

H622

Advanced JavaScript Using Acrobat 5.0

**Kas Thomas
Documentation Manager
eBusiness Integration Products Division
SilverStream Software, Inc.
<kthomas@silverstream.com>**

What This Course is NOT About

- HTML or browser JavaScript
- Beginner-level JavaScript
- Prepress aspects of PDF
- PDF workflows
- Security, encryption, digital signatures
- Acrobat plug-in development with C/C++
- Interapplication communication (VBScript, AppleScript)

What This Course IS About

- JavaScript Core-Language Essentials
- Advanced Acrobat JavaScript Techniques
- New-for-5.0 API Methods
- Boosting Productivity via Bug Avoidance
- Undocumented API Features
- Little-Known Productivity Tips
- Things That Will Take Your JavaScript Skills to the Next Level

Where can I use JavaScript in PDF?

- External **.js** files (must be in the \Acrobat\Javascrpts path)
 - Load at application launch
 - Application-level scope/visibility
- **PDF files (document scripts)**
 - **Document-level** scripts: "top level" scripts
 - Load at doc-open time
 - Scoped to the document
- In Acrobat 5.x, scripts can be assigned to specific event handlers (Doc Will Close, Doc Will Save, Doc Did Save, Doc Will Print, Doc Did Print)
- **Bookmarks** (attach a JS Action in Properties dialog)
- **Links** (attach a JS Action via Links dialog)
- **Form Fields** (event handlers: mouseUp, mouseDown, mouseEnter, mouseExit, onFocus*, onBlur*)
*starting with v4.05
- **FDF & XFDF files** (scripts will be imported into target file)
 - **Note:** This requires hand-editing of FDF/XFDF. There is no UI for this functionality in Acrobat.

What's the Fast Way to Test/Debug Scripts?

ANSWER: Use the console!

- Rapid feedback. Execute code with ENTER key. See immediately whether your code does what you expect.
- Some API functions, such as `app.addItem()`, can't easily be tested any other way.
- 'Esc' will break an infinite loop.
- Lets you modify onscreen items interactively.

Rule of thumb: Make sure it works in the console before attaching code to a form field, bookmark, doc-open action, etc.

AGAIN: USE THE CONSOLE!

- If you aren't sure about the syntax of an API method:
execute: `theMethod("?")` in the console to get syntax help (not helpful for core-JS methods)
- Test whether a method exists in a given object instance. E.g.:
`"openDoc" in app; // returns true`

Data Types in JavaScript

Strings, numbers, and booleans are JavaScript's **atomic** data types. (The periodic table has 3 elements!)

All three are interconvertible at runtime.

```
var n = Math.PI;    // 3.141592653589793
var str = String( n );    // "3.141592653589793"
str = str.substring(0,4);    // "3.14"
--str;    // 2.14
str += true;    // 3.14
```

- You can typecast with `String()` and `Number()` wrapping: `String(3.14)` is a string. `Number("3.14")` is a number.

RUNTIME TYPECASTING

- The rule is:

When numbers and strings occur in the same expression, the interpreter will treat '+' as a *string-concatenation operator*. It will treat '*', '/', and '-' as arithmetic operators. Operands will be typecast accordingly.

```
a = 2; b = 3; c = "4";
a + c;    // "24"
a * c;    // 8
a * b + c;    // "64"
```

The typeof Operator

- Use `typeof` (with or without parentheses) to learn whether a variable is of type 'string', 'number', 'object', 'boolean', 'function', or 'undefined'.

```
typeof Math.PI;    // 'number'  
typeof "abc";     // 'string'  
typeof new String("abc"); // object  
typeof true;      // 'boolean'  
typeof /\w+//;    // 'function'  
typeof undefined; // 'undefined'
```

```
typeof NaN; // Can you guess?  
typeof typeof undefined; // Can you guess?
```

Special Value: `undefined`

- `undefined` (without quotes) is a special value. You can assign this value to a variable to destroy it.

```
var x;    // x is undefined
x = 9;    // x is defined
x = undefined; // x is undefined
```

- The `typeof` operator checks to see if a variable has the special value `undefined`.
- If a variable is undefined, it cannot be used in an expression without causing an exception.

Lesson:

Declaring a variable is not the same as defining it!

Special Value: **null**

- **null** is a special placeholder value that, while not directly usable for anything, can be assigned to a variable without undefining it.
- Many core-language methods return **null** when there is nothing else to return.
- **null** has a type: 'object'.
- **undefined** has no canonical type.

```
typeof null;           // 'object'  
typeof undefined      // 'undefined'  
  
var a = 3.14;          // a is a 'number'  
a = null;              // a is an 'object'  
a = undefined;        // a is typeless
```

- It's interesting to note:

```
null == undefined;    // true  
null === undefined;   // false
```

- Which can only mean that in the first case, the interpreter casts `null` to *undefined* when an untyped comparison is done.
- When the strict equality operator (`===`) is used, the comparands must be equal in value *and* of the same type.

How to Discover Object Properties

- Discover properties of objects by using the core-language **for/in** syntax. For example: To discover all properties in App object, you can do:

```
for (k in app )
try{
    console.println(k + " : " + typeof
    app[k])
}
catch(e) {
    console.println("Couldn't get type of "
+ k)
}
```

NOTE: Be careful about walking an object tree recursively! Some objects have back-references to their parents.

Arrays Are Objects

- Using `typeof` on an array will always give 'object'.
- JavaScript provides no `isArray()` function for testing arrayness.
- To test if something is an array, you can do:

```
function isArray( a )
{
    return /Array/.test(a.constructor);
}

isArray( [] ); // true
```

Objects are Arrays!

- Objects are associative arrays:

```
myObject = { California: "CA", Arizona:
"AZ",
    'New Hampshire': "NH" };

myObject[ 'California' ] // 'CA'
myObject['New Hampshire'] // 'NH'
```

- Notice that you can have spaces in property names!
- When using array notation (square brackets), the "index" value *must* be a string.

Strings are Array-like

- You can use array-like notation on strings:

```
myString = "abcdefg";  
  
myString[3];           // "d"  
myString.length       // 7
```

- To make a true array of letters out of a string (so that you can use Array methods), call `.split("")` on the string.

What Have We Learned So Far?

- To force '+' to be arithmetic, be sure every operand is numeric. Use `Number()` to cast operands as necessary.
- Use atomic data types whenever possible. String objects are of type 'object', not 'string'!

```
typeof new String("asd") // object  
typeof String("asd")    // string
```

- An equality comparison using '==' will result in typecasting if the comparands are not of the same type:

```
"11" == 11; // true (string cast to number)  
"011" == 011; // false (11 != octal 9)
```

- To force a typed (untyped) comparison, use the strict-equality operator:

```
"11" === 11 // false
```

- Use `isNaN()` to check the numberness of a result. Do NOT use `typeof`, since `NaN` is of type "number"!
- When declaring any variable, *give it a default value*. Otherwise, you've declared something that is undefined.
- An uninitialized variable has a value of `undefined`, not `null`.
- Use the `var` declarator unless you are declaring a global.
- Check a variable against the "undefined" string using `typeof`, when you aren't sure if it has been initialized.

```
typeof global.myVar == 'undefined' // true
```

A Word about Scope, Arguments, and Data-Passing

Scope Is Not a Mouthwash

- Scope (visibility and lifetime of variables) is important! Don't be careless. Always use the `var` declarator unless you have a reason not to.
- The `var` declarator gives a variable local or *function-level* scope.

```
    j = 'abc';  
    function a() {  
        var j = 5; // this 'j' is local to  
a()  
        return ++j;  
    }  
  
    a();    // 6  
    j;     // 'abc'
```

Bare variables have app-level scope!

Usually, this is not what you want.

- For app-level scope, you can also declare properties on Acrobat's Global object: `global.myVariable`
- For doc-level scope, you can declare a variable as a property on 'this':

```
    this.count = 99;
```

- Inside a constructor, declare public properties as `this.myProperty`. Declare private variables using `var`.
- EXAMPLE OF HOW TO GET IN TROUBLE:

```
function a() {
    for (i = 0; i < 10; i++)
        b();
}
function b() {
    for (i = 0; i < 3; i++)
        console.println("hello");
}
```

```
a()    // infinite loop!  b() overwrites i
```

Arguments and data-passing

- Function arguments are passed by value, *except for arrays and objects, which are passed by reference.*

```
var abc = ['a','b','c'];

// console-dump an array's contents:
function showArray( ar )
{
    while (ar.length)
        console.println( ar.pop() );
}

showArray(abc);
abc.length    // zero!
```

- All functions have an `arguments` array, which you can inspect at runtime to see how many arguments were passed, and their values.



```
function sum() // takes any number of args
{
    for (var i=0,s=0; i < arguments.length; i++)
        s += arguments[i];
    return s;
}

// thus:
sum(1,2);           // 3
sum(1,2,20,22);    // 45
```

Think General, Not Special-Purpose

- Instead of writing one-off, special-purpose functions, try to write general-purpose utilities. For example . . .

Problem: Need to iterate through an object's properties.

Solution: General-purpose iteration function to list properties in an array.

```
function getObjectProps(obj) {  
  
    var props = []; // blank array  
  
    for (var k in obj) // iterate  
        props.push(k); // accumulate  
  
    return props; // return all props  
}
```

Problem: Need to iterate through a list and process each member of the list.

Solution: General-purpose iteration function to process arrays.

```
// ITERATOR METHOD  
function filterArray( theArray, process )  
{  
    var result = [];  
    for (var i = 0; i < theArray.length; i++)  
        result = result.concat( process(theArray[i]) );  
  
    return result;  
}
```

Note that the "process" arg is a pointer to a custom function that you write. It is a *callback*.

EXAMPLE CALLBACK: Convert characters to hex values

```
function charToHex( ch ) {
    return '0x' + ch.charCodeAt(0).toString(16);
}

// USAGE:
filterArray( "abcdef".split("") , charToHex );

// OUTPUT:
0x61,0x62,0x63,0x64,0x65,0x66
```

- `toString()` can take a "number base" argument.
- You can convert a *string* into an *array* and use the `filterArray()` function to process the string.

ANONYMOUS CALLBACK

```
theArray = ["Kas Thomas", "Blow, Joe", "M.L. King"];

filterArray(theArray,
    function(item) {
        return (item.indexOf(",") == -1) ?
            item.replace(/([\.\w]+\s+(\w+)/, "$2,$1") :
            item;
    }
).join('\n');

// OUTPUT:
Thomas, Kas
Blow, Joe
King, M.L.
```

RECURSIVE CALLBACK

```
// GET ALL BOOKMARKS

// This is our callback, which can,
// in turn, call its caller:

function filter(item)
{
    return item.children ?
        [item.name,filterArray(item.children,filter)]
        :
        item.name;
}

filterArray(bookmarkRoot.children,filter)
```

ADVANCED EXAMPLE

- **Problem:** Find undocumented methods, properties, of a given Adobe API object (such as App, Annot, etc).
- **Solution:** Get documented method/property names from AcroJS.pdf; compare with actual (enumerated) object properties.

```
function getUndocumented( obj )
{
  try {
    // get our object's properties
    var propertiesList = getObjectProps(obj);

    // open AcroJS.pdf
    app.openDoc(app.getPath() +
                "../Help/AcroJS.pdf");

    // get all bookmarks
    var bmArray =
      filterArray(bookmarkRoot.children,filter);

    // remove non-methods & non-properties
    // (need to store result in global
    // so our callback can see it)

    global.api =
      filterArray(
        bmArray,
        function(item) {
          return
            String(item).match(/, [^a-
            z]+[^\,]+/) ?
            item : []; }
      );

    // callback for matching
    // properties against AcroJS bookmarks:
```

```

var theFilter = function(item) {
    return !String(global.api).match(item) ?
        '\n' + item : [];
}

    // finally, perform comparison
var hidden = filterArray(propertiesList,
    theFilter );

    global.api = undefined; // release our
global

    return hidden; // return the list of goodies!

} // try block ends

    // something didn't go right?
catch(e) { app.alert("Problems! " + e); }

    // if we errored out, release the global!
finally { global.api = undefined; }

} // end function

```

- This function looks long and ugly, but in fact there are only ten executable statements, three of which are calls to **filterArray()**.
- In one case, a callback needs to be able to see one of our function's variables, so we create a temporary global, freeing it at the end.
- Lots of stuff here, so we use **try/catch**, with a **finally** statement to free the global no matter what.

```
// EXAMPLE USAGE:

getUndocumented( app );

// OUTPUT:

getString,
getPath,
runtimeHighlight,
runtimeHighlightColor,
fsUseTimer,
fsUsePageTiming,
fsLoop,
fsEscape,
fsClick,
fsTransition,
fsTimeDelay,
fsColor,
fsCursor,
thermometer
```

External Scripts: .js Files

- You can place scripts in your Acrobat\Javascripts folder and they will be executed at program launch.
- Good way to organize and reuse libraries of frequently accessed routines.
- HTML files can also point at these scripts!
- Script files must be plain text, with a file extension of ".js"
- Any functions you place in a **.js** file will have app-wide scope.
- Some AcroJS methods can be called *only* from console or **.js** files. For example, `app.addMenuItem()` will throw an exception if it is called from a form field.
- Reader will load **.js** files, too.
- NOTE: Be sure to use 'this' on the front of Doc methods inside **.js** scripts. Acrobat complains otherwise.

Using a .js Script to Customize Acrobat's Menus

- **Problem:** Add a "New Document" command to File menu.
- **Solution:**

```
app.addItem({  
    cName:"New Document",  
    cParent:"File",  
    cExec:"app.newDoc()"  
});
```

Reading a File via `importTextData()`

- Acrobat 5.0 `importTextData()` method will read a variety of file types. Intended for text only, not binary data.
- You must know the first line in the file.
- There must be a text field by that name in the front doc.
- File will be read line-by-line into the text field.
- Termination mechanism not defined by Adobe. Method chokes on binary data and will not read past a line that consists of two successive space characters.
- `importTextData()` has no meaningful return value.
- One way to terminate is to know in advance how many lines the file has, and read just that many lines. Another tactic is to look for a special EOF marker. You can probably think of others.
- This method is incredibly slow! *Bring green bananas. Watch them ripen.*

importTextData() EXAMPLE:

```
// READ A TEXT FILE INTO AN ARRAY:
function getTextFromFile( file, firstLine,lastLine )
{
    var textArray = [firstLine];

    // create a dummy field
    var field = this.addField(firstLine,
                             "text",this.pageNum, [400,400,500,430]);

    for (var i = 0; field.value != lastLine; i++)
    {
        importTextData(file,i);
        textArray.push(field.value);
    }

    // destroy the dummy field
    this.removeField(firstLine);

    return textArray;
}
```

Creating PDF via Report object

- Acrobat 5.0 will let you "write" to a new PDF file line-by-line via the Report object.
 - Report object must be instantiated with "new".
 - Every call to *writeText()* results in a new line added to the Report.
 - Very little control over styling.
 - Text wraps automatically and page breaks are determined on-the-fly for you.
 - No file appears until you call *open()* on your Report.
-
- EXAMPLE: Using *importTextData()* and the Report object to dump a PDF file's contents

```
// Utility to generate PDF from array of strings:
function reportFromArray( ar, reportTitle )
{
    global.rep = new Report();

    filterArray( ar, function(item) {
        global.rep.writeText(item); return []; }
    );

    global.rep.open(reportTitle);
    global.rep = undefined; // free the global
}

// Grab raw text of the front PDF:
var txt = getTextFromFile(path, "%PDF-1.4", "%EOF");

// Dump it into a Report:
reportFromArray(txt, "TheReport.pdf");
```

Animation and Multithreading

- App object supports `setInterval()`, `setTimeout()`, `clearInterval()`, and `clearTimeout()` methods.
- Works in Reader. Available back to at least 4.05.
- Periodic tasks can be spawned using `setInterval()`.
- One-time "delayed actions" can be spawned with `setTimeout()`.
- Interval is always in milliseconds.
- Return value of "set" methods is a *key* that can be used to abort or turn off the animation via a "clear" method.
- `setInterval()` can be used as a way of implementing "listeners".
- Use caution! Runaway animations are hard to stop!

NOTE: Latency is significant with JavaScript, so do not count on being able to use intervals smaller than about 50 milliseconds.

ALSO: Memory leakage is a problem over time.

EXERCISE: Make the Square Annot tool a cropping tool. When the user drags out a new Square Annot, ask him (in a response dialog) whether the page should be cropped to the dimensions of the annot. If yes, crop.

```
// Utility function: crop current page
// to a given rect

function quickCrop( rect )
{
this.setPageBoxes("Media",
    this.pageNum, this.pageNum, rect);
}

// Utility: Return the most recently added annot

function lastAnnot( page )
{
    var annots = this.getAnnots(page);
    return annots != null ?
        annots[annots.length-1] : null;
}

// Utility: Return boolean indicating whether
// the last annot added was a Square Annot.

function quickCroppable()
{
    if (app.activeDocs.length > 0)
        if (this.getAnnots(this.pageNum) != null)
            if (lastAnnot( this.pageNum ).uiType ==
                'Square')
                return true;
    return false;
}
```

- BUT WAIT! How do we know when the user has created a Square annotation?
- ANSWER: Create a listener function that, once active, detects the addition of annots and queries the user when a Square Annot has been created.

- The listener must run as a background process.
- The listener must NOT query the user more than once per new Square Annot.

setInterval() to the rescue!

```
// LISTENER for crop-to-Square:
function checkCropState()
{
  if (quickCroppable())
    if (Number(new Date) - Number(
      lastAnnot(this.pageNum).modDate )<= 7 * 1000)
      if (app.alert("Crop to annot?",2,2)==4)
        {
          var annot = lastAnnot(this.pageNum);
          quickCrop(annot.rect); // crop to rect
          annot.destroy();
        }
}

// USAGE:
end = app.setInterval("checkCropState()", 4000);
app.clearInterval(end);
```

Some Quick Hacks

```
// Get name of front doc:
function frontDoc() {
    return path.split('/').pop(); }

// Get path to Acrobat folder:
app.getPath(); // undocumented method

// Get path to glob.js:
globPath = app.getPath() +
"..\\Javascripts\\glob.js"

// See if any docs are open:
function docsPresent() {
    return app.activeDocs.length > 0; }

// set the opacity of first annot to 0.5
this.getAnnots()[0].opacity = 0.5

// set the opacity of ALL annots to 0.66
filterArray( getAnnots(), function(item) {
    item.opacity=0.66; return []; } );

// create a random color:
c =['RGB',
Math.random(),
Math.random(),
Math.random()];
```

```

// invert a color:
function invert(rgb)
  { return ['RGB',1-rgb[1], 1-rgb[2], 1-rgb[3] ] }

// urlencode a string:
str = 'www.google.com/search?q="Kas Thomas"';
url = escape(str);

// Result: url ==
"www.google.com/search%3Fq%3D%22Kas%20Thomas%22"

// decode an urlencoded string:
unescape(str);

// General-purpose timing utility
function executionTime( code )
{
  var start = Number(new Date);
  eval(code);
  return Number(new Date) - start;
}

// from among all open docs, get the
// handle for a particular doc (by name)
function getDocObjectFromName( docName )
{
  for (var i = 0, d=app.activeDocs; i < d.length;i++)
    if (d[i].path.split("/").pop() == docName)
      return d[i];
  return null;
}

// alternatively:
doc = filterArray(
  app.activeDocs,
  function(item) {
    return item.path.split("/").pop() ==
'MyNewDoc.pdf' ? item : [];
  }
).join('');

```

```
// USING EXEC TO LOOP OVER A STRING

str = "<P>This is some <B>text</B>.</P>";

// example regex:
r=/<[^>]+>([^<]+)/g;
```

In English: "Less-than, followed by one or more of anything that's not a greater-than, followed by a greater-than, followed by one or more of anything that's not a less-than, and apply globally." I.e., match **<TD>February</TD>** and cache the text following the > (which is to say, remember the February part).

```
// Loop over all occurrences of pattern:
while(r.test(str))
    console.println( r.exec(str)[1] )

// OUTPUT:
This is some
text
.
```

```
// Flatten the current page:

this.flattenPages(this.pageNum,this.pageNum);
```

```
// reverse a string:

str.split('').reverse().join('');
```

```
// truncate a long decimal number:

pi = Math.PI; // 3.141592653589793
shortPi = String(pi).substr(0,4); // 3.14
```

```

// put a new Square annotation at [x,y,x2,y2]

this.addAnnot({
    page: this.pageNum,
    type: "Square",
    rect: [x,y,x2,y2],
});

// inspect the properties of the doc's first annot

with (this.getAnnots()[0]) {
    for (k in getProps())
        console.println(k + ": " + getProps()[k]);
}

// write Hello World as a FreeText Annot

this.addAnnot({
    alignment: 1,
    contents: "Hello World!",
    fillColor: ['T'],
    opacity: 0.6,
    page: this.pageNum,
    rect: [50,400,562,600],
    strokeColor: ['RGB',0.8,0.4,0.15],
    textFont: "Tahoma",
    textSize: 0, /* fit in rect */
    type: "FreeText",
    width: 0
});

```