

JAVASCRIPT



Fashionable and Functional!

JAVASCRIPT

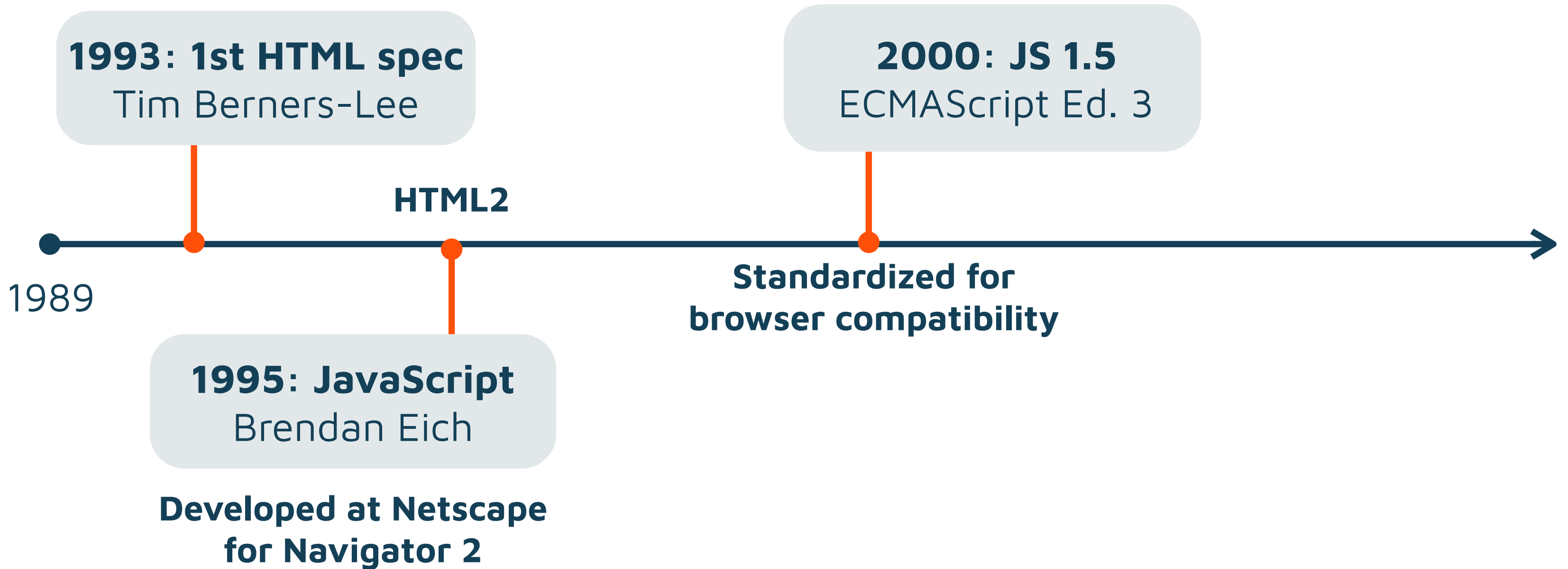
popular scripting language on the Web,
supported by browsers

separate scripting from structure (HTML)
and presentation (CSS)

client- and server-side programming

object-oriented, imperative, functional

Timeline



JAVA : JAVASCRIPT ::



VARIABLES

Dynamically typed: types associated with values, not with variables

Use **var** to define local variables

variables defined implicitly through assignment have global scope

BASIC DATA TYPES

Booleans: **true**, **false**

Number: no integers, 64-bit floating point

String: no char, variable-length

Special Types: **null**, **undefined**



get familiar with String methods

ARRAYS

```
var classes = new Array();
```

```
classes[3] = 'cs498rk';
```

```
var numbers = [5,3,2,6];
```

```
numbers.length;
```

```
other methods: push, pop, sort, ...
```

OBJECTS

collection of properties: name-value pairs

```
llama = {color: 'brown', age:7, hasFur: true};
```

add new properties on the fly

```
llama.family = 'camelid';
```


At first blush...

```
var sum = 0;  
var numbers = [5,3,2,6];  
for (var i=0;i<numbers.length;i++) {  
    sum += numbers[i];  
}
```

...everything seems typical

Functions are first-class objects!

FUNCTIONS ARE OBJECTS

that are callable!

reference by variables, properties of objects

pass as arguments to functions

return as values from functions

can have properties and other functions

DECLARATION

```
function eat() {...}  
  
var sleep = function() {...}  
  
var play = function stop() {...}  
  
console.log(eat.name);  
  
console.log(sleep.name);  
  
console.log(play.name);
```

what will this print?

DECLARATION

name

`function eat() {...}`

`var sleep = function() {...}`

anonymous function


ANONYMOUS FUNCTIONS

create a function for later use

store it in a variable or method of an object

use it as a callback



see more examples next class

SORT COMPARATOR

```
var inventory =[
  {product: "tshirt", price: 15.00},
  {product: "jacket", price: 35.00},
  {product: "shorts", price: 10.00}
]

inventory.sort(function(p1,p2) {
  return p1.price-p2.price;
});
```

VARIABLE NUMBER OF ARGUMENTS

functions handle variable number of arguments

excess arguments are accessed with **arguments** parameter

unspecified parameters are **undefined**

this

the other implicit parameter

a.k.a. **function context**

object that is implicitly associated
with a function's invocation

defined by how the function is
invoked (not like Java)

FUNCTION INVOCATION

```
function eat() {return this;}
```

```
eat();
```

```
var sleep = function()  
{return this;}
```

```
sleep();
```

this refers to the global object

METHOD INVOCATION

```
function eat() {return this;}
```

```
var llama = {  
  graze: eat  
};
```

```
var alpaca = {  
  graze: eat  
};
```

this refers to the object

```
console.log(llama.graze()===llama); true
```

```
console.log(alpaca.graze()===alpaca); true
```

`apply()` *and* `call()`

two methods that exist for every function

explicitly define function context

`apply(functionContext, arrayOfArgs)`

`call(functionContext, arg1, arg2, ...)`

```
function forEach(list, callback) {  
    for (var n = 0; n < list.length; n++) {  
        callback.call(list[n], n);  
    }  
}
```

```
var numbers = [5, 3, 2, 6];  
forEach(numbers, function(index) {  
    numbers[index] = this * 2; });  
console.log(numbers);
```

don't need multiple copies of a function
to operate on different kinds of objects!

```
function forEach(list, callback) {  
  for (var n = 0; n < list.length; n++) {  
    callback.call(list[n], n);  
  }  
}
```

```
var camelids = ["llama", "alpaca", "vicuna"];  
forEach(camelids, function(index) {  
  camelids[index] = this+this; });  
console.log(camelids);
```

Classes are defined through
functions!

OBJECT-ORIENTED PROGRAMMING

new operator applied to a constructor function
creates a new object

no traditional class definition

newly created object is passed to the
constructor as this parameter, becoming the
constructor's function context

constructor returns the new object

CONSTRUCTOR INVOCATION

constructors are given the class name

```
function Llama() {  
  this.spitted = false;  
  this.spit = function() { this.spitted = true; }  
}
```

```
var llama1 = new Llama();  
llama1.spit();  
console.log(llama1.spitted); true
```

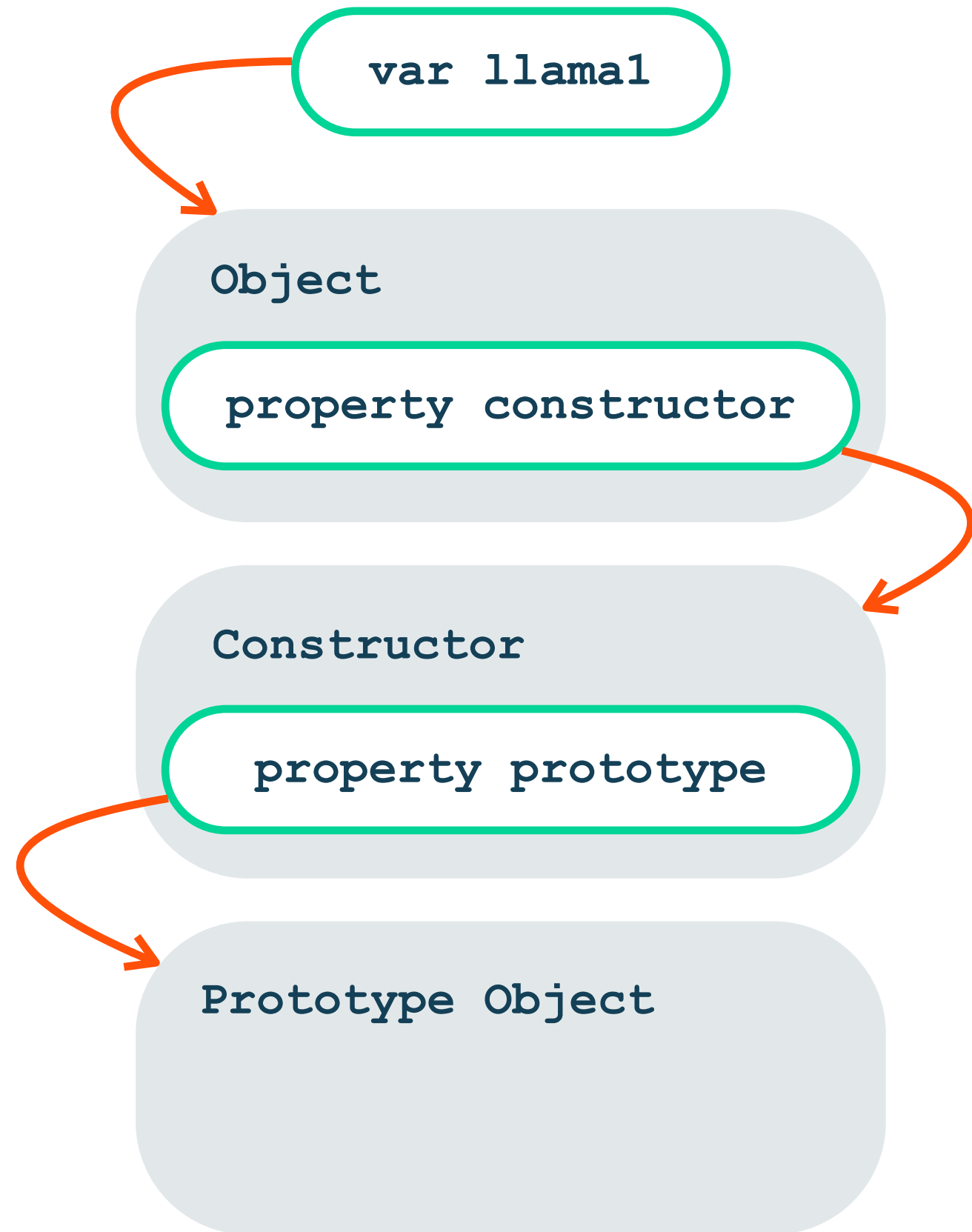
```
var llama2 = new Llama();  
console.log(llama2.spitted); false
```

prototype

prototype is a property of the constructor
another way to add methods to objects

```
function Llama() {  
    this.spitted = false;  
}  
Llama.prototype.spit = function() {  
    this.spitted = true;  
};
```

```
function Llama() {  
    this.spitted = false;  
    this.spit = function() { this.spitted = true; }  
}  
Llama.prototype.spit = function() {  
    this.spitted = false;  
};  
var llama1 = new Llama();  
llama1.spit();  
console.log(llama1.spitted); true
```



binding operations
within the constructor
always take
precedence over those
in its prototype

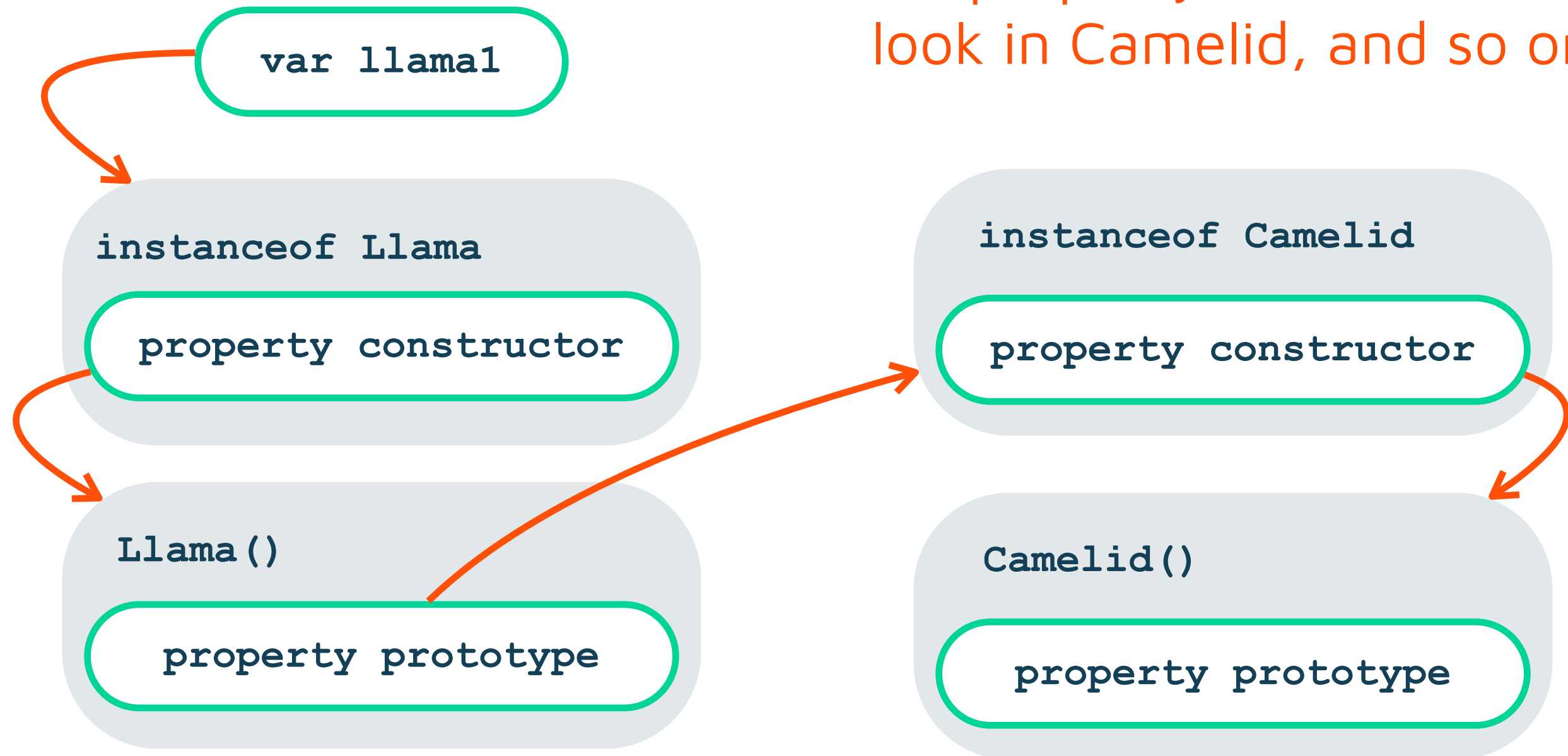
INHERITANCE

create prototype as instance of parent class

```
Llama.prototype = new Camelid();
```

PROTOTYPE CHAINING

if a property isn't in Llama, look in Camelid, and so on



Scoping

SCOPE

```
function outerFunction() {  
    var x = 1;  
    function innerFunction() {...}  
    if(x==1) {var y=2;}  
    console.log(y);    what will it print?  
}  
  
outerFunction();
```

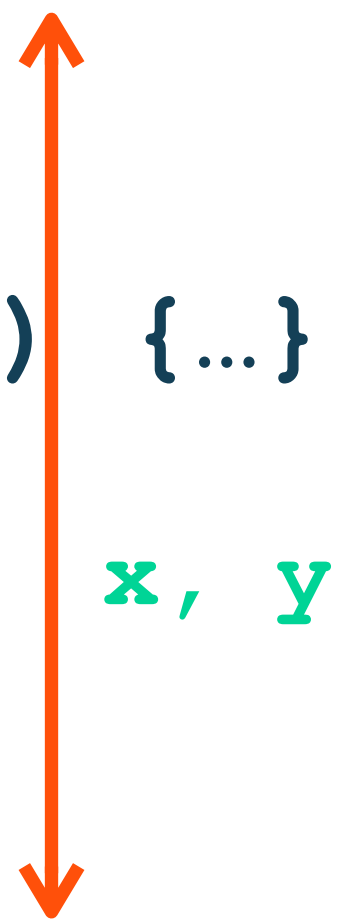

scopes are declared through
functions and not blocks `{}`

HOISTING

Variables and functions are in scope within the entire function they are declared in

SCOPE

```
function outerFunction() {  
    var x = 1;  
    function innerFunction() {...}  
    if(x==1) {var y=2;}  
    console.log(y);  
}  
  
outerFunction();
```



A diagram illustrating variable scope resolution. A vertical orange double-headed arrow spans from the opening curly brace of the `outerFunction()` to the closing curly brace. To the right of the arrow, the text `x, y` is written in green, indicating that both variables are resolved within the scope of `outerFunction()`.

SCOPE

```
function outerFunction() {  
    var x = 1;  
    function innerFunction() {...}  
    if(x==1) {var y=2;}  
    console.log(y);  
}  
outerFunction();
```

The diagram illustrates the scope of the functions in the code. Two orange arrows originate from the right side of the code. The first arrow starts at the opening curly brace of the `innerFunction` definition and points upwards to the word `innerFunction` in green text. The second arrow starts at the closing curly brace of the `outerFunction` definition and points downwards to the word `outerFunction` in green text.

HOISTING

```
function outerFunction() {  
    var x = 1;  
    console.log(y);    what will it print?  
    if (x==1) {var y=2;}  
}  
  
outerFunction();      initializations are not hoisted!
```

closure *scope created when a function is declared that allows the function to access and manipulate variables that are external to that function*

CLOSURES

access all the variables (including other functions) that are in-scope when the function itself is declared

inner function has access to state of its outer function even after the outer function has returned!

Closure Example

```
var outerValue = 'llama';  
var later;  
function outerFunction() {  
    var innerValue = 'alpaca';  
    function innerFunction() {  
        console.log(outerValue);  
        console.log(innerValue);  
    }  
    later = innerFunction;  
}  
outerFunction();  
later();
```

what will this print?

Closure Example

```
var outerValue = 'llama';  
var later;  
function outerFunction() {  
    var innerValue = 'alpaca';  
    function innerFunction() {  
        console.log(outerValue);  
        console.log(innerValue);  
    }  
    later = innerFunction;  
}  
outerFunction();  
later();
```

prints:

llama

alpaca

innerFunction has
access to **innerValue**
through its closure

Closure of innerFunction

```
var outerValue = 'llama';  
var later;  
function outerFunction() {  
  var innerValue = 'alpaca';  
  function innerFunction() {  
    console.log(outerValue);  
    console.log(innerValue);  
  }  
  later = innerFunction;  
}  
outerFunction();  
later();
```

function()
innerFunction
{...}

function
outerFunction

var outerValue

var innerValue

var later

Closure Example

```
var later;  
  
function outerFunction() {  
    function innerFunction(paramValue) {  
        console.log(paramValue);  
        console.log(afterValue);  
    }  
  
    later = innerFunction;  
}
```

what will this print?

```
var afterValue = 'camel';  
  
outerFunction();  
  
later('alpaca');
```

Closure Example

```
var later;  
  
function outerFunction() {  
    function innerFunction(paramValue) {  
        console.log(paramValue);  
        console.log(afterValue);  
    }  
    later = innerFunction;  
}  
  
var afterValue = 'camel';  
outerFunction();  
later('alpaca');
```

prints:
alpaca
camel

Closure Example

```
var later;  
  
function outerFunction() {  
    function innerFunction(paramValue) {  
        console.log(paramValue);  
        console.log(afterValue);  
    }  
    later = innerFunction;  
}
```

Closures include:

Function parameters

All variables in an
outer scope

```
var afterValue = 'camel';  
outerFunction();  
later('alpaca');
```



*declared after the
function declaration!*

SELF-INVOKING FUNCTIONS

```
var add = (function () {  
    var counter = 0;  
  
    return function () {return  
        counter += 1;}  
  
})();  
  
add();
```

self-invoking

PRIVATE VARIABLES

```
function Llama () {
```

```
  var spitted = false;
```

```
  this.spit = function() { spitted =  
    true; }
```

```
  this.hasSpitted = function() { return  
    spitted; }
```

```
}
```

private data member now!

CURRYING

partial evaluation of functions

```
function curriedAdd(x) {  
  return function(y) {  
    return x+y;  
  };  
};  
  
var addTwo = curriedAdd(2) ;  
addTwo(3) ;
```


NEXT CLASS:

JAVASCRIPT *and the Web*

<https://uiuc-web-programming.gitlab.io/fa21/>