# Classical Inheritance in JavaScript

Douglas Crockford
www.crockford.com

*And you think you're so clever and classless and free*

— John Lennon

JavaScript is a *class-free*, object-oriented language, and as such, it uses prototypal inheritance instead of classical inheritance. This can be puzzling to programmers trained in conventional object-oriented languages like C++ and Java. JavaScript's prototypal inheritance has more expressive power than classical inheritance, as we will see presently.

| Java | JavaScript |
|---|---|
| Strongly-typed | Loosely-typed |
| Static | Dynamic |
| Classical | Prototypal |
| Classes | Functions |
| Constructors | Functions |
| Methods | Functions |

But first, why do we care about inheritance at all? There are primarily two reasons. The first is type convenience. We want the language system to automatically *cast* references of similar classes. Little type-safety is obtained from a type system which requires the routine explicit casting of object references. This is of critical importance in strongly-typed languages, but it is irrelevant in loosely-typed languages like JavaScript, where object references never need casting.

The second reason is code reuse. It is very common to have a quantity of objects all implementing exactly the same methods. Classes make it possible to create them all from a single set of definitions. It is also common to have objects that are similar to some other objects, but differing only in the addition or modification of a small number of methods. Classical inheritance is useful for this but prototypal inheritance is even more useful.

To demonstrate this, we will introduce a little sugar which will let us write in a style that resembles a conventional classical language. We will then show useful patterns which are not available in classical languages. Then finally, we will explain the sugar.

## Classical Inheritance

First, we will make a `Parenizor` class that will have set and get methods for its `value`, and a `toString` method that will wrap the `value` in parens.

```
function Parenizor(value) {
    this.setValue(value);
}

Parenizor.method('setValue', function (value) {
    this.value = value;
    return this;
});
```

```
Parenizor.method('getValue', function () {
    return this.value;
});

Parenizor.method('toString', function () {
    return '(' + this.getValue() + ')';
});
```

The syntax is a little unusual, but it is easy to recognize the classical pattern in it. The `method` method takes a method name and a function, adding them to the class as a public method.

So now we can write

```
myParenizor = new Parenizor(0);
myString = myParenizor.toString();
```

As you would expect, `myString` is `"(0)"`.

Now we will make another class which will inherit from `Parenizor`, which is the same except that its `toString` method will produce `"-0-"` if the `value` is zero or empty.

```
function ZParenizor(value) {
    this.setValue(value);
}

ZParenizor.inherits(Parenizor);

ZParenizor.method('toString', function () {
    if (this.getValue()) {
        return this.uber('toString');
    }
    return "-0-";
});
```

The `inherits` method is similar to Java's `extends`. The `uber` method is similar to Java's `super`. It lets a method call a method of the parent class. (The names have been changed to avoid reserved word restrictions.)

So now we can write

```
myZParenizor = new ZParenizor(0);
myString = myZParenizor.toString();
```

This time, `myString` is `"-0-"`.

JavaScript does not have classes, but we can program as though it does.

# Multiple Inheritance

By manipulating a function's `prototype` object, we can implement multiple inheritance, allowing us to make a class built from the methods of multiple classes. Promiscuous multiple inheritance can be difficult to implement and can potentially suffer from method name collisions. We could implement promiscuous multiple inheritance in JavaScript, but for this example we will use a more disciplined form called *Swiss Inheritance*.

Suppose there is a `NumberValue` class that has a `setValue` method that checks that the `value` is a number in a certain range, throwing an exception if necessary. We only want its `setValue` and `setRange` methods for our `ZParenizor`. We certainly don't want its `toString` method. So, we write

```
ZParenizor.swiss(NumberValue, 'setValue', 'setRange');
```

This adds only the requested methods to our class.

## Parasitic Inheritance

There is another way to write `ZParenizor`. Instead of inheriting from `Parenizor`, we write a constructor that calls the `Parenizor` constructor, passing off the result as its own. And instead of adding public methods, the constructor adds privileged methods.

```
function ZParenizor2(value) {
    var that = new Parenizor(value);
    that.toString = function () {
        if (this.getValue()) {
            return this.uber('toString');
        }
        return "-0-"
    };
    return that;
}
```

Classical inheritance is about the *is-a* relationship, and parasitic inheritance is about the *was-a-but-now's-a* relationship. The constructor has a larger role in the construction of the object. Notice that the `uber` née `super` method is still available to the privileged methods.

## Class Augmentation

JavaScript's dynamism allows us to add or replace methods of an existing class. We can call the `method` method at any time, and all present and future instances of the class will have that method. We can literally extend a class at any time. Inheritance works retroactively. We call this *Class Augmentation* to avoid confusion with Java's `extends`, which means something else.

## Object Augmentation

In the static object-oriented languages, if you want an object which is slightly different than another object, you need to define a new class. In JavaScript, you can add methods to individual objects without the need for additional classes. This has enormous power because you can write far fewer classes and the classes you do write can be much simpler. Recall that JavaScript objects are like hashtables. You can add new values at any time. If the value is a function, then it becomes a method.

So in the example above, I didn't need a `ZParenizor` class at all. I could have simply modified my instance.

```
myParenizor = new Parenizor(0);
myParenizor.toString = function () {
    if (this.getValue()) {
        return this.uber('toString');
    }
```

```
        return "-0-";
    };
    myString = myParenizor.toString();
```

We added a `toString` method to our `myParenizor` instance without using any form of inheritance. We can evolve individual instances because the language is class-free.

# Sugar

To make the examples above work, I wrote four sugar methods. First, the `method` method, which adds an instance method to a class.

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

This adds a public method to the `Function.prototype`, so all functions get it by Class Augmentation. It takes a name and a function, and adds them to a function's `prototype` object.

It returns `this`. When I write a method that doesn't need to return a value, I usually have it return `this`. It allows for a cascade-style of programming.

Next comes the `inherits` method, which indicates that one class inherits from another. It should be called after both classes are defined, but before the inheriting class's methods are added.

```
Function.method('inherits', function (parent) {
    var d = {}, p = (this.prototype = new parent());
    this.method('uber', function uber(name) {
        if (!(name in d)) {
            d[name] = 0;
        }
        var f, r, t = d[name], v = parent.prototype;
        if (t) {
            while (t) {
                v = v.constructor.prototype;
                t -= 1;
            }
            f = v[name];
        } else {
            f = p[name];
            if (f == this[name]) {
                f = v[name];
            }
        }
        d[name] += 1;
        r = f.apply(this, Array.prototype.slice.apply(arguments, [1]));
        d[name] -= 1;
        return r;
    });
    return this;
});
```

Again, we augment `Function`. We make an instance of the `parent` class and use it as the new `prototype`. We also correct the `constructor` field, and we add the `uber` method to the `prototype` as well.

The `uber` method looks for the named method in its own `prototype`. This is the function to invoke in the case of Parasitic Inheritance or Object Augmentation. If we are doing Classical Inheritance, then we need to find the function in the `parent`'s `prototype`. The `return` statement uses the function's `apply` method to invoke the function, explicitly setting `this` and passing an array of parameters. The parameters (if any) are obtained from the `arguments` array. Unfortunately, the `arguments` array is not a true array, so we have to use `apply` again to invoke the array `slice` method.

Finally, the `swiss` method.

```
Function.method('swiss', function (parent) {
    for (var i = 1; i < arguments.length; i += 1) {
        var name = arguments[i];
        this.prototype[name] = parent.prototype[name];
    }
    return this;
});
```

The `swiss` method loops through the `arguments`. For each `name`, it copies a member from the `parent`'s `prototype` to the new class's `prototype`.

# Conclusion

JavaScript can be used like a classical language, but it also has a level of expressiveness which is quite unique. We have looked at Classical Inheritance, Swiss Inheritance, Parasitic Inheritance, Class Augmentation, and Object Augmentation. This large set of code reuse patterns comes from a language which is considered smaller and simpler than Java.

Classical objects are hard. The only way to add a new member to a hard object is to create a new class. In JavaScript, objects are soft. A new member can be added to a soft object by simple assignment.

Because objects in JavaScript are so flexible, you will want to think differently about class hierarchies. Deep hierarchies are inappropriate. Shallow hierarchies are efficient and expressive.

I have been writing JavaScript for 8 years now, and I have never once found need to use an `uber` function. The *super* idea is fairly important in the classical pattern, but it appears to be unnecessary in the prototypal and functional patterns. I now see my early attempts to support the classical model in JavaScript as a mistake.