<http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-On-7601>

* [**Introduction**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-Introductio-24059)
* [**The Base Class**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-Th-28215)
* [**Adding a Method**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-Addin-32215)
* [**Creating a Subclass**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-Creatin-49778)
* [**Calling the Base Constructor**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-Callin-58934)
* [**One More Subclass**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-On-7601)
* [**The Code**](http://kevlindev.com/tutorials/javascript/inheritance/index.htm#Anchor-Th-24351)

**Introduction**

The following is a discussion of a technique I've us ed over the years to simulate the feeling of Object-Oriented Programming in languages like C# and Java. The keyword here is "simulation". This is not necessarily equivalent to what you would encounter in those languages. JavaScript is a very expressive language and, as a result, there are a wide variety of techniques you can employ to build re-usable and extensible code. What I describe here is by no means the final answer on that subject.

As a side note, this article will use OOP terminology common to Java and C#; however, this vernacular is technically incorrect when discussing JavaScript in general. So, if you find it offensive to mix that terminology with JS then close this page now! For the rest of you, keep in mind that I am merely presenting an idiom. I tend to build libraries, so this particular style of JS coding fits my needs well. Others will argue that it is pointless to write JS in this way. If you decide to use this approach or even to roll your own OOP JS, be prepared for debates much like the tab vs. whitespace argument. OK, that's enough ranting, so let's jump in!

NOTE: This tutorial has been updated. If you're looking for the original Inheritance tutorial, you can find that [**here**](http://kevlindev.com/tutorials/javascript/inheritance/inheritance10.htm), with warts and all.

**The Base Class**

The first thing we need is a base "class" that can create instances via the "new" operator. The identifier following the "new" operator is the object constructor and any JavaScript function can serve in this capacity. For instance, the following code creates a Person function and then instantiates a new Person object.

function Person(first, last) {

this.first = first;

this.last = last;

}

var person = new Person("John", "Dough");

The "new" operator will invoke your constructor passing in the specified parameters. A special variable named "this" will exist within the scope of the constructor. "This" points to the current object instance (sometimes referred to as the activation object) with which you are working. Using "this" you can add and modify properties on the instance being created by the constructor. We now have our base "class": Person.

**Adding a Method**

Typically you'll want to add methods to your class to manipulate instances of that class. For example, it is often useful to create a "toString" method so you can display the content of your object in an "alert". The following adds such a method to our base class implementation.

Person.prototype.toString = function() {

return this.first + " " + this.last;

}

alert( person ); // displays "John Dough"

This demonstrates one of the unique qualities of the JavaScript language and that is the concept of prototypes. In a nutshell, all JS objects have a private prototype property. I say "private" because this property is not directly visible or accessible from the object itself. When the runtime performs a lookup on a property, it first looks at the instance to see if that property is defined there. If it is not defined there, the runtime looks for the property on the object's private prototype. Prototypes can have their own private prototype, so the search can keep working its way up the prototype chain. This continues until either the property is located or we reach a null private prototype, in which case, the property is not defined.

When we call "new", the runtime creates a new object instance which is passed into the constructor as discussed earlier. However, it also assigns the constructor's prototype property (the publicly accessible one) to the new instance's private prototype. By defining properties on the constructor's prototype, like Person.prototype.toString above, we are creating properties that will be in the private prototype chain of each new instance. These become, in effect, instance methods.

**Creating a Subclass**

Let's say we like the Person object, but we need more information when that person acts as an employee. It's time for a subclass of Person which we define below.

KevLinDev.extend = function(subClass, baseClass) {

function inheritance() {}

inheritance.prototype = baseClass.prototype;

subClass.prototype = new inheritance();

subClass.prototype.constructor = subClass;

subClass.baseConstructor = baseClass;

subClass.superClass = baseClass.prototype;

}

function Employee(first, last, id) {

// initialize properties here

}

KevLinDev.extend(Employee, Person);

We begin by creating a convenience function to setup our inheritance chain. We need the subClass.prototype property to be equivalent to baseClass.prototype. (Remember, when we call "new" that prototype property will be copied to the new instance's private prototype, thus hooking up its inheritance chain.) However, we can't allow both the subClass.prototype and the baseClass.prototype to point to the same prototype object. That would mean adding methods to one class would add it to the other. That's fine if we add to the base class, but the base class would also "inherit" methods we added to the subClass. We need a way disconnect the two prototypes while still allowing the subClass to refer to the baseClass. Once again, the prototype chain comes to the rescue.

We start by defining a nested function called "inheritance". Next, we point its prototype to the baseClass prototype. This means that any "new" instances of "inheritance" will have their private prototype pointing to that base class' public prototype. Finally, we create a new instance of "inheritance" and assign that to the subClass prototype. Now when we create a new instance of subClass, the instance's private prototype will point to the "inheritance" instance. The "inheritance" instance's private prototype points to the base class' public prototype. This means that changes to baseClass.prototype will propagate to subclass instances through the inheritance chain and since we created a new object for the subclass' prototype property, we can add to subClass.prototype without affecting the base class prototype.

Whenever you create a new object instance in JavaScript, the instance's local "constructor" property points to the function (constructor) that was invoked to create the instance. I often compare an instance's constructor property against a function to test the "type" of an object instance. Calling "new inheritance()" points subClass.prototype.constructor to the nested "inheritance" function which would break my tests. I fix this by updating the constructor property to point to my subClass constructor.

The final two lines are used as convenience properties when invoking ancestor constructors and methods. These will be demonstrated in the next section.

**Calling the Base Constructor**

We've defined our new subClass, but we still need to initialize it properly when calling "new". Ideally, we should never have to place code from Person into our subclass, Employee. We need a way to pass "first" and "last" into Person while letting the Employee constructor handle the "id" argument. The following code shows how this is done.

function Employee(first, last, id) {

Employee.baseConstructor.call(this, first, last);

this.id = id;

}

Admittedly, this is a bit ugly. However, we simply invoke the super constructor via "call". This calls the base class constructor as if it were a method of the first parameter, "this". The remaining parameters are passed in as arguments to the function being invoked. So, in this case, the base class constructor, Person, will perform any processing on "first" and "last" and the Employee constructor will handle "id".

**One More Subclass**

I've found that its easy to build a JS OOP idiom to support one level of inheritance that fails with two more more levels of inheritiance. Lets create one more subclass just to make sure our logic is generalizing properly. Our next subclass, Manager, will extend Employee by adding a "department" property. The new subclass is defined below.

function Manager(first, last, id, department) {

Manager.baseConstructor.call(this, first, last, id);

this.department = department;

}

KevLinDev.extend(Manager, Employee);

Nothing new here. This code looks practically identical to our Employee definition and running the sample code below shows that our inheritance is working properly.

**The Code**

You can download the "extend" method and a sample test file here: [**inheritance.zip**](http://kevlindev.com/tutorials/javascript/inheritance/inheritance.zip). The results of that test can be seen on [**example.htm**](http://kevlindev.com/tutorials/javascript/inheritance/example.htm) (very simple). I've also included the sample JS code inline on this page. Additionally, this code shows the idiom used to invoke overridden base methods from sub-class (see the toString methods).

/\*\*

\* inheritance

\*

\* @author Kevin Lindsey

\* @version 1.0

\*

\* copyright 2006, Kevin Lindsey

\*

\*/

// namespace placeholder

KevLinDev = {};

/\*\*

\* A function used to extend one class with another

\*

\* @param {Object} subClass

\* The inheriting class, or subclass

\* @param {Object} baseClass

\* The class from which to inherit

\*/

KevLinDev.extend = function(subClass, baseClass) {

function inheritance() {}

inheritance.prototype = baseClass.prototype;

subClass.prototype = new inheritance();

subClass.prototype.constructor = subClass;

subClass.baseConstructor = baseClass;

subClass.superClass = baseClass.prototype;

}

/\*

\* Person class

\*/

/\*\*

\* Person constructor

\*

\* @param {String} first

\* The person's first name

\* @param {String} last

\* The person's last name

\*/

function Person(first, last) {

this.first = first;

this.last = last;

}

/\*\*

\* Create a string representation of this object

\*

\* @return {String} A string representation of this object

\*/

Person.prototype.toString = function() {

return this.first + " " + this.last;

};

/\*

\* Employee class

\*/

/\*\*

\* Employee constructor

\*

\* @param {String} first

\* The employee's first name

\* @param {String} last

\* The employee's last name

\* @param {Number} id

\* The employee's number

\*/

function Employee(first, last, id) {

Employee.baseConstructor.call(this, first, last);

this.id = id;

}

// subclass Person

KevLinDev.extend(Employee, Person);

/\*\*

\* Create a string representation of this object

\*

\* @return {String} A string representation of this object

\*/

Employee.prototype.toString = function() {

return Employee.superClass.toString.call(this) + ": " + this.id;

};

/\*

\* Manager

\*/

/\*\*

\* Manager constructor

\*

\* @param {String} first

\* The manager's first name

\* @param {String} last

\* The manager's last name

\* @param {Number} id

\* The manager's employee number

\* @param {String} department

\* This manager's department

\*/

function Manager(first, last, id, department) {

Manager.baseConstructor.call(this, first, last, id);

this.department = department;

}

// subclass Employee

KevLinDev.extend(Manager, Employee);

/\*\*

\* Create a string representation of this object

\*

\* @return {String} A string representation of this object

\*/

Manager.prototype.toString = function() {

return Manager.superClass.toString.call(this) + ": " + this.department;