



You are here: [Home](#) > [Developers](#) > [View Source Articles](#) > Server-Side JavaScript View Source Article

Developer Program
[Membership](#)
[One-to-One Support](#)
[Newsgroups](#)

Developer Publications
[View Source](#)
[Developer News](#)

Documentation
[Technical Manuals](#)
[White Papers](#)
[TechNotes](#)
[Sample Code](#)
[FAQs](#)
[Books](#)

Technologies
[CORBA](#)
[Directory & LDAP](#)
[Dynamic HTML](#)
[Java](#)
[JavaScript](#)
[Linux](#)
[Security](#)
[SSJS](#)
[XML](#)

Developer Downloads
[Tools & SDKs](#)
[Patches](#)

iPlanet Products
[Technical Resources](#)

Extending the Capabilities of the Server-Side JavaScript File Object

By [Duane K. Fields](#)

[Send comments and questions](#) about this article to [View Source](#).

Many Web developers are familiar with server-side JavaScript's built-in `File` object, which enables reading and writing to files on the Web server. Java and Perl aficionados may feel that the `File` object lacks some of the power and elegance of their file access implementations. In this article I'll show you how to use SSJS's object inheritance model to extend the functionality of the `File` object in a powerful and elegant way.

Throughout this article I'll be working with the [SuperFile](#) module. You can compile this module straight into your applications and take advantage of the new capabilities the `SuperFile` object offers, or extend it further to meet your specific needs. The approach I use here--building on the functionality of an existing object--can be applied to any of JavaScript's built-in objects, or even objects that you've created yourself.

If you've never used the `File` object, the excellent documentation in Chapter 7 of Netscape's [Writing Server-Side JavaScript Applications](#) will get you up and running quickly. You should be familiar with basic file operations before reading this article, as I'll be showing you how to add to (and in some cases enhance) existing functionality.

JAVASCRIPT'S OBJECT MODEL

While Java uses a class-based system for creating objects and their relationships, JavaScript employs a prototype approach. In Java, you define classes that describe objects, then create instances of each class to bring the objects themselves into being. Object classes can inherit functionality from each other through subclassing--defining a class as being built upon an existing class. Under JavaScript's prototype approach, by contrast, there are no classes, just objects. But objects can inherit the properties of their "prototype object" in a way analagous to the class-subclass relationship in Java. So the object inheritance relationship in Java and JavaScript is architecturally the same, if not syntactically so.

I'll be using a Java-style object-oriented programming (OOP) vocabulary throughout this article; when I refer to classes and subclasses in JavaScript, I mean the words in the OOP sense, not literally. Of course, OOP is a lengthy topic. For a detailed discussion of OOP in JavaScript, you may want to read [Object Hierarchy and Inheritance in JavaScript](#).

THE FILE OBJECT

SSJS's file system services are provided through the `File` object. To work with a file, you must first create a `File` object (an "instance," if you will) by invoking the `File()` function and passing in the file name:

```
myFile = new File("/tmp/data.txt");
```

The `File()` function is a constructor—a special type of function that returns an object with properties and methods (functions that operate on objects) defined inside the constructor. So the `File()` constructor returns an instance of the `File` object, which we call `myFile`. As an instance of the `File` object, `myFile` will have properties and methods associated with it.

For example, the following statement uses the object's `getLength()` method to print the length of the file represented by the `myFile` instance of the `File` object:

```
write ("Length is " + myFile.getLength() + " bytes.");
```

The `getLength()` method is defined as part of the `File` object and has no meaning outside of it. A full complement of methods and properties associated with the `File` object enable you to open the file, read data from it, and so on. What we will explore in the rest of this article is adding file-related functionality to our programs by building on the existing `File` object.

CREATING A NEW SUPERFILE CLASS

When defining a new class, we don't have to start from scratch each time—we can build on an existing class through inheritance. Inheritance, simply stated, means that a subclass automatically has all the same properties and methods as its parent. Thus, an instance of the subclass has a ready-made set of data and operations. We can add entirely new methods to this object as well, or alter existing methods to specialize, refine, or customize their behavior.

SSJS already has the basic functionality we're looking for—the ability to read and write to files on the server. Suppose we want to add some new functionality and create customized operations. We'll create a new `SuperFile` class based on the existing `File` object to avoid having to rewrite all of the existing, and perfectly good, `File` object methods. We'll define our constructor to create a `File` object and pass it back directly.

```
function SuperFile(filename)
{
    var f = new File(filename);
    return f;
}
```

Now that we've defined our `SuperFile` object as a subclass of JavaScript's built-in `File` object, we can create a new `SuperFile` instance just like we would create a regular `File` object:

```
myFile = new SuperFile("/tmp/data.txt");
```

The `myFile` object can now be used just as if it were a `File` object—as a subclass of `File`, `SuperFile` has inherited all the properties and methods of `File`. Of course, this only becomes useful when we begin extending the capabilities of `SuperFile`.

ADDING NEW METHODS TO SUPERFILE

Now we want to extend the functionality of our `SuperFile` object by adding new methods. We'll create a `getContents()` method that will allow us to read the contents of the entire file in one step. We'll make use of existing methods like `readln()` to get the job done.

The first step in adding a method to our `SuperFile` class is to create a function that's designed to operate on a referenced object rather than to

receive an explicit object through its arguments. To refer to the object we're working with inside the function, we use the `this` keyword. Once the function is assigned to a particular object (like our `SuperFile` object), the `this` keyword can be used to act on the object's properties. For example, `this.x` would refer to property `x` of whatever object the function was assigned to.

So we create our `getContents()` function -- referring to the `SuperFile` object's `readln()` and `eof()` methods (inherited from the `File` object) through the `this` keyword--as follows:

```
function SuperFile_getContents()
{
    var contents = "";
    var line = "";
    while (! this.eof())
    {
        if ((line = this.readln()) &&
!this.eof())
            contents += line;
    }
    return contents;
}
```

It's common practice to name functions that will be used as methods of objects in the form `Classname_function`. This lets anyone working with the code know that this is a method to be used in association with an object and not to be called directly.

Once the function has been defined, we tie it to our `SuperFile` object by mapping the function name to an object property in the constructor:

```
function SuperFile(filename)
{
    var f = new File(filename);
    this.getContents = SuperFile_getContents;
    return f;
}
```

Notice that we didn't say `SuperFile_getContents()` or `this.getContents()`, because in this case (with the parentheses added) JavaScript would try to evaluate the function immediately. Instead, we're simply mapping the function's *name* to the object.

Once we've set up this mapping, we can call the method on the object just like we would call native methods. For example, with our `myFile` object, we could say

```
=>contents_of_file = myFile.getContents();
```

This is a powerful technique indeed. We can create as many methods as we need, encapsulating all of the added functionality directly into the object. By keeping our routines generic, we open the door to code reuse. Anytime we need to perform file operations in the future, for example, we can just throw our `SuperFile` object into the mix.

USING LIVECONNECT TO CALL JAVA METHODS

The Java programming language's `File` class has a number of very useful methods. For example, the class contains the `canRead()` and `canWrite()` methods, which test a file to determine if the system can access it. But because these are methods of a Java (not a JavaScript) `File` object, we can't access them directly in JavaScript.

What we need, then, is a way to somehow associate the complementary Java-accessible version of the `File` object with our JavaScript one. LiveConnect can provide us access to Java objects (as described in the

TechNote [Java to JavaScript Communication Using Server-Side LiveConnect](#), so this is the likely solution. All we have to do is provide instances of our `SuperFile` class with access to the Java `File` object corresponding to the file we're working with. This will open up most of the Java `File` object's methods to us.

Let's modify the `SuperFile()` constructor to include a new property called `javaFile` that gives ready access to the Java version of our `File` object:

```
function SuperFile(filename)
{
    var f = new File(filename);
    this.getContents = SuperFile_getContents;
    f.javaFile = new java.io.File(filename);
    return f;
}
```

You may be tempted to do something like this in your program now:

```
if (myFile.javaFile.canRead())
    write ("File is Readable!");
```

This will work, but it forces you to remember the details of the Java method and doesn't allow you to add any functionality (such as trying alternate paths or printing debugging information) to the `canRead()` operation. A better way to approach this problem is to add a method to your `SuperFile` class to handle the `canRead()` test. By encapsulating the Java calls inside your own method, you're defining an *interface*, or set of operations, that won't have to change if you wish to add functionality later to the `canRead()` test itself.

```
function SuperFile(filename)
{
    var f = new File(filename);
    this.getContents = SuperFile_getContents;
    f.javaFile = new java.io.File(filename);
    f.canRead = SuperFile_canRead;
    return f;
}
function SuperFile_canRead()
{
    return this.javaFile.canRead();
}
```

One thing to keep in mind when working with Java calls through LiveConnect is that any arrays or strings returned from Java calls will be Java (not JavaScript) objects, and as such will have to be converted. This can be accomplished with simple utility functions such as the following:

```
function javaArrayTojsArray(javaArray)
{
    var jsArray = new Array();
    for (var i=0; i < javaArray.length; i++)
        jsArray[i]=javaArray[i];
    return jsArray;
}
function javaStringTojsString(javaString)
{
    var jsString = javaString + "";
    return jsString;
}
```

Booleans, dates, and integers will automatically be converted into the appropriate JavaScript object and thus don't require special handling.

OVERLOADING PARENT CLASS METHODS

OVERLOADING PARENT CLASS METHODS

Many of the operations we'd like to perform on files are already defined by the `File` object. However, we might wish to alter an operation's behavior or add functionality to it. For example, we might want the operation to record debug information, perform additional validity checks, or modify data before or afterward.

Say we wanted to modify the `File` object's `open()` method to verify first that the file is indeed readable and to log an error message otherwise. We first need to modify the constructor to assign the built-in `File` object's `open()` method to `SuperFile` objects, since we're about to create our own `open()` method. If we fail to do this, we won't be able to make calls back to the "old" `open()` method since we're essentially redefining it.

```
function SuperFile(filename)
{
  var f = new File(filename);
  this.getContents = SuperFile_getContents;
  f.javaFile = new java.io.File(filename);
  f.canRead = SuperFile_canRead;
  f.parent_open = f.open;
  f.open = SuperFile_open;
  return f;
}
```

We can then define our own `open()` method, with whatever extras we desire:

```
function SuperFile_open(args)
{
  if (this.canRead())
    return this.parent_open(args)
  else
    java.lang.System.out.println("Cannot
read"+this+", read access not allowed");
}
```

GOING FURTHER

While the `File` object isn't available in client-side JavaScript, the techniques introduced in this article can generally be applied to any objects in order to extend their functionality. You can extend the `Array` object, for example--adding your own methods to replace any functionality you feel is missing or confusing. (To find out about client-side JavaScript arrays, see Danny Goodman's [Primer on JavaScript Arrays](#) in his *View Source* JavaScript Apostle column.) Or you can add new features to the `String` object.

By packaging functionality into reusable objects, you can develop robust code modules once, then reuse them from project to project. This style of programming not only gets you up and running quickly but also encourages elegant, clear, robust program design. The included `SuperFile` object is a good start. Simply compile the [SuperFile.js](#) module into your own projects and invoke it to instantiate your own `SuperFile` objects.

FURTHER RESOURCES

- [Java to JavaScript Communication Using Server-Side LiveConnect](#)
- [Writing Server-Side JavaScript Applications](#), Chapter 7

[write to us](#) and let us know
what you think of this article.

[Duane K. Fields](#) is an Internet systems engineer with Tivoli Systems, an IBM Company, in Austin, Texas. Duane has been developing network applications in one form or another since the early days of the Internet.

Related Readings:

- [Frequently Asked Questions](#): Find the answers to your questions on Internet technologies.
- [Documentation](#): A library of technical information.
- [Developer Central](#): Collections of developer resources organized by Internet technology areas.

Any sample code included above is provided for your use on an "AS IS" basis, under the [Netscape License Agreement - Terms of Use](#)
