CSC 470 – Section 3

Topics in Computer Science: Advanced Browser Technologies

Mark F. Russo, Ph.D.

Spring 2016

Lecture 8

Eloquent JavaScript: Chapter 6

Functions and Context

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Ex 1</title>
  </head>
  <body>
    <button type="button" id="b1" name="George">Click Me</button>
    <button type="button" id="b2" name="Stuart">Click Me Too</button>

    <script type="text/javascript">
      var speak = function(msg) {
        console.log(this.name + " says '" + msg + "'");
      }

      var name = "Paul";
      window.name = "John";
      var beatle = {name: "Ringo", action: speak};

      speak("I am the walrus");
      beatle.action("It Don’t Come Easy");

      document.getElementById('b1').onclick = speak;
      document.getElementById('b2').onclick = beatle.action;
    </script>
  </body>
</html>
```

Output?

08/ex1.html

Functions and Execution Context

```
John says 'I am the walrus'
Ringo says 'It Don’t Come Easy'
George says '[object MouseEvent]'
Stuart says '[object MouseEvent]'
```

Things we know…

- "Global" variables are properties of the Global Object
- Functions executed in the global context have '`this`' assigned to the Global Object
- Functions assigned as object properties and executed using dot-notation are executed in the object's context
- Just because a function is a property of an object, does not mean it will execute within the context of the object
- Functions assigned as event listeners execute with the event dispatcher object as its context

Recall: Setting Function Context

- The context of a function can be set if invoked using the `call()` or `apply()` methods of a Function object.
- In either case the value of `this` within the Function object will be set.
- Non-local variables referenced within the Function are expected to be properties of the `this` context

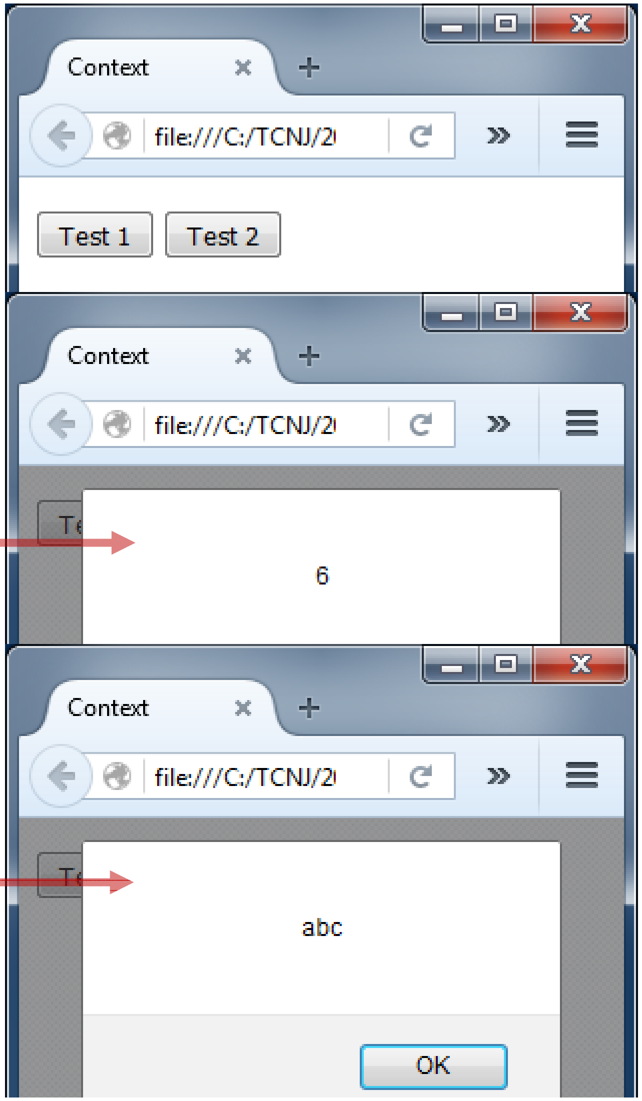
```
fun.call(  thisArg[, arg1[, arg2[, ...]]] )

fun.apply( thisArg, [arg1, ..., argN] );
```

Recall: Custom Function Context

Any object may be provided to `call()` or `apply()` as a function context, including a custom Object.

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Context</title>
    <script type="text/javascript">

      var addABC = function() {
        return this.A + this.B + this.C;
      };

      var test1 = function() {
        var ob = {'A':1, 'B':2, 'C':3};
        var result = addABC.call( ob );
        alert(result);
      }

      var test2 = function() {
        var ob = {'A':'a', 'B':'b', 'C':'c'};
        var result = addABC.call( ob );
        alert(result);
      }
    </script>
  </head>
  <body>
    <p>
      <button onclick="test1();">Test 1</button>
      <button onclick="test2();">Test 2</button>
    </p>
  </body>
</html>
```

05/context.html



```
6
```

```
abc
```

Permanently assigning (binding) `this`

- The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments

Syntax

```
fun.bind( thisArg[, arg1[, arg2[, ...]]] )
```

Important Points

- A new function is created; the existing function is not affected
- `this` cannot be overridden
- Unlike `.call()` and `.apply()`, `.bind()` is permanent

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

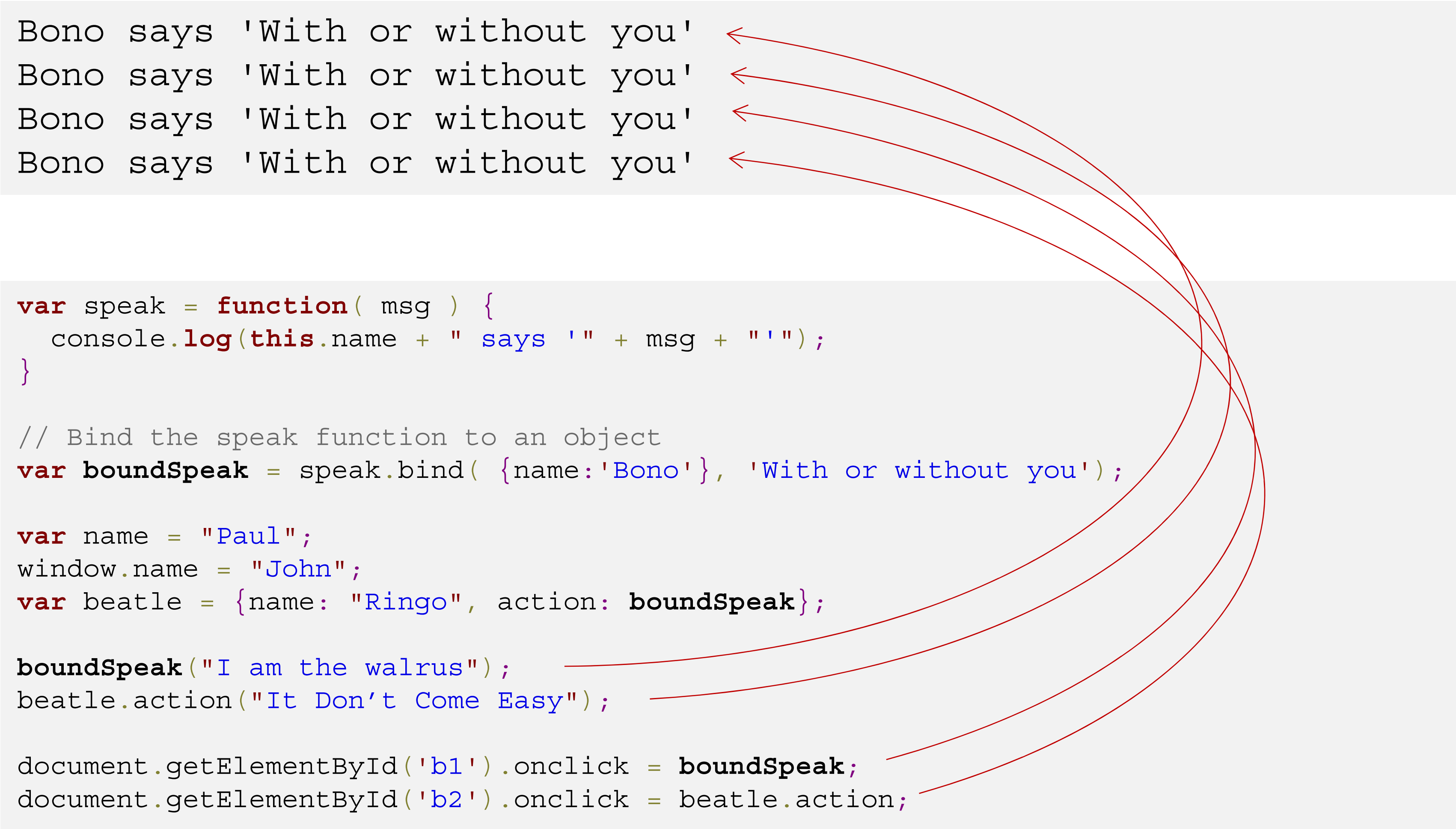
Functions and Bound Context

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Ex 2</title>
  </head>
  <body>
    <button type="button" id="b1" name="George">Click Me</button>
    <button type="button" id="b2" name="Stuart">Click Me Too</button>

    <script type="text/javascript">
      var speak = function( msg ) {
        console.log(this.name + " says '" + msg + "'");
      }

      // Bind the speak function to an object
      var boundSpeak = speak.bind( {name:'Bono'}, 'With or without you');

      var name = "Paul";
      window.name = "John";
      var beatle = {name: "Ringo", action: boundSpeak};

      boundSpeak("I am the walrus");
      beatle.action("It Don’t Come Easy");

      document.getElementById('b1').onclick = boundSpeak;
      document.getElementById('b2').onclick = beatle.action;

    </script>
  </body>
</html>
```

Output?

08/ex2.html

Functions and Bound Context



```
Bono says 'With or without you'
Bono says 'With or without you'
Bono says 'With or without you'
Bono says 'With or without you'
```

```
var speak = function( msg ) {
  console.log(this.name + " says '" + msg + "'");
}

// Bind the speak function to an object
var boundSpeak = speak.bind( {name:'Bono'}, 'With or without you');

var name = "Paul";
window.name = "John";
var beatle = {name: "Ringo", action: boundSpeak};

boundSpeak("I am the walrus");
beatle.action("It Don’t Come Easy");

document.getElementById('b1').onclick = boundSpeak;
document.getElementById('b2').onclick = beatle.action;
```

08/ex2.html

Avoiding Context Ambiguity

We can avoid "`this` ambiguity" by creating objects with bound methods

```
// Compute the length of a vector
var vector_length = function() {
  return Math.sqrt( this.x*this.x + this.y*this.y );
};

// Print a vector
var print_vector = function() {
  console.log( this.name, this.x, this.y, this.length() );
};

// Function to create a new vector with bound methods
var new_Vector = function(name, x, y) {
  var v = {name:name, x:x, y:y};      // New object with properties
  v.length = vector_length.bind(v);   // Bound length() method
  v.print  = print_vector.bind(v);    // Bound print() method
  return v;                           // Return new object
};
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

08/ex3.html

Avoiding Context Ambiguity

We can avoid "`this` ambiguity" by creating objects with bound methods

```
// Create new vectors and print
var v1 = new_Vector('v1', 3, 4);
v1.print();

var v2 = new_Vector('v2', 5, 12);
v2.print();

// Invoke object methods out of object context
var print1 = v1.print;
var print2 = v2.print;
print1();

// Attach event listener
document.getElementById('b1').onclick = print2;
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

08/ex3.html

Avoiding Context Ambiguity

We can avoid "`this` ambiguity" by creating objects with bound methods

```
// Create new vectors and print
var v1 = new_Vector('v1', 3, 4);
v1.print();

var v2 = new_Vector('v2', 5, 12);
v2.print();

// Invoke object methods out of object context
var print1 = v1.print;
var print2 = v2.print;
print1();

// Attach event listener
document.getElementById('b1').onclick = print2;
```

```
v1 3 4 5
v2 5 12 13
v1 3 4 5
v2 5 12 13
```

What do you think?

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

08/ex3.html

Constructors and Built-in Objects

- Recall the two ways we were able to invoke built-in objects:

  1. As a function - used to perform type conversion
  2. As a constructor - used to create new objects

- This dichotomy explains the differing output from the `typeof` operator

```
// Create a number using Number() for type conversion
var n1 = Number("2");

// Create a number Object using the Number constructor
var n2 = new Number("2");

// Inspect type
console.log( typeof n1 );  // -> "number"
console.log( typeof n2 );  // -> "object"
```

Creating User-defined Objects

- In JavaScript, a user-defined class-like thing is defined using a single function called a "constructor"
- Constructor functions are executed in the context of a newly created object bound to `this`. ***This new object is created automatically***.
- For that to happen, a constructor function must be invoked with the `new` operator
- The newly created and initialized object is returned from invoking the function with `new` automatically (unless something else is returned)

Two steps required to define and create user-defined objects

1. Define the object type by writing a (constructor) function.
2. Create an instance of the object with the `new` operator.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new

Creating User-defined Objects

```
// Vector constructor function
var Vector = function(name, x, y) {
  // Vector properties
  this.name = name;
  this.x = x;
  this.y = y;

  // Compute the length of a vector
  this.length = function() {
    return Math.sqrt( this.x*this.x + this.y*this.y );
  };

  // Print a vector
  this.print = function() {
    console.log( this.name, this.x, this.y, this.length() );
  };
};

// Create new vectors and print
var v1 = new Vector('v1', 3, 4);
v1.print();

var v2 = new Vector('v2', 5, 12);
v2.print();
```

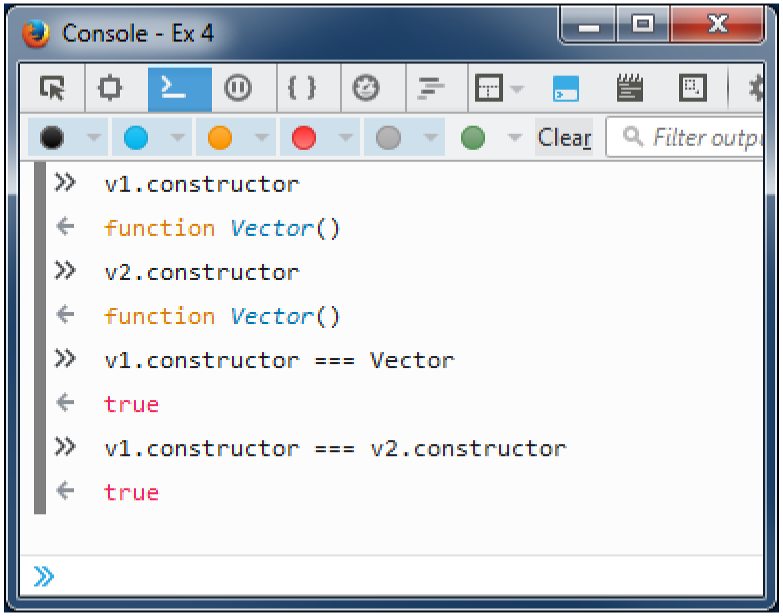
1. Define constructor function. `this` is bound to new object. Assign new properties as necessary.

2. Create new instances by invoking constructor using the `new` operator

08/ex4.html

`constructor` Property

- When a constructor function is invoked, and a new object is created, the object's `constructor` property is set to a reference to the constructor function
- Objects created from the same constructor function have identical `constructor` property values
- The `constructor` property is one way to determine if objects are of a type
- Not useful for inheritance (later)



```
» v1.constructor
← function Vector()
» v2.constructor
← function Vector()
» v1.constructor === Vector
← true
» v1.constructor === v2.constructor
← true
```

Question...

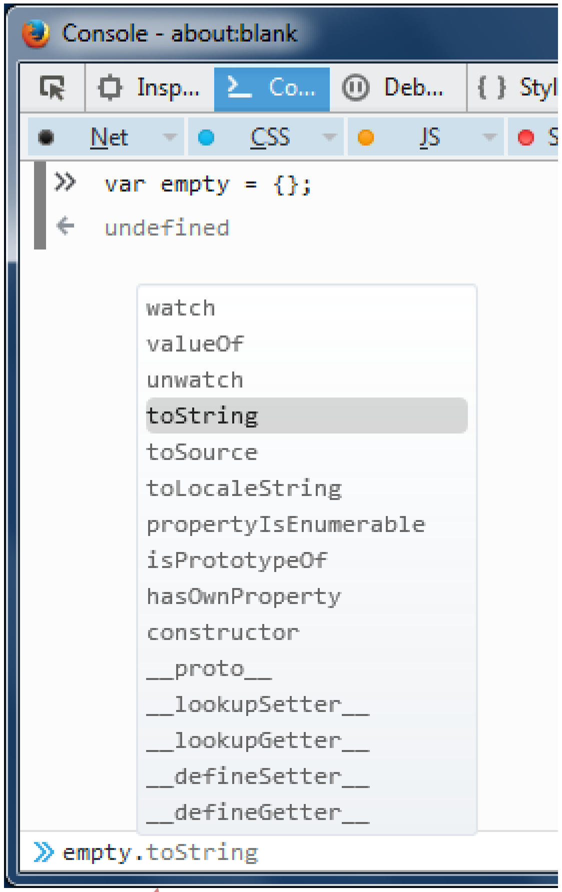
```
var empty = {};

console.log( empty.toString );
// → function toString(){…}

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

- How can an empty object have a `.toString()` method?
- Where did `.toString()` come from?

Question…

```
var empty = {};

console.log( empty.toString );
// → function toString(){…}

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

- How can an empty object have a `.toString()` method?
- Where did `.toString()` come from?



```
» var empty = {};
← undefined
» empty.toString
```

- There are more methods other than `.toString()`

JavaScript: A Prototype-based Language

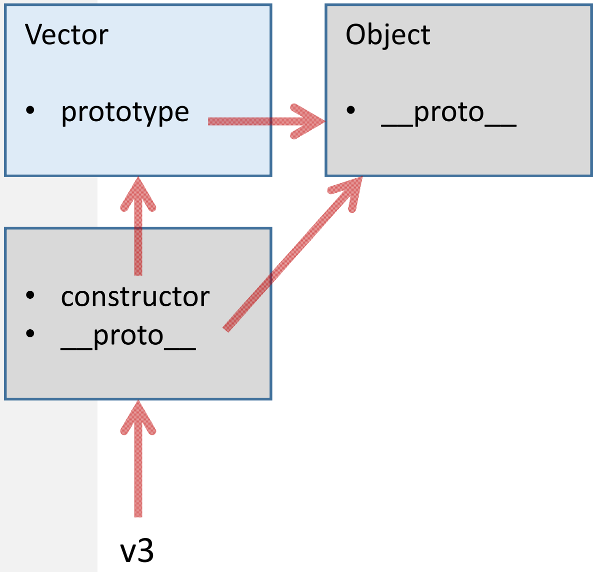
- There are no classes in JavaScript – objects "inherit" from other objects
- A prototype is an object from which another object inherits properties
- (Nearly) all objects have a prototype (there are rare exceptions)
- An object's prototype can be accessed with the `Object.getPrototypeOf(…)` method (also the non-standard `__proto__` property)

```
Object.getPrototypeOf(1)
[object Number]
Object.getPrototypeOf(true)
[object Boolean]
Object.getPrototypeOf("")
[object String]
Object.getPrototypeOf(parseFloat)
function ()
Object.getPrototypeOf([])
Array [  ]
Object.getPrototypeOf({})
Object { , 15 more… }
```

prototype of a New Object

- Functions have a special property called `prototype`
- *When an object is created by invoking a constructor function with* `new`*, the newly created object's prototype is set to the constructor function's* `prototype` *property value*
- This prototype object provides inherited properties for the new object
- `__proto__` is a non-standard reference to an object's prototype

```
var v3 = new Vector('v3', 30, 20);

Vector.prototype
Object { , 1 more… }

Object.getPrototypeOf( v3 )
Object { , 1 more… }

v3.__proto__ === Vector.prototype
true

v3.toString()
"[object Object]"
```



Because Object implements `toString()`, v3 now inherits that function as well

08/ex4.html

Adding Properties to a Prototype

A property added to a constructor's prototype, will be inherited by all objects created using the constructor function … even existing ones!

Recall Vector…

```
// Vector constructor function
var Vector = function(name, x, y) {
  // Vector properties
  this.name = name;
  this.x = x;
  this.y = y;

  // Compute the length of a vector
  this.length = function() {
    return Math.sqrt( this.x*this.x + this.y*this.y );
  };

  // Print a vector
  this.print = function() {
    console.log( this.name, this.x, this.y );
  };
};
```

08/ex5.html

Adding Properties to a Prototype

1. Create an instance of the Vector class
2. Add a new method to Vector's prototype
3. Invoke the new method on the existing object

```
// Create new vectors and print
var v1 = new Vector('v1', 3, 4);
v1.print();

// Add a new method to all vector objects
Vector.prototype.scale = function(factor) {
  this.x *= factor;
  this.y *= factor;
};

// Invoke new methods on existing objects
v1.scale(2);
v1.print();
```

```
v1 3 4
v1 6 8
```

Modifying a constructor function prototype is the correct way to define properties that are shared by all objects created using the constructor.

An object's prototype provides properties to be shared by all inherited objects.
-> When an object gets a request for a property that it does not have, its prototype will be searched

08/ex5.html

Adding Properties to a Prototype

Properties can be added through the prototype reference from an object

```
// Create new vectors and print
var v1 = new Vector('v1', 3, 4);
v1.print();

var v2 = new Vector('v2', 5, 6);
v2.print();

// Add methods to prototype through instance
// ... careful. Better to do through Vector
v1.__proto__.add = function( v ) {
  this.x += v.x;
  this.y += v.y;
};

// Add vectors and print result
v2.add( v1 );
v2.print();
```

```
v1 3 4
v2 5 6
v2 8 10
```

Think about all the damage we can do with this approach …
surreptitiously changing properties and the definition of methods!

08/ex6.html

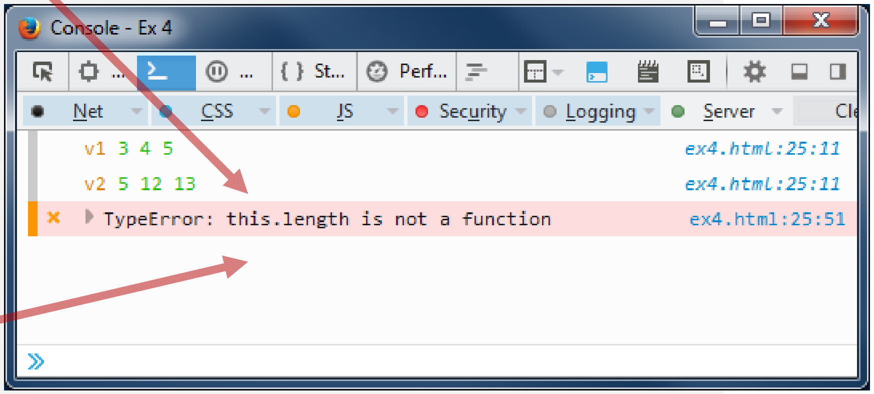
Constructor Summary

Invoking a constructor with `new`…

1. Creates a new object and sets to `this`.
2. Sets the `constructor` property of the new object to the constructor function.
3. Sets the new object's prototype to the constructor function's `prototype` property value.
4. Invokes the constructor function in the context of the new object.

new is not as Strong as bind()

```
…

  // Print a vector
  this.print = function() {
    console.log( this.name, this.x, this.y, this.length() );
  };
};

// Create new vectors and print
var v1 = new Vector('v1', 3, 4);
v1.print();

var v2 = new Vector('v2', 5, 12);
v2.print();

// Invoke object methods out of object context
var print1 = v1.print;
var print2 = v2.print;

print1();
print2();
```



```
v1 3 4 5                                          ex4.html:25:11
v2 5 12 13                                        ex4.html:25:11
✕  TypeError: this.length is not a function       ex4.html:25:51
```

- With `new` and a constructor function, `this` is bound to the new object while the constructor executes…
- When complete, we're back to standard behavior

08/ex7.html

Bound Functions as Constructors

- `bind()` is strong, when it is used to construct a new object, the provided `this` object is ignored.

```
// Bind the vector to an object
var BoundVector = Vector.bind( {}, 'Rick', 0, 0 );

// Create new vectors using bound and unbound constructors
var v1 = new Vector('v1', 1, 2);
var v2 = new BoundVector('v2', 3, 4);
var v3 = new BoundVector('v3', 5, 6);
var v4 = new BoundVector('v4', 7, 8);

v1.print();    // Okay
v2.print();    // Rick-rolled by a bound constructor
v3.print();    // Rick-rolled by a bound constructor
v4.print();    // Rick-rolled by a bound constructor
```

```
v1 1 2
Rick 0 0
Rick 0 0
Rick 0 0
```

08/ex8.html

`Object.create`

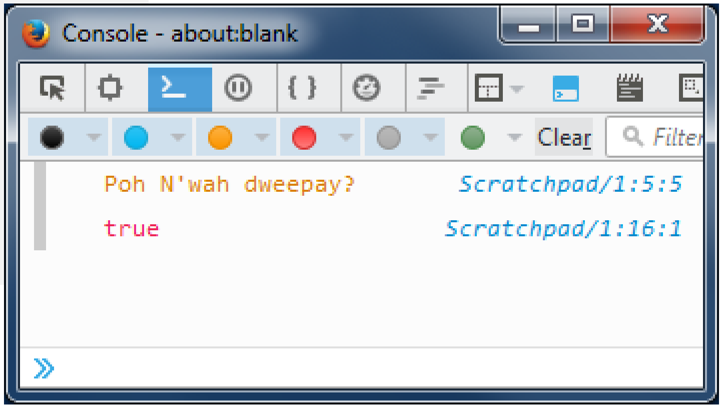
- `Object.create()` creates a new object with the specified prototype object and properties.
- This method can be very useful, because it allows you to **choose the prototype object** for the object you want to create, without having to define a constructor function.

```
Object.create( proto[, propertiesObject ])
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create

Object.create Example

```
// Create a constructor function
var character = {
  name :'Jabba the Hutt',
  speak:function() {
    console.log("Poh N'wah dweepay?");
  }
};

// Create a new object with defined prototype
var o = Object.create(character);

// Invoke inherited method
o.speak();

// Test prototype identity
console.log( o.__proto__ == character );
```



```
Poh N'wah dweepay?          Scratchpad/1:5:5
true                        Scratchpad/1:16:1
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create

08/create.html

Prototypal Inheritance

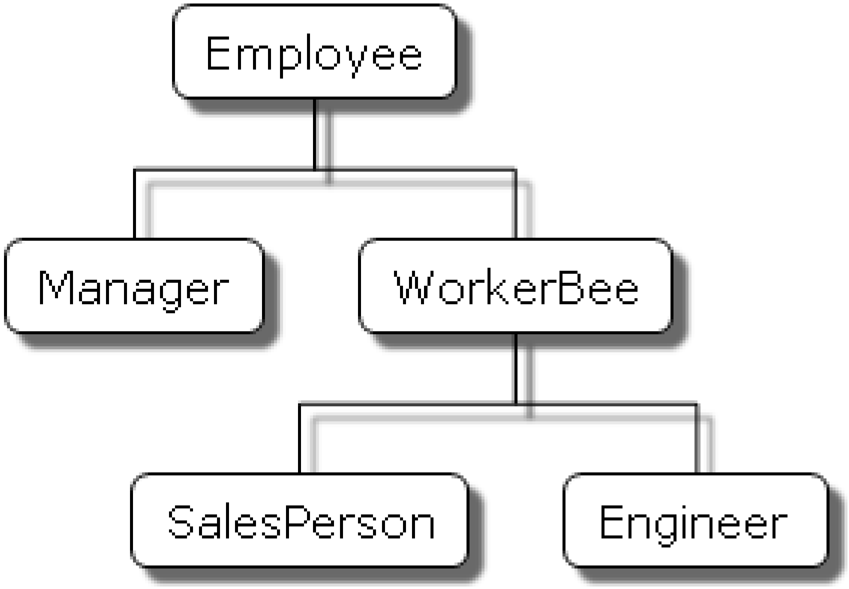
- Prototypes are just objects
  - Therefore, prototype objects themselves have prototypes
- Using this fact, multi-level prototype hierarchies can be constructed
  - One object can acquire properties of all prototypes in a hierarchy
- Multi-level prototype hierarchies are called **prototype chains**

```
myObject.__proto__.__proto__.__proto__.__proto__
```

- `myObject` "inherits" properties from all objects in its **prototype chain**

Prototypal Inheritance: Example

- The Employee Hierarchy



Prototypal Inheritance: Example

```
// Employee object
var Employee = function() {
  this.name = "";
  this.dept = "general";
}
Employee.prototype.print = function() {
  console.log('name:', this.name, ', dept:', this.dept);
};

// Employee has a prototype
console.log( 'prototype:', Employee.prototype );

// Instances refer to the constructor
// and to the same prototype as the Employee
var ernie = new Employee();
ernie.print();
```

```
prototype: Object { , 1 more… }
name:  , dept: general
```

… as expected

- a.k.a. *Pseudo-Classical Inheritance*

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

08/ex9.html

Setting Up Prototypal Hierarchies

In "subclass" constructor function …

1. Set inherited properties in "subclass" by invoking the "superclass" constructor function in the context of the subclass
   - use `Function.call(…)`
2. Set any additional properties specific to subclass in constructor function
3. Set `prototype` property of subclass constructor function to a new object created using superclass prototype
   - use `Object.create(…)`

Prototypal Inheritance: Example

- Manager is a subclass of Employee
- Manager has an additional property named 'reports' that holds an Array of all Manager reports, initialized to an empty Array.

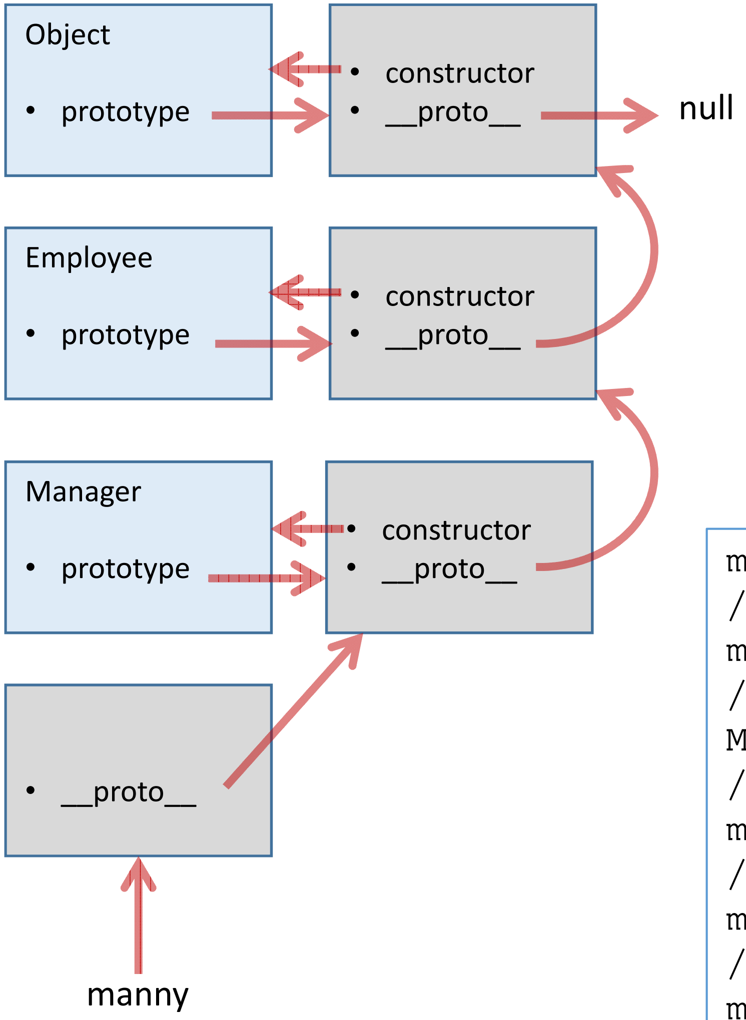
```
var Manager = function() {
  Employee.call(this);                                                          // 1.
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);                          // 2.
Manager.prototype.print = function() {
  console.log('name:', this.name, 'dept:', this.dept, 'reports:', this.reports); // 3.
};

// Test Manager
var manny = new Manager();
manny.print();
```

```
name:   , dept: general , reports: Array [  ]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

08/ex9.html

Prototypal Inheritance: Example



Object
- prototype

- constructor
- __proto__

null

Employee
- prototype

- constructor
- __proto__

Manager
- prototype

- constructor
- __proto__

- __proto__

manny

…Inspecting the **prototype chain**

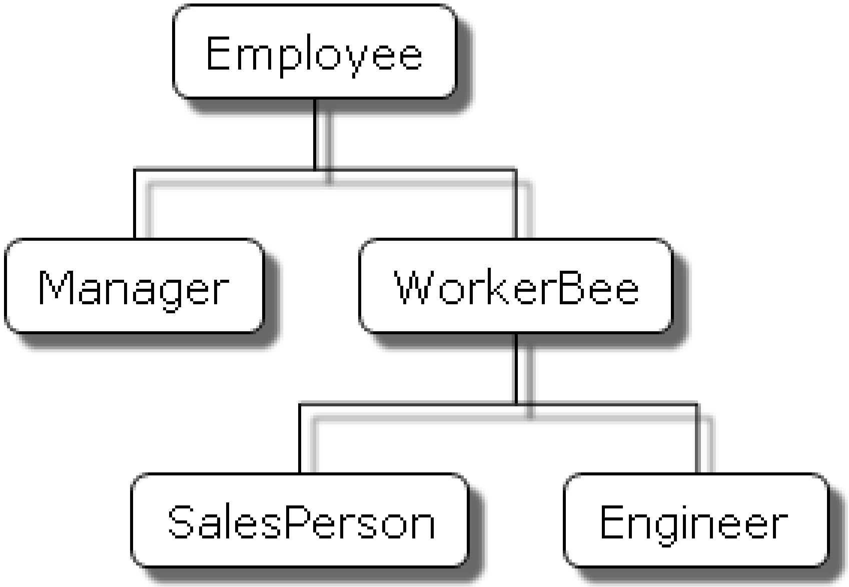
```
manny.__proto__
//-> Object {   }
manny.__proto__.constructor
//-> function Employee()
Manager.prototype === manny.__proto__
//-> true
manny.__proto__.__proto__.constructor
//-> function Employee()
manny.__proto__.__proto__.__proto__.constructor
//-> function Object()
manny.__proto__.__proto__.__proto__.__proto__
//-> null
```

08/ex9.html

Prototypal Inheritance: Example

- Remaining hierarchy

```
var WorkerBee = function() {
  Employee.call(this);
  this.projects = [];
}
WorkerBee.prototype =
Object.create(Employee.prototype);

var SalesPerson = function() {
   WorkerBee.call(this);
   this.dept = "sales";
   this.quota = 100;
}
SalesPerson.prototype =
Object.create(WorkerBee.prototype);

var Engineer = function() {
   WorkerBee.call(this);
   this.dept = "engineering";
   this.machine = "";
}
Engineer.prototype =
Object.create(WorkerBee.prototype);
```



- `WorkerBee` objects have a projects Array
- `SalesPerson` objects have "sales" `dept` and a `quota`
- Engineer objects have "engineering" `dept` and a `machine`

08/ex9.html

Prototypal Inheritance: Example

- SalesPerson and Engineer objects <u>inherit</u> `print()` but <u>override</u> `dept`.

```
var ellen = new Engineer();

ellen.print()
//-> name: , dept: engineering

ellen.projects
//-> Array [  ]

ellen.quota
//-> undefined
```

- `ellen` the Engineer is in 'engineering'
- She has `projects`
- She has no sales `quota`

Properties not possessed by the object directly are searched for by moving "up" the ***prototype chain***

Inheritance: The Prototype "Chain"

- The prototype relations of a JavaScript object hierarchy forms a tree-shaped structure
  - at the root of the tree sits `Object.prototype`.
  - `Object.prototype` provides a few methods that show up in all objects, such as `toString()`, which converts an object to a string representation

- The prototype of `Object.prototype` is `null`

Properties and prototype

- You can always add a property to an object by assignment
- This will not affect any other existing objects created with the same constructor
- New objects created with the associated constructor function will not have this new property
- To add a property to all new objects of a constructor function and all "subclasses", add the property to an object's prototype
- You can even add a new property to a constructor function prototype and all existing objects will get that property.

Properties and prototype

- New properties added to the WorkerBee prototype are inherited by Engineer and SalesPerson

```
var WorkerBee = function() {
  Employee.call(this);
  this.projects = [];
}
WorkerBee.prototype =
Object.create(Employee.prototype);

// Add a project to the projects Array
WorkerBee.prototype.addProject =
function( proj ) {
  this.projects.push( proj );
};

// Return a count of all projects
WorkerBee.prototype.projectCount =
function( proj ) {
  return this.projects.length;
};
```

```
var ellen = new Engineer();

ellen.projects
//-> Array [  ]

ellen.projectCount()
//-> 0

ellen.addProject('React');
ellen.projectCount()
//-> 1

var sal = new SalesPerson();

sal.projectCount()
//-> 0

sal.addProject('expand to
Europe')

sal.projectCount()
//-> 1
```

08/ex10.html

Properties and prototype

- Properties added after objects are created are inherited be existing objects as well.

```
WorkerBee.prototype.itsFriday = function() { console.log('Yipee!'); };

ellen.itsFriday()
//-> Yipee!

sal.itsFriday()
//-> Yipee!
```

instanceof Operator

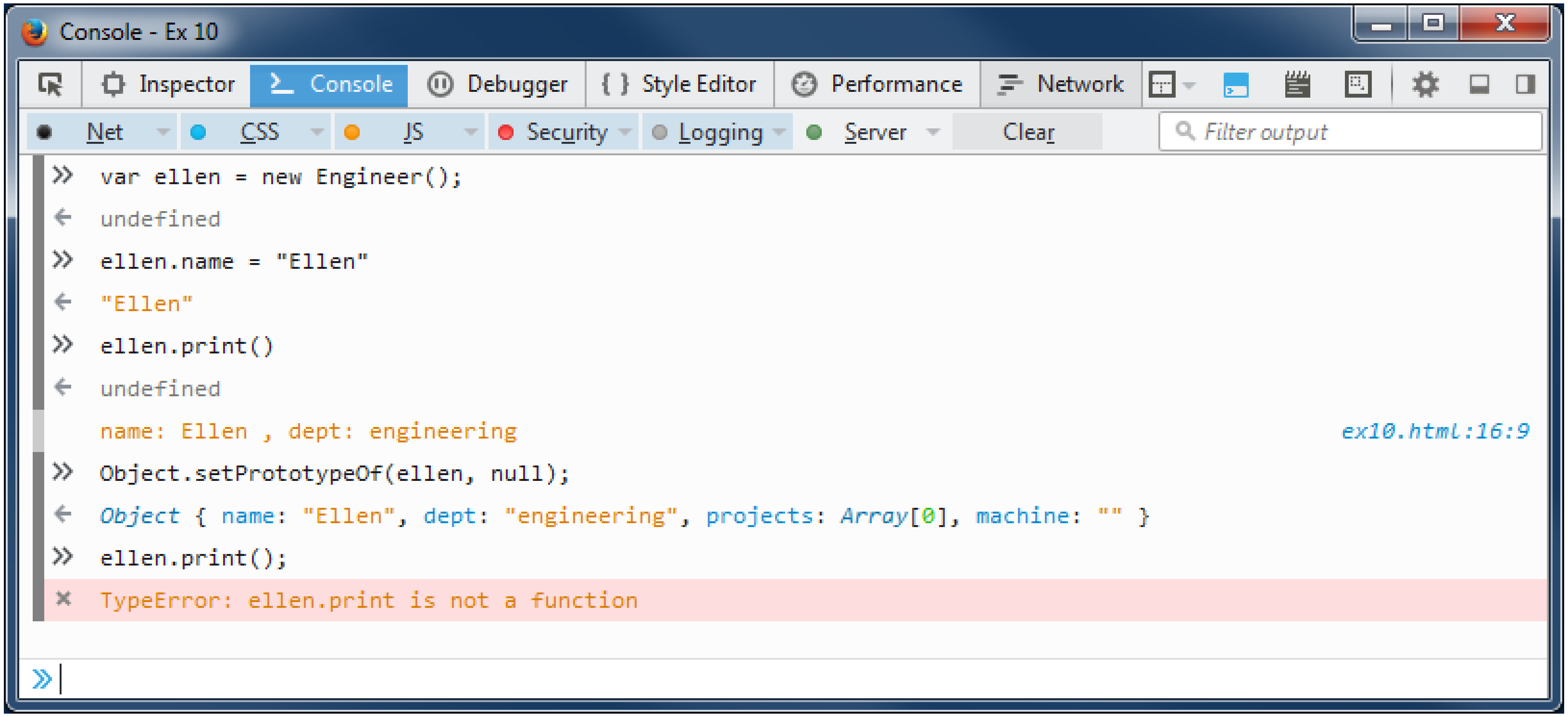
- Used to determine whether an object was derived from a specific object
- The difference between the instanceof operator and the constructor property is that instanceof inspects all objects in the prototype chain.

```
var ellen = new Engineer();

ellen instanceof Engineer
//-> true

ellen instanceof WorkerBee
//-> true

ellen instanceof Employee
//-> true

ellen instanceof Manager
//-> false
```

Breaking the prototype chain with `null`

- It is possible to break an object's prototype chain by setting is prototype to `null`
- This eliminates any inherited properties all-together

```
Object.setPrototypeOf(obj, prototype);
```



```
>> var ellen = new Engineer();
←  undefined
>> ellen.name = "Ellen"
←  "Ellen"
>> ellen.print()
←  undefined
   name: Ellen , dept: engineering                                                          ex10.html:16:9
>> Object.setPrototypeOf(ellen, null);
←  Object { name: "Ellen", dept: "engineering", projects: Array[0], machine: "" }
>> ellen.print();
✕  TypeError: ellen.print is not a function
```

hasOwnProperty()

- Indicates if object itself has a property, as opposed to being accessible through the prototype chain
- Useful for enumerating "owned" properties vs. inherited properties

```
obj.hasOwnProperty(prop)
```

```
var ellen = new Engineer();

ellen.hasOwnProperty(print)
//-> false
ellen.hasOwnProperty("machine")
//-> true

for (prop in ellen) {
  if ( ellen.hasOwnProperty(prop) ) {
    console.log(prop);
  };
};
```

```
name
dept
projects
machine
```

Includes nothing from prototype chain

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty

getOwnPropertyNames()

- The `Object.getOwnPropertyNames()` method returns an array of all properties (as string) found directly upon a given object.

```
Object.getOwnPropertyNames(obj)
```

```
Object.getOwnPropertyNames(ellen);

// ->Array [ "name", "dept", "projects", "machine" ]
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames

enumerable vs. nonenumerable properties

- Enumerable properties are those properties whose internal [[Enumerable]] flag is set to true (set by default)
- Enumerable properties show up in `for...in` loops
- Ownership of properties is determined by whether the property belongs to the object directly and not to its prototype chain

Rules:

- User-defined properties are generally enumerable
- Built-in properties are not enumerable
- Own properties are enumerable
- Inherited properties are not enumerable

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties

propertyIsEnumerable()

```
var ellen = new Engineer();

ellen.propertyIsEnumerable('print')     // inherited property
//-> false

ellen.propertyIsEnumerable('dept')      // own property
//-> true

var arr = ['a', 'b', 'c'];

arr.propertyIsEnumerable('length')      // built-in property
//-> false

arr.propertyIsEnumerable(0)             // user-define property
//-> true
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/propertyIsEnumerable