www.phparch.com

April 2015
VOLUME 14 - ISSUE 4

Front-end Polish

php[architect]

Object Oriented JavaScript (Part the Second)

The Browser Capabilities Project in 2014

UX Without the Process

FREE Article!

ALSO INSIDE

PHP Extensions—Class and Object Management

PHP Conference Newbies 101

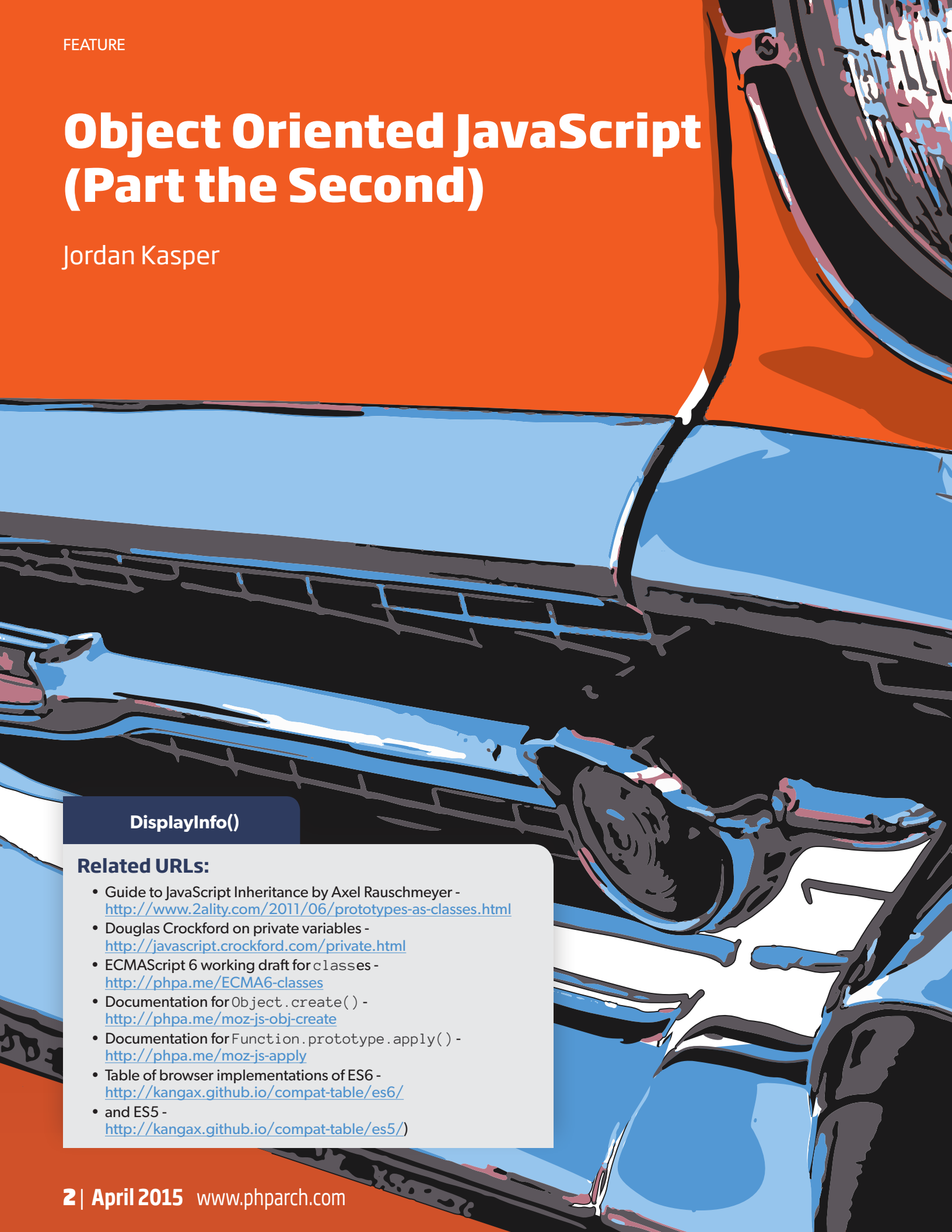
Leveling Up:
TDD with phpspec

Community Corner:
Midwest PHP Report

finally{}:
JavaScript—It's Already Won the Web (But So Has PHP)

Object Oriented JavaScript (Part the Second)

Jordan Kasper



DisplayInfo()

Related URLs:

- Guide to JavaScript Inheritance by Axel Rauschmeyer - http://www.2ality.com/2011/06/prototypes-as-classes.html
- Douglas Crockford on private variables - http://javascript.crockford.com/private.html
- ECMAScript 6 working draft for `classes` - http://phpa.me/ECMA6-classes
- Documentation for `Object.create()` - http://phpa.me/moz-js-obj-create
- Documentation for `Function.prototype.apply()` - http://phpa.me/moz-js-apply
- Table of browser implementations of ES6 - http://kangax.github.io/compat-table/es6/
- and ES5 - http://kangax.github.io/compat-table/es5/)

Recap and Reintroduction

Welcome to Part Two in our series on object-oriented JavaScript! In the March 2015 issue of php[architect] we introduced readers to the core principles of OOP and started discussing the core nature of objects and functions in JavaScript. We moved onto constructors, the `new` keyword, and different types of object members. In this issue we're going deeper into the `prototype` object to discuss prototypical inheritance and polymorphism. We may refer to some nomenclature and syntax from Part One, so be sure to review that content first! We'll finish up Part Two with a peek into ECMAScript 6 and the plan for JavaScript objects in the future.

Our Dog Example

Before we jump into the meat of this article, let's review the example code we were working with in Part One of our series. We began with a simple `Dog` object constructor and a few basic properties and methods (see Listing 1).

LISTING 1

```
01. // Our Dog constructor...
02. function Dog(name) {
03.     if (name) { this.name = name; }
04.
05.     var alive = true;                          // private variable
06.     this.isAlive = function() { return alive; }  // privileged method
07.     this.die = function() { alive = false; }     // privileged method
08. }
09.
10. // Public members
11. Dog.prototype.name = "Bubbles";
12. Dog.prototype.speak = function() {
13.   return this.name + " says woof";
14. };
15.
16. // Static members
17. Dog.genus = "Canis";
18. Dog.mergeBreeds = function(dogA, dogB) {
19.     // TODO: implement this ;)
20. }
```

And we can see how this `Dog` constructor is used below. Don't forget our discussion of closures in Part One—that's what makes our "privileged" methods work!

```
// Using our Dog object...
var v = new Dog("Vincent");
console.log( v.name );       // "Vincent"
console.log( v.speak() );    // "Vincent says woof"

console.log( v.alive );      // undefined!
console.log( v.isAlive() );  // true
v.die();
console.log( v.isAlive() );  // false
console.log( v.alive );      // still undefined!

// alternate syntax for `new Dog()`
var b = Object.create(Dog.prototype);
b.constructor();

// "Bubbles says woof"
// (using the default/prototype `name` property
console.log( b.speak() );
```

And last, be sure to remember what our `prototype` object looks like for our `Dog` constructor function:

```
> console.log(Dog.prototype);
{
    constructor: Dog(name) {
        if (name) { this.name = name; }
        // ...
    },
    name: "Bubbles",
    speak: function() {
        return this.name + " says woof";
    },
    __proto__: Object { ... }
}
```

Prototypical Inheritance

Some people can grok prototypical inheritance (or "prototypal"—both are correct) by truly understanding the word itself. Remember how we mentioned the `prototype` object as being the exemplar of what it means to be a `Dog` (or `Employee`, etc.)? Consider the word "prototype" outside of a programming context. Hearing the word in general use is clear: a prototype is what all future things of that type should look like. If we had a new phone prototype we would expect phones in that model line to look and act like the prototype, with some small enhancements, perhaps.

This is the nature of prototypical languages. There are no classes and instances, only objects, some created from a blank slate (object literals, perhaps) and some created from an example object: the `prototype`.

Let's continue with our `Dog` example: we've already abstracted away what is common to each `Dog` into a prototype, but what if we wanted to abstract away what all animals share so that we can have `Dog` and `Cat` instances? This is the point of inheritance in programming, and prototypical inheritance accomplishes this the same way it accomplishes the abstraction of instances to their prototypes.

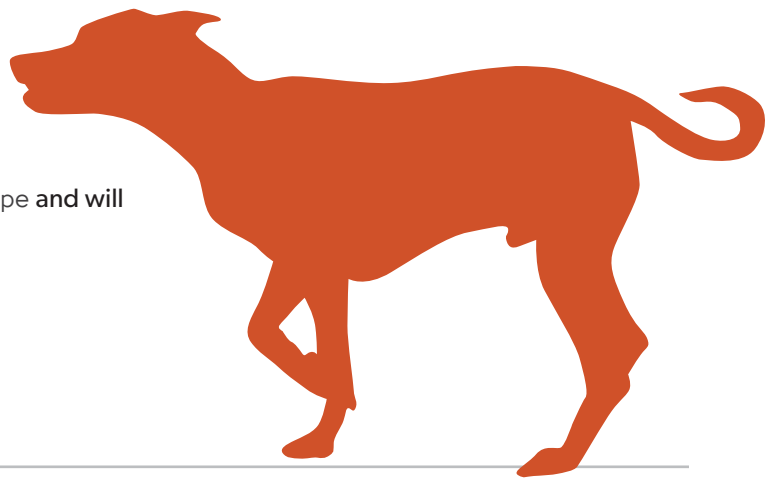
```
function Animal(age) {
    if (age) { this.age = age; }

    var alive = true;
    this.isAlive = function() { return alive; }
    this.die = function() { alive = false; }
}
Animal.prototype.age = 1;

function Dog(name) {
    if (name) { this.name = name; }
}
```

We can see how the `alive` private variable and its privileged methods have been moved into a new constructor function for `Animal`, and that we've added a new `age` property to those instances (defaulting to `1`). But right now, our two object types (`Animal` and `Dog`) are not related. There is no connection between them, and although we can create instances of both, a `Dog` is not a type of `Animal` in any way. To do this, we need to specify that the `prototype` of a `Dog` derives from the prototype of an `Animal`:

```
function Animal(age) { /* ... */ }
Animal.prototype.age = 1;

function Dog(name) { /* ... */ }

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.name = "Bubbles";
// The remainder of Dog.prototype members...

var v = new Dog("Vincent");
```

After we define the two constructor functions, we then set the initial `Dog.prototype` to be equal to a new object that is created from the `Animal.prototype`. In other words, "what it means to be a `Dog`" starts out as "what it means to be an `Animal`." Next, we have to revert the constructor back to the `Dog` function itself. If we don't, then the `Dog.prototype.constructor` method will still point to the `Animal.prototype.constructor`! Finally, we begin to set all of the other members of the new `Dog.prototype` (the `name` property, `speak` method, etc.).

The key to understanding this relationship is understanding that the `prototype` object is a fallback mechanism. (I told you we'd get back to it!) Let's say we want to get the age of a `Dog`: `console.log(v.age);`. If the current instance object (`v`) does **not** have an `age` property directly on it, JavaScript will "fall back" to the first prototype (`Dog.prototype`) and look for `age` there. If that object doesn't have an `age` property, JavaScript will "fall back" to the next prototype, in this case `Animal.prototype` (defined by the `__proto__` member of `Dog.prototype`). There it is! JavaScript found a member called `age` on the `Animal.prototype` and will use that value.

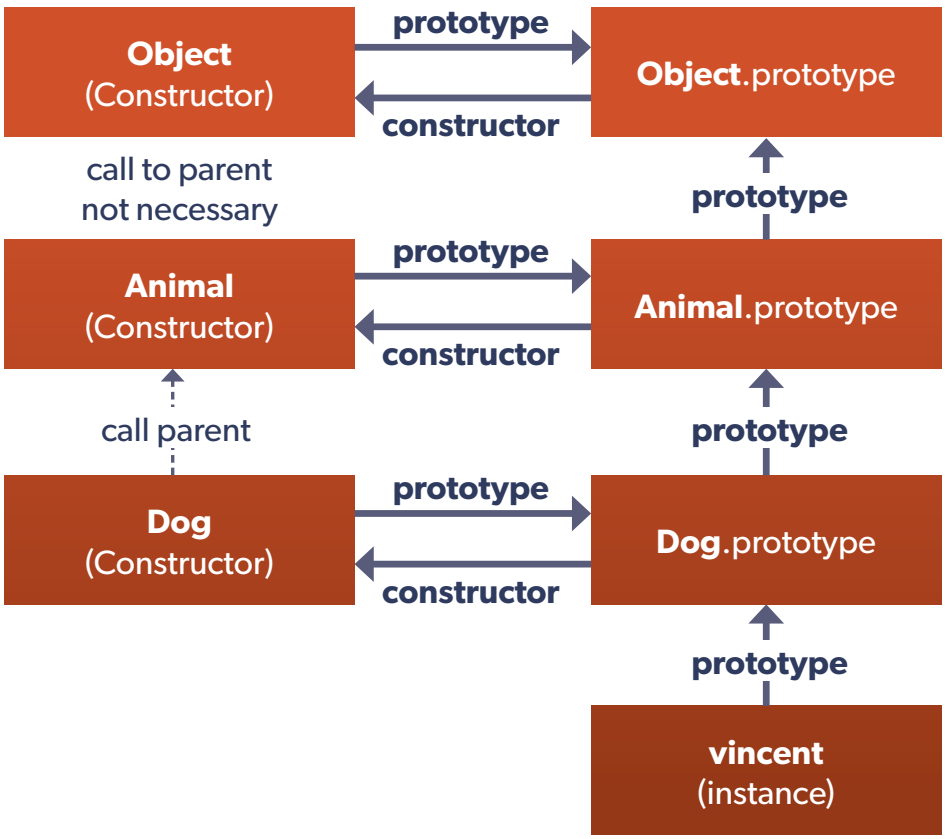


This structure is referred to as the **prototype chain** and falling back to successive prototypes is the mechanism by which JavaScript creates inheritance. Below we can see this chain from the instance of a `Dog` down to the core `Object` prototype:

```
> var v = new Dog("Vincent");
> console.log(v);
{
    name: "Vincent",
    __proto__: Dog {
      constructor: function Dog(name) { ... }
      name: "Bubbles"
      speak: function () { ... }
      __proto__: Animal {
        constructor: function Animal(age) { ... }
        age: 1
        __proto__: Object { ... }
      }
    }
}
```

This output shows how the `__proto__` property is used to reference the `prototype` object that each level was created from. That "chain" of `__proto__` properties is our inheritance path. For those of us that are more visually inclined, perhaps this diagram is better. Notice that our `Dog` instance (`v`) is at the bottom and the chain proceeds up with the top level being the "root" JavaScript `Object`.

The JavaScript prototype chain **FIGURE 1**



We can show that JavaScript recognizes this prototype chain by using the `instanceof` operator on our newly created objects. In each case, JavaScript is simply looking up the prototype chain to see if it finds a matching object. The last line in the example below simply shows how you can reverse the logic to go from the prototype down to the instance.

```
var v = new Dog();
console.log(v instanceof Dog);      // true
console.log(v instanceof Animal);   // true
console.log(v instanceof Object);   // true

Animal.prototype.isPrototypeOf(v);  // true
```

Calling Parent Methods

One thing we left out of our Animal-to-Dog relationship above is how we actually call the `Animal` constructor when a new `Dog` is created. The parent method **will not** be automatically called for you! The difficulty is that we can't simply call `Animal()` directly, because the context inside that function will not be correct. Calling a function like that would typically set `this` inside the function to point to the `window` object, our default context. That is not helpful. Instead, we have to manually call the `Animal` constructor function while changing its context to point to our new `Dog` instance.

```
function Animal(age) { /* ... */ }
// ...
function Dog(name, age) {
    Animal.apply(this, [age]);
    // ...
}

var v = new Dog("Vincent", 10);
```

By using the `apply` method on the `Animal` function object, we can change the context within the execution of the function. In other words, we change the value of `this` inside of the `Animal` function body. That's what the first argument to the `apply()` method is: our new context. From within the `Dog` function `this` points to the newly created `Dog` instance, and we are simply passing that reference as the context for the `Animal` function. The second argument to `apply()` is an array of the arguments that will be passed on to the called function (the `age` for the new `Dog`/`Animal`).

The `Animal` constructor function is now operating on the newly created `Dog` instance object instead of the `window`, and anything it adds to `this` will actually be added to the `Dog` instance.

A Side Note on `apply()`

The `apply()` method of functions is used to change context, and every single function object will have this method. However, you can also use the `call()` method to achieve the same goal; the arguments are just slightly different. Additionally, many frameworks and utility libraries have a way to "bind" a function to a different context.

Polymorphism

In the example above we see how we can call a parent method (the `Animal` constructor) from a child method (the `Dog` constructor). But this is not exactly polymorphism, since the two functions do not share a name or form (different method signatures). To show how JavaScript supports this concept, we need another example.

Notice how the `Animal` object has an `age` property. This is great, but perhaps we want the age to always be represented in "human years" (which we'll assume is the default). To do this, we might create an accessor method for the property. This does not need to be "privileged" because `age` is not private, but we could do that as well. Here is our new `Animal` definition:

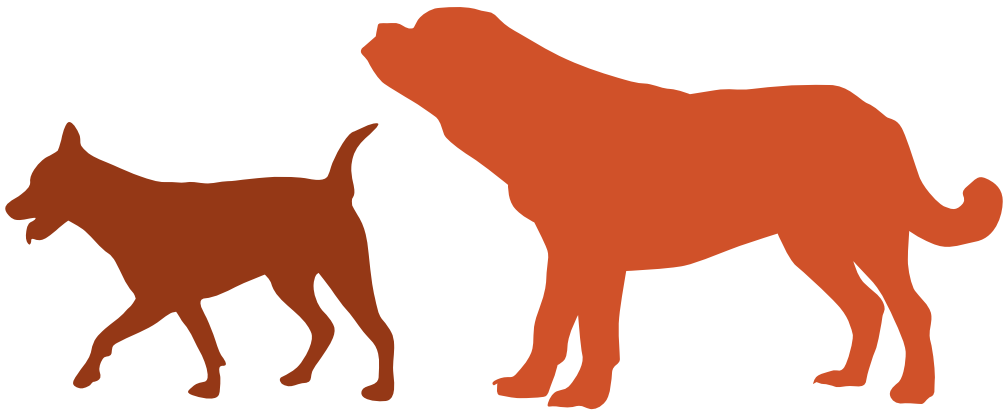
```
function Animal(age) {
    var alive = true;
    if (age) { this.age = age; }

    this.isAlive = function() { return alive; }
    this.die = function() { alive = false; }
}
Animal.prototype.age = 1;
Animal.prototype.getAge = function() { return this.age; }
```

We can see above that the `getAge` method is pretty simple currently: it returns the `age` property on `this` (the current `Animal` or one of its descendants). But for our `Dog` we want to return the age in human years, but store it in "dog years." This means our accessor method must multiply the `Dog`'s age by about 7. To accomplish this we'll override the `getAge` method, call the parent `Animal` `getAge` method, and then return the proper value in "human years."

```
function Dog(name, age) { /* ... */ }

Dog.prototype.getAge = function() {
    var dogYears = Animal.prototype.getAge.apply(this);
    return (dogYears * 7);
}
```

There are a couple things to note in the example above. First, we must use the method as it is defined on the `Animal.prototype` in order to call the parent method. If we try to simply call `this.getAge()` JavaScript will think we want the `Dog.prototype.getAge` method and we'll enter a recursive, infinite loop! Second, we have to use the `apply` method of the `getAge` function object in order to switch the context again. Remember, if we were to simply call the method as is (`Animal.prototype.getAge()`), then the context would be the object on the left side of the dot: `Animal.prototype`. It would not be our `Dog` instance!

Once we have retrieved the stored `age` value, we can simply return that value multiplied by 7. In this manner we are able to achieve polymorphism by having a method with the same name and same function signature, but with different implementations depending on the type of object (`Animal` versus `Dog` versus `Cat`).



Wrapping Things Up

There are many facets to object-oriented programming and we've really only hit on the big concepts in these two articles. We hope we've shown how the JavaScript object model implements those core concepts and have given you a deeper understanding of what is going on under the covers. We can see how the prototype allows for abstraction of the essence of modeled objects and encapsulation through object methods. We see inheritance in our prototype chain, and we can implement polymorphism through overridden methods within links of that chain.

That all said, you should not be implementing these practices directly in application code! Instead, use one of the many very good frameworks out there for dealing with object types, instances, inheritance, polymorphism, and other aspects of OOP. If you're curious to see how these are implemented (and seeing different implementations of each) you can look at prototypejs, Ember, dojo, MooTools, or ExtJS. Understanding what is happening under the hood should give you valuable insight when your framework of choice isn't doing what you think it should. It can help debug those situations and create workarounds, and could be very useful for library or framework authors. But implementing OOP on this level is cumbersome and fragile. Work within a framework to develop good practices within your application.

The Future, the Year 2000

There are some useful reference links at the end of this article, but before we get there I'd like to show you where ECMAScript (ES6, the specification that informs JavaScript) is headed with respect to OOP. In ES6 some new keywords will be added to make the creation of prototypes a bit easier for developers coming from class-based languages. However, this all just syntactic sugar. Nothing you have learned in this article is changing! Instead, a layer is being added on top of the `prototype` structure to simplify integration into JavaScript from other languages.

In the future, you will be able to create a "class" and specify various pieces, including the prototype chain, with a new `extends` keyword:

```
class Dog extends Animal {
  constructor(name, age) {
    super(age);  // Call parent method of the same name
    if (name) { this._name = name; }
  }

  get name() {
    return this._name + " (the dog)";
  }

  set name(value) {
    this._name = value.toLowerCase();
  }

  speak() {
    return this.name + " says woof";
  }
}
// for data members, we still have to do this...
Dog.prototype.name = "Bubbles";
```

There are lots of new things in here. Obviously we have the `class` and `extends` keywords, but also the explicitly named `constructor` function. Additionally, we can now use the `super` special method within a "class" to call parent methods with the correct context. The `get` and `set` keywords specify getter and setter methods for a given property such that when we use `someDog.name = "Bob";` our setter method is called, rather than direct property access. Technically, those last two keywords (`get` and `set`) are part of ES5 and are available in all evergreen browsers!

This may look very appealing to you, especially coming from a class-based language such as PHP. However, don't forget that this is just syntactic sugar! If you run into problems you'll still need to know how prototypical inheritance works, because that's what's going to be happening under the hood!

Shortly after it arrived at his home in 1993, **JORDAN** began disassembling his first computer - his mother was not happy. She breathed more easily when he moved from hardware into programming, starting with BASIC. Jordan's experience includes startups, companies large and small, and universities. He contributes to open source projects and participates in local user groups, barcamps, and hackathons. Jordan's primary mission for over 10 years has been to use JavaScript, HTML, and CSS to elevate web applications above their desktop rivals. He currently works as a Sr. JavaScript Engineer and Team lead for appendTo, a leader in front-end software solutions, specializing in Full Stack JavaScript, responsive web design, and mobile development. In his down time he enjoys puzzles of all sorts and board games.

Twitter: @jakerella

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine

books

conferences

training

phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.