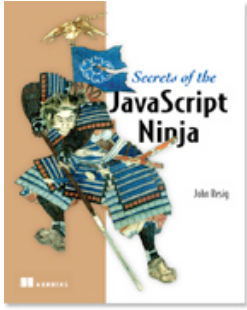


# Instantiation and Prototypes

	<p><b>RELATED READING</b></p> <p>Secrets of the JavaScript Ninja John Resig</p> <p>MEAP Release: March 2008 Paper book: January 2009 (est.)   300 pages ISBN: 193398869X</p>
---	--

*This article is excerpted from the upcoming title *The Secrets of JavaScript Ninjas* by John Resig and published by Manning Publications. This article introduces the technique of instantiating a function to give its prototype property functionality. For the table of contents, source code, the Author Forum, and other resources, go to [www.manning.com/resig](http://www.manning.com/resig).*

All functions have a `prototype` property which, by default, contains an empty object. This property doesn't serve a purpose until the function is instantiated. In order to understand what it does, it's important to remember the simple truth about JavaScript: Functions serve a dual purpose. They can behave both as a regular function and as a "class" (where they can be instantiated).

## Instantiation

Let's examine the simplest case of using the prototype property to add functionality to an instance of a function (see listing 1).

### Listing 1: Creating an instance of a function that has a prototyped method

```
function Ninja() {}

Ninja.prototype.swingSword = function() {
  return true;
};

var ninja1 = Ninja();
assert( !ninja1, "Is undefined, not an instance of Ninja." );

var ninja2 = new Ninja();
assert( ninja2.swingSword(), "Method exists and is callable." );
```

There's two points to learn from this example: First that in order to produce an instance of a function (something that's an object, has instance properties, and has prototyped properties) you have to call the function with the `new` operator. Second we can see that the `swingSword` function has now become a property of the `ninja2` `Ninja` instance.

In some respects the base function (in this case, `Ninja`) could be considered a "constructor" (since it's called whenever the `new` operator is used upon it). This also means that when the function is called with the `new` operator it's `this` context is equal to the instance of the object itself. Let's examine this a little bit more, as shown in listing 2.

### Listing 2: Extending an object with both a prototyped method and a method within the constructor function

```
function Ninja() {
  this.swung = false;

  // Should return true
  this.swingSword = function() {
    return !!this.swung;
  };
}
```

```
// Should return false, but will be overridden
Ninja.prototype.swingSword = function(){
    return this.swung;
};

var ninja = new Ninja();
assert( ninja.swingSword(), "Calling the instance method, not the prototype method." );
```

In this listing we're doing an operation very similar to the previous example: We add a new method by adding it to the prototype property. However, we also add a new method within the constructor function itself. The order of operations is as such:

1. Properties are bound to the object instance from the prototype.
2. Properties are bound to the object instance within the constructor function.

This is why the `swingSword` function ends up returning `true` (since the binding operations within the constructor always take precedence). Since this context, within the constructor, refers to the instance itself we can feel free to modify it until our heart's content.

An important thing to realize about a function's prototype is that, unlike instance properties which are static, it will continue to update, live, as its changed--even against all existing instance of the object. For example, if we were to take some of the code in Listing 2 and re-order it, it would still work as we would expect it to, as shown in listing 3.

### Listing 3: The prototype continues to update instance properties live even after they've already been created

```
function Ninja(){
    this.swung = true;
}

var ninja = new Ninja();

Ninja.prototype.swingSword = function(){
    return this.swung;
};

assert( ninja.swingSword(), "Method exists, even out of order." );
```

These, seamless, live updates gives us an incredible amount of power and extensibility, to a degree at which isn't typically found in other languages. Allowing for these live updates it makes it possible for you to create a functional framework which users can extend with further functionality - even well after any objects have been instantiated.

## Object Type

Since we now have this new instance of our function it becomes important to know which function constructed the object instance, so we can refer back to it later (possibly even performing a form of type checking, if need be). See listing 4.

### Listing 4: Examining the type of an instance and its constructor

```
function Ninja(){}
```

```
var ninja = new Ninja();
assert( typeof ninja == "object", "However the type of the instance is still an object." );
assert( ninja instanceof Ninja, "The object was instantiated properly." );
assert( ninja.constructor == Ninja, "The ninja object was created by the Ninja function." );
```

In this listing, we start by examining the type of a function instance. Note that the `typeof` operator isn't of much use to us here; all instance will be objects, thus always returning "object" as its result. However, the `instanceof` operator really helps in this case, giving us a clear way to determine if an instance was created by a particular function.

On top of this we also make use of an object property: `.constructor`. This is a property that exists on all objects and offers a reference back to the original function that created it. We can use this to verify the origin of the instance (much like how we do with the `instanceof` operator). Additionally, since this is just a reference back to the original function, we can, once-again, instantiate a new Ninja object, like in listing 5.

### Listing 5: Instantiating a new object using only a reference to its constructor

```
var ninja = new Ninja();
var ninja2 = new ninja.constructor();

assert( ninja2 instanceof Ninja, "Still a ninja object." );
```

What's especially interesting about this is that we can do this without ever having access to the original function; taking place completely behind the scenes.

## Inheritance and the Prototype Chain

There's an additional feature of the `instanceof` operator that we can use to our advantage when performing object inheritance. In order to make use of it, however, we need to understand how the prototype chain works in JavaScript (see listing 6).

### Listing 6: Different ways of copying functionality onto a function's prototype

```
function Person(){
  Person.prototype.dance = function(){};

  function Ninja(){

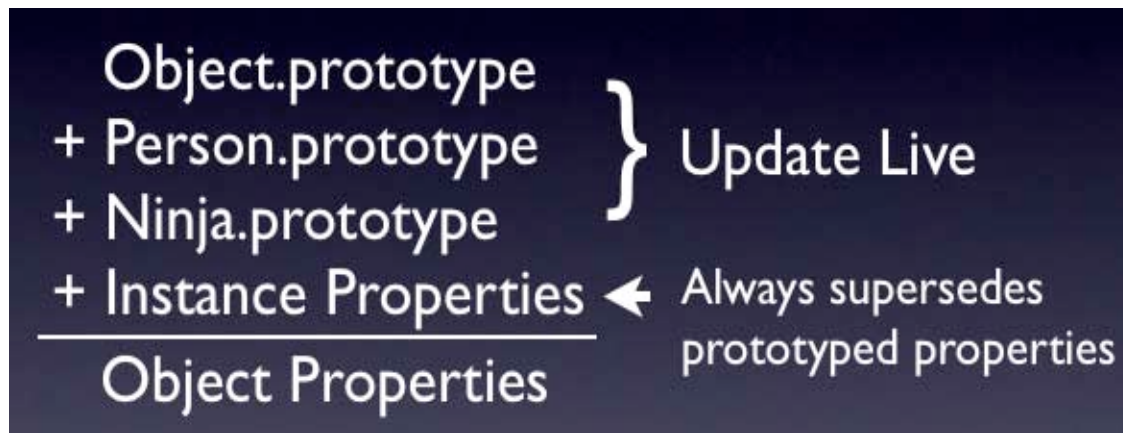
    // Achieve similar, but non-inheritable, results
    Ninja.prototype = Person.prototype;
    Ninja.prototype = { dance: Person.prototype.dance };

    // Only this maintains the prototype chain
    Ninja.prototype = new Person();

    var ninja = new Ninja();
    assert( ninja instanceof Ninja, "ninja receives functionality from the Ninja prototype" );
    assert( ninja instanceof Person, "... and the Person prototype" );
    assert( ninja instanceof Object, "... and the Object prototype" );
  }
}
```

Since the prototype of a function is just an object, there are multiple ways of copying functionality (such as properties or methods). However only one technique is capable of creating a prototype 'chain': `SubFunction.prototype = new SuperFunction()`; This means that when you perform an `instanceof` operation you can determine if the function inherits the functionality of any object in its prototype chain.

An additional side-effect of doing prototype inheritance in this manner is that all inherited function prototypes will continue to update live. The result is something akin to what's shown in the following illustration:



It's important to note in this figure that our object has properties that are inherited from the Object prototype. All native objects (such as Object, Array, String, Number, RegExp, and Function) have prototype properties which can be manipulated and extended. This proves to be an incredibly powerful feature of the language. Using this you can extend the functionality of the language itself, introducing new, or missing, pieces of the language.

For example, one such case where this becomes quite useful is with some of the upcoming features of JavaScript 2. The next version of the JavaScript language introduces a couple helper methods, including some for Arrays. We can duplicate this functionality, right now, completely circumventing the need to wait until the next version of the language is ready, as shown in listing 7.

### Listing 7: Implementing the JavaScript 2 array forEach method in a future-compatible manner

```
if (!Array.prototype.forEach) {
  Array.prototype.forEach = function(fn) {
    for ( var i = 0; i < this.length; i++ ) {
      fn( this[i], i, this );
    }
  };
}

["a", "b", "c"].forEach(function(value, index, array){
  assert( value, "Is in position " + index + " out of " + (array.length - 1) );
});
```

Since all the built-in objects include these prototypes it ends up giving you all the power necessary to extend the language to your desire.

An important point to remember when implementing properties or methods on native objects is that introducing them is every bit as dangerous as introducing new variables into the global scope. Since there's only ever one instance of a native object there still significant possibility for naming collisions to occur.

When implementing features on native prototypes that are forward-looking (such as the previously-mentioned implementation of `Array.prototype.forEach`) there's a strong possibility that your implementation won't match the final implementation (causing issues to occur when a browser finally does implement the functionality correctly). You should always take great care when treading in those waters.

## HTML Prototypes

There's a fun feature in Internet Explorer 8, Firefox, Safari, and Opera which provides base functions representing objects in the DOM. By making these accessible the browser is providing you with the ability to extend any HTML node of your choosing, like in listing 8.

### Listing 8: Adding a new method to all HTML elements via the HTML element prototype

```
<div id="a">I'm going to be removed.</div> <div id="b">Me too!</div>
<script>
HTMLElement.prototype.remove = function(){
    if ( this.parentNode )
        this.parentNode.removeChild( this );
};

// Old way
var a = document.getElementById("a");
a.parentNode.removeChild( a );

// New way
document.getElementById("b").remove();
</script>
```

More information about this particular feature can be found in the HTML 5 specification:

<http://www.whatwg.org/specs/web-apps/current-work/multipage/section-elements.html>

One library that makes heavy use of this feature is the Prototype library, adding all forms of functionality onto existing DOM elements (including the ability to inject HTML, manipulate CSS, amongst other features). The most important thing to realize, when working with these prototypes, is that they don't exist in older versions of Internet Explorer. If that isn't a target platform for you, then these features should serve you well.

Another point that often comes in contention is the question of whether HTML elements can be instantiated directly from their constructor function, perhaps something like this:

```
var elem = new HTMLElement();
```

However, that does not work at all. Even though browsers expose the root function and prototype, they selectively disable the ability to actually call the root function (presumably to limit element creation to internal functionality, only).

Save for the overwhelming difficulty that this feature presents, in platform compatibility, the benefits of clean code can be quite dramatic and should be strongly investigated for specific situations, where applicable.