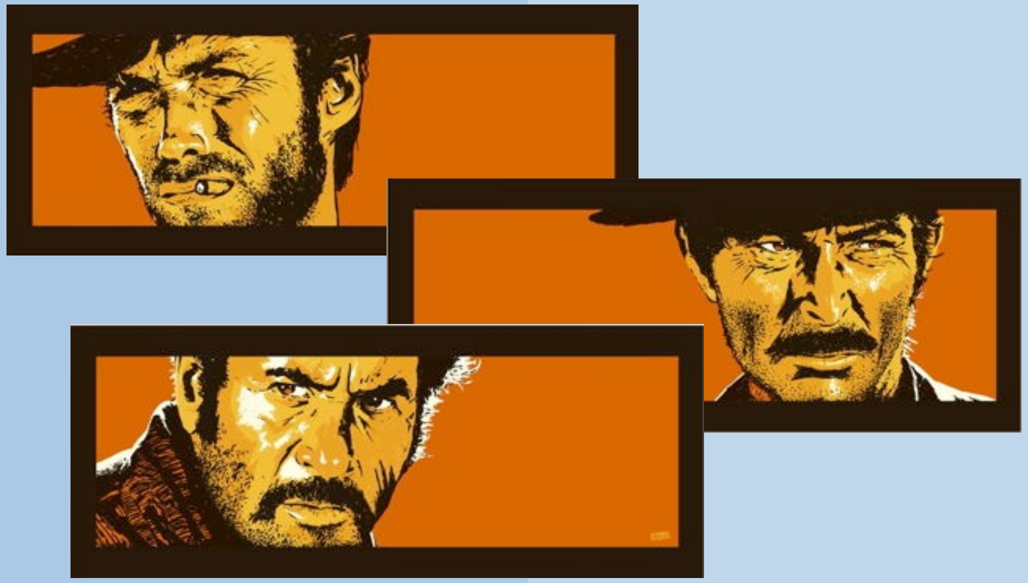
8. Objects and Prototypes

Oscar Nierstrasz

Based on material by Adrian Lienhard



All examples can be found in the usual git repo,

```
git@scg.unibe.ch:lectures-pl-examples
```

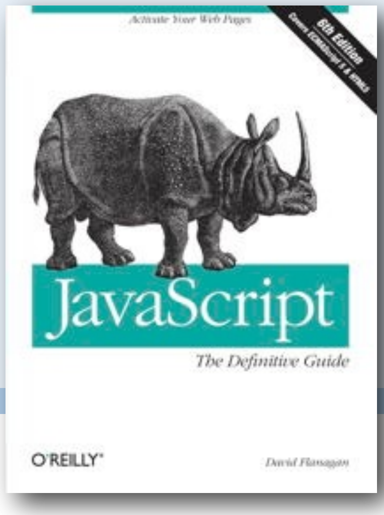
as well as online:

http://scg.unibe.ch/download/lectures/pl-examples/JavaScript/

Roadmap

> Class- vs. prototype-based languages
> Objects, properties and methods
> Delegation
> Constructors
> Closures
> Snakes and Ladders with prototypes
> The Good, the Bad and the Ugly

References



> *JavaScript: The Definitive Guide*, D. Flanagan, O'Reilly, 5th edition

> *JavaScript: The Good Parts*, D. Crockford, O'Reilly

> *JavaScript Guide & Reference*, Mozilla Developer Center, http://developer.mozilla.org/en/docs/JavaScript

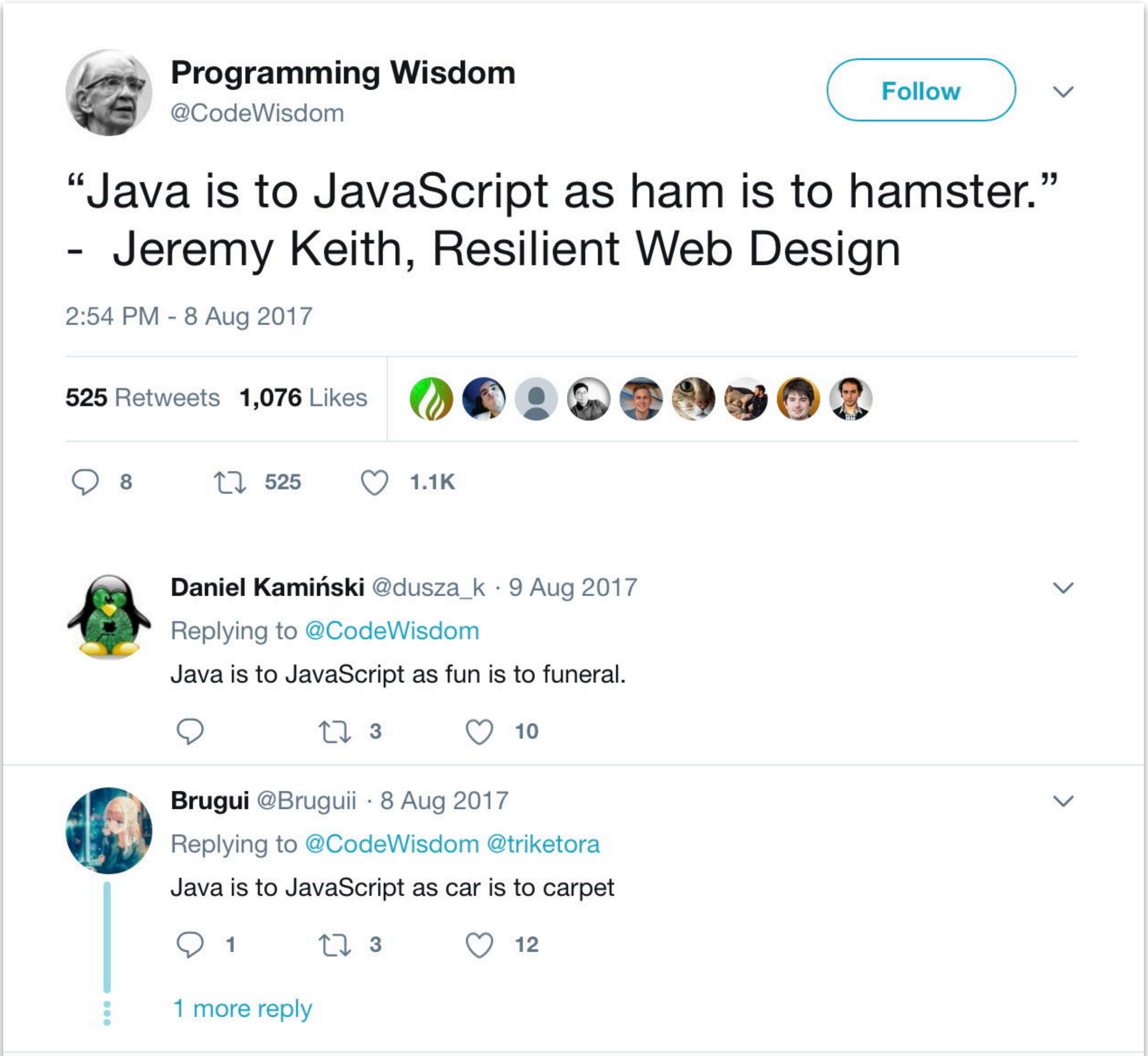
> *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, H. Lieberman, OOPSLA'86

> *ECMAScript Language Specification* — 5th edition, http://www.ecma-international.org/publications/standards/Ecma-262.htm

> *ECMAScript 6 features* —http://es6-features.org/#Constants

Roadmap

> **Class- vs. prototype-based languages**
> Objects, properties and methods
> Delegation
> Constructors
> Closures
> Snakes and Ladders with prototypes
> The Good, the Bad and the Ugly



Programming Wisdom
@CodeWisdom

Follow

“Java is to JavaScript as ham is to hamster.”
- Jeremy Keith, Resilient Web Design

2:54 PM - 8 Aug 2017

525 Retweets 1,076 Likes

8 525 1.1K

Daniel Kamiński @dusza_k · 9 Aug 2017
Replying to @CodeWisdom
Java is to JavaScript as fun is to funeral.

3 10

Brugui @Bruguii · 8 Aug 2017
Replying to @CodeWisdom @triketora
Java is to JavaScript as car is to carpet

1 3 12

1 more reply

Class-based vs. Prototype-based languages

Class-based:

> Classes share methods and define common properties
> *Inheritance* along class chain
> Instances have exactly the properties and behavior defined by their class
> Structure typically cannot be changed at runtime

Prototype-based:

> No classes, only objects
> Objects define their own properties and methods
> Objects *delegate* to their prototype(s)
> Any object can be the prototype of another object

Prototype-based languages unify objects and classes

In class-based languages, classes may (e.g., in Smalltalk) or may not be (e.g., in Java) be *first-class objects*. Classes enable sharing of behaviour and structure. There is no inherent reason why these responsibilities could not be part of ordinary objects, which leads to the idea of prototype-based languages.

The approach was first proposed by Henry Lieberman in an 1986 OOPSLA paper. The central idea is that an object can delegate any responsibility to another object. If an object does not understand a particular message (request), it can delegate it to another object. (Delegation is similar to forwarding, except that the identity of the original object is bound to self — "`this`" in JS — so self-sends will be correctly handled, as in class-based languages.)

Prototype-based Languages

No classes ⇒simpler descriptions of objects

Examples first (vs. abstraction first)

Fewer concepts ⇒simpler programming model

Many languages (but only few used outside research):

> JavaScript, Self, Io, NewtonScript, Omega, Cecil, Lua, Object-Lisp, Exemplars, Agora, ACT-I, Kevo, Moostrap, Obliq, Garnet

Roadmap

> Class- vs. prototype-based languages
> **Objects, properties and methods**
> Delegation
> Constructors
> Closures
> Snakes and Ladders with prototypes
> The Good, the Bad and the Ugly

What is JavaScript?

> Introduced in 1995 by Netscape
> Minimal object model:
  — everything is an object (almost)
  — functions are first-class objects, closures
> Prototype-based delegation
> Dynamically typed, interpreted
> Platforms: web browsers and servers, Java 6, embedded in various applications (Flash, Photoshop, ...)
> What it is not:
  — No direct relationship to *Java*
  — Not only for *scripting* web browsers

Syntax

Comments:	`// single line comment`
	`/* multi`
	`line comment */`

Identifiers:	First character must be a letter, _, or `$`; subsequent characters can be digits: `i, v17, $str, __proto__`

Basic literals:	`‘a string’, “another string”, “that’s also a string”`
	`17, 6.02e-32`
	`true, false, null, undefined`

Object literals:	`var point = { x:1, y:2 }`
	empty: `{}`
	nested: `var rect = {`
	`upperLeft: { x:1, y:2 },`
	`lowerRight: { x:4, y:5 } }`

Function literals:	`var square = function(x) { return x*x; }`
Array literals:	`[1,2,3] []`
Operators:	assignment: `=` equality: `==` strict equality (identity): `===`

Object Properties

Reading properties

```
var book = { title: 'JavaScript' };
book.title; //=>'JavaScript'
```

Adding new properties (at runtime)

```
book.author = 'J. Doe';
'author' in book; //=>true
```

Inspecting objects

```
var result = '';
for (var name in book) {
 result += name + '=';
 result += book[name] + ' ';
};
//=>title=JavaScript author=J. Doe
```

Deleting properties

```
delete book.title;
'title' in book; //=>false
```

In JavaScript (JS), an object consists of a set of properties. These may be either data values (like strings, numbers, or other objects), or they may be functions (lambdas). In other words, *an object is a dictionary*.

Methods

Methods are just *functions* that are assigned to properties of an object

At runtime the keyword `this` is bound to the object of the method

```
var counter = { val : 0 }
counter.inc = function() {
  this.val++;
}

counter.inc();
counter.val; // => 1
```

Accessing (vs. executing) methods

```
var f = counter.inc;
typeof f; //=> 'function'
```

Property and method slots are unified. Functions are first-class objects.

A prototype, unlike an object in a class-based language, may hold its methods as values of its properties. No distinction is made between executable properties and other data values. In practice, methods are generally stored in shared prototypes, but this is not required by the object model.

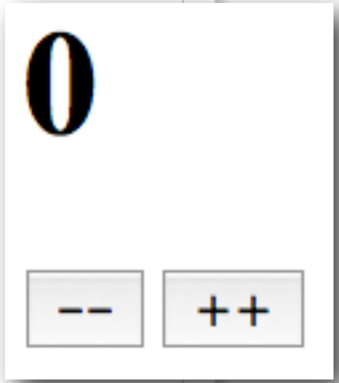
In the example we see that properties can be added incrementally to objects in JS. The `inc` property is a function (a lambda) that is added to the `counter` object after it is first created.

Note how the variable “`this`” (AKA *self*) refers to the object itself (the original receiver).

NB: the `var` declaration creates a variable in the current lexical scope (more on this later). Without “`var`”, JS will search for it in enclosing scopes, and if it does not find the variable, will (silently) create it in the global scope.

Scripting a web browser

```
<html>
<head>
<title>Counter</title>
<script type="text/javascript">
var counter1 = {
   val : 0,
   name : "counter1",
   inc : function() { this.val++; update(this); },
   dec : function() { this.val--; update(this); }
}
function update(counter) {
   document.getElementById(counter.name).innerHTML=counter.val;
}
</script>
</head>
<body>
<h1 id="counter1">0</h1>
<button type="button" onclick="counter1.dec()">--</button>
<button type="button" onclick="counter1.inc()">++</button>
</body>
</html>
```



JS was designed as client-side scripting language for web pages, somewhat like Java “applets”, but specified as dynamic scripts within HTML. JS can control functionality of the client’s web browser, and (to some extent) has access to server side. Unlike PHP (which is executed server-side), JS is downloaded as part of web page, and can be viewed in source form on the client.

This example illustrates how JS can control updates of the DOM (Domain Object Model) of the web page, modifying the value of the header with id “`counter1`” when the `--` and `++` buttons are clicked.

Roadmap

- Class- vs. prototype-based languages
- Objects, properties and methods
- **Delegation**
- Constructors
- Closures
- Snakes and Ladders with prototypes
- The Good, the Bad and the Ugly

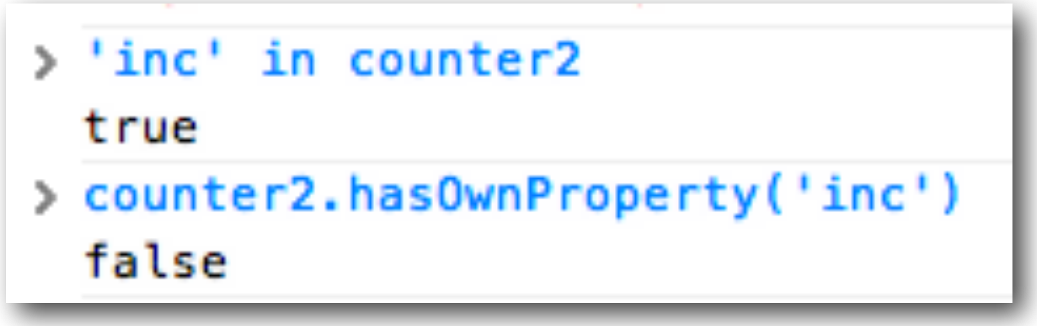
Delegation

The mechanism to share data and behavior is *delegation*

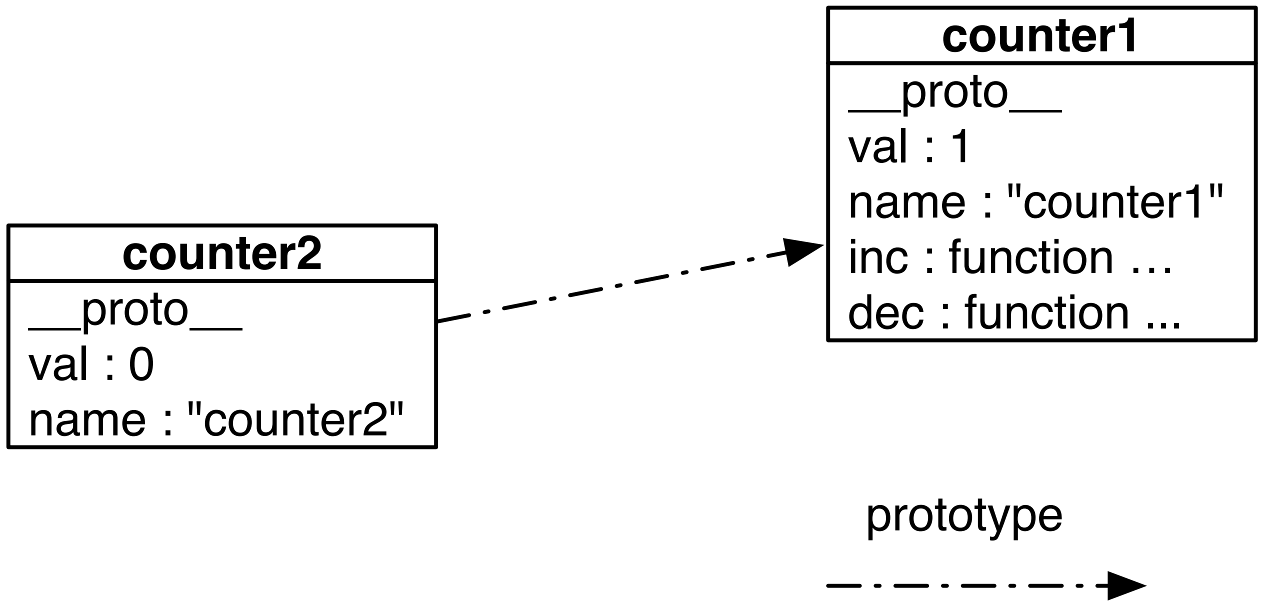
An object delegates to its *prototype* object (the Mozilla interpreter allows one to access the prototype through the property `__proto__`)

```
var counter2 = Object.create(counter1);
counter2.val = 0;
counter2.name = "counter2";
```

Binding not found in object, then *lookup* in prototype



```
> 'inc' in counter2
  true
> counter2.hasOwnProperty('inc')
  false
```



In principle, an object may be self-contained, holding all its data and methods as properties. In practice, it make sense to share methods between similar objects, just as in class-based languages. Every object can have a prototype, to which it delegates requests.

The JS console shown in the slide is provided by the Chrome web browser. It offers a convenient interface for inspecting and interacting with JS objects in the current web page. Here we are inspecting the Counter.html example from the PL examples repo.

```
git@scg.unibe.ch:lectures-pl-examples
```

Delegation of Messages

The key aspect of *prototype delegation* is that `this` in the prototype is bound to the *receiver* of the original message.

```
counter2.inc();
counter2.val; //=>1
```

The method `inc()` is executed in the context of `counter2`, the receiver, rather than in `counter1`, the object to which the message `inc()` is delegated!

The invocation `Object.create(counter1)` will create a new object that has the object `counter1` as its prototype. All requests to `counter2` that it does not find in its own dictionary of properties will be *delegated* to its prototype.

The key difference between forwarding and delegation is that the original receiver is remembered in the "`this`" variable. When `counter2.inc()` is evaluated, it is delegated to `counter1`. When the `inc()` function is evaluated, however, `this` is bound to `counter2` and not `counter1`, so the correct `val` property is updated. With simple forwarding, the `object1.val` would be updated instead.

Roadmap

> Class- vs. prototype-based languages
> Objects, properties and methods
> Delegation
> **Constructors**
> Closures
> Snakes and Ladders with prototypes
> The Good, the Bad and the Ugly

Constructor Functions

Constructors are functions that are used with the `new` operator to create objects

The operator `new` creates an object and binds it to `this` in the constructor. By default the return value is the new object.

```
function Counter(name) {
  this.val = 0;
  this.name = name;
}
```

```
var counter3 = new Counter("counter3");

counter3.val; //=>0
counter3.name; //=>"counter3"
```

Note that simply calling the function will not create a new object. Worse, it will update properties of `this` in the current context (whatever that might be).

Constructor.prototype

- All objects created with a constructor share the same prototype
- Each constructor has a `prototype` property (which is automatically initialized when defining the function)

Instead of creating a new method for each object, add one to the prototype

```
Counter.prototype.inc = counter1.inc;
Counter.prototype.dec = counter1.dec;
```

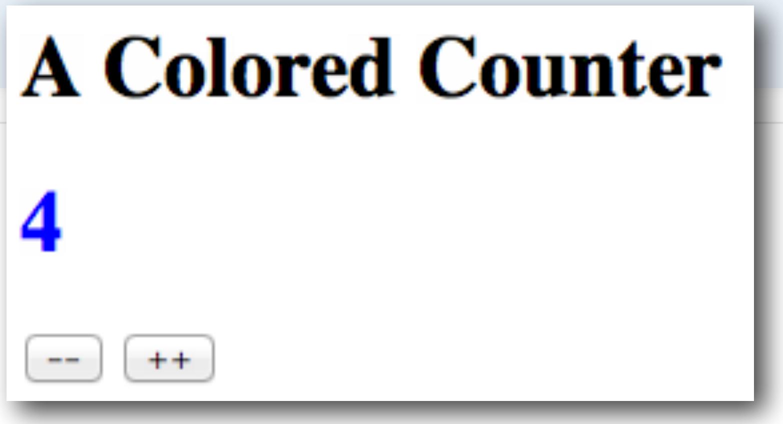
```
var counter3 = new Counter("counter3");

counter3.inc();
counter3.val; //=>1
```

Note how the functions `inc` and `dec` in `counter1` are copied to the `Counter` prototype. These functions each make use of a `this` variable, which will be bound to any object created using the `Counter` constructor.

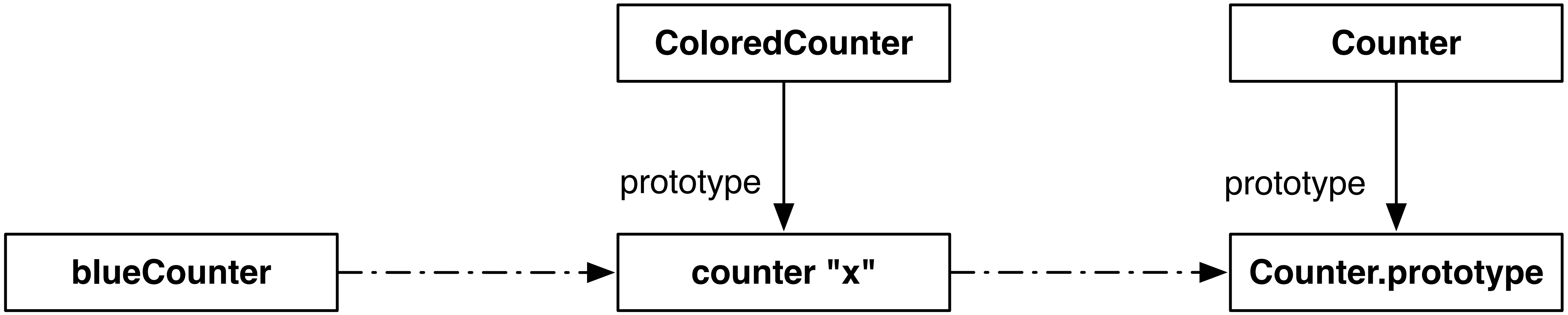
We also see clearly in this example that *functions are object too*! In particular, the `Counter` function (also a constructor) has a `prototype` field, so it is also an object.

Constructor.prototype

```
function ColoredCounter(name, color) {
  this.val = 0;
  this.name = name;
  this.color = color;
  var that = this;
  window.onload = function() {
    document.getElementById(that.name).style.color = that.color;
  };
}
ColoredCounter.prototype = new Counter('x');
var blueCounter = new ColoredCounter('blueCounter', 'Blue');
```


A Colored Counter

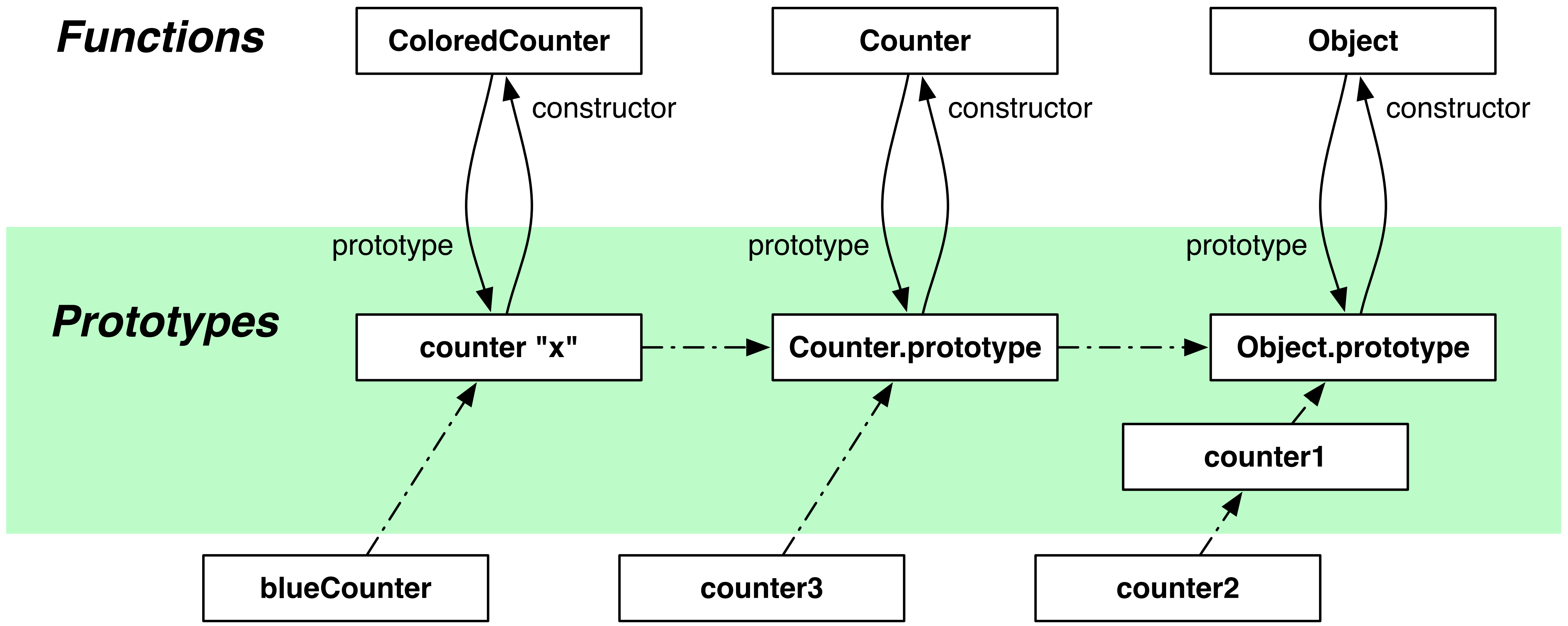
4



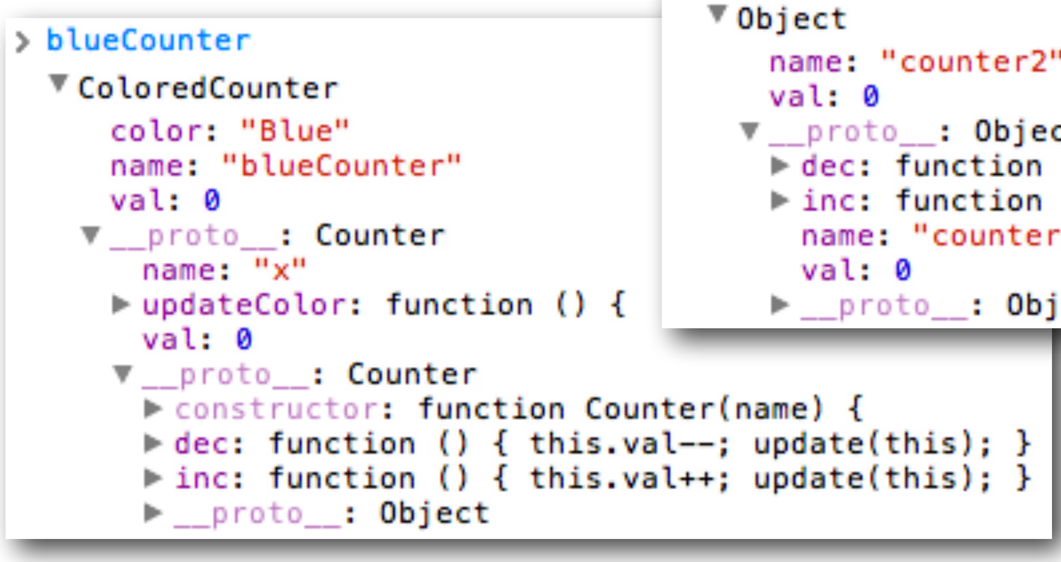
In this example we reassign the default prototype of `ColoredCounter` to be an instance of `Counter` (`counter “x”`). Its prototype, in turn, is that of `Counter`.

This is a classical prototype chain. Every property that is looked up in the `blueCounter` but not found, is delegated to the next prototype in the chain.

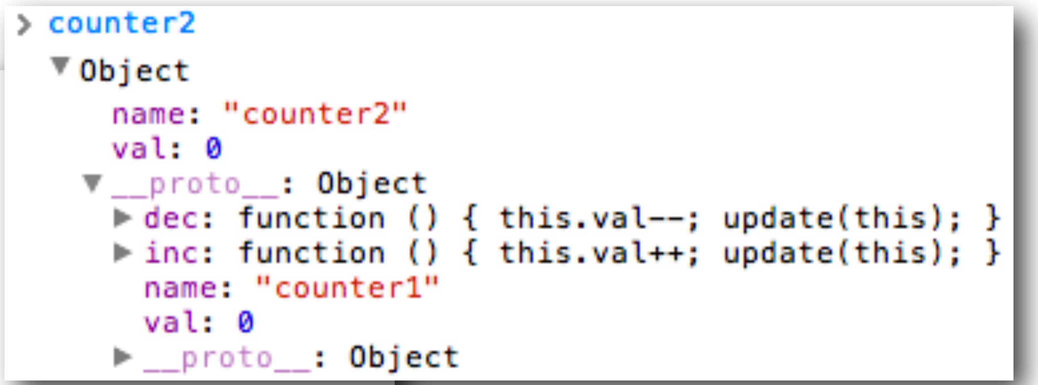
Object Model



NB: This is incomplete. What are the prototypes of Object, Counter … ?



```
> blueCounter
  ▼ColoredCounter
     color: "Blue"
     name: "blueCounter"
     val: 0
   ▼__proto__: Counter
      name: "x"
    ▶updateColor: function () {
      val: 0
    ▼__proto__: Counter
     ▶constructor: function Counter(name) {
     ▶dec: function () { this.val--; update(this); }
     ▶inc: function () { this.val++; update(this); }
     ▶__proto__: Object
```



```
> counter2
  ▼Object
     name: "counter2"
     val: 0
   ▼__proto__: Object
    ▶dec: function () { this.val--; update(this); }
    ▶inc: function () { this.val++; update(this); }
      name: "counter1"
      val: 0
    ▶__proto__: Object
```

Every object has a prototype. Every prototype is an object, and any object may also serve as a prototype for another object.

Objects may be created (i) as a literal, (ii) using Object.create(), or (iii) using a constructor.

Literal objects (e.g., `counter1`) have `Object.prototype` as their prototype.

Objects created with `Object.create()` have the argument as their prototype (e.g., `counter2` has `counter1` as its prototype).

Objects that are created using a constructor function get their (initial) prototype from that constructor (e.g., `counter3`, `counter4`).

Predefined Objects

> Global functions: Array, Boolean, Date, Error, Function, Number, Object, String,... eval, parseInt, ...

> Global objects: Math

Extending Predefined Objects

Extending all objects

The last object in the prototype chain of every object is `Object.prototype`

```
Object.prototype.inspect = function() {
  alert(this);
};
'a string'.inspect();
true.inspect();
(new Date()).inspect();
```

Extending arrays

```
Array.prototype.map = function(f) {
  var array = [];
  for (var n = 0; n < this.length; n++) {
    if (n in this) { array[n] = f(this[n]); };
  };
  return array;
};
[1.7, -3.1, 17].map(Math.floor); //=>[1, -4, 17]
[1.7, -3.1, 17].map(function(x) { return x+1; });
  //=>[2.7, -2.1, 18]
```

Since all objects have `Object.prototype` at the end of their prototype chain, one can add properties to all objects simply by extending that object.

In the first example we define an `inspect()` method for all objects that will pop up a browser alert dialog.

In the second example we add a `map()` function to all JS arrays.

(Hint: you can try these out in the Chrome browser JS console.)

The arguments object

`arguments` object

you can call a function with more arguments than it is formally declared to accept

```
function concat(separator) {
  var result = '';
  for (var i = 1; i < arguments.length; i++)
    result += arguments[i] + separator;
  return result;
};
concat(";", "red", "orange", "blue");
//=>"red;orange;blue;"
```

`arguments.callee` returns the currently executing function

```
var f = function() {
  if (!arguments.callee.count) {
    arguments.callee.count = 0; };
  arguments.callee.count++; };
f(); f(); f();
f.count; //=>3
```

In JS you can pass any number of arguments to a function. Those not declared explicitly as arguments in the function's definition can be accessed by the `arguments` variable.

`arguments.callee` is the function being called.

NB: the function cannot be accessed using "`this`" since that would give the enclosing object!

Roadmap

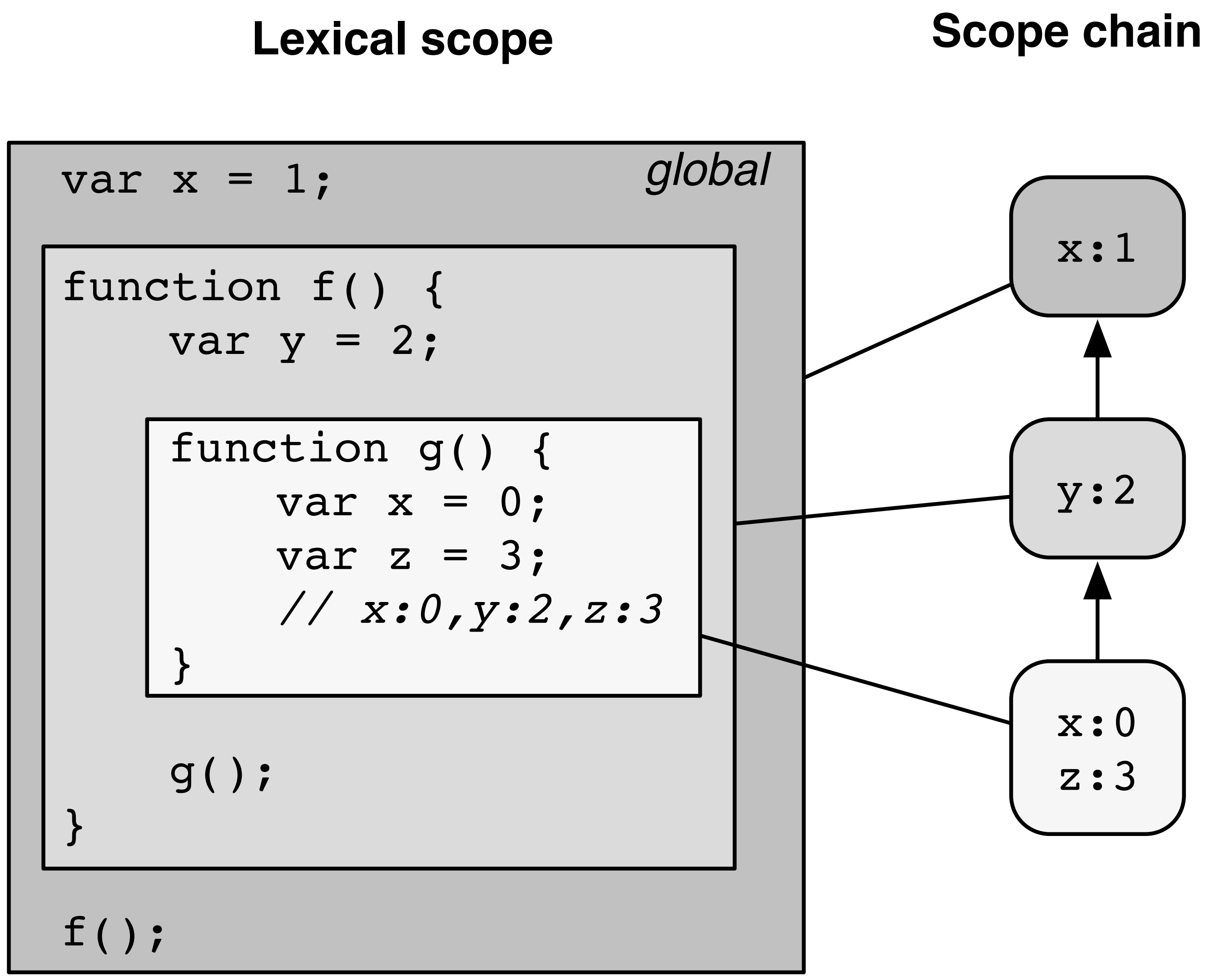
- > Class- vs. prototype-based languages
- > Objects, properties and methods
- > Delegation
- > Constructors
- > **Closures**
- > Snakes and Ladders with prototypes
- > The Good, the Bad and the Ugly

Variable Scopes

> The *scope* of a variable is the region of the program in which it is defined

> Scopes in JavaScript
  — *function*
  — *global*
  — no block-level scope(!)

> Identifier resolution: lookup along scope chain



```
var x = 1;

function f() {
    var y = 2;

    function g() {
        var x = 0;
        var z = 3;
        // x:0,y:2,z:3
    }

    g();
}

f();
```

Program languages distinguish between *lexical* and *dynamic scope*. Lexical scope is determined purely by the program source code, while dynamic scope is determined at run time. Each nested scope may introduce new names (i.e., variables or functions), which are not defined outside of that scope. This effectively gives rise to a stack of scopes, as shown in the diagram. The innermost scope of `g()` defines `x` and `z`, which are not visible in the enclosing scope of `f()`. Note that there is a global `x` in the outermost scope that is *shadowed* by the `x` within `g()`. (We also say that the `x` in `g()` *masks* the global `x`.)

Modern programming languages mostly rely on lexical scoping, as dynamic scoping makes programs hard to understand. The only common example of dynamic scoping is exception handlers, which are set dynamically by a calling scope, affecting dynamically called inner scopes.

Closures

> Functions are *lexically scoped* (rather than dynamically). Functions are executed in the scope in which they are *created*, not from which they are called.

> Inner functions are *closures*

When the anonymous function is created, the current scope chain is saved. Hence, the scope of f(x) continues to exist even after f(x) returned.

```
function f(x) {
    var y = 1;
    return function() { return x + y; };
};
closure = f(2);
var y = 99;
closure(); //=>3
```

A *closure* is a *function* (or an object) together with an *environment* that binds the free names within it (i.e., the environment *closes* those names).

Here `x` and `y` are free in

```
function() { return x + y; }
```

but they are both bound in the enclosing environment of `f()`.

When we evaluate

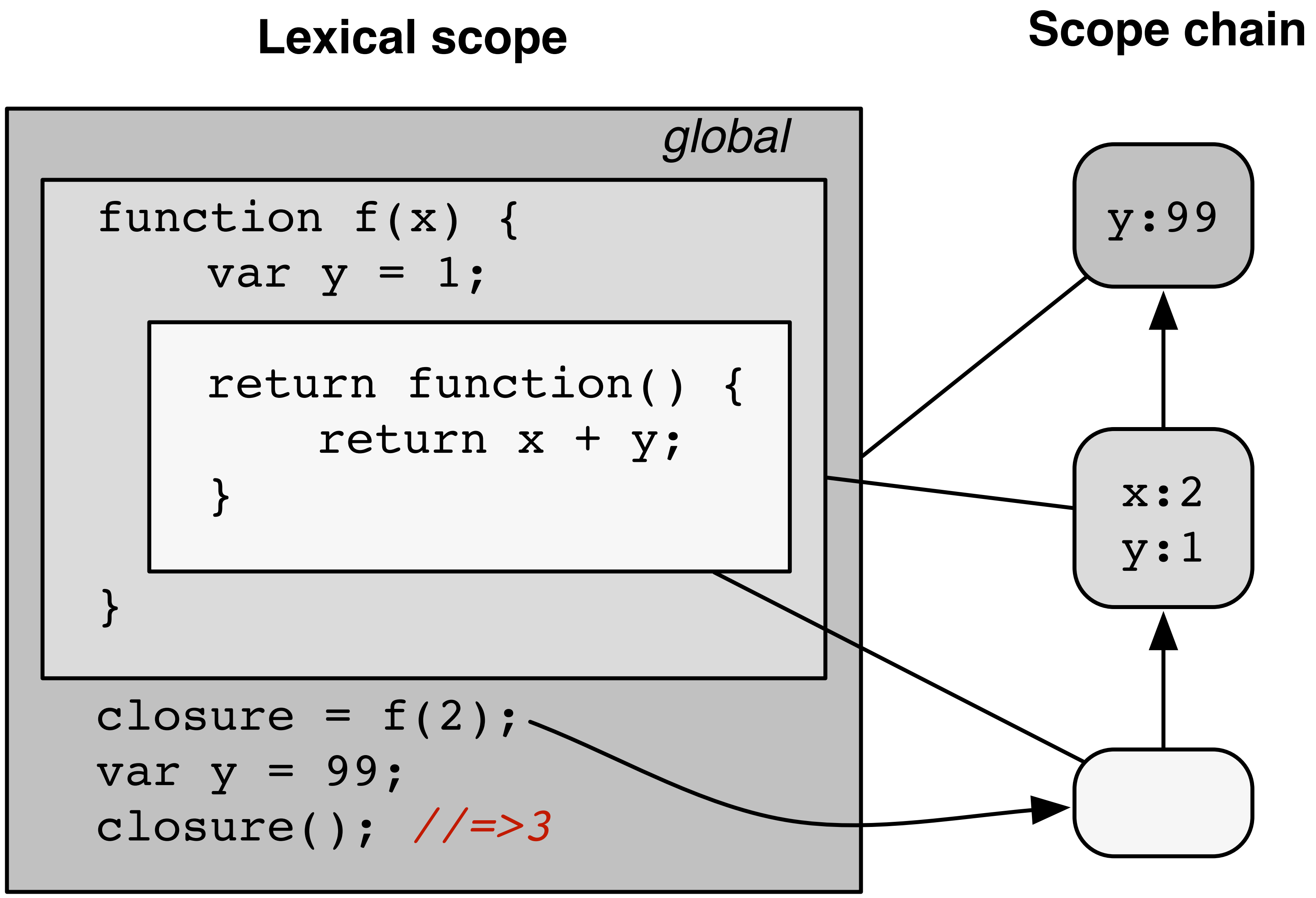
```
closure = f(2);
```

we obtain a closure consisting of the anonymous function together with the environment that binds `x` to `2` and `y` to `1`.

Defining `y` to be `99` in the global scope does *not* affect the environment of the closure, so evaluating it will yield `3`, not `101`.

(With dynamic scoping, names would be looked up in the current scope, not the lexical one.)

Closures



```
function f(x) {
    var y = 1;
    return function() {
        return x + y;
    }
}
closure = f(2);
var y = 99;
closure(); //=>3
```

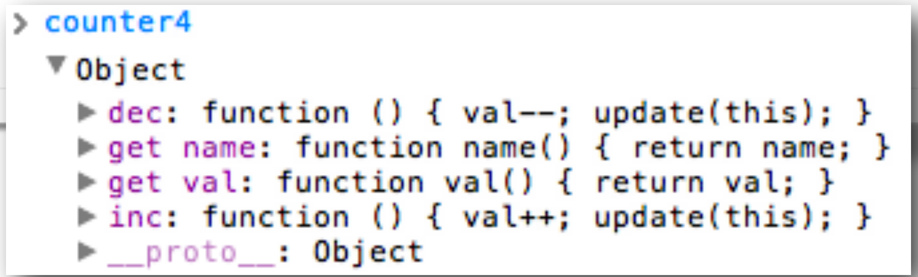
A closure is a function whose free variables are bound by an associated environment.

Closures and Objects

By default all properties and methods are public.
Using closures, properties and methods can be made private.

The variables `val` and `name` are only accessible to clients via the exported methods.

```
var counter4 = (function(name) {
  var val = 0;
  var name = name;
  return {
    inc : function() { val++; update(this); },
    dec : function() { val--; update(this); },
    get val() { return val; },
    get name() { return name; }
  }
})('counter4');
```



```
> counter4
  ▼Object
    ▶dec: function () { val--; update(this); }
    ▶get name: function name() { return name; }
    ▶get val: function val() { return val; }
    ▶inc: function () { val++; update(this); }
    ▶__proto__: Object
```

NB: ECMAScript 5 getters are used to allow `val` and `name` to be accessed as (immutable) fields.

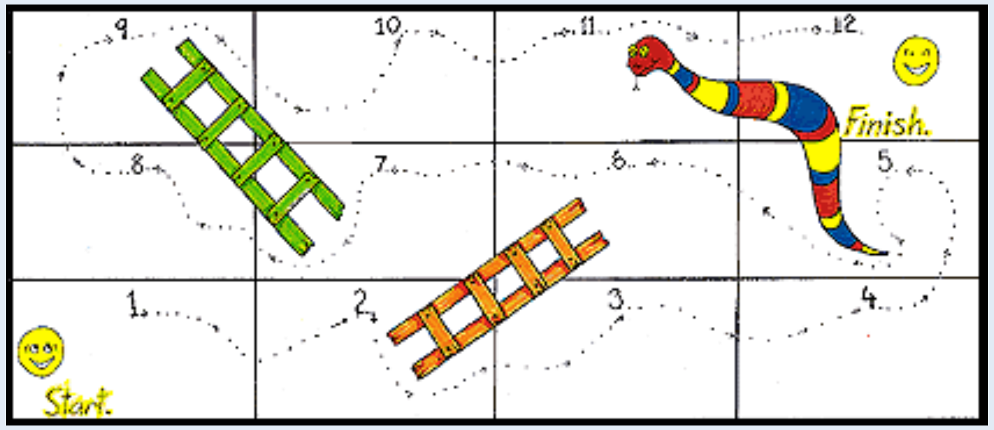
Although all properties are public by default, we can use the mechanism of closures to provide a form of information hiding. The trick is to define a function that returns an object containing only public properties, but whose methods are closures, with (private access) to a hidden environment.

This is supported by a JS idiom in which a function, which defines the private environment within its own lexical scope, is immediately called, and returns a closure (a literal object) with access to that environment.

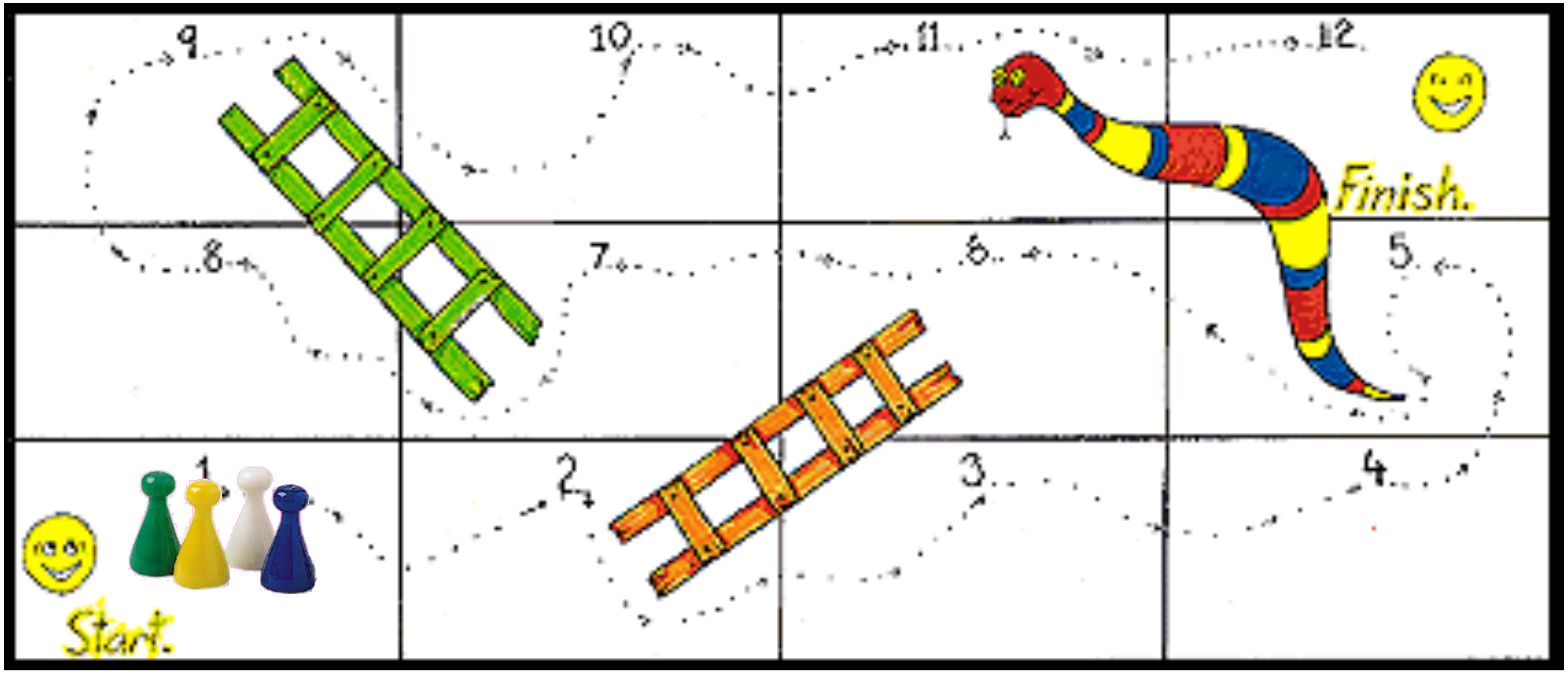
Here an unnamed function defines an environment with the properties `val` and `name`. It is called and returns a literal object with access to these names, but they are not (public) properties of the object itself.

Roadmap

> Class- vs. prototype-based languages
> Objects, properties and methods
> Delegation
> Constructors
> Closures
> **Snakes and Ladders with prototypes**
> The Good, the Bad and the Ugly



Snakes and Ladders (see P2 lecture)



http://en.wikipedia.org/wiki/Snakes_and_ladders

We will reimplement the Snakes and Ladders games from lecture 2 of the P2 OO design course.

http://scg.unibe.ch/teaching/p2

The original design was class-based and implemented in Java. Here we will adopt a prototype-based design.

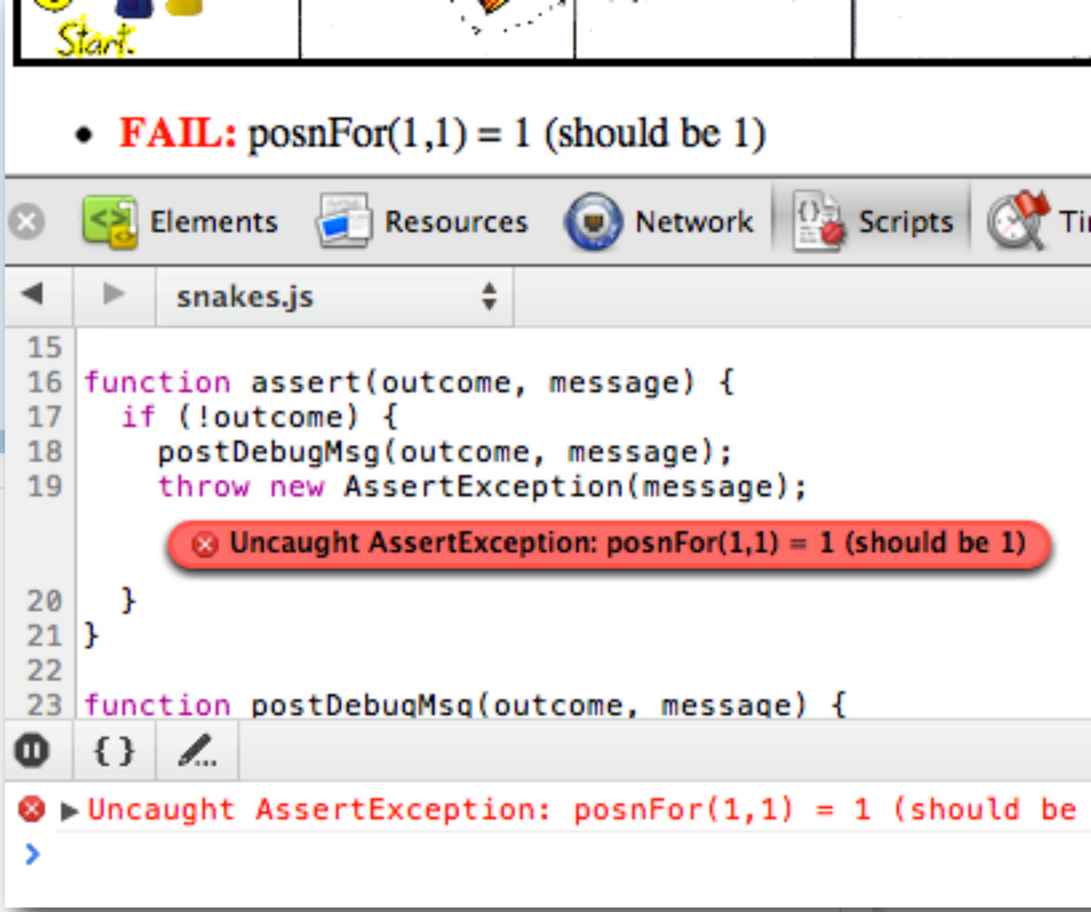
See the full source code in the examples repo:

```
git@scg.unibe.ch:lectures-pl-examples
```

Or try it out online (with links to source code):

http://scg.unibe.ch/download/lectures/pl-examples/JavaScript/Snakes/snakes.html

Assertions and Exceptions


- FAIL: posnFor(1,1) = 1 (should be 1)

```
function assert(outcome, message) {
  if (!outcome) {
    postDebugMsg(outcome, message);
    throw new AssertException(message);
      Uncaught AssertException: posnFor(1,1) = 1 (should be 1)
  }
}

function postDebugMsg(outcome, message) {
```

```
Uncaught AssertException: posnFor(1,1) = 1 (should be
>
```

```
function assert(outcome, message) {
  if (!outcome) {
    postDebugMsg(outcome, message);
    throw new AssertException(message);
  }
}

function postDebugMsg(outcome, message) {
  var debug = document.getElementById('debug');
  var li = document.createElement('li');
  li.className = outcome ? 'pass' : 'fail';
  li.appendChild( document.createTextNode( message ) );
  debug.appendChild(li);
};

function AssertException(message) { this.message = message; }
AssertException.prototype.toString = function () {
  return 'AssertException: ' + this.message;
}
```

We will adopt *Design by Contract*, so we need a way to write *assertions* (there wasn't one yet when we wrote this code).

If an assertion fails, we post a message in the `debug` element of the current web page.

Closures

```
function makeDisplay(config, player1, player2) {
  var rows = config.rows;
  var cols = config.cols;
  ...

  var canvas = document.getElementById('display');
  var c = canvas.getContext('2d');

  var boardImg = new Image();
  boardImg.src = config.board;
  ...

  function repaint() {
    c.drawImage(boardImg,0,0);
    piece1.draw();
    piece2.draw();
  }
  ...
```

```
...
  // only return the public fields
  return {
    repaint: repaint,
    test: test
  }
}

...

display = makeDisplay(config, jack, jill);
```

The display is fully encapsulated as a closure.

The `makeDisplay()` constructor returns an object with just two properties, but the functions these properties are bound to have access to the lexical context of `makeDisplay()` holding further private properties, such as `rows`, `cols`, `canvas` and so on.

`canvas.getContext('2d')` will return an object supporting 2D drawing operations.

Asynchrony

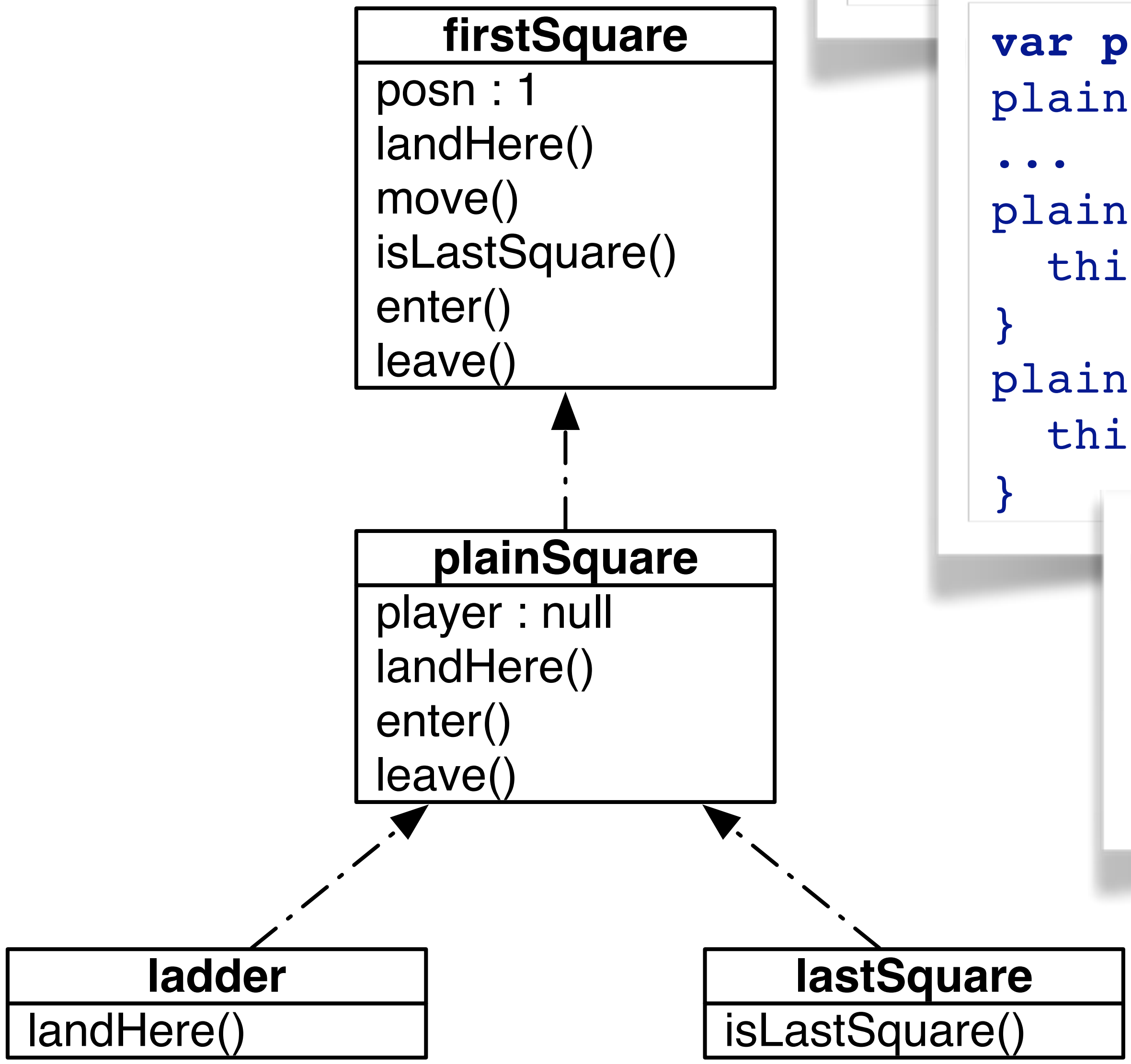
Files are loaded asynchronously
triggering a callback on completion

```
boardImg.onload = function () {
  c.canvas.height = this.naturalHeight;
  c.canvas.width = this.naturalWidth;
  c.drawImage(boardImg,0,0);
};
```

```
piece1.onload = function () {
  var scale;
  this.dh = rowHeight() / 2;
  scale = this.dh / this.naturalHeight;
  this.dw = this.naturalWidth * scale;
  repaint();
}
```

Both `boardImg` and `piece1` are instances of `Image`. The `onload()` method will be called automatically when the Image is loaded. Note how the piece auto-adjusts its size.

Delegation vs Inheritance



```
var firstSquare = {
  posn : 1,
  landHere : function() { return this; },
  ...
  enter : function() {},
  leave : function() {}
}
```

```
var plainSquare = Object.create(firstSquare);
plainSquare.player = null;
...
plainSquare.enter = function(player) {
  this.player = player;
}
plainSquare.leave = function() {
  this.player = null;
}
```

```
var ladder = Object.create(plainSquare);
ladder.landHere = function() {
  return board.squares
    [this.posn + this.forward].landHere();
}
```

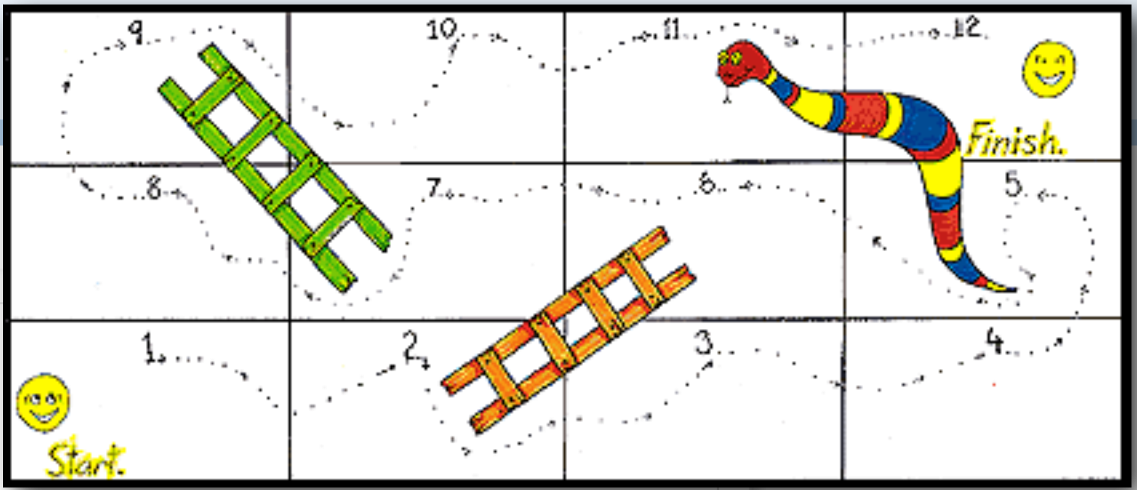
```
var lastSquare = Object.create(plainSquare);
lastSquare.isLastSquare = function() {
  return true;
}
```

In the Java version of the game, we have a class hierarchy of different kinds of squares.

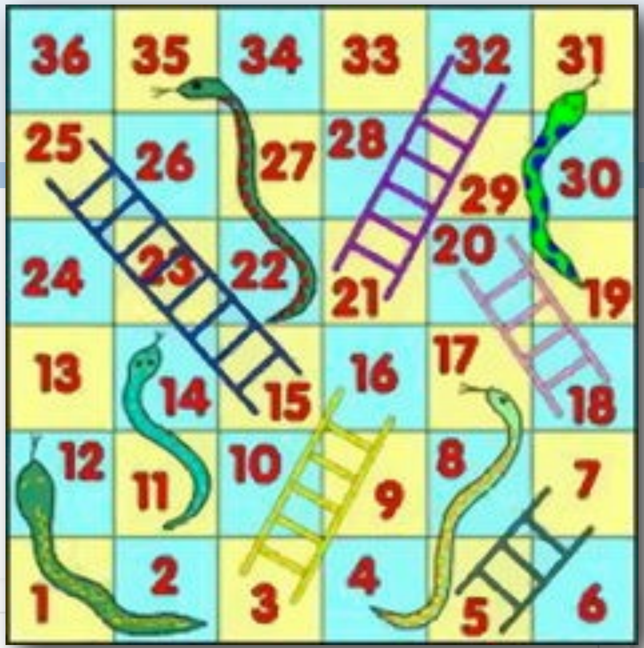
In JS we have no classes, but instead a delegation chain. The `firstSquare` serves as a prototype for a `plainSquare`, and so on. Each new kind of square only redefines the methods that require special handling, for example: the first square does not care who enters or leaves it, but a plain square must ensure that at most one player occupies it at a time.

The objects `plainSquare` and `ladder` serve as prototypes for the actual squares of a given board.

JSON



```
{
  "rows" : 3,
  "cols" : 4,
  "board" : "board12.png",
  "piece1" : "blue.png",
  "piece2" : "yellow.png",
  "ladders" : [
    { "posn" : 2, "forward" : 4 },
    { "posn" : 7, "forward" : 2 },
    { "posn" : 11, "forward" : -6 }
  ]
}
```



```
{
  "rows" : 6,
  "cols" : 6,
  "board" : "board36.png",
  "piece1" : "blue.png",
  "piece2" : "yellow.png",
  "ladders" : [
    { "posn" : 3, "forward" : 13 },
    { "posn" : 5, "forward" : 2 },
    { "posn" : 15, "forward" : 10 },
    { "posn" : 18, "forward" : 2 },
    { "posn" : 21, "forward" : 11 },
    { "posn" : 12, "forward" : -10 },
    { "posn" : 14, "forward" : -3 },
    { "posn" : 17, "forward" : -13 },
    { "posn" : 31, "forward" : -12 },
    { "posn" : 35, "forward" : -13 }
  ]
}
```

JSON is a lightweight standard for data exchange based on object literals

The configuration is a literal object stored as a JSON (JavaScript Object Notation) file. It is passed to the `newGame()` function using jQuery in the main HTML file:

```
 jQuery.getJSON("board12.json", newGame);
```

jQuery parses the JSON file, converts it into a literal JS object and passes it as an argument to `newGame()`.

Interpreting JSON

```
function makeBoard(config) {
  var squares = [];
  ...
  squares[1] = firstSquare;
  for (var i=2;i<=size;i++) {
    square = Object.create(plainSquare);
    squares[i] = square;
    square.posn = i;
  }
  var entry;
  for (i = 0; i<config.ladders.length; i++) {
    entry = config.ladders[i];
    newLadder = Object.create(ladder);
    ...
  }
  ...
  return {
    size : size,
    squares : squares,
    find : find,
    home : home
  }
}
```

The `newGame()` function invokes `makeBoard()` which interprets the configuration and creates a closure representing the configured board. Here we see that the `plainSquare` and `ladder` objects are used as prototypes for the squares of the board. (We directly use `firstSquare` and `lastSquare`, as we only need one of each.)

“Adam’s rib”

```
var jack, jill;
jack = {
  name : 'Jack',
  square : firstSquare,
  move : function(moves) {
    this.square.leave();
    this.square = this.square.move(moves);
    this.piece.posn = this.square.posn;
    this.square.enter(this);
    display.repaint();
  },
  wins : function() {
    return this.square.isLastSquare();
  }
}

jill = Object.create(jack);
jill.name = 'Jill';
jill.square = firstSquare;
```

Player `jack` can serve as a prototype for `jill`, as they have the same behavior and only different state. Note again how the pseudo-variable `this` is correctly resolved in `jill` to refer to her state, not `jack`'s.

The top level

```
var game = {
  player : jack,
  isOver : false,
  swapTurn : function () {
    this.player = (this.player === jack) ? jill : jack;
  },
  move : function () {
    var moves;
    var win = '';
    if (!this.isOver) {
      moves = die.roll();
      this.player.move(moves);
      if (this.player.wins()) {
        win = ' &mdash; ' + this.player.name + ' wins!'
        this.isOver = true;
      }
      this.status(this.player.name + ' rolls ' + moves + win);
      this.swapTurn();
    } else {
      this.status('The game is over!');
    }
  },
  status : function(msg) {
    document.getElementById('status').innerHTML=msg;
  }
}
```

The `game` object keeps track of the state of the game and implements the main `move()` function, which is invoked from the web interface. The `status()` function injects feedback messages into the web page.

The top level

snakes.html

```
<html>
...
<body>
...
<canvas id="display"></canvas>
<ul id="debug"></ul>
<script type="text/javascript" src="jquery-1.7.2.min.js"></
script>
<script type="text/javascript" src="snakes.js"></script>
<script type="text/javascript">
jQuery.getJSON("board12.json", newGame);
</script>
<button type="button" onclick="game.move()">move</button>
...
</body>
</html>
```

snakes.js

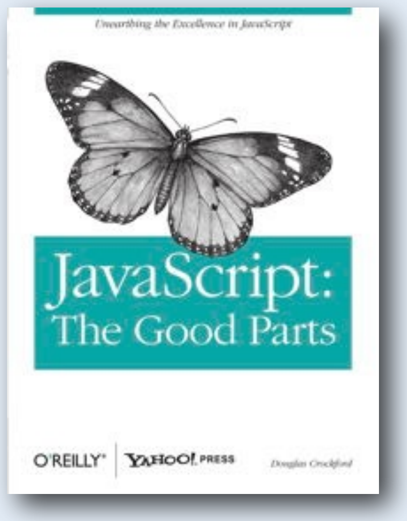
```
function newGame(config) {
  board = makeBoard(config);
  jack.square = firstSquare;
  jill.square = firstSquare;
  game.player = jack;
  game.isOver = false;
  display = makeDisplay(config, jack, jill);
  display.test();
  display.repaint();
}
```

jQuery is a JS library to simplify client-side scripting

The JS implementation resides in the snakes.js file, while the main web page (snakes.html) includes it, loads the configuration, and links a button to the `game.move()` method.

Roadmap

> Class- vs. prototype-based languages
> Objects, properties and methods
> Delegation
> Constructors
> Closures
> Snakes and Ladders with prototypes
> **The Good, the Bad and the Ugly**



The Good

> Object literals, JSON
> Object.create()
> Dynamic object extension
> First-class functions
> Closures

These features are used extensively in idiomatic JS code. JSON is widely used as a human readable format for expressing and exchanging data objects. It has the advantage of being far more readable than XML.

Object literals combined with `Object.create()` offer an expressive way to create delegation chains. Dynamic object extension offers a convenient way to incrementally define and update objects (and prototypes).

First class functions and closures offer a very expressive way to control visibility of properties.

The Bad (globals)

```
function ColoredCounter(name, color) {
  this.val = 0;
  this.name = name;
  this.color = color;
  window.onload = function() {
    document. ... .color = this.color;
  };
}
```

fails miserably

> Global variables
  — belong to `window`
  — `this` may bind to global object
> Undeclared variables are new globals (!)
> No nested block scopes!

Caveat: ES6 fixes some of these, e.g., block scopes.

As in all programming languages, globals are evil. In the example, the `ColoredCounter` constructor will create a new object with `val`, `name` and `color` properties. But the `window.onload` function will be bound in the context of the global `window` object, not the object being created, so it refers to the *global* `color` property, not that of the object being created. This kind of confusion arises frequently in JS.

Forgetting to declare a variable with `var` will just cause JS to try to update it in the enclosing lexical scope. If it is not found, it will (eventually) silently create it in the global context.

Blocks do not have their own lexical scope, but share that of the enclosing function. EC6 however fixes this:

https://www.freecodecamp.org/news/5-javascript-bad-parts-that-are-fixed-in-es6-c7c45d44fd81/

The Bad (arrays)

- Arrays are not real arrays [p 105]
  - Slow, inefficient
- Function arguments are not arrays [p 31]
  - Just objects with a `length` property
- For-in loops don’t work well with arrays
  - Iterates over all properties, not length of array

"Arrays" are actually objects, and are not implemented efficiently. This article explains some of the problems:

http://thecodeship.com/web-development/common-pitfalls-when-working-with-javascript-arrays/

In particular, the typeof operator does not distinguish between arrays and objects.

The Bad (...)

> Semicolon insertion [p 102]
  — Don’t rely on it
> Type confusion [p 103]
  — `typeof` returns strange results
> Equality [p 109]
  — Doesn’t play well with implicit coercion
  — Not symmetric!
  — Use === instead
> Constructors [p 29]
  — Calling without `new` will give unexpected results!

Sometimes JS can insert semi-colons where they are not welcome. For example, be sure that a `return` statement is followed on the same line by the value it returns, otherwise JS will insert a semi-colon, causing it to return undefined.

The `typeof` operator says `null` is an `object`, not a `null`.

The `==` equality operator will try to coerce arguments if they are not of the same type, possibly leading to strange results. Better use `===`.

Forgetting to call a constructor with `new` will cause very strange results. For this reason it is important to follow the convention that constructors start with an upper-case letter, while functions that directly return literal objects or closures should not.

The Ugly

- No standard for setting the prototype of an object
  - __proto__ is browser specific
  - No setter in metaprogramming API
- Single prototype chain
  - No multiple delegation
- new Object style confuses programming styles
  - Simulates class-based programming
  - Use Object.create() instead!

Some browsers will let you set the prototype of an object by updating the `__proto__` property, but this is not standard.

The `new` keyword that lets you create objects from constructors that simulate classes seems to be an attractive way of translating class-based designs to JS, but it is really just a hack. The prototype-based style offers an alternative way of programming that is very expressive. Use `Object.create()` instead of `new` to stick to this style.

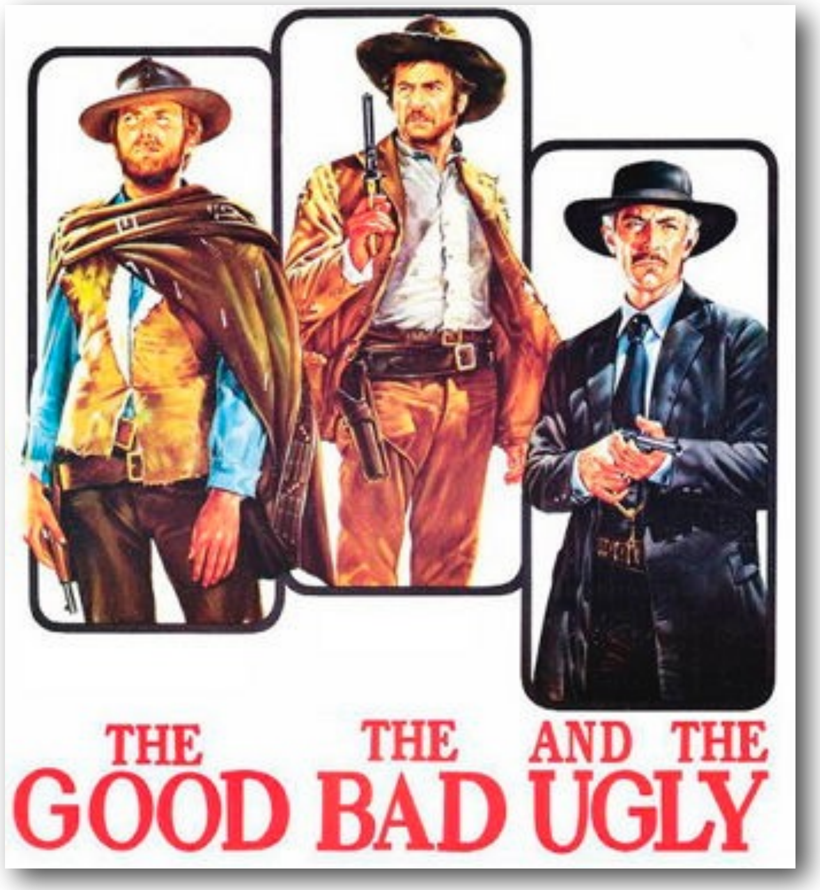
What you should know!

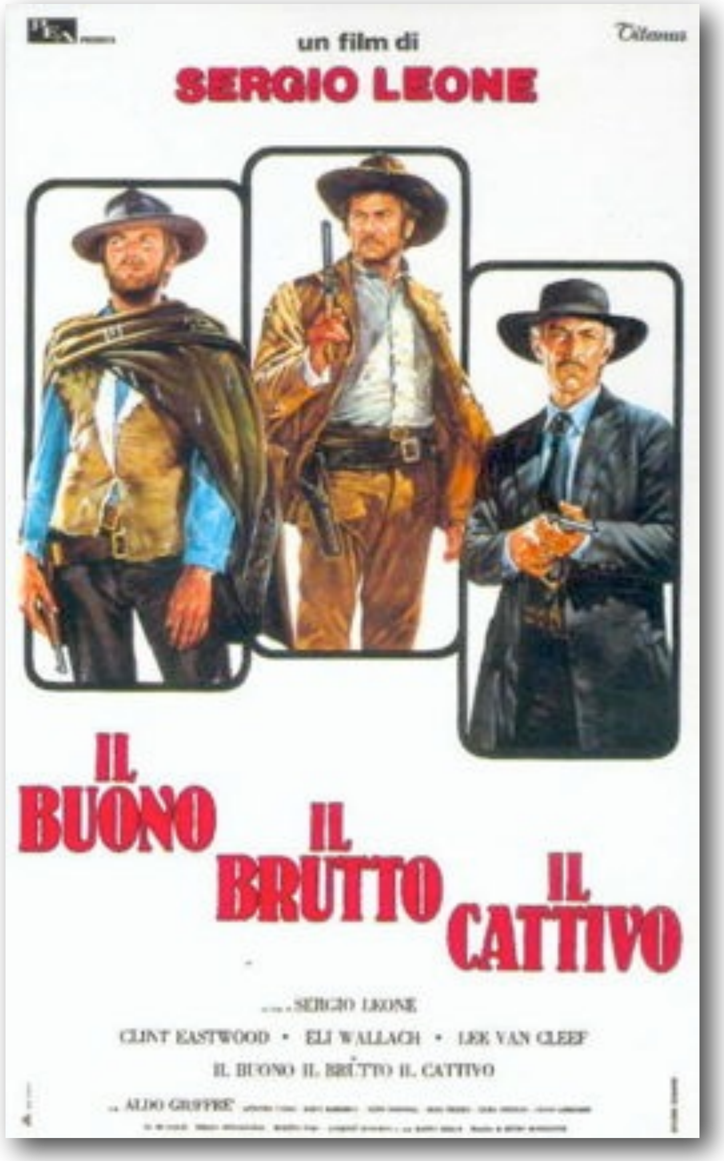
> What is the difference between delegation and inheritance?
> Which object is modified when changing the value of a property within a delegated method?
> How do you extend all objects created with a specific constructor?
> Where do you define properties that are shared between a group of objects (i.e., static members in Java)?
> How does variable scoping work?
> What is a closure?

Can you answer these questions?

> What is the prototype of the global object Function?
> How would you implement private static properties?

Which one is "bad" and which is "ugly"?




un film di
SERGIO LEONE
IL BUONO
IL BRUTTO
IL CATTIVO
SERGIO LEONE
CLINT EASTWOOD • ELI WALLACH • LEE VAN CLEEF
IL BUONO IL BRUTTO IL CATTIVO
ALDO GIUFFRÈ

creative commons

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/