

*Unearthing the Excellence in JavaScript*



# JavaScript: The Good Parts

O'REILLY®

YAHOO! PRESS

*Douglas Crockford*

## JavaScript: The Good Parts

by Douglas Crockford

Copyright © 2008 Yahoo! Inc. All rights reserved.  
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

**Editor:** Simon St.Laurent

**Production Editor:** Sumita Mukherji

**Copyeditor:** Genevieve d'Entremont

**Proofreader:** Sumita Mukherji

**Indexer:** Julie Hawks

**Cover Designer:** Karen Montgomery

**Interior Designer:** David Futato

**Illustrator:** Robert Romano

### Printing History:

May 2008: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *JavaScript: The Good Parts*, the image of a Plain Tiger butterfly, and related trade dress are trademarks of O'Reilly Media, Inc.

Java™ is a trademark of Sun Microsystems, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51774-8

[M]

This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to [permissions@oreilly.com](mailto:permissions@oreilly.com).

## CHAPTER 5

---

# Inheritance

*Divides one thing entire to many objects;  
Like perspectives, which rightly gazed upon  
Show nothing but confusion...*

—William Shakespeare, *The Tragedy of King Richard the Second*

Inheritance is an important topic in most programming languages.

In the classical languages (such as Java), inheritance (or extends) provides two useful services. First, it is a form of code reuse. If a new class is mostly similar to an existing class, you only have to specify the differences. Patterns of code reuse are extremely important because they have the potential to significantly reduce the cost of software development. The other benefit of classical inheritance is that it includes the specification of a system of types. This mostly frees the programmer from having to write explicit casting operations, which is a very good thing because when casting, the safety benefits of a type system are lost.

JavaScript, being a loosely typed language, never casts. The lineage of an object is irrelevant. What matters about an object is what it can do, not what it is descended from.

JavaScript provides a much richer set of code reuse patterns. It can ape the classical pattern, but it also supports other patterns that are more expressive. The set of possible inheritance patterns in JavaScript is vast. In this chapter, we'll look at a few of the most straightforward patterns. Much more complicated constructions are possible, but it is usually best to keep it simple.

In classical languages, objects are instances of classes, and a class can inherit from another class. JavaScript is a prototypal language, which means that objects inherit directly from other objects.

# Pseudoclassical

JavaScript is conflicted about its prototypal nature. Its prototype mechanism is obscured by some complicated syntactic business that looks vaguely classical. Instead of having objects inherit directly from other objects, an unnecessary level of indirection is inserted such that objects are produced by constructor functions.

When a function object is created, the Function constructor that produces the function object runs some code like this:

```
this.prototype = {constructor: this};
```

The new function object is given a prototype property whose value is an object containing a constructor property whose value is the new function object. The prototype object is the place where inherited traits are to be deposited. Every function gets a prototype object because the language does not provide a way of determining which functions are intended to be used as constructors. The constructor property is not useful. It is the prototype object that is important.

When a function is invoked with the constructor invocation pattern using the new prefix, this modifies the way in which the function is executed. If the new operator were a method instead of an operator, it could have been implemented like this:

```
Function.method('new', function () {  
  
    // Create a new object that inherits from the  
    // constructor's prototype.  
  
    var that = Object.beget(this.prototype)  
  
    // Invoke the constructor, binding -this- to  
    // the new object.  
  
    var other = this.apply(that, arguments);  
  
    // If its return value isn't an object,  
    // substitute the new object.  
  
    return (typeof other === 'object' && other) || that;  
});
```

We can define a constructor and augment its prototype:

```
var Mammal = function (name) {  
    this.name = name;  
};  
  
Mammal.prototype.get_name = function () {  
    return this.name;  
};
```

```
Mammal.prototype.says = function () {
    return this.saying || '';
};
```

Now, we can make an instance:

```
var myMammal = new Mammal('Herb the Mammal');
var name = myMammal.get_name(); // 'Herb the Mammal'
```

We can make another pseudoclass that inherits from `Mammal` by defining its constructor function and replacing its prototype with an instance of `Mammal`:

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
};

// Replace Cat.prototype with a new instance of Mammal
Cat.prototype = new Mammal();

// Augment the new prototype with
// purr and get_name methods.

Cat.prototype.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s += '-';
        }
        s += 'r';
    }
    return s;
};

Cat.prototype.get_name = function () {
    return this.says() + ' ' + this.name +
        ' ' + this.says();
};

var myCat = new Cat('Henrietta');
var says = myCat.says(); // 'meow'
var purr = myCat.purr(5); // 'r-r-r-r-r'
var name = myCat.get_name();
//      'meow Henrietta meow'
```

The pseudoclassical pattern was intended to look sort of object-oriented, but it is looking quite alien. We can hide some of the ugliness by using the `method` method and defining an `inherits` method:

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent();
    return this;
});
```

Our `inherits` and `method` methods return `this`, allowing us to program in a cascade style. We can now make our `Cat` with one statement.

```
var Cat = function (name) {
  this.name = name;
  this.saying = 'meow';
}.
  inherits(Mammal).
  method('purrr', function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
      if (s) {
        s += '-';
      }
      s += 'r';
    }
    return s;
  }).
  method('get_name', function () {
    return this.says() + ' ' + this.name +
      ' ' + this.says();
  });
```

By hiding the prototype jazz, it now looks a bit less alien. But have we really improved anything? We now have constructor functions that act like classes, but at the edges, there may be surprising behavior. There is no privacy; all properties are public. There is no access to super methods.

Even worse, there is a serious hazard with the use of constructor functions. If you forget to include the `new` prefix when calling a constructor function, then `this` will not be bound to a new object. Sadly, `this` will be bound to the global object, so instead of augmenting your new object, you will be clobbering global variables. That is really bad. There is no compile warning, and there is no runtime warning.

This is a serious design error in the language. To mitigate this problem, there is a convention that all constructor functions are named with an initial capital, and that nothing else is spelled with an initial capital. This gives us a prayer that visual inspection can find a missing `new`. A much better alternative is to not use `new` at all.

The pseudoclassical form can provide comfort to programmers who are unfamiliar with JavaScript, but it also hides the true nature of the language. The classically inspired notation can induce programmers to compose hierarchies that are unnecessarily deep and complicated. Much of the complexity of class hierarchies is motivated by the constraints of static type checking. JavaScript is completely free of those constraints. In classical languages, class inheritance is the only form of code reuse. JavaScript has more and better options.

# Object Specifiers

It sometimes happens that a constructor is given a very large number of parameters. This can be troublesome because it can be very difficult to remember the order of the arguments. In such cases, it can be much friendlier if we write the constructor to accept a single object specifier instead. That object contains the specification of the object to be constructed. So, instead of:

```
var myObject = maker(f, l, m, c, s);
```

we can write:

```
var myObject = maker({
  first: f,
  last: l,
  state: s,
  city: c
});
```

The arguments can now be listed in any order, arguments can be left out if the constructor is smart about defaults, and the code is much easier to read.

This can have a secondary benefit when working with JSON (see Appendix E). JSON text can only describe data, but sometimes the data represents an object, and it would be useful to associate the data with its methods. This can be done trivially if the constructor takes an object specifier because we can simply pass the JSON object to the constructor and it will return a fully constituted object.

# Prototypal

In a purely prototypal pattern, we dispense with classes. We focus instead on the objects. Prototypal inheritance is conceptually simpler than classical inheritance: a new object can inherit the properties of an old object. This is perhaps unfamiliar, but it is really easy to understand. You start by making a useful object. You can then make many more objects that are like that one. The classification process of breaking an application down into a set of nested abstract classes can be completely avoided.

Let's start by using an object literal to make a useful object:

```
var myMammal = {
  name : 'Herb the Mammal',
  get_name : function () {
    return this.name;
  },
  says : function () {
    return this.saying || '';
  }
};
```

Once we have an object that we like, we can make more instances with the `Object.beget` method from Chapter 3. We can then customize the new instances:

```
var myCat = Object.beget(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
myCat.purr = function (n) {
  var i, s = '';
  for (i = 0; i < n; i += 1) {
    if (s) {
      s += '-';
    }
    s += 'r';
  }
  return s;
};
myCat.get_name = function () {
  return this.says + ' ' + this.name + ' ' + this.says;
};
```

This is *differential inheritance*. By customizing a new object, we specify the differences from the object on which it is based.

Sometimes it is useful for data structures to inherit from other data structures. Here is an example: Suppose we are parsing a language such as JavaScript or T<sub>E</sub>X in which a pair of curly braces indicates a scope. Items defined in a scope are not visible outside of the scope. In a sense, an inner scope inherits from its outer scope. JavaScript objects are very good at representing this relationship. The `block` function is called when a left curly brace is encountered. The parse function will look up symbols from scope, and augment scope when it defines new symbols:

```
var block = function () {

  // Remember the current scope. Make a new scope that
  // includes everything from the current one.

  var oldScope = scope;
  scope = Object.beget(scope);

  // Advance past the left curly brace.

  advance('{');

  // Parse using the new scope.

  parse(scope);

  // Advance past the right curly brace and discard the
  // new scope, restoring the old one.

  advance('}');
  scope = oldScope;
};
```

# Functional

One weakness of the inheritance patterns we have seen so far is that we get no privacy. All properties of an object are visible. We get no private variables and no private methods. Sometimes that doesn't matter, but sometimes it matters a lot. In frustration, some uninformed programmers have adopted a pattern of *pretend privacy*. If they have a property that they wish to make private, they give it an odd-looking name, with the hope that other users of the code will pretend that they cannot see the odd looking members. Fortunately, we have a much better alternative in an application of the module pattern.

We start by making a function that will produce objects. We will give it a name that starts with a lowercase letter because it will not require the use of the new prefix. The function contains four steps:

1. It creates a new object. There are lots of ways to make an object. It can make an object literal, or it can call a constructor function with the new prefix, or it can use the `Object.getPrototypeOf` method to make a new instance from an existing object, or it can call any function that returns an object.
2. It optionally defines private instance variables and methods. These are just ordinary vars of the function.
3. It augments that new object with methods. Those methods will have privileged access to the parameters and the vars defined in the second step.
4. It returns that new object.

Here is a pseudocode template for a functional constructor (boldface text added for emphasis):

```
var constructor = function (spec, my) {  
  var that, other private instance variables;  
  my = my || {};  
  
  Add shared variables and functions to my  
  
  that = a new object;  
  
  Add privileged methods to that  
  
  return that;  
}
```

The `spec` object contains all of the information that the constructor needs to make an instance. The contents of the `spec` could be copied into private variables or transformed by other functions. Or the methods can access information from `spec` as they need it. (A simplification is to replace `spec` with a single value. This is useful when the object being constructed does not need a whole `spec` object.)

The `my` object is a container of secrets that are shared by the constructors in the inheritance chain. The use of the `my` object is optional. If a `my` object is not passed in, then a `my` object is made.

Next, declare the private instance variables and private methods for the object. This is done by simply declaring variables. The variables and inner functions of the constructor become the private members of the instance. The inner functions have access to `spec` and `my` and `that` and the private variables.

Next, add the shared secrets to the `my` object. This is done by assignment:

```
my.member = value;
```

Now, we make a new object and assign it to `that`. There are lots of ways to make a new object. We can use an object literal. We can call a pseudoclassical constructor with the `new` operator. We can use the `Object.getPrototypeOf` method on a prototype object. Or, we can call another functional constructor, passing it a `spec` object (possibly the same `spec` object that was passed to this constructor) and the `my` object. The `my` object allows the other constructor to share the material that we put into `my`. The other constructor may also put its own shared secrets into `my` so that our constructor can take advantage of it.

Next, we augment `that`, adding the privileged methods that make up the object's interface. We can assign new functions to members of `that`. Or, more securely, we can define the functions first as private methods, and then assign them to `that`:

```
var methodical = function () {  
    ...  
};  
that.methodical = methodical;
```

The advantage to defining `methodical` in two steps is that if other methods want to call `methodical`, they can call `methodical()` instead of `that.methodical()`. If the instance is damaged or tampered with so that `that.methodical` is replaced, the methods that call `methodical` will continue to work the same because their private `methodical` is not affected by modification of the instance.

Finally, we return `that`.

Let's apply this pattern to our `mammal` example. We don't need `my` here, so we'll just leave it out, but we will use a `spec` object.

The `name` and `saying` properties are now completely private. They are accessible only via the privileged `get_name` and `says` methods:

```
var mammal = function (spec) {  
    var that = {};  
  
    that.get_name = function () {  
        return spec.name;  
    };  
};
```

```

    that.says = function () {
        return spec.saying || '';
    };

    return that;
};

var myMammal = mammal({name: 'Herb'});

```

In the pseudoclassical pattern, the `Cat` constructor function had to duplicate work that was done by the `Mammal` constructor. That isn't necessary in the functional pattern because the `Cat` constructor will call the `Mammal` constructor, letting `Mammal` do most of the work of object creation, so `Cat` only has to concern itself with the differences:

```

var cat = function (spec) {
    spec.saying = spec.saying || 'meow';
    var that = mammal(spec);
    that.purr = function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
            if (s) {
                s += '-';
            }
            s += 'r';
        }
        return s;
    };
    that.get_name = function () {
        return that.says() + ' ' + spec.name +
            ' ' + that.says();
    };
    return that;
};

var myCat = cat({name: 'Henrietta'});

```

The functional pattern also gives us a way to deal with super methods. We will make a superior method that takes a method name and returns a function that invokes that method. The function will invoke the original method even if the property is changed:

```

Object.method('superior', function (name) {
    var that = this,
        method = that[name];
    return function () {
        return method.apply(that, arguments);
    };
});

```

Let's try it out on a `coolcat` that is just like `cat` except it has a cooler `get_name` method that calls the super method. It requires just a little bit of preparation. We will declare a `super_get_name` variable and assign it the result of invoking the superior method:

```

var coolcat = function (spec) {
  var that = cat(spec),
      super_get_name = that.superior('get_name');
  that.get_name = function (n) {
    return 'like ' + super_get_name() + ' baby';
  };
  return that;
};

var myCoolCat = coolcat({name: 'Bix'});
var name = myCoolCat.get_name();
//      'like meow Bix meow baby'

```

The functional pattern has a great deal of flexibility. It requires less effort than the pseudoclassical pattern, and gives us better encapsulation and information hiding and access to super methods.

If all of the state of an object is private, then the object is tamper-proof. Properties of the object can be replaced or deleted, but the integrity of the object is not compromised. If we create an object in the functional style, and if all of the methods of the object make no use of this or that, then the object is *durable*. A durable object is simply a collection of functions that act as *capabilities*.

A durable object cannot be compromised. Access to a durable object does not give an attacker the ability to access the internal state of the object except as permitted by the methods.

## Parts

We can compose objects out of sets of parts. For example, we can make a function that can add simple event processing features to any object. It adds an `on` method, a `fire` method, and a private event registry:

```

var eventuality = function (that) {
  var registry = {};

  that.fire = function (event) {

    // Fire an event on an object. The event can be either
    // a string containing the name of the event or an
    // object containing a type property containing the
    // name of the event. Handlers registered by the 'on'
    // method that match the event name will be invoked.

    var array,
        func,
        handler,
        i,
        type = typeof event === 'string' ?
            event : event.type;

```

```

// If an array of handlers exist for this event, then
// loop through it and execute the handlers in order.

    if (registry.hasOwnProperty(type)) {
        array = registry[type];
        for (i = 0; i < array.length; i += 1) {
            handler = array[i];

// A handler record contains a method and an optional
// array of parameters. If the method is a name, look
// up the function.

            func = handler.method;
            if (typeof func === 'string') {
                func = this[func];
            }

// Invoke a handler. If the record contained
// parameters, then pass them. Otherwise, pass the
// event object.

            func.apply(this,
                handler.parameters || [event]);
        }
    }
    return this;
};

that.on = function (type, method, parameters) {

// Register an event. Make a handler record. Put it
// in a handler array, making one if it doesn't yet
// exist for this type.

    var handler = {
        method: method,
        parameters: parameters
    };
    if (registry.hasOwnProperty(type)) {
        registry[type].push(handler);
    } else {
        registry[type] = [handler];
    }
    return this;
};
return that;
};

```

We could call `eventuality` on any individual object, bestowing it with event handling methods. We could also call it in a constructor function before that is returned:

```
eventuality(that);
```

In this way, a constructor could assemble objects from a set of parts. JavaScript's loose typing is a big benefit here because we are not burdened with a type system that is concerned about the lineage of classes. Instead, we can focus on the character of their contents.

If we wanted eventuality to have access to the object's private state, we could pass it the `my` bundle.