Dynarch JavaScript Toolkit

www.jstoolkit.com

Mihai Bazon

May 25, 2007

Preface

DynarchLIB is a JavaScript framework for building rich, desktop-like Web applications. It provides the following features:

- a suitable syntax for defining JavaScript objects
- a coherent event system
- various utilities to make your JavaScript life easier
- an easy way to communicate with a backend server via (XML)RPC requests
- a collection of easy to use widgets[1]

DynarchLIB is released under the terms of the GNU General Public License, version 2 (`www.gnu.org/licenses/gpl.html`). Commercial licensing is also available and we encourage you to support our efforts by purchasing one (`www.jstoolkit.com/licensing`).

Visit the project website at `www.jstoolkit.com` for news and the latest version.

[1]user interface controls such as: menu, dialog, toolbar, calendar, layout manager, etc.

Contents

Chapter 1

DynarchLIB overview

DynarchLIB extends the classic JavaScript objects with a few useful methods that are used all over the place. In this chapter we present the basic building blocks of DynarchLIB, which are necessary in order to create applications on top of it.

1.1 General purpose utilities

Appendix A

Advanced JavaScript primer

In order to use DynarchLIB (or any other AJAX framework) you need to program in JavaScript. It is not a complicated language, but it can be easily confusing for programmers coming from a different background—such as Java or PHP.

In this section we want to explain a few JavaScript techniques that are heavily used with our toolkit. We assume the reader has some basic knowledge about JavaScript. We won't try to describe all the tidbits of this language, but only a few that we consider to be of high importance for understanding the language and DynarchLIB.

If you are an expert JavaScript programmer, feel free to jump to section 1.1. Otherwise, please let me start by saying that learning JavaScript is a somewhat recursive process. You can't understand functions until you understand objects, and you can't understand objects until you understand functions. In essence, in JavaScript everything is an object. However, we could also say that everything is a function.

If this section seems to be a mess, I apologize and may I kindly ask you to read it again. Test the samples and experiment on them too.

A.1 Variable scope

In JavaScript, variables have function or global scope. When you declare a variable using the `var` keyword, it has function scope (or global if it is defined outside any function). When you don't specify the `var` keyword, or when variables are defined outside a function, they are global. In practice, I always use `var`. Here is a basic scope sample.

```
var a; // global variable, because it's not inside a function

function f() {
  alert(a); // accesses the global variable above
  var b = "This is B"; // local variable
  alert(b);
};

a = "This is A";

// Calling f() now will display "This is A",
// then "This is B".
f();

// This will display "undefined" because b is not available here.
alert(b);
```

We cannot access the variable *b* outside the f() function. It is created while the function is running, and effectively destroyed as soon as it finishes running.

A.1.1 Shadowing variables

When you use a variable, JS will always look it up in the most nested function block that contains it. Here is a somewhat trickier example:

```
var a = "This is global A";
function f() {
  alert(a);
  var a = "This is local A";
  alert(a);
}
f();
```

When we call f(), the JavaScript interpreter notices that we have a declaration for the variable *a* inside the function, so it will use that one instead of the global. Therefore the code above will never display "This is global A". The first `alert(a)` line will display "undefined"—because the variable wasn't initialized by that time, even though its stack space was reserved. The second `alert(a)` call will display "This is local A".

If, in any function, you declare a variable having the same name as a global variable, then you shadow the global and cannot access it by standard means. In the example above we could have used `window.a` to access the global[1].

[1] `window` is "the Global Object". All global variables are actually properties of this object. Try to avoid it when it's not absolutely necessary.

A.1.2 Function arguments

It might not be very obvious, but function arguments are implicit local variables. The following two functions are perfectly equivalent, but the first one is uselessly verbose:

```
function square(x) {
  var local_x = x;
  return local_x * local_x;
}

function square(x) {
  return x * x;
}
```

x is a local variable in both cases, behaving exactly the same as if it were declared with `var`. The notes about variable shadowing in the previous section apply to function arguments as well:

```
var a = "This is global A";
function f(a) {
  alert(a);
};
f("Hello World"); // displays "Hello World"
```

The function argument a shadows the global variable and makes it inaccessible to the function.

A.2 JavaScript object system

The JavaScript object system is quite simple: everything is an object. Including constants and including functions.

An object is an hash that maps property names to property values. Objects can contain methods—but they should be regarded as ordinary properties that just happen to be functions.

Here's the classical way to define and instantiate objects in JavaScript:

```
// declaring the object and its methods

function Computer(manufacturer, cpu) {
  this._manufacturer = manufacturer;
  this._cpu = cpu;
}

```

```
Computer.prototype.getManufacturer = function() {
  return this._manufacturer;
};

Computer.prototype.getCPU = function() {
  return this._cpu;
};

// instantiate some objects

var my_laptop = new Computer("IBM", "Intel Centrino Duo / 2GHz");
var my_old_laptop = new Computer("Dell", "Intel Centrino / 1.7GHz");

alert(my_laptop.getManufacturer()); // displays "IBM"
alert(my_old_laptop.getCPU()); // displays "Intel Centrino / 1.7GHz"
```

The constructor of this object is the function `Computer`. It receives two arguments that are saved as properties of the newly created object. In order to instantiate objects, we need to call the function using the "`new`" syntax. If we would call simply: `Computer("IBM", "Pentium")` then it would fail because `this` won't be defined[2].

As you can see, the constructor has a special property called "`prototype`". When we add methods and properties to it, they will be magically available in all objects of that type, even if they were instantiated *before* the new properties were added. Therefore, an object definition can be modified at any time. To exemplify, we can continue our example like this:

```
Computer.prototype.reboot = function(reason) {
  // reboot the computer here..
};

my_laptop.reboot();
```

A.2.1 Inheritance

JavaScript supports object inheritance in a very ~~rudimentary~~ simple and natural way. You just need to state that the prototype of a derived object initially contains the properties of the base object.

```
// declare that Server inherits Computer.
// note that we can do this even before the constructor is defined.
```

[2]Actually, `this` will refer to the global object, as we will see later in section A.3.1. However, it's best not to rely on this.

```
Server.prototype = new Computer;

// define the constructor of the derived object
function Server(manufacturer, cpu, ip) {
  // call the base class constructor.
  Computer.call(this, manufacturer, cpu);
  // any additional initialization here..
  this._ip = ip;
}

Server.prototype.getIP = function() {
  return this._ip;
};

var my_server = new Server("Apple", "G5", "192.168.1.1");

// note how my_server inherits these methods from the Computer object
alert(my_server.getManufacturer()); // displays "Apple"
alert(my_server.getCPU()); // displays "G5"

// and here's the new method that Server has and Computer doesn't
alert(my_server.getIP()); // displays "192.168.1.1"
```

A.2.2 Overriding methods

When a server is rebooted, we want our sysadmin to know that, so we write the reboot method this way:

```
Server.prototype.reboot = function(reason) {
  email_sysadmin("Reboot reason: " + reason);
  // call the base class' method
  Computer.prototype.reboot.call(this);
};

my_server.reboot("Water in the coprocessor");
```

Next, when the `reboot` method is called on a `Server` object, our sysadmin would get an email about it.

A.2.3 Run-time type information

We can use the instanceof operator in order to check what type is a certain object instance.

```
alert(my_laptop instanceof Computer); // displays true
alert(my_laptop instanceof Server); // displays false
alert(my_server instanceof Server); // displays true
alert(my_server instanceof Computer); // displays true
```

Quite clearly, my_server is an instance of both Computer and Server, while my_laptop is an instance of Computer but not of Server.

Implicitly, all objects inherit from a standard Object class, which is defined in JavaScript. Therefore, foo instanceof Object will always be true regardless of what object type “foo” is.

Good practice rules dictate that you should avoid calling instanceof unless you absolutely need it. Best rule is, you should never need it.

A.3 Objects and methods

As you could see above, an object can have methods. Methods are functions, properties of the object prototype (though as we will see later, an object can also have methods that are not part of its prototype).

Where JavaScript gets interesting is that any function can be called “against” any arbitrary object. Above we have this.getManufacturer(), which is OK because getManufacturer is a method defined in the object prototype. However, we can define a new, totally independent function, and by way of calling it, it can access that object using this, just like an object method. Example:

```
function getManufacturerAndCPU() {
  return this._manufacturer + ", " + this._cpu;
};

// the following will display:
// "IBM, Intel Centrino Duo / 2GHz"
alert(getManufacturerAndCPU.call(my_laptop));
```

In the case above, getManufacturerAndCPU is effectively defined as if it were an object method. It really doesn’t make any difference *until we actually call it.* So in short, in JavaScript the object that a function acts upon (that is, the value of “this”) is determined *at the function call time*, rather than at the definition time. This, more or less, happens in any OOP language, but the syntax of JavaScript makes it a lot more obvious than in other languages.

A.3.1 The Function object

So in order to call a function in the context of some object, we can use the function's call method. Sounds crazy, but yes—functions have methods, because functions are objects.

```
function whoIsThis() {
  alert(this);
}
```

That is a plain function which uses the this keyword. Note, however, that it's not linked to any object (it's not in a constructor's prototype as we did before). We can call it in a few ways:

```
// since "this" is not specified in any way, the JavaScript
// interpreter will automatically use the global object
whoIsThis(); // displays "[object Window]"

// here we explicitly pass a "null" value for "this"
whoIsThis.call(null); // displays "null"

// here we pass the string "Me" for "this"
whoIsThis.call("Me"); // displays "Me"

var a = "Foo";
// here we assign that function as a method of a String
// object. This syntax doesn't actually call the
// function--since the "()" are not present.
a.test = whoIsThis;
// now we can call it normally and it will be in the
// context of "a"
a.test(); // displays "Foo"

// this sample is the same as the one above, but uses a
// Number object instead.
var n = 10;
n.test = whoIsThis;
n.test(); // displays "10"
```

So a certain function can be called like f() – in which case it *shouldn't* access the `this` keyword[3], or like f.call(obj) – when `this` is explicitly assigned to *obj*, or like obj.f() – where `this` is implicitly assigned to *obj*.

Any additional arguments passed to `call` will be passed over to the function, so you can write:

```
function f(arg1, arg2) {
  alert(this + " " + arg1 + " " + arg2);
}
var obj = new String("I am");
f.call(obj, "a", "string");
```

There is also the `apply` variant, which is similar to `call` but the additional arguments are passed in an array:

```
var a = new Array("Brad", "Pitt");
f.apply(obj, a);
```

This is extremely useful for functions that receive a variable number of arguments.

A.4 Literal notation

JavaScript syntax provides some handy shortcuts to instantiate arrays or hashes (which are, well, objects). Literal arrays and hashes form the basis of JSON[4].

Literal arrays

Here are various ways to create an array:

```
var a = new Array();
a[0] = "Foo";
a[1] = "Bar";
a[2] = "Baz";

var b = new Array("Foo", "Bar", "Baz");

var c = [ "Foo", "Bar", "Baz" ];
```

They are perfectly equivalent. All variables are `Array` objects and contain the same data. However, the definition of *c* is the most elegant.

[3]It will be the global object, but good code should not rely on it
[4]JavaScript Object Notation — a highly compact and readable data interchange format which almost obsoletes XML

Literal objects (hashes)

In JavaScript you can instantiate objects that don't actually have a type using "the hash notation". They will be of the implicit type (`Object`). Again, the advantage is that the syntax is extremely clean and readable.

```
var a = new Object();
a.foo = "foo";
a.bar = "bar";

// the same, using the hash notation:
var b = { foo: "foo", bar: "bar" };
alert(b instanceof Object); // displays "true"
```

The definition of *b* is equivalent to the definition of *a*, but it's a lot more compact and readable. DynarchLIB uses hashes to pass arguments to most constructors and functions. Here's an example of how we create a dialog box:

```
var dlg = new DlDialog({
  title     : "Enter your name",
  resizable : false,
  modal     : true
});
```

A.5 Lexical closures

Closures are the most interesting feature of JavaScript. They are present in some other languages as well, but—curiously—not in the most popular ones. In JavaScript they play an essential role, therefore it's something you must understand in order to make anything useful with JS.

Closures are possible because JavaScript has two fundamental features:

- Functions are *first class*. This means that you can pass around a reference to a function the same way you pass an ordinary object or a constant.

  You can define a function inside another function and nest them indefinitely, depending on your needs.

- It has *lexical scope*. A function will have access to any variables defined outside it.

First, here is how to define an inner function.

```
function makeGreeter() {
  return function() {
    alert("Hello World");
  }
}

// this doesn't display anything
var greeter = makeGreeter();

// now we actually call the inner function,
// which displays "Hello World"
greeter();
```

This wasn't so useful in itself, but shows that, basically, we can have a `function` declaration anywhere an expression is expected:

```
function makeGreeter2() {
  var foo = function() {
    alert("Hello World");
  };
  return foo;
}
```

The above does the same as makeGreeter(). Furthermore, we can use a different syntax:

```
function makeGreeter3() {
  function foo() {
    alert("Hello World");
  };
  return foo;
}
```

Again, the meaning is exactly the same as before.

OK, what are closures? Combining inner functions with local variables, we get closures. Here's another greeter example:

```
function makeGreeter(greeting) {
  return function() {
    alert(greeting);
  };
}

var f1 = makeGreeter("Good afternoon");
var f2 = makeGreeter("Good evening");
var f3 = makeGreeter("And Good night");
```

```
// now if we call them...
f1(); // displays "Good afternoon"
f2(); // displays "Good evening"
f3(); // displays "And Good night"
```

So each time we call `makeGreeter(greeting)`, it returns a function that, when invoked, will display that greeting. What is interesting is that the function "remembers" the greeting, since when we call *f1*, *f2* and *f3* we don't need to pass any arguments.

Here's how to create a counter—a function that will return consecutive numbers each time it's called.

```
function makeCounter(start) {
  return function() {      // closure for "start"
    return start++;
  };
}

var counter = makeCounter(1);
alert(counter()); // displays "1"
alert(counter()); // displays "2"
alert(counter()); // displays "3"
```

`makeCounter()` creates and returns a function that, when subsequently called, will return and increment the value of a *local* variable (*start*).

```
var otherCounter = makeCounter(10);
alert(otherCounter()); // displays "10"
alert(otherCounter()); // displays "11"
alert(counter());     // displays "4"
alert(otherCounter()); // displays "12"
alert(counter());     // displays "5"
// etc.
```

So calling `makeCounter()` multiple times will instantiate separate counters, each of them having access to its own instance of the `start` variable. They are functions that encapsulate state—this is so useful that we could even create an object system with it, if JavaScript didn't have one. Fortunately, it does and it's pretty good. Here's how we can create the example above using the object system:

```
function Counter(start) {
  this.iterator = start;
};

Counter.prototype.getNext = function() {
  return this.iterator++;
};

var counter = new Counter(1);

alert(counter.getNext()); // displays "1"
alert(counter.getNext()); // displays "2"
```

As far as I am concerned, I prefer the variant using closures. It's a lot more simple and natural.

A.6 Real-world closure examples

I have many friends that, once here, will ask "OK, I finally got it. I understand what closures are all about. Now, why in the world would I want to use them?".

Here are some more useful samples—but note that they don't necessary teach you good style. They're just examples.

A.6.1 String buffer

Everyone knows by know that Internet Explorer is extremely slow when concatenating strings. The fastest idea on IE seems to be to create an Array and call join() at the end. The following code implements a string buffer "object" (yes I can call it so—it even has methods—but it's really a closure).

```
function makeStringBuffer() {
  var a = new Array(), i = 0, f = function(val) {
    a[i++] = val;
  };
  f.join = function(sep) {
    return a.join(sep || "");
  };
  return f;
};

var buffer = makeStringBuffer();
buffer("Hello");
buffer("World!");
alert(buffer.join(" ")); // displays "Hello World!"
```

A.6.2 Iterators

It's common to have to iterate arrays. You usually write a for loop that starts with zero and ends when the integer iterator has reached `array.length`. It's so common that it's almost boring to have to write that for over and over.

Here's an implementation of an iterator, not necessarily better than the classical for variant, but interesting.

```
function makeIterator(array) {
  var i = 0;
  var f = function() {
    return array[i];
  };
  f.done = function() {
    return i < 0 || i >= array.length;
  };
  f.next = function() {
    i++;
  };
  return f;
};
var i = makeIterator([ "Check", "this", "out" ]);
while (!i.done()) {
  alert(i());
  i.next();
}
```

Combining the above with the facility to extend JS objects, we can also write:

```
Array.prototype.makeIterator = function() {
  var i = 0, self = this;
  var f = function() {
    return self[i];
  };
  f.done = function() {
    return i < 0 || i >= self.length;
  };
  f.next = function() {
    i++;
  };
  return f;
};
var i = [ "Check", "this", "out" ].makeIterator();
while (!i.done()) {
  alert(i());
  i.next();
}
```

A.6.3 The `foreach` function

Going further with iterating arrays, here is the `foreach` function. Very simple and very useful.

```
Array.prototype.foreach = function(f) {
  for (var i = 0; i < this.length; i++) {
    f(this[i]);
  }
};
```

It receives a function and it applies that function for each element of the array. Now let's say we want to quickly sum the elements of an array. We define this helper function:

```
function makeAccumulator() {
  var sum = 0;
  var f = function(val) {
    sum += val;
  };
  f.total = function() {
    return sum;
  };
  f.reset = function() {
    sum = 0;
  };
  return f;
};
```

And now we can easily apply it to any array:

```
var accumulator = makeAccumulator();

[ 1, 2, 3, 4 ].foreach(accumulator);
alert(accumulator.total()); // displays "10"

[ 10, 20, 30, 40 ].foreach(accumulator);
alert(accumulator.total()); // displays "110"

// reset to zero
accumulator.reset();
```

```

[ 2, 4, -1, -5 ].foreach(accumulator);
alert(accumulator.total()); // displays "0"
```

A.6.4 Private property

In languages such as C++ or Java, it's very easy to define private object methods or variables by just specifying the "private" access type. In JavaScript we cannot do this. But we have closures. This is the first example where we can't get away without them.

```
function Person(birth_date) {
  var birth_year = birth_date.getFullYear();
  this.getAge = function() {
    return (new Date()).getFullYear() - birth_year;
  };
};

var me = new Person(new Date(1979, 3, 8));
alert(me.getAge()); // displays "28" at the time of writing
```

So in this example, `getAge` is a closure that can access the `birth_year` inner variable. We can display a `Person`'s age by calling this method, but it's absolutely impossible to modify ones age from the outside code, because it's not even a property of the object. It's just a variable that gets created along with a new object—but a reference to it is not stored anywhere.

`birth_year` is therefore a fully private variable.

A.6.5 File local variables

JavaScript does not have the notion of "packages". That is, all ".js" files share the same memory; a global variable defined in one file can be accessed in another. This can many times create problems—global variables should be avoided in most cases.

Suppose you want to define a few functions in a ".js" file and all of them operate on some global variables. You only want to export the functions, not the variables.

Here is how it can be done.

Listing A.1: File local variables example

```
// we will create a Person object and we declare it here so
// we export it to other scripts. This variable is global.
var Person;

```

```
// using this syntax, we create and call an anonymous
// function. Its only purpose is to hide a few variables
// from outside code.
(function(){

  // These are some file local variables. Note that all of the
  // Person object's methods can access them. In a way, they are
  // global--but only for this script.
  var population = new Array();
  var males = 0;
  var females = 0;
  var totalAge = 0;

  // not using "var" here since we actually want to export this
  // variable--it is a global.
  Person = function(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
    population.push(this);
    totalAge += age;
    if (sex == "M")
      males++;
    else
      females++;
  };

  Person.getPopulationSize = function() {
    return population.length;
  };

  Person.getManCount = function() {
    return males;
  };

  Person.getWomenCount = function() {
    return females;
  };

  Person.getAverageAge = function() {
    return totalAge / Person.getPopulationSize();
  };

})();
```

After loading this script, we have access to the Person object and its "static" methods. These methods can access the variables *population*, *males* and *females*—but we can't access them directly through other means. In a way, we can say that we defined *private static members* of the Person object.

```
new Person("John", 10, "M");
new Person("Diane", 22, "F");
new Person("David", 25, "M");
new Person("Bobby", 28, "M");

alert(Person.getPopulationSize()); // displays "4"
alert(Person.getManCount()); // displays "3"
alert(Person.getWomenCount()); // displays "1"
alert(Person.getAverageAge()); // displays "21.25"
```