Dojo concepts for Java developers

Declaring classes and setting context

Dave Draper
WebSphere Application Server Administrative Console Developer
IBM

Skill Level: Intermediate

Date: 14 Oct 2008

Dojo is being used more and more in Web-based applications. Many developers have strong skills in Java™ programming, but only limited experience in JavaScript. They can struggle with the conceptual leap from a strongly typed, object-oriented compilation language to a dynamic, weakly typed scripting language. This confusion can make it difficult for developers to correctly declare Dojo classes. This article helps clear up this confusion, shows why it may be necessary to set context, and describes how to go about it.

Introduction

Get started with Dojo development http://www.ibm.com/developerworks/training/kp/wa-kp-dojo/index.html
If you're a Java programmer coming to Dojo with little or no experience of JavaScript, chances are you're going to struggle with some of the concepts that enable it to work. The main concerns with Dojo are that — at the time of writing — it is still in its infancy (version 1.0 was only released in February 2008) and the documentation available is still somewhat limited. This article helps you bridge the gap from Java code to Dojo so that you can get up to speed quickly and use the toolkit when developing your applications.

This article does not describe how to obtain the Dojo toolkit or the necessary statements needed to use it, because there are many other resources available that provide that information. The article is written for Web developers who are coming to Dojo from a servlet development background.

The JavaScript hash

One of the first challenges is to understand the syntax that is used when invoking Dojo functions, in particular the use of the "hash" or JavaScript object. A hash is

expressed as a set of attributes delimited by commas between curly braces. A simple example is shown in Listing 1, declaring a hash consisting of 6 attributes: a string, an integer, a boolean, an undefined attribute, another hash, and a function.

Listing 1. Example JavaScript hash

```
var myHash = {
    str_attr : "foo",
    int_attr : 7,
    bool_attr : true,
    undefined_attr : null,
    hash_attr : {},
    func_attr : function() {}
};
```

It is important to remember that JavaScript is weakly typed, so although each attribute has been initialized to a value linked to its name, there is no reason why the initial `str_attr` attribute cannot be subsequently set to an integer or boolean (or any other type for that matter). Each attribute in the hash can be accessed or set using the dot operator (as shown in Listing 2).

Listing 2. Accessing and setting hash attributes

```
// Accessing a hash attribute...
console.log(myHash.str_attr);

// Setting a hash attribute...
myHash.str_attr = "bar";
```

The first four attributes of `myHash` should be self-explanatory. The fact that a hash can have attributes that are also hashes should be no surprise. (This can be thought of as analogous to Java classes referencing both primitives and objects.) It is the final attribute that is most important to understand.

Functions are objects

Although in Java code there is a `java.reflection.Method` class, it essentially acts as wrapper to a method. In JavaScript the function is an object like any other that can be set, referenced, and passed as an argument to other functions. Often it is necessary to declare new functions in function calls in much the same way as an anonymous inner class can be declared in a Java method call.

Another important difference between Java methods and JavaScript functions is that JavaScript functions can be run in different contexts. In Java programming the use of the keyword `this` refers to the current instance of the class where it is used. When used in a JavaScript function, `this` refers to the context in which that function is running. A function will run in the closure that defines it unless otherwise specified.

In the simplest terms, a closure can be considered to be any JavaScript code contained within curly braces (`{}`). A function declared inside a JavaScript file can use `this` to access any variable declared in the main body of that file, but a function

declared inside a hash can only use `this` to reference variables declared inside that hash, unless it is provided with an alternative context to work in.

Because enclosed functions are often required as arguments to Dojo functions, understanding how to set their context will save a lot of unnecessary debugging.

The main Dojo function that is used to assign context is `dojo.hitch`. You may never use `dojo.hitch`, but it is important to understand that it is one of the cornerstones of Dojo, and many other functions are invoking it under the covers.

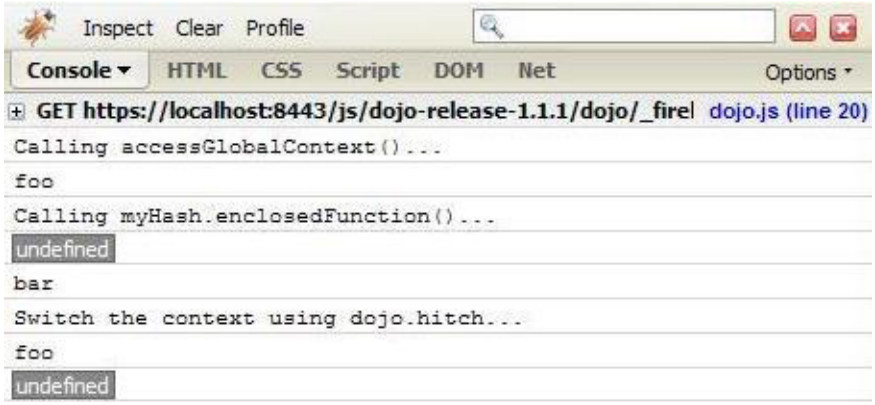
Listing 3 shows how context hitching works (its output is shown in Figure 1):

- A variable is defined at the global context (`globalContextVariable`) and another variable is declared in the context of a hash (`enclosedVariable`).
- The function `accessGlobalContext()` can successfully access `globalContextVariable` and display its value.
- But, `enclosedFunction()` can only access its local variable `enclosedVariable` (note that the value of `globalContextVariable` is displayed as "undefined").
- Using `dojo.hitch` to "hitch" `enclosedFunction()` to the global context allows `globalContextVariable` to be displayed (note however that `enclosedVariable` is now undefined because it is not declared in the context that `enclosedFunction()` is running.

Listing 3. Closures and context

```
var globalContextVariable = "foo";

function accessGlobalContext() {
    // This will successfully output "foo"...
    console.log(this.globalContextVariable);
};

var myHash = {
    enclosedVariable : "bar",
    enclosedFunction : function() {
                           // Display global context variable...
                           console.log(this.globalContextVariable);

                           // Display enclosed context variable...
                           console.log(this.enclosedVariable);
                       }
};

console.log("Calling accessGlobalContext()...");
accessGlobalContext();

console.log("Calling myHash.enclosedFunction()...");
myHash.enclosedFunction();

console.log("Switch the context using dojo.hitch...");
var switchContext = dojo.hitch(this, myHash.enclosedFunction);
switchContext();
```

Figure 1. How contect hitching works



Declaring classes

Class declaration tips

- While `myClass` is a perfectly valid name, it is a better practice to declare names using a fully qualified class name style, for example, `com.ibm.dojo.myClass`. This does not mean that the class should be deployed to the file system under the relative path of "./com/ibm/dojo/"; it simply reduces the chances of there being naming collisions with any other classes.
- There must never be a `,` (comma) after the last attribute because some browsers will ignore it (FireFox), but others will blow-up (Internet Explorer). This rule also applies to declarations of hash objects anywhere.

The reason why this hitching is so important will become apparent as soon as you start to declare Dojo classes or to create your own widgets. One of the greatest powers of Dojo is the ability to "wire" objects together through the use of the `dojo.connect` function and the built-in pub/sub model.

Declaring a class requires three objects:

1. A unique name for the class
2. A parent class to extend function from (plus any "mix-in" classes to simulate multiple inheritance)
3. A hash defining all the attributes and functions.

The simplest possible class that can be declared in shown in Listing 4, and its instantiation in Listing 5.

Listing 4. Basic class declaration

```
dojo.declare(
   "myClass",
   null,
   {}
);
```

Listing 5. Basic class instantiation

```
var myClassInstance = new myClass();
```

If you want to declare a "real" (that is, useful) Dojo class, it's important to understand constructors. In Java code you can declare multiple overloaded constructors to enable the instantiation of the class by a variety of different signatures. In a Dojo class you can declare a `preamble`, a `constructor`, and a `postscript`, but in the majority of cases you will only need to declare a constructor.

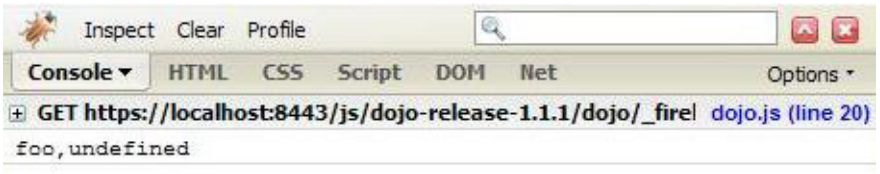
- Unless you're mixing-in other classes to simulate multiple inheritances, you are unlikely to require `preamble`, as it allows you to manipulate your `constructor` arguments before they are actually passed to the extended and mixed-in classes.
- `postscript` drives the Dojo widget life cycle methods, but provides no benefit to a standard Dojo class.

It is not essential to declare any of them, but to pass any values into an instance of the class the `constructor` function must be declared as a minimum. If the `constructor` arguments are to be accessed by any other method of the class, they must be assigned to declared attributes. Listing 6 shows a class that assigns only one of its `constructor` arguments to a class attribute and then attempts to reference both in another method.

Listing 6. Assigning constructor arguments

```
dojo.declare(
    "myClass",
    null,
    {
        arg1 : "",
        constructor : function(arg1, arg2) {
                          this.arg1 = arg1;
                      },
        myMethod : function() {
                       console.log(this.arg1 + "," + this.arg2);
                   }
    }
);

var myClassInstance = new myClass("foo", "bar");
myClassInstance.myMethod();
```

Figure 2. Output from assigning constructor arguments



Complex attribute rules

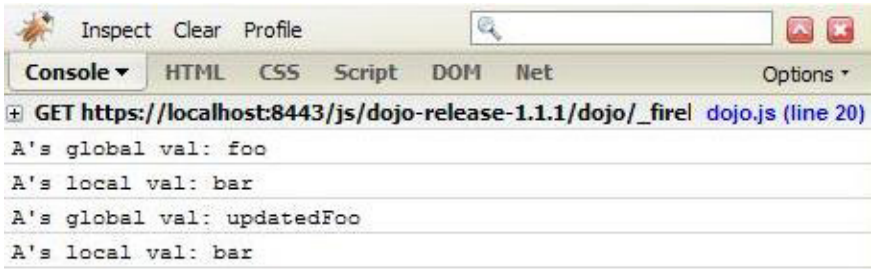
Class attributes can be initialized when declared, but if the attribute is initialized with a complex object type (such as a hash or an array) that attribute becomes analogous to a public static variable in a Java class. This means that whenever any

instance updates it, the change will be reflected in all other instances. To avoid this problem, complex attributes should be initialized in the constructor; however, this is not necessary for simple attributes such as strings, booleans, and so on.

Listing 7. Global class attributes

```
dojo.declare(
    "myClass",
    null,
    {
        globalComplexArg : { val : "foo" },
        localComplexArg : null,
        constructor : function() {
                          this.localComplexArg = { val:"bar" };
                      }
    }
);

// Create instances of myClass A and B...
var A = new myClass();
var B = new myClass();

// Output A's attributes...
console.log("A's global val: " + A.globalComplexArg.val);
console.log("A's local val: " + A.localComplexArg.val);

// Update both of A's attributes...
A.globalComplexArg.val = "updatedFoo";
A.localComplexArg.val = "updatedBar";

// Update B's attributes...
console.log("A's global val: " + B.globalComplexArg.val);
console.log("A's local val: " + B.localComplexArg.val);
```

Figure 3. Class attributes



Overriding methods

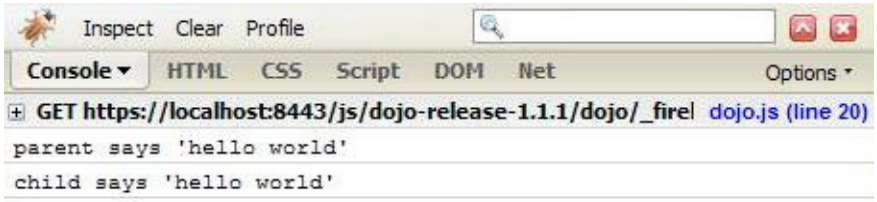
A method of a superclass can be extended by declaring an attribute with the same name. There is no concept of overloading, as JavaScript ignores any unexpected arguments and substitutes null for any that are missing. In Java code, to invoke the overridden method you call the method on super (that is, `super().methodName(arg1, arg1);`), but in Dojo you use the `inherited` method (`this.inherited(arguments);`). Listing 8 shows two classes declared, where `child` extends `parent`, overriding its `helloWorld` method, but calls `inherited` to access the function of `parent`.

Listing 8. Invoking superclass method in Dojo

```
dojo.declare(
```

```
    "parent",
    null,
    {
        helloWorld : function() {
                        console.log("parent says 'hello world'");
                    }
    }
);

dojo.declare(
    "child",
    parent,
    {
        helloWorld : function() {
                        this.inherited(arguments); // Call superclass method...
                        console.log("child says 'hello world'");
                    }
    }
);

var child = new child();
child.helloWorld();
```

Figure 4. Output from invoking superclass method in Dojo



Setting method context

Listing 9 shows a Java class that, upon instantiation, copies the elements from the supplied string array to an ArrayList of strings. It would not be unreasonable to assume that the same functionality can be provided in Dojo with the code in Listing 10. (Note the instantiation of `targetArray` in the constructor function to prevent it being global.) Unfortunately, it will result in the error message shown in Figure 5, because the function declared in the `dojo.forEach` method call creates a closure that defines `this` as referring to its own body.

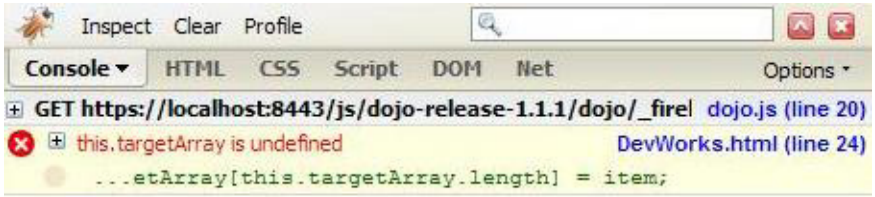
Listing 9. Accessing class scoped variable in Java code

```
import java.util.ArrayList;

public class MyClass
{
    // Declare an ArrayList of Strings...
    private ArrayList<String> targetArray = new ArrayList<String>();

    public MyClass(String[] sourceArray)
    {
        // Copy each element of a String[] into the ArrayList...
        for (String val: sourceArray)
        {
            this.targetArray.add(val);
        }
    }
}
```

Listing 10. Missing context in Dojo

```
dojo.declare(
    "myClass",
    null,
    {
        targetArray: null,
        constructor: function(source) {
                        // Initialise in constructor to avoid making global
                        this.targetArray = [];

                        // Copy each element from source into target...
                        dojo.forEach(source,
                                    function(item) {
                                        this.targetArray[this.targetArray.length] = item;
                                    });
                    },
    }
);

// This will cause an error!
var myClass = new myClass(["item1","item2"]);
```

Figure 5. Output from missing context in Dojo



Although `targetArray` is not defined at the context enclosed by the function, it is possible to pass the context where it is defined as an argument to the Dojo function. This means the `this` keyword can then access any objects (including functions) that have been declared at that context. The correct implementation is shown in Listing 11 (note the additional code in bold).

Listing 11. Setting correct context in Dojo

```
dojo.declare(
    "myClass",
    null,
    {
        targetArray: null,
        constructor: function(source) {
                        // Initialise in constructor to avoid making global
                        this.targetArray = [];

                        // Copy each element from source into target...
                        dojo.forEach(source,
                                    function(item) {
                                        this.targetArray[this.targetArray.length] = item;
                                    }, this);
                    },
    }
);
```

Context is not always passed as the same argument in a Dojo functions signature:

- In `dojo.subscribe` the context is passed **before** the function declaration (Listing 12).
- In `dojo.connect` both the context where the trigger method is defined and the context where the target method is defined should be supplied. Listing 13 shows an example where `obj1` is the context where `methodA` is defined and `obj2` is the context where `methodB` is defined. Calling `methodA` on `obj1` will result in `methodB` being invoked on `obj2`.

Listing 12. Setting context in dojo.subscribe

```
dojo.declare(
    "myClass",
    null,
    {
        subscribe : function() {
                        dojo.subscribe("publication",
                                       this,
                                       function(pub) {
                                           this.handlePublication(pub);
                                       });
                    },

        handlePublication : function(pub) {
                                console.log("Received: " + pub);
                            }
    }
);
```

Listing 13. Setting context in dojo.connect

```
dojo.connect(obj1, "methodA", obj2, "methodB");
```

Conclusion

JavaScript will never come naturally to developers who have become used to the more structured environment of Java code. Yet the implementation of Dojo, with its class declaration capabilities, does make the leap to client-side development considerably easier. A good understanding of context, and when and how to set it, will save a lot of pain for the Java developer and help them confidently add JavaScript to their toolbox.

Resources

- All the information and resources for getting started can be found at DojoToolkit.org.
- Find out more about Eclipse-based tooling to help write JavaScript in the developerWorks article Meet the JavaScript Development Toolkit (developerWorks, May 2008).
- Browse the technology bookstore for books on these and other technical topics.
- More information on other Ajax technologies (including Dojo) can be found in the developerWorks Ajax resource center.
- You can also get a complete reference of the Dojo API.
- Some excellent examples of Dojo coding are available at the Dojo campus.

About the author

Dave Draper

Dave Draper has been a developer of the WebSphere Application Server Administrative Console for the past 6 years. He is a Sun Certified Web Component Developer and has extensive experience developing Web-based tooling.