Objects

Object Creation and Prototypal Inheritance

Moving Forward

Today's the first "intermediate" course on javascript fundamentals.

We might be talking about concepts you already know, so as a group feel free to decide that we're moving too slow or too fast, today is a gauge.

Today's Outline

- Principles of Object Oriented Code
- Object Creation
- Prototypal Inheritance

Object Oriented Programming

Three major principles of writing Object Oriented Code

1. Abstraction
2. Inheritance
3. Encapsulation

We will touch on concepts in Abstraction and Inheritance today.

Abstraction

Abstraction is a means of controlling complexity in computer systems.

Oftentimes we deal with codebases with hundreds of thousands of lines of code. In order to mitigate risk of bugs and cut the time to add features, we separate code into modules and subsystems that represent a smaller part of the larger whole.

Abstraction begins with modeling our system. In order to effectively model our system, we create objects.

Object Creation

We use objects to organize code into structures that represent their counterpart in the real-world.

This is the beginning of writing object-oriented code.

In Javascript, everything is an object, including functions.

Object Creation

Objects in javascript can be created by using object literal notation.

```
var name = {
    first: “Kyle”,
    last: “Pace”,
    title: “Mr”
};
```

Object Creation

Objects can be made up properties, functions, arrays of objects and nested objects.

```
var engineer = {
        name: { first: “Kyle”, last: “Pace” },

        responsibilities: [ “Write Code”, “Learn about Python” ],

        writeCode: function () {
                console.log(“Writing code”);
        }
};
```

Object Creation

In order to access an object's properties or methods, we use the dot-operator.

```
var engineer = {
        name: { first: "Kyle", last: "Pace" },
        responsibilities: [ "Write Code", "Learn about Python" ],
        writeCode: function () {
                console.log("Writing code");
        }
};

engineer.writeCode();                   // writes "Writing Code" out to the console.
console.log(engineer.name.first)        // writes "Kyle" out to the screen
```

Object Creation

As we said earlier, functions are technically objects as well. We can create a function object by using what is known as a Function Literal.

```
var engineer = function ( ) {
        console.log("I'm an engineer!");
};
```

This isn't any different from how we were creating functions in the previous lectures.

Object Creation

Objects without properties or functions are boring and serve only to perform one job. In order to create a proper object using functions, we need the keyword this and the Constructor Invocation Pattern.

```
var Engineer = function (firstName, lastName) {
      this.name = { first: firstName, last: lastName };
      this.yearsOfExperience = 6;
};

var eng = new Engineer('Kyle', 'Pace');

console.log(eng.yearsOfExperience);
```

Object Creation

Now that we have an engineer function object, we need to add some functions. You do so by modifying the engineers Prototype (we will go into this next).

```
Engineer.prototype.sayName = function () {
      console.log(this.name.first + ‘ ‘ this.name.last);
};

var engi = new Engineer(‘John’, ‘Smith’);

engi.sayName()    // What does this do?
```

Object Creation

Pay careful attention to the new keyword with the usage of this within a function.

If you create an object using a function that relies on the keyword this, you need to instantiate the object with the keyword new, otherwise you will be polluting the global namespace.

```
var eng = Engineer();          // This would be very bad.
```

Objects created using the Constructor Invocation Pattern should be UpperCamelCase for this very fact.

Object Creation

You can also use the this keyword when creating an object using the object literal pattern.

```
var engineer = {
        name: { first: 'Kyle', last: 'Pace' },
        sayName: function () {
                console.log(this.name.first, this.name.last);
        }
};

engineer.sayName()    // Writes "Kyle Pace" to the console.
```

Object Creation

Demo

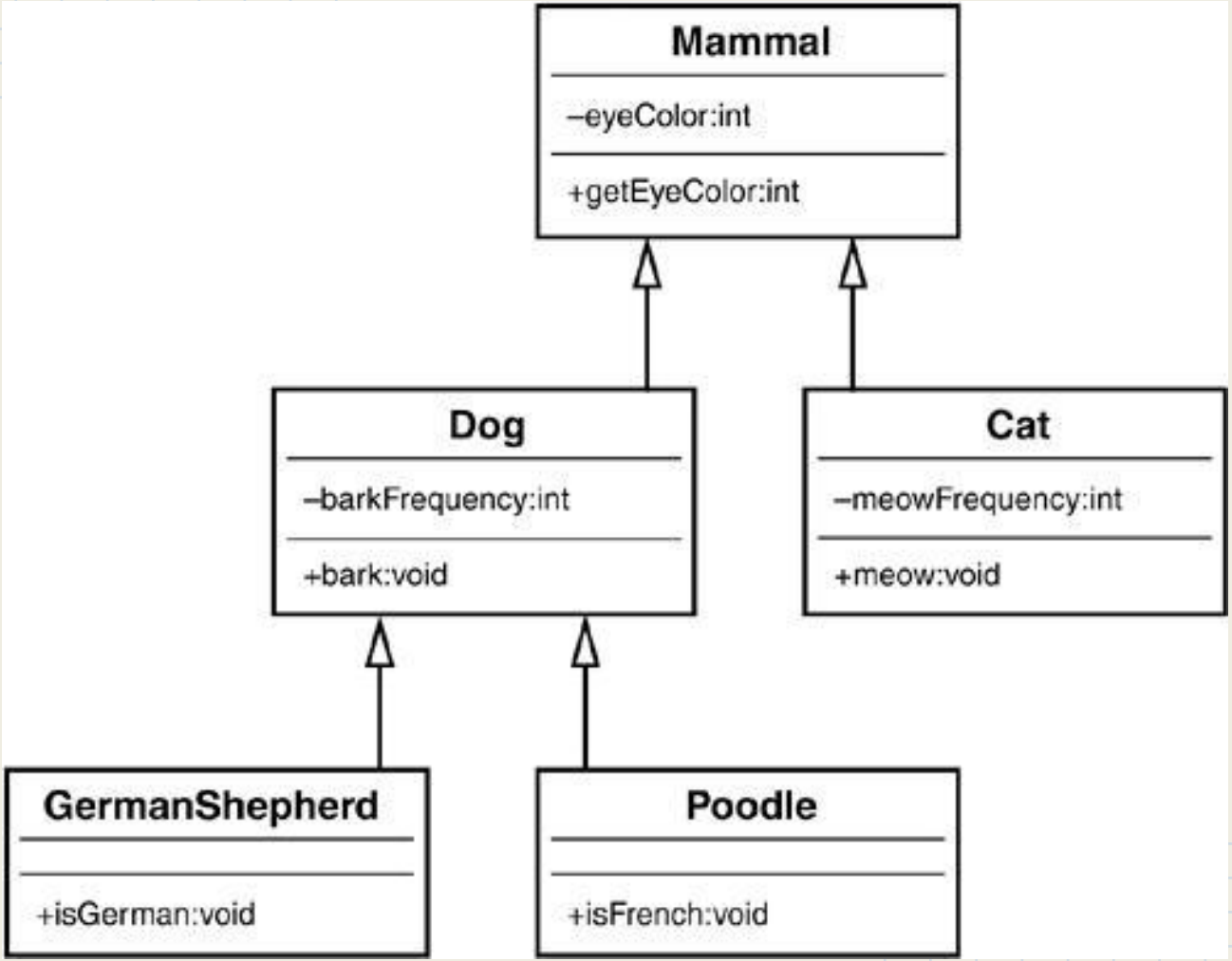
Prototypes - Inheritance

Inheritance is one of the most important object-oriented programming concepts.

It is the fundamental way of sharing code between objects.

After creating an object, we can reuse that object by inheriting its properties and methods.

When an object is based on another object, we say it IS that object, or it inherits from that object.


Mammal
–eyeColor:int
+getEyeColor:int
Dog
–barkFrequency:int
+bark:void
Cat
–meowFrequency:int
+meow:void
GermanShepherd
+isGerman:void
Poodle
+isFrench:void

Prototypes - Inheritance

Most programming languages implement Classical Inheritance, since Javascript doesn't have classes and everything is an object, we have Prototypal Inheritance.

Every object in javascript has a prototype. Instead of implementing a class, we can reuse code by assigning the prototype onto a new object.

Think of this as copying an object in memory and expanding that object with new methods.

Copying prototypes is a way of cloning behavior.

Prototypes - Usage

Currently there are two ways of writing code to extend javascript objects.

1. Constructor Functions
2. Object.create()

We'll only be going over Constructor Functions with respect to prototypes today. Object.create() was introduced in ES5, is not supported with IE < 10 and is not as widespread.

Having said that, it provides an alternative to dealing with the somewhat misleading new keyword.

Prototypes - Usage

```
var Mammal = function () {
        this.hasHair = true;
}

Mammal.prototype.speak = function () {
        console.log("Hi I'm a " + this.name);
};

var Cat = function (name) {
        this.name = name;
};

Cat.prototype = new Mammal();
var cat = new Cat('Harold');
console.log(cat.hasHair);      // Writes out "true" to console.
cat.speak();                   // What does this print out?
```

Prototypes - Usage

We can now create a Dog type from the same mammal object.

```
var Dog = function () { };

Dog.prototype = new Mammal();

Dog.prototype.bark = function () {
      console.log('woof');
};

var dog = new Dog();

console.log(dog.hasHair)   // writes "true" to the console.
```

Prototypes - Usage

Now we're able to share code between objects and extend those objects with unique behavior.

If the Cat object were to try and call bark(), the javascript runtime would throw an exception.

```
cat.bark();      // death and destruction.
```

Prototypes - Usage

The Constructor Function and new keyword syntax is meant to pseudo-mirror classical inheritance.

Object.create() is a more typical “javascript” approach but is not as widely adopted.

If you want typical class-based syntax, wait for ES-6, it’s coming.

Prototypes - Usage

Demo

Prototypes - WTF

Why would we ever in the world do any of this crazy stuff?

1. Code Reuse. Less code is better. The more code, the more maintenance, the greater the probability of breaking things.
2. Speed and Performance. Prototypes also save big-time on memory. Every javascript object has a memory imprint, prototypes help to share the workload.

When to avoid

- Small apps or limited functionality. When dealing with a system that can be described in a few hundred lines of code, save yourself the thoughtwork, it's probably overkill.

Questions

If there's time….

- What is the new keyword doing underneath the covers
- How do prototypes help save on memory?
- How can you use SUPER from javascript in inherited objects?
- When do you use a particular pattern?

Workshop

Modelling exercises

https://github.com/walterg2/EverCraft-Kata

Citations

Javascript: The Good Parts, Douglas Crockford

http://www.2ality.com/2015/02/es6-classes-final.html