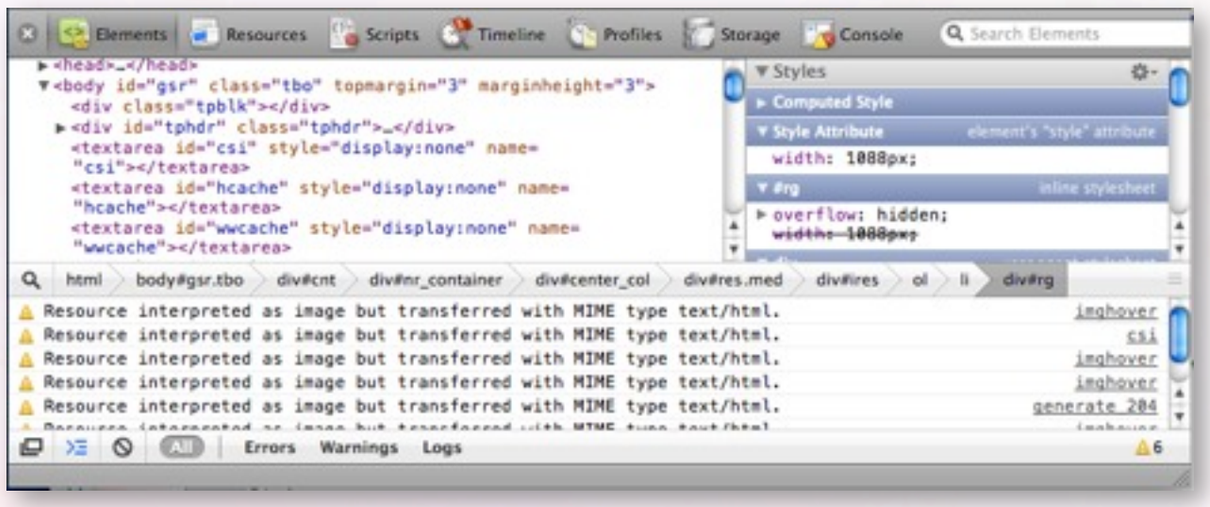


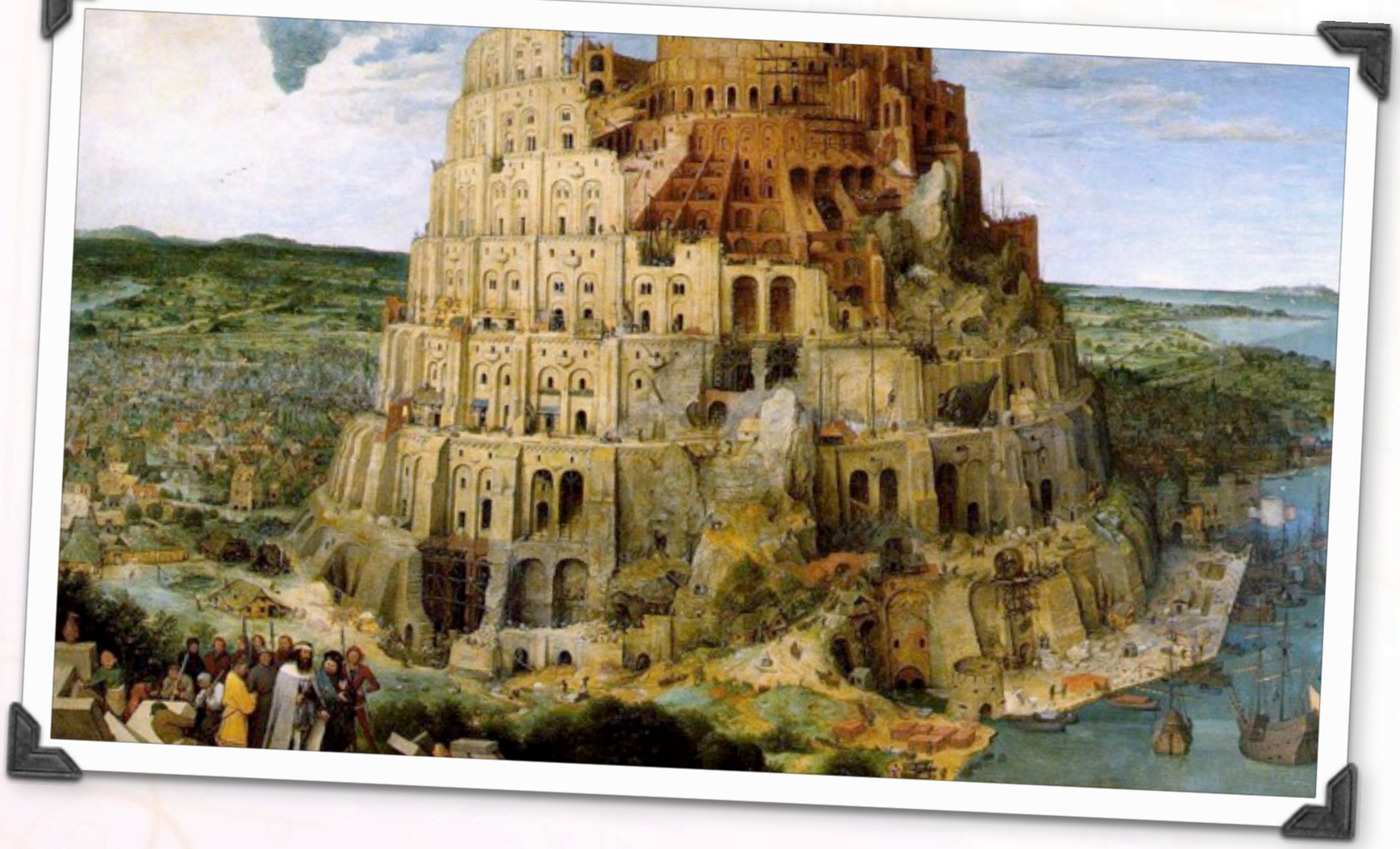
nodeJS

Tooling support

Firebug
(Firefox)

WebKit Console
(Chrome, Safari)





Part I

JavasScript the Language

Ancestry

Functions from Scheme

Prototypes from Self

Syntax from Java

Event-driven model from HyperCard

Types

String	`"hello", 'world'`	Object	`{ hello: 'world' }`
Number	`1, 3.14, 2e11, NaN`	Array	`[1, 2, 3]`
Boolean	`true, false`	Function	`function(x) { return x*x; }`
null	`null`		
undefined	`undefined`		

A little word about functions

Functions are first-class!

```
function greet() {
    alert('Hi there!');
}
```

Is the same as:

```
var greet = function() {
    alert('Hi there!');
}
```

Semicolon insertion

```
function foo(x)
{
    var y = x*2
    return
    {
        x: x,
        y: y
    }
}

alert(foo(3)) // ?
```

Semicolon insertion

```
function foo(x)
{
    var y = x*2
    return
    {
        x: x,
        y: y
    }
}

alert(foo(3)) // ?
```

ParseError!

Semicolon insertion

```
function foo(x)
{
    var y = x*2
    return;
    {
        x: x,
        y: y
    }
}

alert(foo(3)) // ?
```

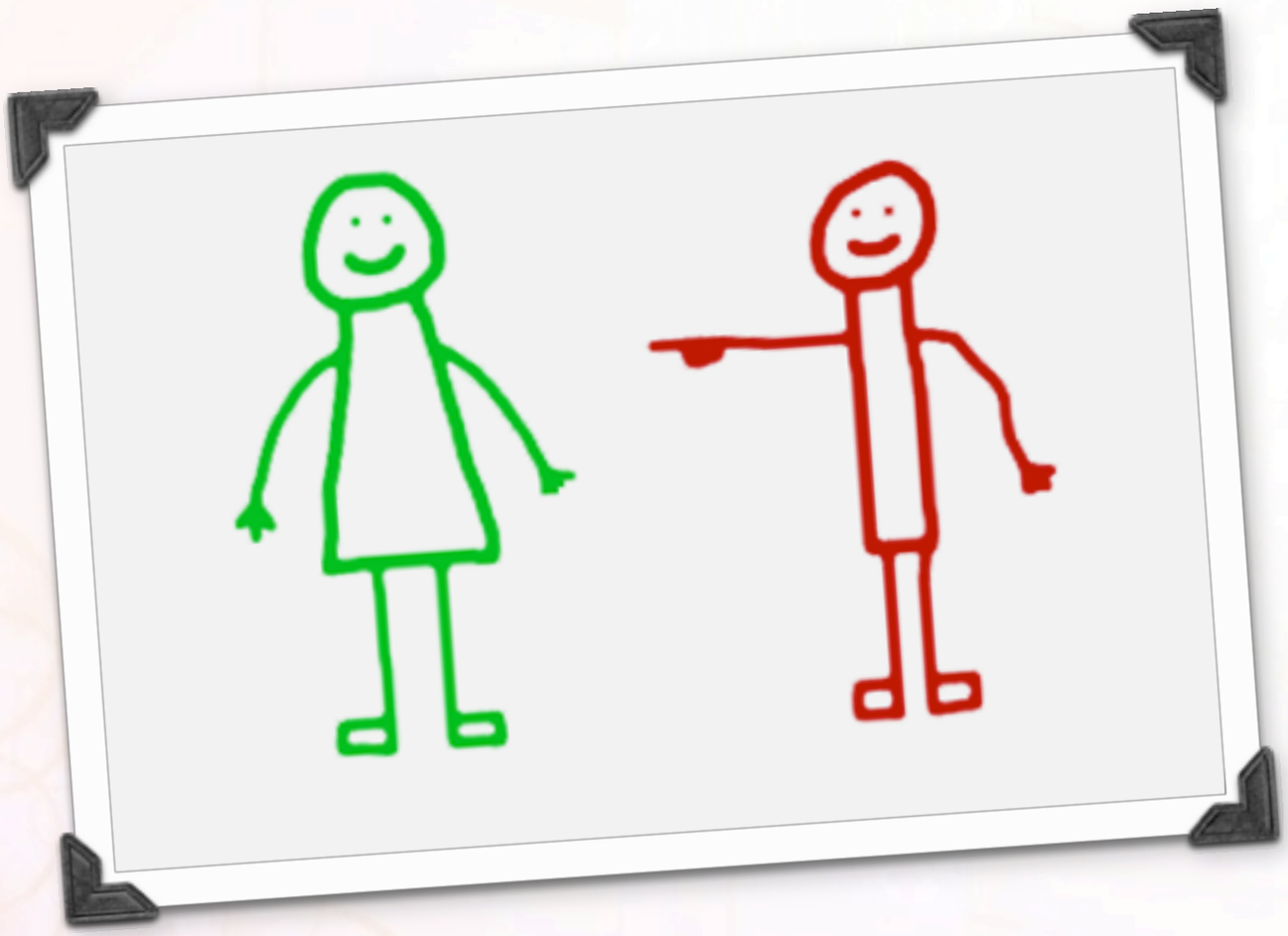
ParseError!

Semicolon insertion

```
function foo(x) {
    var y = x*2
    return {
        x: x,
        y: y
    }
}

alert(foo(3)) // { x: 3, y: 6 }
```

Use **K&R** style braces!



Objects & Prototyping

Objects

Object is a property bag (dictionary):

```
world['greeting'] = 'hello';
world.greeting = 'hello';
world = { greeting: 'hello' };
```

Methods are simply values that happen to be callable:

```
world.greet = function(hello) {
    alert(hello + ' world!');
};
world.greet('Cheerio');
```

Objects are open

All object members can always be replaced.

How does a method know on which object it is invoked? (Answer: `this`)

```
earth.age = 4e9;
earth.birthday = function() {
    this.age++;
};

earth.birthday(); // 4000000001
```

Exercise 1

Add a sum() method to an array

http://rix0r.nl/bootcamp

Hints:

Use `this` to refer to the current object.

Array iteration:

```
for (var i = 0; i < array.length; i++) {
    alert(array[i]);
}
```

“this” is dynamically bound :)

Same `birthday`, different object.

```
dog.age = 2;
dog.birthday = earth.birthday;

dog.birthday();
alert(dog.age); // 3
```

Bound at moment of invocation.

“this” is dynamically bound :(

Only if you call `object.method()` exactly:

```
dog.age = 3;
dog_birthday = dog.birthday;

dog_birthday();
alert(dog.age); // ?

alert(age); // ?
```

“this” is dynamically bound :(

Only if you call `object.method()` exactly:

```
dog.age = 3;
dog_birthday = dog.birthday;

dog_birthday();
alert(dog.age); // ?
```

3

```
alert(age); // ?
```

“this” is dynamically bound :(

Only if you call `object.method()` exactly:

```
dog.age = 3;
dog_birthday = dog.birthday;

dog_birthday();
alert(dog.age); // ?
```

3

```
alert(age); // ?
```

NaN (“undefined + 1”)
(Should have been ReferenceError)

“this” is dynamically bound :(

Only if you call `object.method()` exactly:

```
dog.age = 3;
dog_birthday = dog.birthday;

dog_birthday();
alert(dog.age); // ?
```

3

```
alert(age); // ?
```

NaN (“undefined + 1”)
(Should have been ReferenceError)

We’ll see a solution later on!

JavaScript Object Pattern

Invoke a function with new, and it gets called on a new object (as the constructor):

```
function Dog(name) {
    this.age  = 0;
    this.name = name;
}

var pooch = new Dog('pooch');
alert(pooch.age); // 0
```

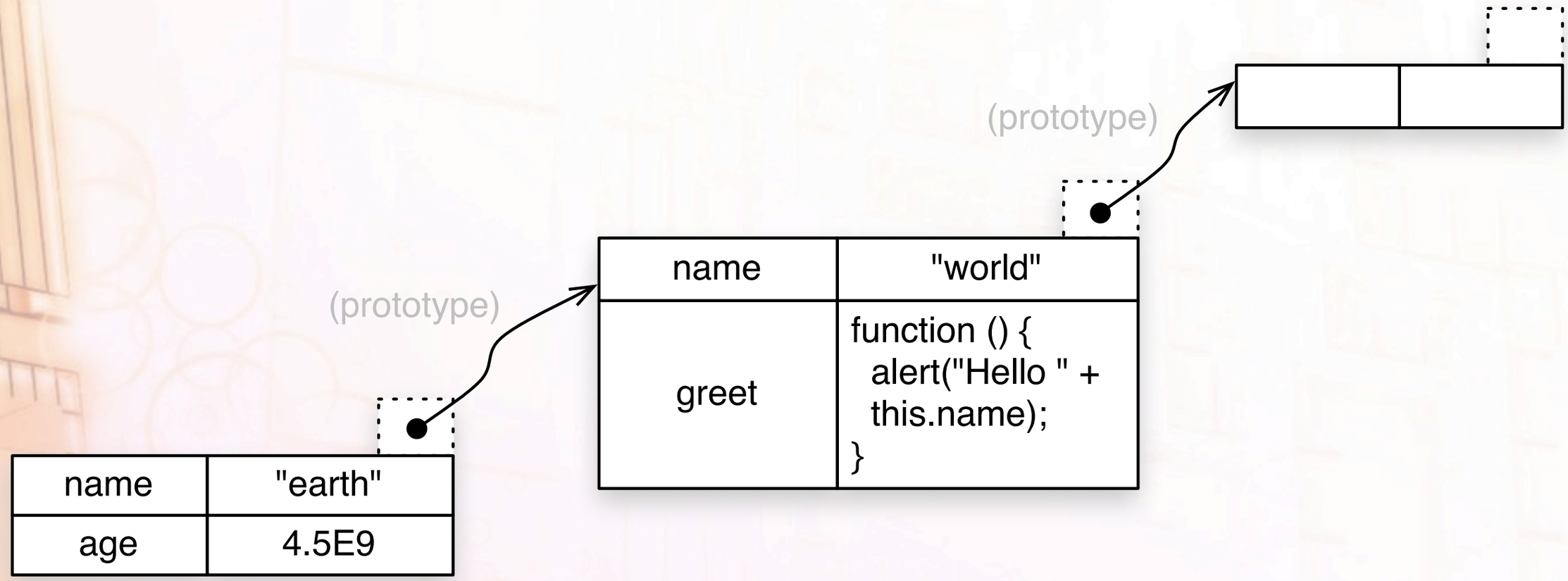
Prototyping

Mechanism for code reuse.

No classes.

(But you can emulate classes if you want to)

Prototype chain




```
earth.greet(); // "Hello earth"
```

Setting the prototype

`prototype` attribute of constructor function:

```
World.prototype.greet = /* ... */;
Earth.prototype = new World();
var earth = new Earth();
earth.greet();
```

The prototype of the object is set to whatever the `prototype` attribute of the `Earth` function points to.

Exercise 2

The Counter object

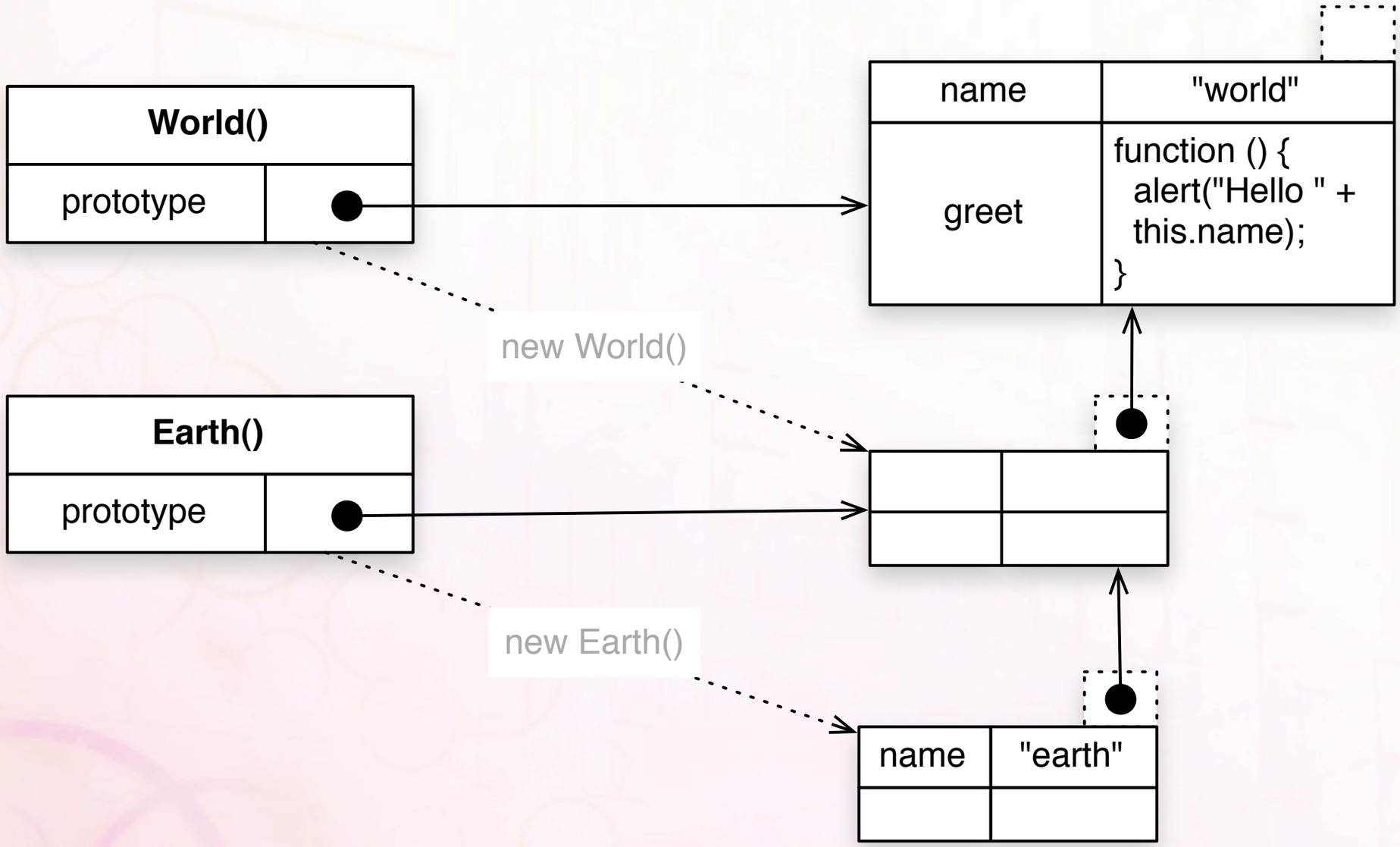
Write a constructor for an object that will return sequential numbers.

Hints:

Initialize the object in the constructor.

Use the prototype to share methods between objects.

Setting the prototype, the old way



Setting the prototype, the old way

Useless extra object in the inheritance chain.

If you forget `new`, it's a regular function call and won't do what you want!

Assignments to `Function.prototype.whatever` are quite ugly.

Setting the prototype, the new way

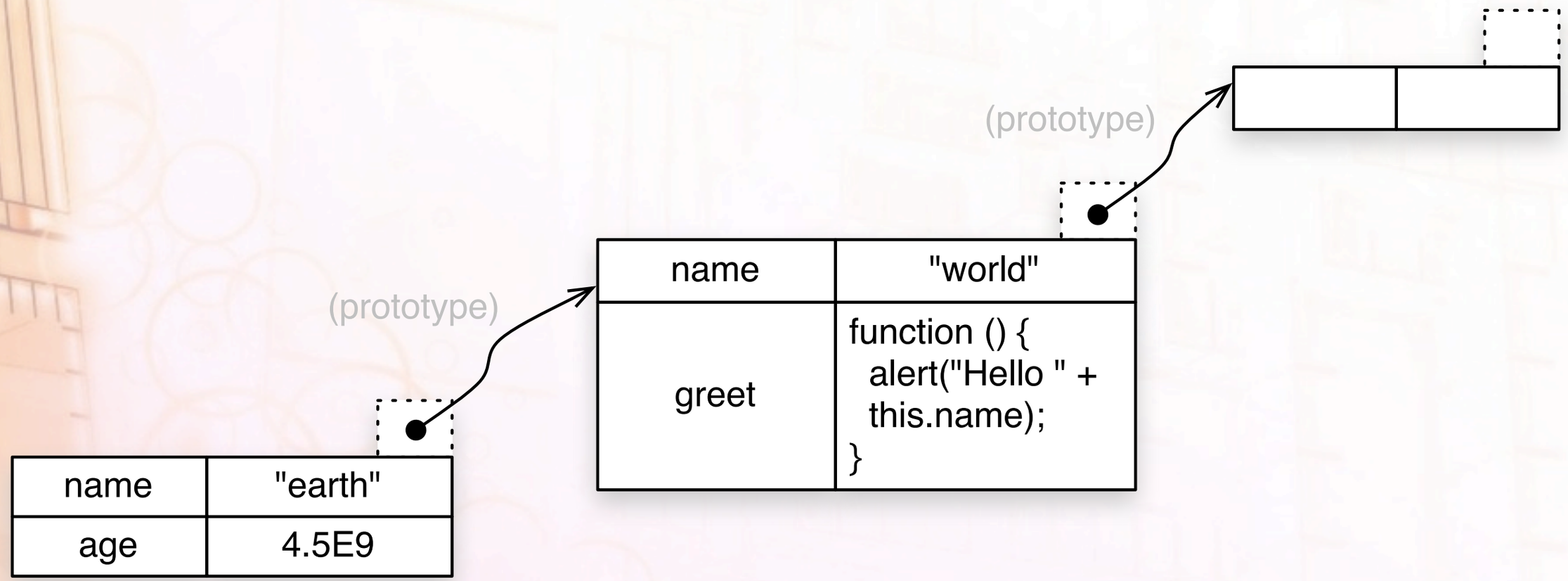
Object.create(proto, properties):

```
var world = {
    name: 'world',
    greet: /* ... */
};

var earth = Object.create(world, {
    name: 'earth'
});

earth.greet(); // ‘Hello earth’
```

Setting the prototype, the new way



Prototypes are open

Prototypes are just objects, and hence open as well!

Which object is the prototype cannot be changed after instantiation, but the members of the prototype can!

Exercise 3

Monkey patching

Adding useful methods to built-ins like String or Array.

(In this case, add the sum() method from exercise 1 to all arrays)

Used a lot by JavaScript libraries to paper over browser incompatibilities!

(For example, `Array.forEach()`...)

Advantages of prototyping

Lightweight and flexible

Some patterns are much easier with prototypes (factory, prototype(*), decorator, ...)

Prototypes can emulate classes but not the other way around!

(*) Well duh!

Functions & Closures

Functions

Functions are first-class

Functions are functions

Functions are methods

Functions are constructors

Functions define scope

Scope

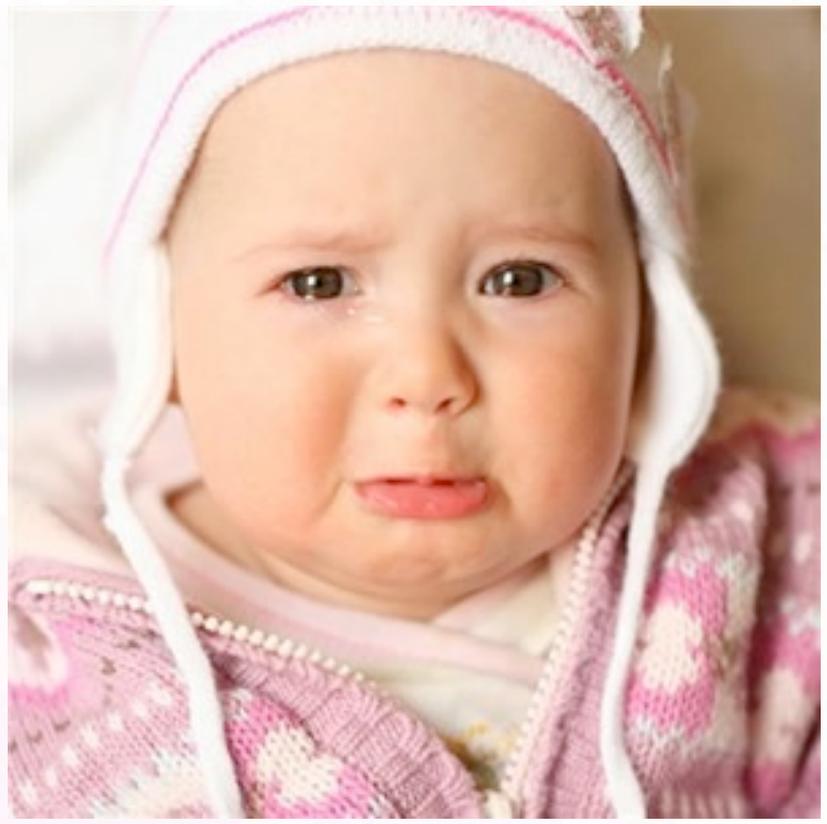
By default, everything goes in the “global object”.

No modules!

Undeclared variables go in global scope!

Only function scope exists.

“this” is dynamically bound.



Undeclared variables go in global scope

```
function foo() {
    a = 1;
}

function bar() {
    alert(a);
}

foo();
bar(); // ?
```

Undeclared variables go in global scope

```
function foo() {
    a = 1;
}

function bar() {
    alert(a);
}

foo();
bar(); // ?
```

1

Undeclared variables go in global scope

```
function foo() {
    var a = 1;
}

function bar() {
    alert(a);
}

foo();
bar(); // ?
```

Undeclared variables go in global scope

```
function foo() {
    var a = 1;
}

function bar() {
    alert(a);
}

foo();
bar(); // ?
```

ReferenceErrror

Undeclared variables go in global scope

```
function foo() {
    var a = 1;
}

function bar() {
    alert(a);
}

foo();
bar(); // ?
```

ReferenceErrror

Always use var!

Function scope

```
var a = 1;

function foo() {
    alert(a);   // ?
    var a = 2;
    alert(a);   // ?
}

foo();
```

Function scope

```
var a = 1;

function foo() {
    alert(a);   // ?
    var a = 2;
    alert(a);   // ?
}

foo();
```

2

Function scope

```
var a = 1;

function foo() {
    alert(a);   // ?
    var a = 2;
    alert(a);   // ?
}

foo();
```

ReferenceErrror

2

Closures

Functions retain references to the environment of their definition

```
var make_greeter = function(greeting) {
    return function(who) {
        alert(greeting + ' ' + who);
    }
};

var briton = make_greeter('Cheerio');
briton('mate');      // Cheerio mate
briton('old chap'); // Cheerio old chap
```

Closures

Functions retain references to the environment of their definition

```
var make_greeter = function(greeting) {
    return function(who) {
        alert(greeting + ' ' + who);
    }
};

var briton = make_greeter('Cheerio');
briton('mate');      // Cheerio mate
briton('old chap'); // Cheerio old chap
```

Immediately invoked functions = modules

```
var days = (function() {
  var names = [ 'Mon', 'Tue', 'Wed', 'Thu',
                'Fri', 'Sat', 'Sun'];

  return {
    name: function(nr) { return names[nr]; },
    nr: function(nm) { return names.indexOf(nm); }
  };
}());

alert(days.name(3));    // 'Thu'
alert(days.nr('Thu')); // 3
```

Exercise 4

Closures can emulate objects

Rewrite the Counter object from exercise 2 as a closure.

Hint:

An outer function creates a scope with private variables and returns an inner function that provides access to this scope.

Closures close over variables, not values

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push(function() { alert(i); });
}

funcs[0](); // ??
```

Closures close over variables, not values

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push(function() { alert(i); });
}

funcs[0](); // ??
```

10

Closures close over variables, not values

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    funcs.push(function() { alert(i); });
}

funcs[0](); // ??  10
```

Every function has a reference to the same `i`!

Function calls make fresh variables

```
var funcs = [];

for (var i = 0; i < 10; i++) {
    (function(j) {
        funcs.push(function() { alert(j); });
    }(i));
}

funcs[0](); // 0
```

First-class functions

Convenient callbacks

Very useful for asynchronous programming.

```
var screen;

retrieveFromNetwork(function(result) {
    screen.display(result);
});
```

First-class functions

Capabilities

Fine-grained access control to objects.

```
counter.next();
counter.reset();

var x = new Collaborator(counter.next);
x.doSomething();
```

First-class functions

Functional programming

Abstract out behavioral patterns.

```
var x2 = function(x) { return x * 2; }

var a = [1, 2, 3];
var b = a.map(x2);
// b = [2, 4, 6]
```

First-class functions

Higher-order functions

Functions that take, manipulate and return other functions.

(Practical example and exercise on the next slides)

Wrap existing methods

```
// Decorate function call with logging.
function add_logging(object, member) {
    var fn = object[member];

    object[member] = function() {
        alert(member + ' called.');
        fn.apply(this, arguments);
    }
}

add_logging(Array.prototype, 'push');

[].push(1);  // 'push called.'
```

Exercise 5 (optional)

Memoization

Write a higher-order function that makes arbitrary functions more efficient by caching their results.

Binding of "this"

`this` will only be bound if you call `object.method()` exactly.

An inner function in a method will lose the reference to `this`!

Binding of “this”

```
Object.prototype.method = function() {
    this.x = 0;
    var inc = function() {
        this.x++;
    };
    inc();
    alert(x); // ?
};
```

You’ll usually pass `inc` away as a callback (and it will fail).

Binding of “this”

```
Object.prototype.method = function() {
    this.x = 0;
    var inc = function() {
        this.x++;
    };
    inc();
    alert(x); // ?
};
```

0

You’ll usually pass `inc` away as a callback (and it will fail).

Binding of “this”

```
Object.prototype.method = function() {
    this.x = 0;
    var inc = functi
        this.x++;
    };
    inc();
    alert(x); // ?
};
```

inc() called without object so ‘this’ got bound to global object.

0

You’ll usually pass `inc` away as a callback (and it will fail).

Capture “this” in closure

```
Object.prototype.method = function() {
    this.x = 0;
    var self = this;
    var inc = function() {
        self.x++;
    };
    inc();
    alert(x); // 1
};
```

Passing around methods

MooTools has `bind()` method that retains the reference to `this`:

```
dog.age = 3;
dog_birthday = dog.birthday.bind(dog);

dog_birthday();
alert(dog.age); // 4
```

jQuery doesn't but we can make it!

bind() function

```
Function.prototype.bind = function(inst) {
    var fn = this;
    return function() {
        fn.apply(inst, arguments);
    }
}
```

Closes over `inst`, the instance to be applied to, as well as `this`.

Implicit type conversions

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

If `A == B` and `B == C` then `A == C` **doesn't hold!**

Use `===` for strict equality checking.

Good Parts

Literals for Objects, Arrays, Functions

Closures

Duck typing

Flexible object pattern

Bad parts

Global namespace by default

Implicit type conversions

Semicolon insertion

Binding of “this”

Coffee Break