JAVASCRIPT

the language

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

CREATING FUNCTIONS

name

Declaration: `function eat() {…}`

Expression: `var sleep = function() {…}`

anonymous function

VARIABLE NUMBER OF ARGUMENTS

functions handle variable number of arguments

excess arguments are accessed with **`arguments`** parameter

unspecified parameters are **`undefined`**

VARIABLE NUMBER OF ARGUMENTS

```
function power(base, exponent){
    if (exponent == undefined){
        exponent = 2;
    }

    …

}
```

```
power(3,2)

   //arguments.length -> 2
   //arguments[0] -> 3
```

Scoping

SCOPE

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

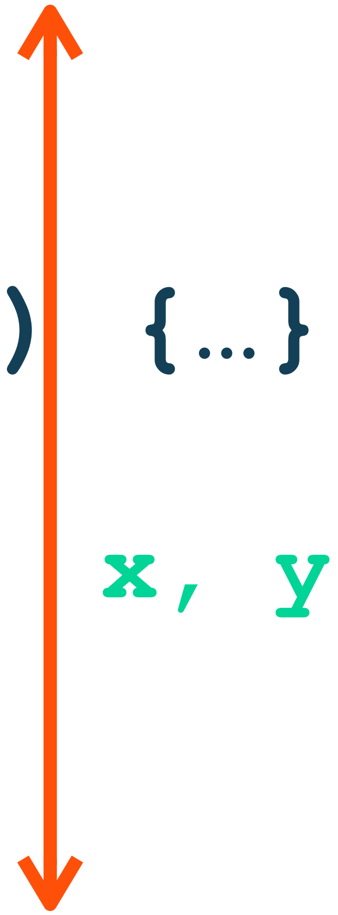
scopes are declared through functions and not blocks { }

HOISTING

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

SCOPE

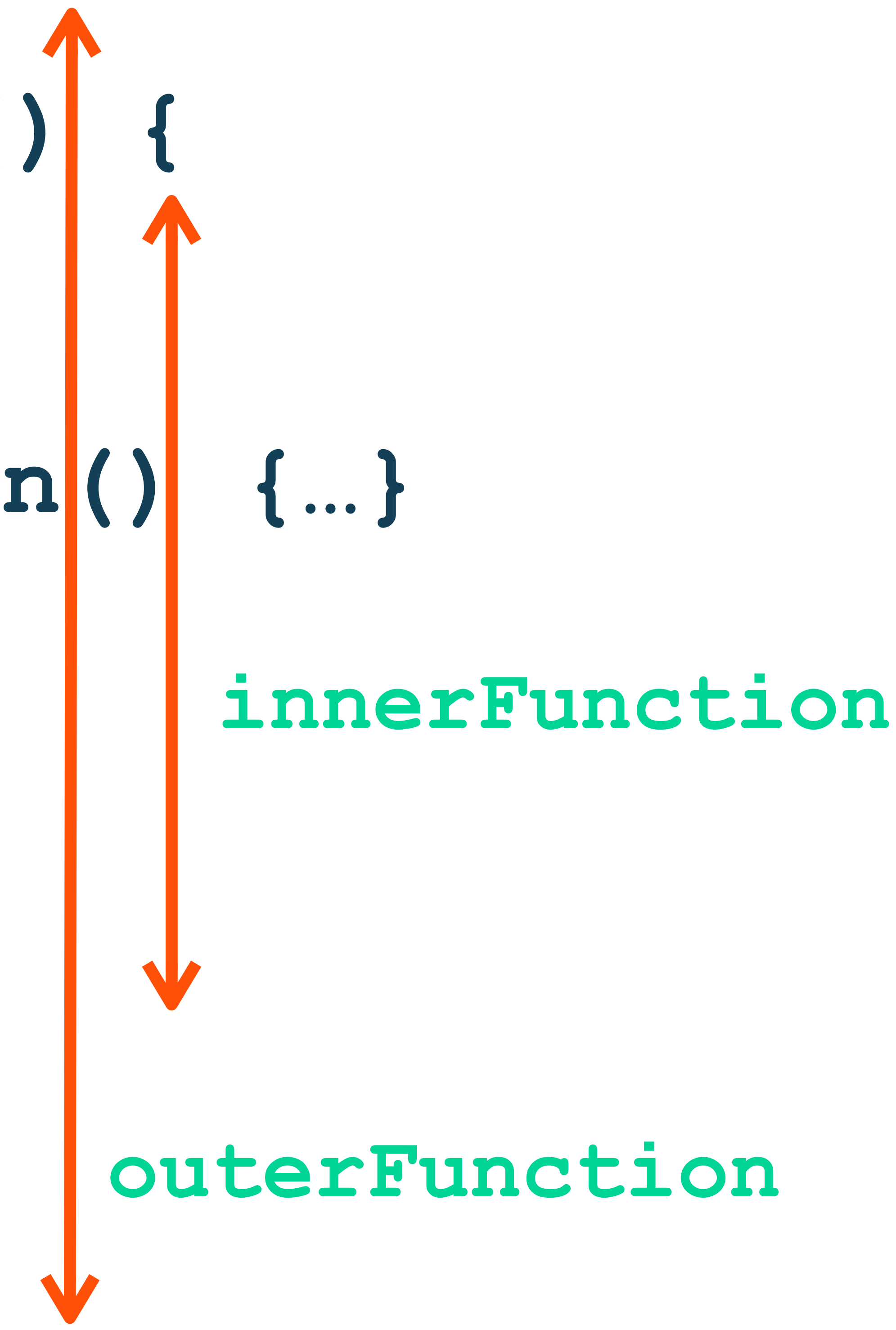
```
function outerFunction() {
  var x = 1;
  function innerFunction() {…}
  if(x==1) {var y=2;}
  console.log(y);
}
outerFunction();
```



x, y

SCOPE

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



HOISTING

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

what will it print?

initializations are not hoisted!

Before you came into my life

I missed you so bad...

```
function foo() {
 x = 10;
 var bar = x + 5;
 var x;
}
```

Because Fuck Logic

CREATING FUNCTIONS

Declaration:

```
function eat() {…}
```

Expression:

```
var sleep = function() {…}
```

Declarations are hoisted. Expressions are not.

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

`this` refers to the object

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

two methods that exist for every function

explicitly define function context

```
fn.apply(functionContext,arrayOfArgs)
```

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

CODEPEN

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

implemented in Javascript 1.6

```
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);
```

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

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

Classes are defined through functions

OBJECT-ORIENTED PROGRAMMING

`new` operator applied to a function (called constructor) creates an 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 generally start with uppercase
(think of this as a 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
```

```
var empty = {};
console.log(empty.x);  undefined

console.log(empty.toString());    [object Object]
```

Where did toString come from?

`prototype`

In addition to their properties, all objects have another object called a *prototype*.

When an object does not have a requested property, its prototype is searched, then the prototype’s prototype, and so on.

prototype

```
console.log(Object.getPrototypeOf({}) == Object.prototype)    true
```

contains the toString property

SPECIFYING PROTOTYPES

```
var protoLlama = {
  spit: function(){
    this.spit = true;
  }
}

var llama = Object.create(protoLlama);
```

SPECIFYING PROTOTYPES USING THE CONSTRUCTOR

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

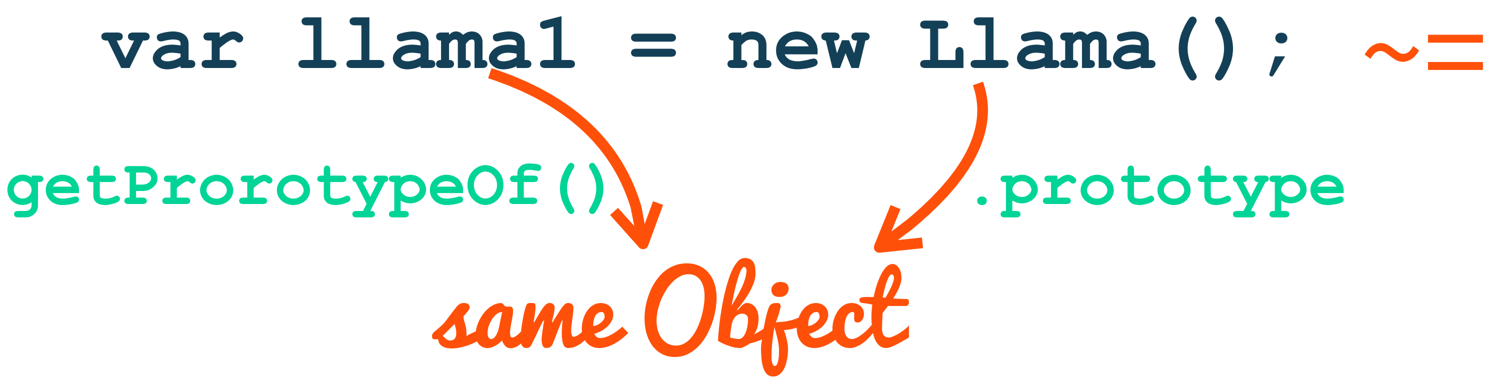
All objects created using this constructor will have a prototype that can be accessed with a property of this function: Llama.prototype

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

this adds the spit function to the prototypes of all Llama instances

SPECIFYING PROTOTYPES USING THE CONSTRUCTOR

What is the prototype of Llama instances?



```
var llama1 = new Llama(); ~=
```

```
var llama1 = new Object();
llama1.[[Prototype]] = Llama.prototype
Llama.call(llama1);
```

CODEPEN

What is the prototype of Llama (the constructor)?

```
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
```

Properties present in the prototype can be overridden

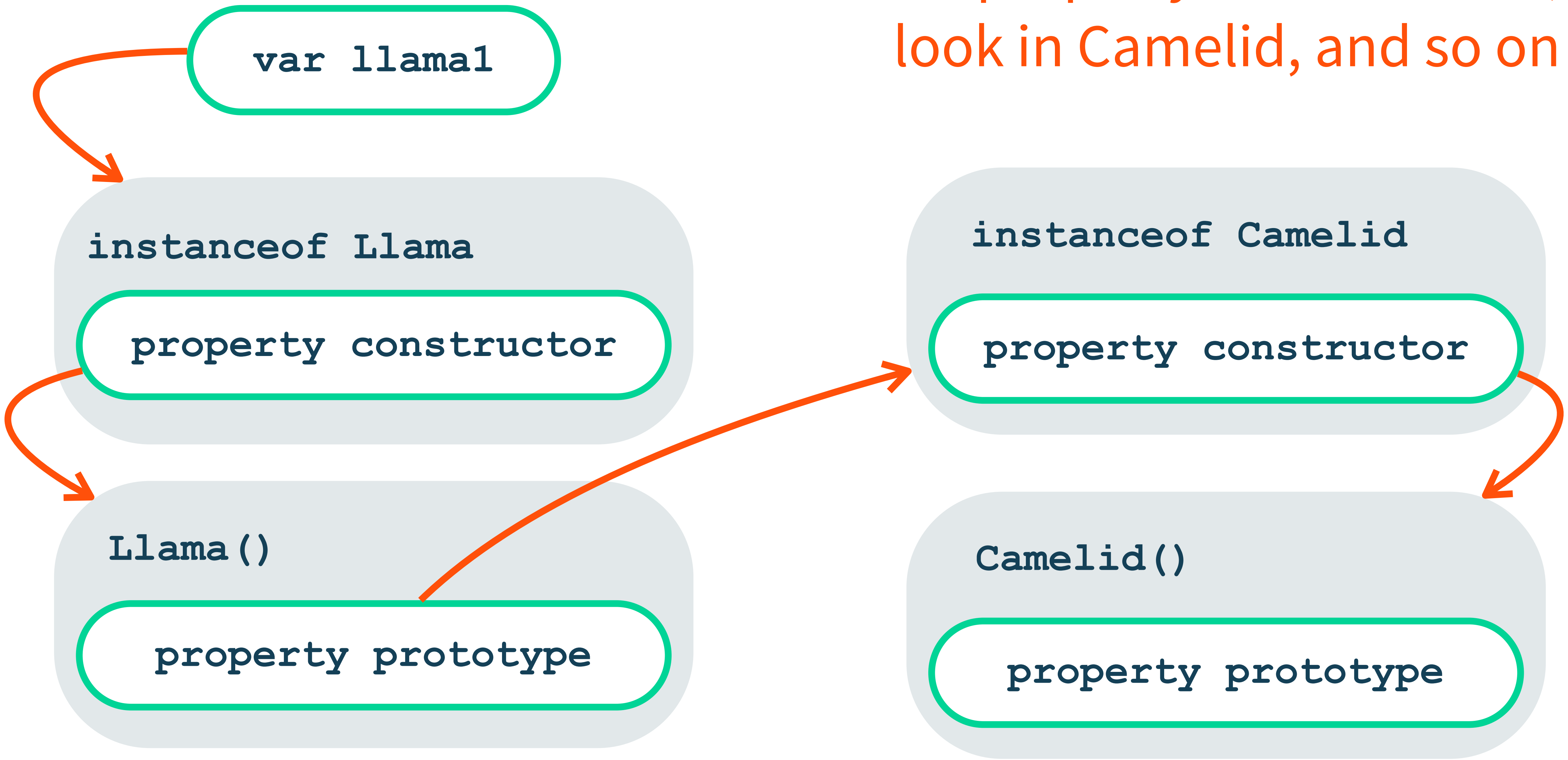
INHERITANCE

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



`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:

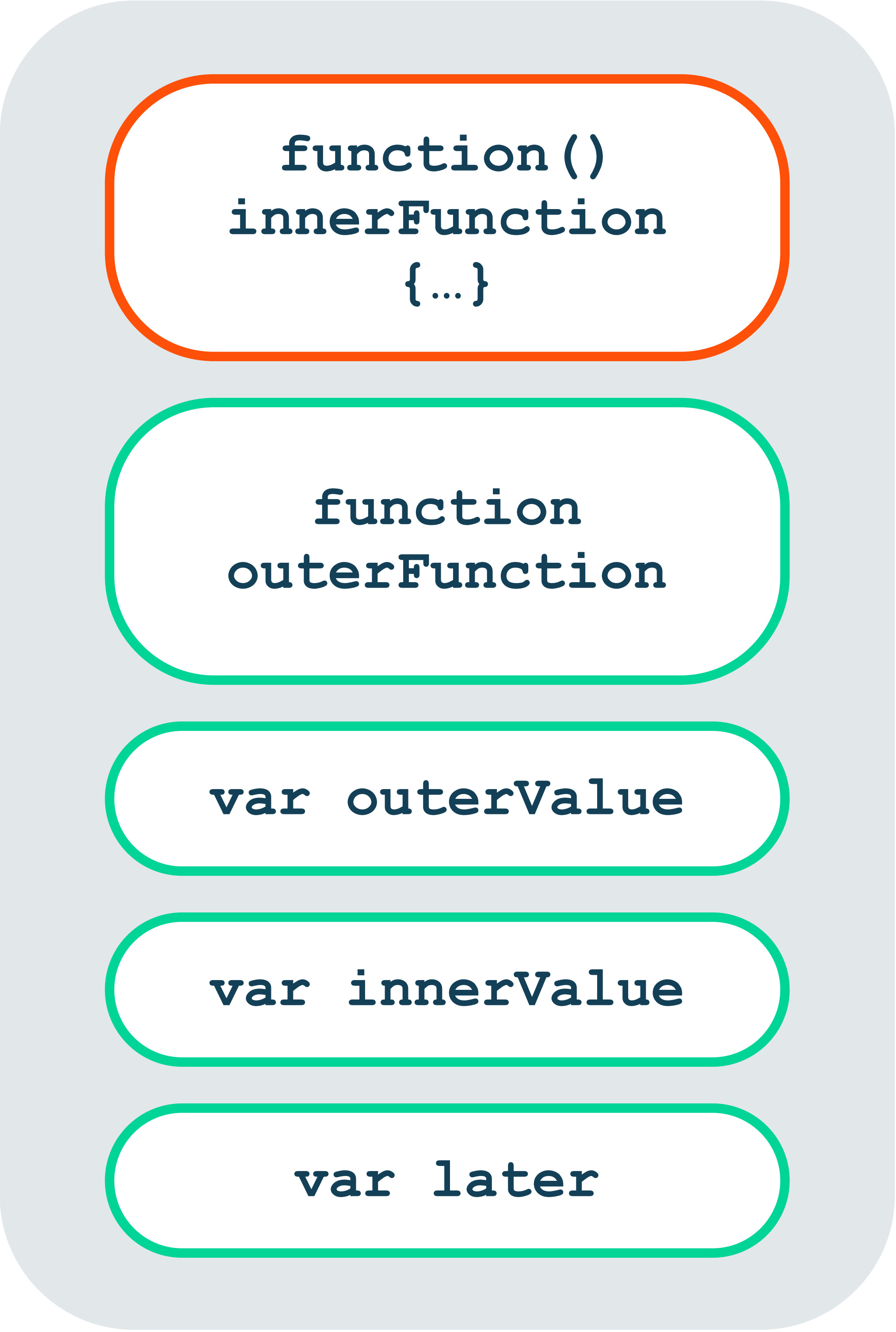
```
llama
alpaca
```

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

I just met you, and this is crazy

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

Closure of innerFunction



Closure Example

CODEPEN

Closure Example

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

what will this print?

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;
}
var afterValue = ‘camel’;
outerFunction();
later(‘alpaca’);
```

Closures include:

Function parameters

All variables in an outer scope

declared after the function declaration!

PRIVATE VARIABLES

```
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);
var addFive = curriedAdd(5);

addTwo(3);
```

Event Example 1

CODEPEN

Anonymous Functions

```
function animateIt(elementId, speed) {
  var elem = document.getElementById(elementId);
  tick = 0;
  var timer = setInterval(function() {
    if (tick <100) {
      elem.style.left = tick*speed + "px";
      tick++;
    }
    else {clearInterval(timer);}
    }, 30);
}
```

Closures

```
function animateIt(elementId, speed) {
  var elem = document.getElementById(elementId);
  tick = 0;
  var timer = setInterval(function() {
    if (tick <100) {
      elem.style.left = tick*speed + "px";
      tick++;
    }
    else {clearInterval(timer);}
    }, 30);
}
```

TIPS & TRICKS

Scoping cheatsheet

developers.google.com/speed/articles/optimizing-javascript

jonraasch.com/blog/10-javascript-performance-boosting-tips-from-nicholas-zakas