Web Development Intensive

Macy's

Part 1

- Web Development Basics
- Browser Developer Tools
- HTML Introduction
- Semantic HTML5 Elements
- CSS Introduction

Part 2

- CSS Selectors and CSS3 in-depth
- CSS Specificity & the Cascade
- CSS Layout: Box Model, Display & Positioning
- Browser Dependencies

Part 3

- JavaScript Introduction
- Basic Objects
- Control Flow
- Arrays
- Document Object Model (DOM) Manipulation
- jQuery Introduction

Part 4

- JavaScript Built-in Objects
- Basic Event Handling
- Browser Object Model (BOM)
- Objects In-depth
- JavaScript Inheritance

Part 4

Macy’s

Part 4

- JavaScript Built-in Objects
- Basic Event Handling
- Browser Object Model (BOM)
- Objects In-depth
- JavaScript Inheritance

Review

Best Practices

- Avoid polluting the global namespace
- Define variables at top of a scope
- Use === and !== for comparison
- Avoid primitive object wrappers like Number() or String()
- Include implicit ;
- Always open and close blocks with { }
- Indent and empty lines ensure readability

Built-in Objects

Built-in Objects

- JavaScript gives us built-in objects

- The objects have instance properties
- The objects have instance methods

Built-in Objects [cont.]

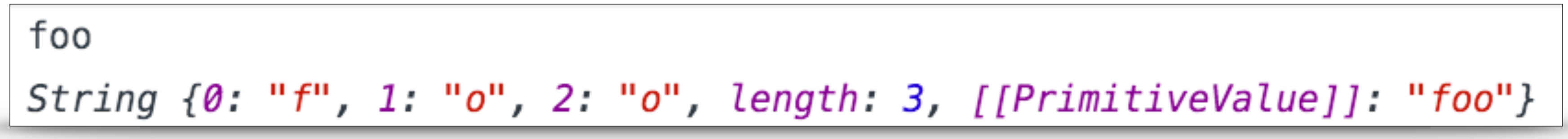
- String
- Number
- Math
- Array
- Date

String

- Global constructor for strings (e.g. a sequence of characters)
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

```
//Global constructor
var foo = new String('foo');
```

Chrome Debugger



```
foo
String {0: "f", 1: "o", 2: "o", length: 3, [[PrimitiveValue]]: "foo"}
```

String [cont.]

- Instance properties

```
//Global constructor
new String('foo').length;

//or…

//String primitive
var foo = 'foo';
foo.length;
```

String [cont.]

- Instance methods
  - charAt : Returns the specified character
  - concat: Combines 2 strings and returns a new string
  - indexOf: Returns the first occurrence of value

```
var str = new String('hello world!');

//The output is the return of the method call
str.charAt(0);          // 'h'
str[0]                  // 'h'
str.concat('!');        // 'hello world!!'
str.indexOf('w');       // 6
str.lastIndexOf(‘l');   // 9
```

String [cont.]

- More instance methods
  - slice: Returns a portion of the string as a new string
  - substr: Returns a new string starting with the location through the allotted number of characters
  - toUpperCase: Returns a new string the has been uppercased
  - trim: Returns a new string with removed trailing and leading whitespace

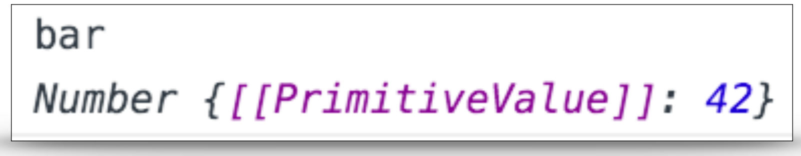
```
var str = new String('hello world! ');

str.slice(0, 5);      // 'hello'
str.substr(6, 5);     // 'world'
str.toUpperCase();    //'HELLOWORLD! '
str.trim();           // 'hello world!'
```

Number

- Global constructor for numbers
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

```
//Global constructor
var bar = new Number(42);
```

Chrome Debugger



```
bar
Number {[[PrimitiveValue]]: 42}
```

Number [cont.]

- Generic properties
  - Properties available directly on the Object itself (i.e. no object instance needed)

```
Number.MIN_VALUE; // 5e-324
Number.MAX_VALUE; // 1.7976931348623157e+308
Number.NaN;       // NaN
Number.POSITIVE_INFINITY; //Infinity
Number.NEGATIVE_INFINTY;  //-Infinity
```

Number [cont.]

- Number helpers
  - parseInt: Returns the integer portion of the value as a number
  - parseFloat: Returns the floating point portion of the value as a number
  - isNaN: Returns a boolean based off a whether the value literally is **NaN**

```
parseInt(42.53, 10);    //42
parseFloat('42.53t');   //42.53
isNaN(NaN);             //True
isNaN('5x');            //True
isNaN(Number('5x');     //True
```

Number [cont.]

- Instance Methods
  - toExponential: Returns a string representation of exponent notation
  - toFixed: Returns a string representation of fixed-point notation
  - toPrecision: Returns a string representation of the specified precision

```
var num = new Number(3.1415);

num.toExponential();  // "3.1415e+0"
num.toFixed(3);       // 3.142
num.toPrecision(3);   // 3.14
```

Math

- Object for mathematical constants
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math
  - Math is not a constructor

Math

- Generic properties
  - PI: Numeric representation of PI
  - SQRT2: Numeric representation of the Square root of 2

```
Math.PI      // ~3.14159
Math.SQRT2   // ~1.414
```

Math

- Generic methods
  - abs: Returns a numeric absolute value
  - max: Returns the maximum number in a set
  - min: Returns the minimum number in a set
  - pow: Returns a numeric base raised to the exponent power

```
Math.abs(-14);    // 14
Math.max(0, 10, 15, 3) // 15
Math.min(0, 10, 15, 3) // 0
Math.pow(2, 3)    // 8
```

Math

- Generic methods
  - sqrt: Returns the square root
  - floor: Returns largest integer less than or equal to a value
  - ceil: Returns the smallest integer greater than or equal to a value
  - random: Returns a random number between 0 and 1

```
Math.sqrt(2)        // ~1.414
Math.floor(42.45)   // 42
Math.ceil(42.45)    // 43
Math.random()       // A number between 0 and 1
```

Array

- We have seen a lot of Array methods already
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray
- Array.isArray() : Returns a boolean of the checked value

```
Array.isArray([]);    //True
Array.isArray(new Array());   //True
```

Exercise: Array

- Goal: Gain more familiarity with Arrays

- Arrays

```
var array1 = [1,2,3,4,5,6,7,8,9,0];
var array2 = ['aa','b','cccc','dddddd','eeee','fffff'];
```

- Specifications:
  - Create a filtered array1 that only contains even values
  - Create a sorted array2 that sorts the values in descending order by length

Date Object

Date Object

- A JavaScript Object
- Snapshot of the current date and time
- Useful for working with date and time
  - Calendar apps
  - date/time stamping
  - form submissions

Date Object [cont.]

- Create an instance
  - Requires use of **new** operator to call the Date constructor
  - No arguments gives you an object based of the current timestamp
- Just want a string representation?
  - Don't use the **new** operator
  - Gives a the current date and time
  - Can't take any arguments

```
var theDate = new Date();
var stringDate = Date();
```

Date Object [cont.]

- ECMAScript 5 addition
- Need the milliseconds without wanting to use the **new** operator
  - Useful if you need a comparison against another time in milliseconds

```
var theDate = Date.now();
```

Date Object [cont.]

- Date.now polyfill
  - For those legendary browsers ;)

```
if (!Date.now) {
  Date.now = function() { return new Date().getTime(); };
  Date['now'] = Date.now;
}
```

Date Object [cont.]

- Can be created with arguments
- Allows you to create a date instance at a specific point in time
  - **millisecond_value**: the number of milliseconds between the desired date and Jan. 1, 1970 @ midnight
  - **date_string**: a string format "Month dd, yy" or "Month dd, yy hh:mm:ss"
    - Jan. 3, 1970
    - January 3, 1970
    - Jan 3, 1970 23:08:01

```
new Date(millisecond_value);
new Date(date_string);
```

Date Object [cont.]

- Allows you to create a date instance at a specific point in time
  - **year**: a four-digit number representation (e.g. 1970)
  - **month**: an integer from 0 (i.e. Jan) to 11 (i.e. Dec)
  - **day**: an integer from 1 to 31 representing calendar day
  - **hour**: an integer in 24 hour format from 0 (i.e. midnight) to 23 (i.e. 11 PM)
  - **minutes**: an integer from 0 to 59
  - **seconds**: an integer from 0 to 59

```
new Date(year, month, day);
new Date(year, month, day, hour, minutes, seconds);
```

Date Object Methods

- **getTime**: returns the millisecond representation of the Date object
- **getFullYear**: returns the year of the Date object in four-digit format
- **getMonth**: returns the month from 0 (i.e. Jan) to 11 (i.e. Dec)
- **getDate**: returns the day of the month from 1 to 31

Date Object Methods [cont.]

- **getDay**: returns the day of the week from 0 (i.e. Sunday) to 6 (i.e. Saturday)
- **getHours**: returns the hours in 24-hour format from 0 to 23
- **getMinutes**: returns the minutes from 0 to 59
- **getSeconds**: returns the seconds from 0 to 59
- There are also similar **set** methods

<time>

- A semantic way to display the time, date or
- Contains an optional **datetime** property
  - If not used the **<time>** must be a valid date

```
<time>1970-01-01</time>
```

- Some valid dates:
  - 1970-01-01 a valid year, month, day
  - 1970-01 a valid year, month
  - 01-01 a valid year-less string
  - 12:24 a valid time (i.e. a time based on a 24hour clock)
  - 12:24:32 a valid time
  - 4h 31m 22s a valid duration

<time> [cont.]

- A **datetime** property can be used to describe the date given
- Gives a standard microformat way to consume the data for browsers, search engines …
  - http://microformats.org/wiki/Main_Page

```
<time datetime="1970-01-01">January 1st</time>
```

JavaScript IIFE Pattern

```
(function() {
  'use strict';

   //Variables
   var hatColor = 'blue';

   //Functions
   function changeColor() {
     hatColor = 'brown';
   }

   //Main functionality
   (function() {
     changeColor();
   })();
});
```

JavaScript DOM Ready Pattern

```
(function() {
  'use strict';

   //Variables
   var hatColor = 'blue';

   //Functions
   function changeColor() {
     hatColor = 'brown';
   }

   //Main functionality
   $(function() {
     changeColor();
   });
});
```

Lab 5

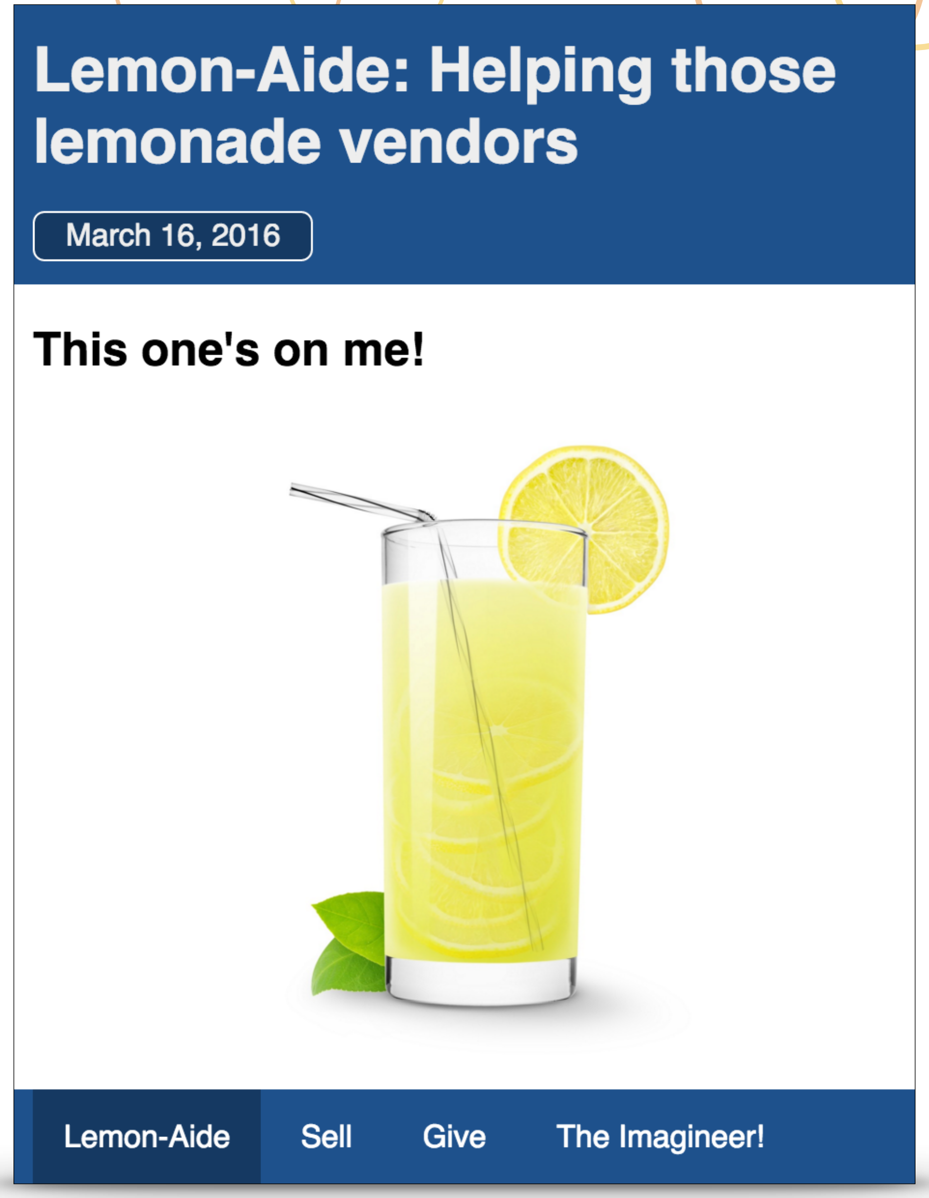
- Think about the previous code
  - Make sure none of the variables / functions are global
  - Only invoke the script once the page has loaded
  - What could you make as constants?

- Use the JavaScript Date object to fill-in the time with today’s date

Lab 5b

- Use moment.js to fill in the date instead of the date functionality you just created
  - http://momentjs.com/

Lab 5



Lemon-Aide: Helping those lemonade vendors

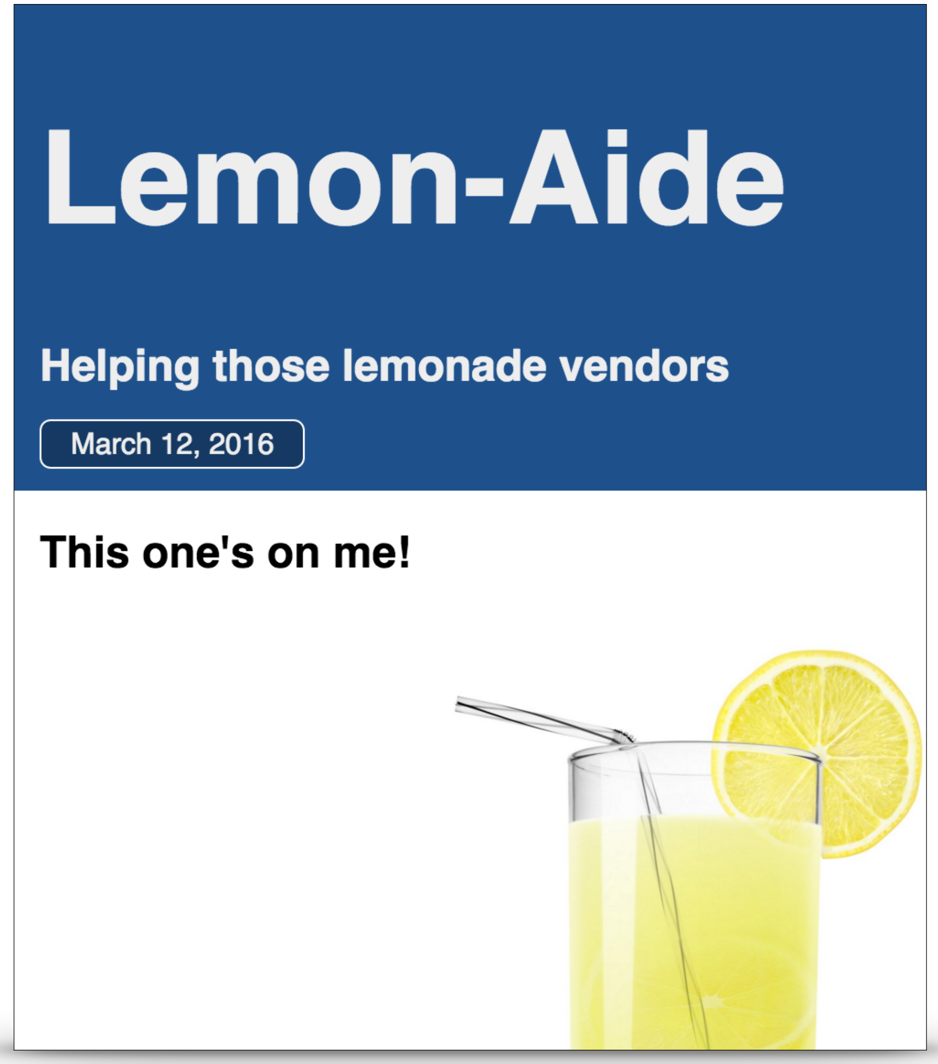
March 16, 2016

This one's on me!

Lab 5c

- Use FitText to make your header grow and shrink based on screen size
  - http://fittextjs.com/
  - Only make the “Lemon-Aide” grow and shrink
  - Modify your date functionality to make that work still

Lab 5c


Lemon-Aide

Helping those lemonade vendors

March 12, 2016

This one's on me!

Locales

- date.toLocaleDateString:
  - Allows for a formatted date string based on language
- date.toLocaleTimeDate
  - Allows for a formatted time string based on language
- http://jsfiddle.net/kamrenz/LrFp4/1/

```
var theDate = new Date();
var options = {
  weekday: "short",
  year: "numeric",
  month: "short",
  day: "numeric"
};

console.log("Date: " + theDate.toLocaleDateString('en-us', options));

console.log("Time: " + theDate.toLocaleTimeString(navigator.language,
  {hour: '2-digit', minute:'2-digit', second:'2-digit'}));
```

jQuery Events

The Basics

- Use the $() to grab an element or a group of elements to setup the event listeners
- Use the jQuery **on** method to register event listeners
  - Similar to addEventListener in POJS

```
$('p').on('click', function(event) {
  //Process the click
});
```

Event properties

- important event properties in the callback

```
$('p').on('click ', function(event) {
  //Process the click
});
```

- **target**: The element that initiated the event
  - **this**: In the callback scope is the same as event.target
- **preventDefault()**: Prevents the default action like a link taking us to a new page
- **stopPropagation()**: Stops the event from continuing to bubble up
- **type**: The type of the event … like a click
- **target.nodeName**: Node name of the element clicked

Event Types

- Form Events
  - **submit**:
    - Submits the form for processing to the defined backend action
    - Capture a form submission at the form level not the submit button level
  - **reset**:
    - Capture a form reset at the form level not at the submit button level
    - Resets form to its original state

```
$("form").on('submit', function () { … });
$("form").on('reset', function () { … });
```

Logging

- **console:** Gives access to the browser's console
  - Make sure to take these out when going to production
- **log:** Method used to give general logging output

```
console.log(“Hello”);
```

Browser Objects

More Browser Objects

- We have already interacted the **document** object
  - document.getElementById(‘user’)
- This document object really lives off of the **window** object
  - window.document.getElementById(‘user’)
- The window object is the parent of other objects
  - https://developer.mozilla.org/en-US/docs/Web/API/Window

More Browser Objects

- **history**
  - This is read-only
  - https://developer.mozilla.org/en-US/docs/Web/API/History

```
//Go back a page
window.history.back();

//Go back 2 pages
window.history.go(-2);

//Go forward a page
window.history.forward();

//Go forward 2 pages
window.history.go(2);
```

More Browser Objects

- **location**
  - Allows for the redirection of a URL via location.href
  - https://developer.mozilla.org/en-US/docs/Web/API/Location

```
//Get the URL URL
var url = window.location.href;

//Change to a new URL
window.location.href="http://www.macys.com"

//Change to a new URL
window.location.assign(“http://www.macys.com");

//Change to a new URL without adding to history
window.location.replace(“http://www.macys.com");

//Reload the current page
window.location.reload();
```

More Browser Objects

- **navigator**
  - Allows for the redirection of a URL via location.href
  - https://developer.mozilla.org/en-US/docs/Web/API/Location
  - Get battery status, get geolocation, check if you are online

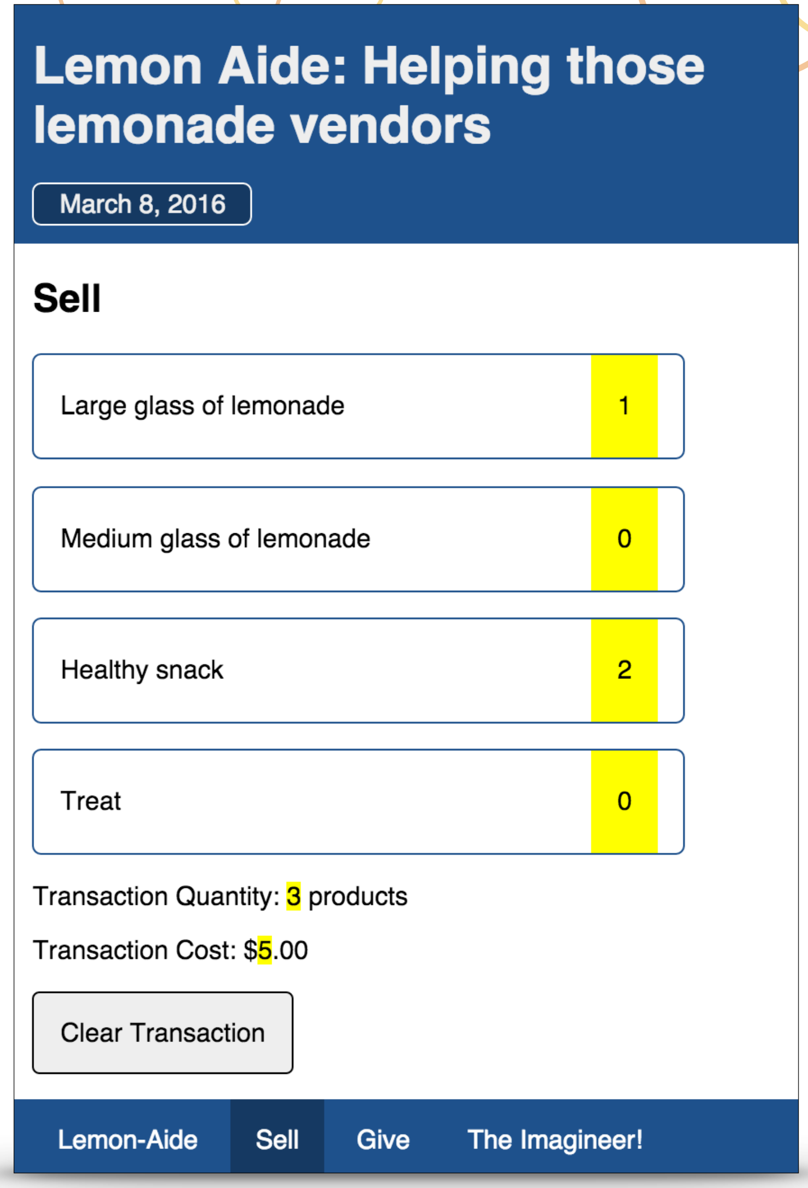
```
//Get the language "en-us"
var language = window.navigator.language;

//User agent
//"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
48.0.2564.116 Safari/537.36"
var agent = window.navigator.userAgent;

//Online
window.navigator.onLine
```

Lab 6

- Create Transaction logic for the **Sell Page**
  - Update the button totals when they are clicked
  - Add a running transaction cost on the screen
  - Add a running transaction quantity on the screen
  - Add a button to reset everything

- Log the button that is clicked to the console
  - Take a look at Mozilla Developer Network (i.e. MDN) to see what other categories of output you can have besides console.log()

Lab 6



Lemon Aide: Helping those lemonade vendors

March 8, 2016

Sell

Large glass of lemonade	1
Medium glass of lemonade	0
Healthy snack	2
Treat	0

Transaction Quantity: 3 products

Transaction Cost: $5.00

POJSO

Constructor

- JavaScript function used to create objects

```
function **City**() {
  //Stuff in here to construct
}

var lansing = new City();

console.log(“instanceof: " + (lansing instanceof City));
console.log("constructor check: " +
  (lansing.constructor === City));
```

Object/Utility Property

- Properties off of the City object not off the instance

```
//We could just say var City = {} if we are doing utilities
//  instead of a Constructor function
function City(numOfPeople) {
  //Stuff in here to construct
}

//Class/Utility property: Useful for configuration
//  Uppercase convention to designate it should be a constant
//  No instantiation of City to use this method
City.HOUSEHOLD_DIVISOR = 2;
console.log(City.HOUSEHOLD_DIVISOR);
```

Instance Property

- Objects all have separate copies of their instance properties
  - 20 cities objects have 20 different numbers of people

```
//We need to create a new City to use instance properties
function City(numOfPeople) {
  //Instance Property: Needs to create a City instance for
access
  //  Every City object instance will have this property
  this.numOfPeople = numOfPeople;
}

City.HOUSEHOLD_DIVISOR = 2;
```

Scope-Safe Constructor

- Allow for creation of objects without **new** operator

```
function City(numOfPeople) {
  if(this instanceof City) {
    this.numOfPeople = numOfPeople;
  } else {
    return new City(numOfPeople)
  }
}

var lansing = new City(110000);
var boulder = City(101808);

console.log(lansing instanceof City);
console.log(boulder instanceof City);
```

Object/Utility Method

- Methods off of the City object not off the instance

```
//We could just say var City = {} if we are doing utilities
//  instead of a Constructor function
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}


//Class/Utility method
//  This method exists only on the City object not on the
instances
City.moreHouseHolds = function (cityA, cityB) {
  if (cityA.numOfPeople > cityB.numOfPeople) {
    return cityA;
  } else {
    return cityB;
  }
}
```

Instance Method

- Methods created in the constructor that are public to anyone

```
//We need to create a new City to use instance methods
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
  this.getNumOfPeople = function() {
    return this.numOfPeople;
  }
}

var lansing = new City(110000);
console.log("Number of people: " + lansing.getNumOfPeople());
```

Object

- Objects have constructor properties
  - **constructor:** reference to the creating function

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}

var boulder = new City(101808);
var aNew = new Object();

console.log(boulder.constructor);
console.log(City.constructor);
console.log(aNew.constructor);
```

Object Prototype Property

- Let’s investigate the Object prototype property
  - contains the method **hasOwnProperty**()

```
var city = { name: "Lansing" };

console.log("Is name property in city: " + ("name" in city));
console.log("Does city own name: " +
  city.hasOwnProperty("name"));
console.log("Does city own hasOwnProperty: " +
  city.hasOwnProperty("hasOwnProperty"));
console.log(“Is hasOwnProperty in city: " +
  ("hasOwnProperty" in city));
console.log("Does the Object prototype own hasOwnProperty" +
  Object.prototype.hasOwnProperty(“hasOwnProperty"));
console.log("Does the Object own hasOwnProperty: " +
Object.hasOwnProperty("hasOwnProperty"));
```

Prototypes

- JavaScript core objects also have prototypes
  - Change them sparingly

```
Array.prototype.clear = function() {
     this.length = 0;
}

var anArray = [1,2]

console.log("length:" + anArray.length);
anArray.clear();
console.log("length:" + anArray.length);
```

Prototypes [cont.]

- We also could redefine an existing method
  - Yikes!!

```
Function.prototype.toString = function () {
  return "I am sam";
};
function showMe(){ return "I am bill"; }
console.log(showMe);
```

```
String.prototype.toString = function () {
  return "I am sam";
};
var aString = "42";
console.log(aString.toString());
```

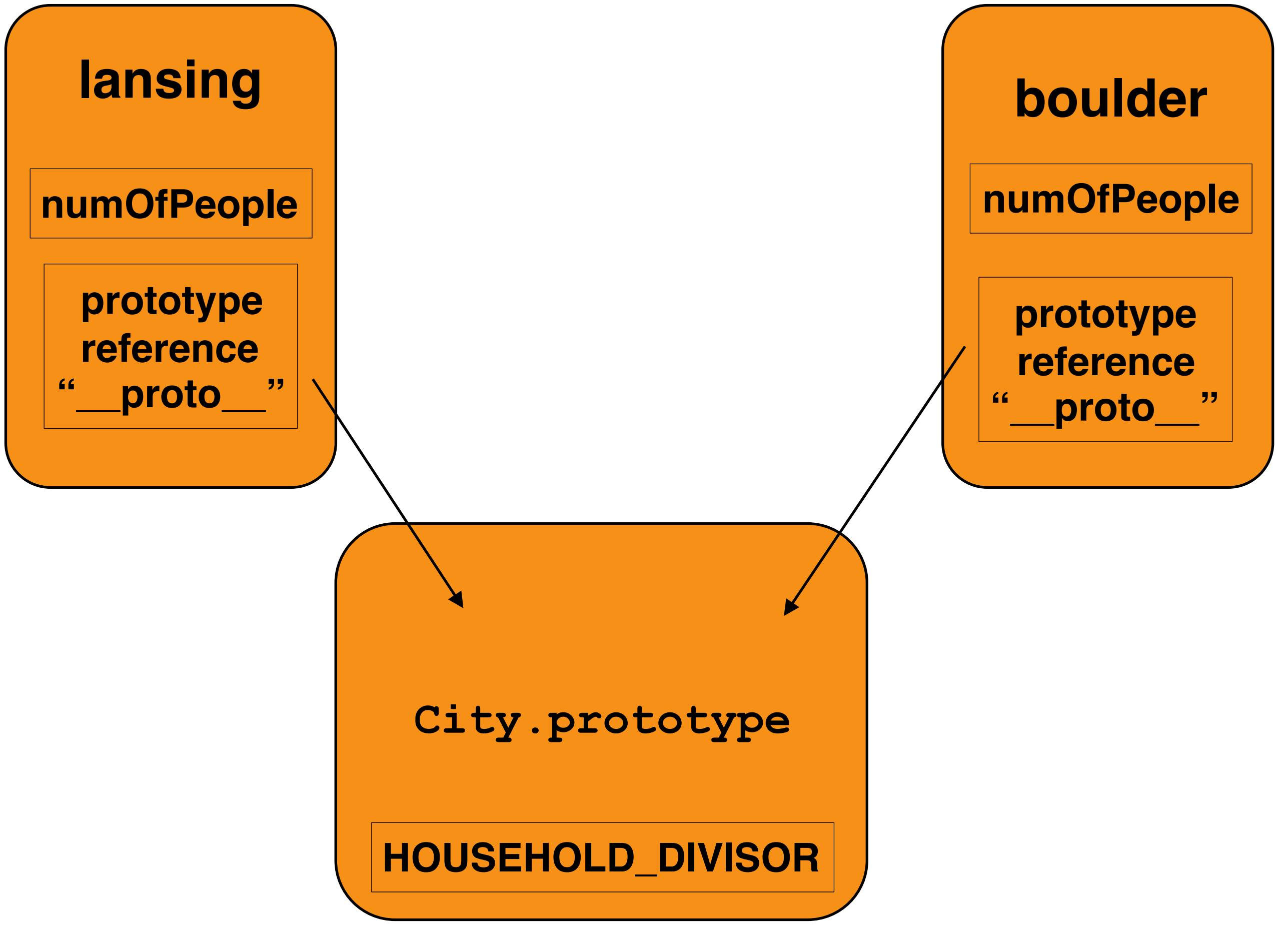
Prototypes [cont.]

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}

//A shared property between instances
//  Only 1 copy exists on City.prototype property
City.prototype = {
  HOUSEHOLD_DIVISOR: 2
}

var boulder = new City(101808);
console.log('Household Divisor: ' + boulder.HOUSEHOLD_DIVISOR +
  ‘\n__proto__ property: ’ + boulder.__proto__.HOUSEHOLD_DIVISOR +
  '\ngetPrototypeOf: ' +
      Object.getPrototypeOf(boulder).HOUSEHOLD_DIVISOR +
  ‘\nCity prototype property: ’ + City.prototype.HOUSEHOLD_DIVISOR);
```

Prototypes [cont.]

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}

//A better Class Method
//  Only 1 copy exists on City.prototype property
City.prototype = {
  HOUSEHOLD_DIVISOR: 2,
  population: function () { return this.numOfPeople; },
  houseHolds: function () {
    return this.numOfPeople / this.HOUSEHOLD_DIVISOR; }
}
var boulder = new City(101808);
console.log('Population:' + boulder.population());
console.log('HouseHolds:' + boulder.houseHolds());
```

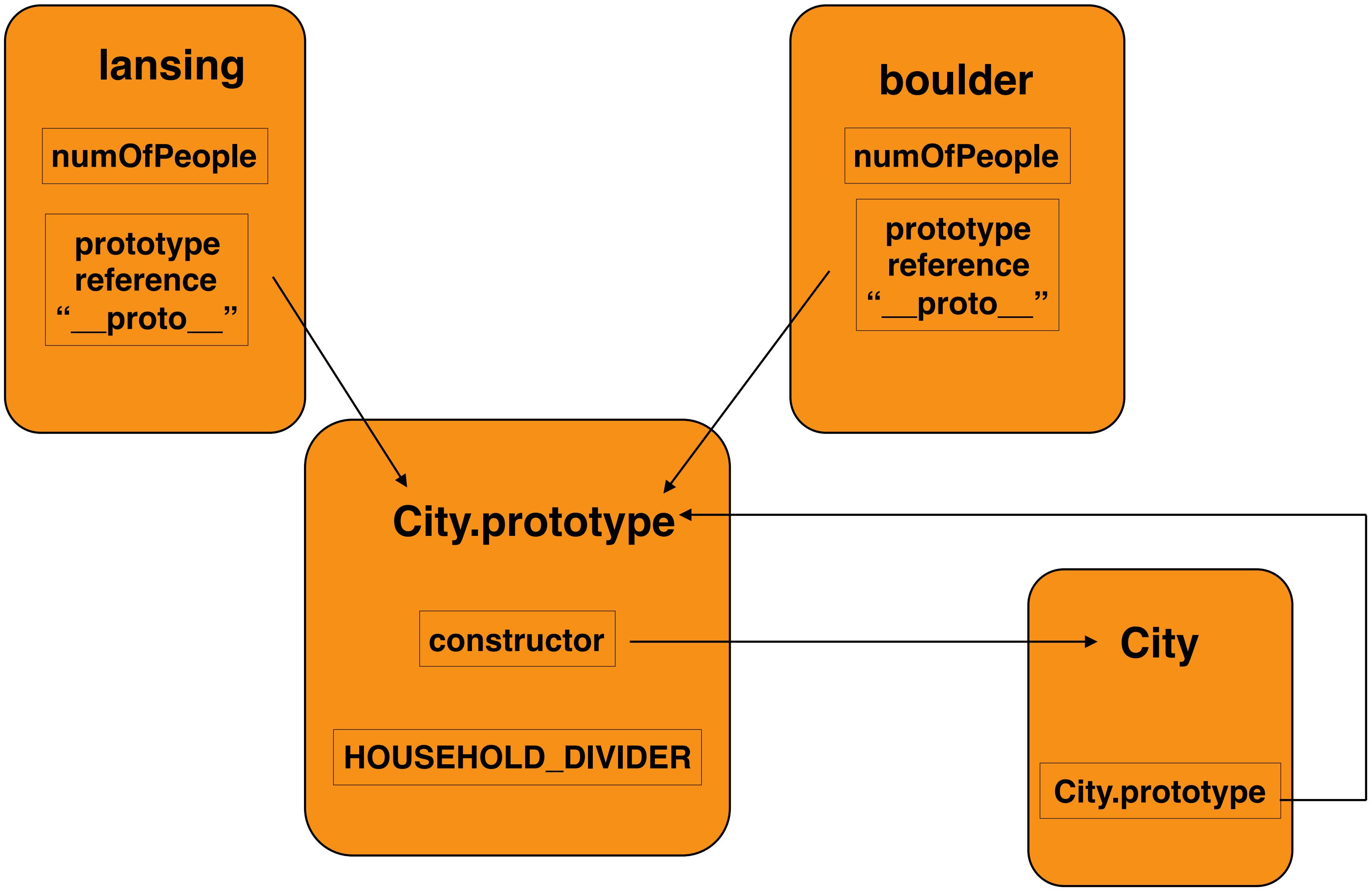
Prototypes [cont.]



Whoops Constructor confusion

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}
//We overwrote the prototype completely!
City.prototype = {
  constructor: City,  //Makes sure the City knows it’s a City
  HOUSEHOLD_DIVISOR: 2,
  population: function () { return this.numOfPeople; },
  houseHolds: function () {
    return this.numOfPeople / this.HOUSEHOLD_DIVISOR; }
}

var lansing = new City(110000);
console.log(‘lansing constructor:' +
 (lansing.constructor === City));
```

Prototypes [cont.]



What if?

- We have a shadow…

```
function City() {
  this.DEFAULT_POPULATION = 20000;
}

City.prototype = {
  DEFAULT_POPULATION: 50000,
  getPopulation: function () {
    return this.DEFAULT_POPULATION;
  }
}

var generic = new City();
console.log(generic.getPopulation());
console.log(Object.getPrototypeOf(generic).DEFAULT_POPULATION);
```

Prototype Recap

- Prototype properties/methods shared between all instances

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople || this.DEFAULT_POPULATION;
}
City.prototype.DEFAULT_POPULATION = 50000;
City.prototype.getPopulation = function () {
  return this.numOfPeople;
}
var lansing = new City(114000);
var boulder = new City(101808);
var generic = new City();
console.log(‘Lansing Population:' + lansing.getPopulation());
console.log(‘Generic Population:’ + generic.getPopulation());
```

Public Members

- Methods accessible to the world

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
}

City.prototype.population = function () {
  return this.numOfPeople;
}
var lansing = new City(110000);

console.log('Num of People:' + lansing.numOfPeople);
console.log('Population:' + lansing.population());
```

Private Members

- Variables not accessible to the world

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;
  var **houseHolds** = this.numOfPeople / 2;
}

var lansing = new City(110000);

console.log('Population:' + lansing.numOfPeople);
console.log('Households:' + lansing.houseHolds);
```

Private Methods

- Methods not accessible to the world

```
function City(numOfPeople) {
  this.numOfPeople = numOfPeople;

  function calcHouseHolds() {
    console.log("not reachable");
  }
}

var lansing = new City(110000);
console.log(lansing.calcHouseHolds());
```

Privileged Methods

- Public methods to access private variables

```
function City(numOfPeople) {
    var that = this;
    this.numOfPeople = numOfPeople;
    var divisor = 3;

    function calcHouseHolds() {
      return that.numOfPeople / divisor;
    }

    this.getHouseHolds = function () {
       return calcHouseHolds();
    }
}
var lansing = new City(110000);
console.log(lansing.getHouseHolds());
```

Object

- As a note:
  - Public members can be added anytime
  - Private and Privileged members can only be created during object construction

Objects Without **new**

- A trending pattern in JavaScript is to create objects without utilizing the **new** operator
  - Utilizing the **new** operator can give people the idea classes are really being created

- The Object.create() method is used for creating objects
  - It is ECMAScript 5 functionality

Objects Without **new** [cont.]

- First let's create our prototype object
  - We just create an Object Literal

```
//Prototype to be used for city instances
var proto = {
  hasMayor: true,
  toString: function() {
    return this.name + ' has ' + this.population + ' people';
  }
};
```

Objects Without **new** [cont.]

- Second let’s create a factory function to create our instances

```
//Factory Function to create instances
var makeCity = function(name, population) {
  //Creating object instance via proto: Shared between all instances
  var city = Object.create(proto);

  //Instance variables
  city.name = name;
  city.population = population;

  //Return the object instance
  return city;
};
```

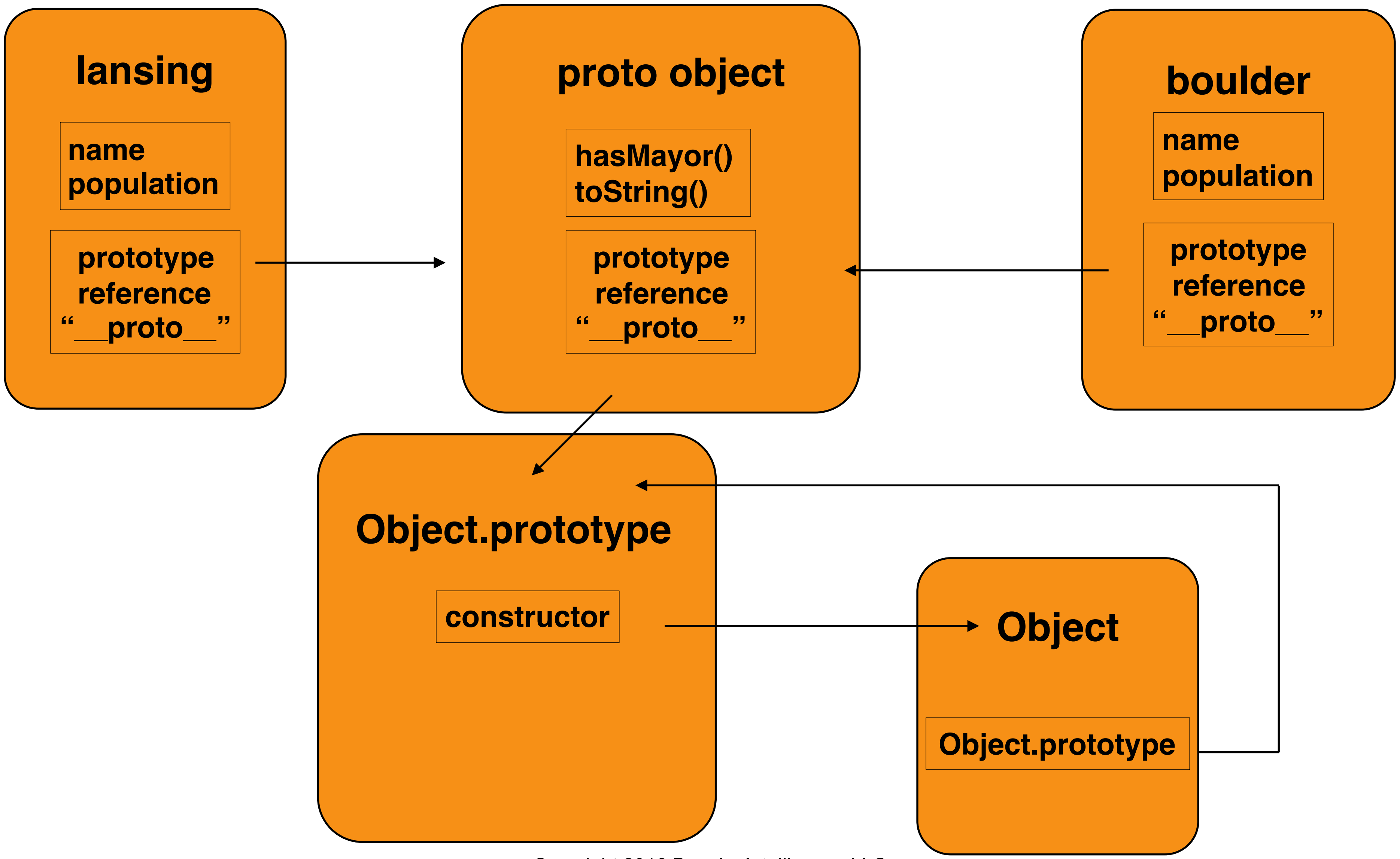
Objects Without **new** [cont.]

- Last we create instances

```
var lansing = makeCity('Lansing', 110000);
var boulder = makeCity('Boulder', 103000);
```

- Instances
  - Will share the same **prototype** object proto
  - Instances will be of type Object
  - Instances will have their constructor pointing to Object

Objects Without **new** [cont.]



Objects Without **new** [cont.]

- What if are programming for IE8?
  - We need to create our own object.create()
  - Based off of Crockford's Object.create
    - http://javascript.crockford.com/prototypal.html

```
var objectCreate = function( objectPrototype ) {
  if (!objectPrototype) { return {}; }
  function F() {}
  F.prototype = objectPrototype;
  return new F();
};
var newObject = objectCreate(proto);
console.log(newObject);
```

this Reference

Functions: Let's backtrack

```
aFunction;

aFunction()

aFunction.call()
```

- Line 1 refers to the function object
- Line 2 calls the function
- Line 3 calls the function with a different context

JavaScript call()

- **call** allows us to indirectly invoke a function as if it were a method of another object
  - https://jsbin.com/suyiho/edit?js,console

```
function Square (x) { this.x = x; }
Square.prototype = {
     perimeter: function () {return 4 * this.x }
}

var square = new Square(4);
var perimeter = square.perimeter();
console.log("perimeter 1: " + perimeter);

var myPoint = {y:3, x:5};
var perimeter2 = square.perimeter(myPoint);
console.log(“perimeter 2: “ + perimeter2);

var myPoint = {y:3, x:5};
var perimeter3 = square.perimeter.call(myPoint);
console.log(“perimeter 3: “ + perimeter3);
```

JavaScript call() [cont.]

- Without using call()
  - https://jsbin.com/wogafi/edit?js,console

```
function Square (x) { this.x = x; }
Square.prototype = {
     perimeter: function () {return 4 * this.x }
}

var square = new Square(4);

//  With a call()
var myPoint = {y:3, x:5};
var perimeter2 = square.perimeter.call(myPoint);
console.log(“perimeter 2: “ + perimeter2);

//  Without a call() we need a temporary property
myPoint.perimeter = square.perimeter;
var perimeter3 = myPoint.perimeter();
console.log(“perimeter 3: “ + perimeter3);
delete myPoint.perimeter;
```

call and apply

- Only difference is the way the arguments are arranged

```
//Let’s say our square.perimeter method
//  took an argument called size and color

var myPoint = {y:3, x:5};

//Comma separated arguments
square.perimeter.call(myPoint, “Big”, “Blue”);

//Array of arguments
square.perimeter.apply(myPoint, [“Big”, “Blue”]);
```

Loosing Our Reference

- “this” can lose its scope … sort of
  - https://jsbin.com/jegaro/edit?js,console

```
function City(numOfPeople) {
  var that = this;  //Happy fun times!
  this.numOfPeople = numOfPeople;
  var divisor = 3;

  var calcHouseHolds = function() {
    return that.numOfPeople / divisor;
  };

  this.getHouseHolds = function () {
    return calcHouseHolds();
  };
}

var lansing = new City(110000);
console.log(lansing.getHouseHolds());
```

Loosing Our Reference [cont.]

- Using call() to save “this” reference
  - https://jsbin.com/gizuvi/edit?js,console

```
function City(numOfPeople) {
  //var that = this;
  this.numOfPeople = numOfPeople;
  var divisor = 3;

  var calcHouseHolds = function() {
    //return that.numOfPeople / divisor;
    return this.numOfPeople / divisor;
  };

  this.getHouseHolds = function () {
    //return calcHouseHolds();
    return calcHouseHolds.call(this);
  };
}
var lansing = new City(110000);
console.log(lansing.getHouseHolds());
```

Loosing Our Reference [cont.]

- Using bind() to save "this" reference
  - https://jsbin.com/ceyoce/edit?js,console

```
function City(numOfPeople) {
  //var that = this
  this.numOfPeople = numOfPeople;
  var divisor = 3;

  var calcHouseHolds = function() {
    //return that.numOfPeople / divisor;
    return this.numOfPeople / divisor;
  }.bind(this);

  this.getHouseHolds = function () {
    return calcHouseHolds();
  };
}

var lansing = new City(110000);
console.log(lansing.getHouseHolds());
```

Delete

- Did the whole object get deleted?

```
var square = new Square(4);
var perimeter = square.perimeter();
var myPoint;

myPoint.perimeter = square.perimeter;
var perimeter2 = myPoint.perimeter();

delete myPoint.perimeter;
```

Exercise: Objects Part A

- Goal: Gain familiarity with Object Creation via new
- Specifications:
  - Create a MacBook Object Constructor
    - Create a small screen check function
      - If screen size is less than 13 it is good for travel
      - Make that private
    - Create a public screenSize property
    - Create a public type property
    - Create a public method exposing if it is good for travel

Exercise: Objects Part A [cont.]

- Goal: Gain familiarity with Object Creation
- Specifications:
  - Every MacBook instance has
    - A make of MacBook
    - A retina screen
    - An SSD harddrive
    - A color of Gray
      - We are only thinking of MacBook Pros and Airs :)
    - A toString showing make, type and screen size

Exercise: Objects Part A [cont.]

- Goal: Gain familiarity with Object Creation
- Specifications:
  - Create 4 MacBooks
    - Pro 15 inch model
    - Pro 13 inch model
    - Air 13 inch model
    - Air 11 inch model

Exercise: Objects Part A [cont.]

- Goal: Gain familiarity with Object Creation
- Specifications:
  - Log information about your 4 MacBooks
    - What is the make, type, screen size,
    - Do they have SSD drives?
    - Is it good for travel?
    - Is it an instance of MacBook?
    - Is it and instance of Object?
    - What is it’s constructor?

Exercise: Objects Part A [cont.]

- Goal: Gain familiarity with Object Creation via Object.create()
- Specifications:
  - Build your object instances via Object.create()
  - Remove your private method

Tightly Controlled Objects

Object Creation

- So far the objects we have created with constructors and prototypes
- JavaScript gives us another way to create objects
  - Allowing for more control over how the object will be used

Defining Properties

- We have seen

```
var obj = {};
obj.x = 42;
```

- We can have more control

```
var obj = {};
Object.defineProperty(obj, 'x', {
  value: 42,
  writable: true,
  enumerable: true,
  configurable: true
});
```

Defining Properties

- Writeable
  - Specifies if the value may be changed via assignment
- Configurable
  - Specifies if the property descriptor may be changed and the property may be deleted
- Enumerable
  - Specifies if the property will show up during enumeration of the properties
- Value
  - Specifies the value associated with the property

Getters and Setters

- We have seen

```
var rectangle = { width:10, height:10, area: null }
```

- With more control

```
var rectangle = {};
Object.defineProperty(rectangle, ‘area’, {
  enumerable: true,
  configurable: true,
  get: function() {
    return this.width * this.height
  },
  set: function() {
    console.log(‘You cannot set the area directly’);
  }
});
```

Getters and Setters

- Handling multiple properties

```
var rectangle = {};
Object.defineProperties(rectangle, {
  width: {
    value: true,
    writeable: true
  },
  height: {
    value: ‘Hello’,
    writeable: false
  }
});
```

Object.create with Descriptors

- Object.create takes a second parameter beyond a simple object prototype

- Allow for instance properties to be assigned to the objects

Object.create with Descriptors

- Object.create takes a second parameter beyond a simple object prototype

```
var obj = Object.create(Object.prototype, {
  init: {
    value: function(foo) { this.foo = foo; }
  },
  // foo is a data property
  foo: { writable: true, configurable: true, value: 0 },
  // properties can have any value, including functions
  baz: {
    value: function(x) { console.log('baz says', x); }
  }
});
//No constructor… no new… we need to initialize it ourself
obj.init(42)
```

Interrogating Objects

- Get a list of keys in an object
  - Object.keys(obj)
  - The keys need to be enumerable

- Get a list of all properties in an object
  - Object.getOwnPropertyNames(obj)
  - All properties including those with enumerability set to false

Interrogating Objects [cont.]

- Get the property descriptor
  - Object.getOwnPropertyDescriptor(obj, ‘foo’)
  - Shows the state of the object
- Check if the object owns the property
  - obj.hasOwnProperty(‘foo')
  - Objects on the prototype are not directly owned

Preventing Extensions Objects

- Prevents object properties from being added
  - Object properties can be deleted
- Allows values that are already present to be changed

```
var obj = { foo: 1, bar: 2};
Object.preventExtensions(obj);

console.log(Object.isExtensible(obj));
```

Sealing Objects

- Prevents object properties from being added and/or deleted
- Makes all objects properties non-configurable
- Allows values that are already present to be changed

```
var obj = { foo: 1, bar: 2};
Object.seal(obj);

console.log(Object.isSealed(obj));
```

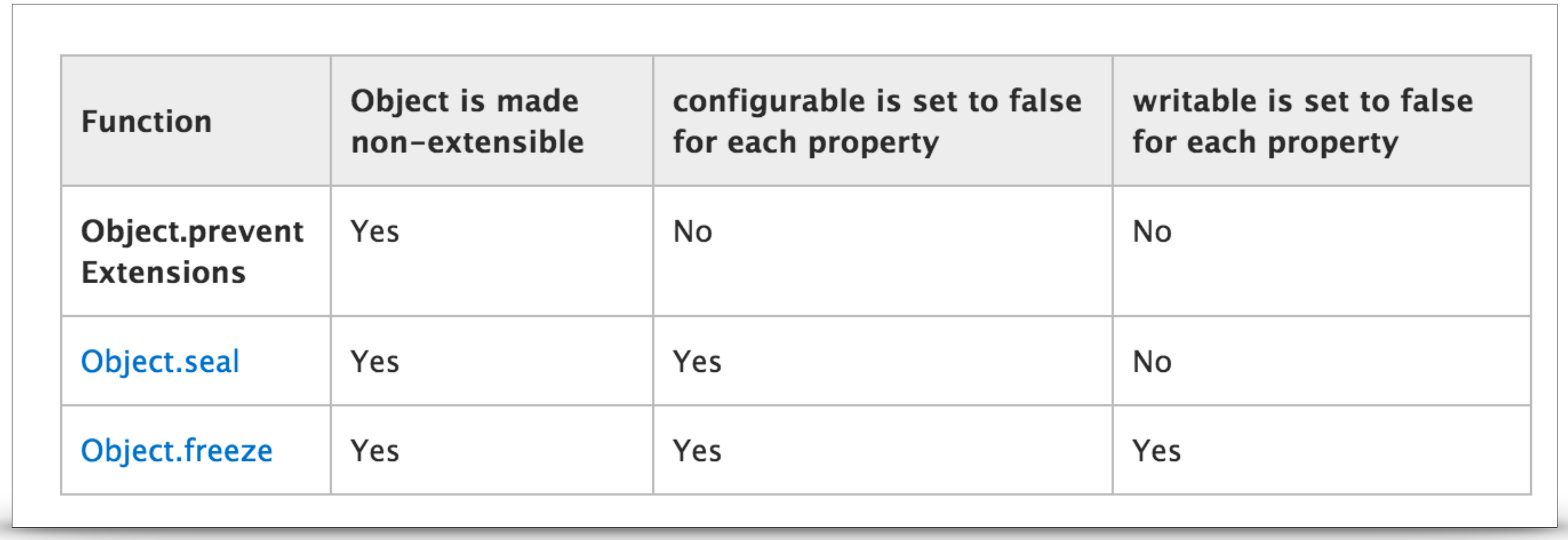
Freezing Objects

- Prevents object properties from being added/ deleted
- Prevents object property values from being changed
- Makes all objects properties non-configurable
  - Essentially objects become immutable

```
var obj = { foo: 1, bar: 2};


Object.freeze(obj);
Object.isFrozen(obj);
```

Comparison

- Compare object methods
  - https://msdn.microsoft.com/en-us/library/ff806191(v=vs.94).aspx



Function	Object is made non-extensible	configurable is set to false for each property	writable is set to false for each property
Object.preventExtensions	Yes	No	No
Object.seal	Yes	Yes	No
Object.freeze	Yes	Yes	Yes

Exercise: Objects Part B

- Goal: Gain familiarity with Object property descriptors

- Specifications:
  - Change Math.PI to 42
  - Make it configurable and writeable

Exercise: Objects Part B

- Goal: Gain familiarity with Object Creation via Object.create()

- Specifications:
  - Return to your previous exercise:
    - The Object.create()
  - Take the object you created with Object.create() and use Object descriptors directly on your prototype, rather than simply adding properties to the prototype

Exercise: Objects B

- Goal: Gain familiarity with Object Creation

- Specifications:
  - Return to your previous exercise:
    - The objects instances created with **new**
  - Add to private variables: price and buyIt
  - Expose buyIt via a privileged method
  - Set the price via an Object property descriptor
    - When the price is set check if it is below the price point
    - If it is set buyIt to true
  - A good price point could be $1,200
  - Print out the object keys (price should be shown)

Exercise: Objects Part C

- Goal: Gain familiarity with Object property descriptors

- Specifications:
  - Create a way to completely make an object a constant
    - This would include objects the have references to other objects

End