Apple JavaScript Coding Guidelines

User Experience

2009-05-05

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Tables

Introduction to Apple JavaScript Coding Guidelines

This document provides an overview of the JavaScript language as provided by Apple in the WebKit and JavaScript Core frameworks. It looks at JavaScript language basics, object-oriented programming using JavaScript, and best practices to follow when writing JavaScript code.

Who Should Read This Document?

You should read this document if you are a webpage or Dashboard widget developer who wants to learn JavaScript and best practices for Apple's JavaScript runtime. Also, this document is useful if you're using an application that offers a scripting environment that uses Apple's JavaScript Core framework, like Installer or Quartz Composer.

Organization of This Document

This document contains the following chapters:

- "JavaScript Basics" (page 9) introduces you to the JavaScript language and its basic structures.
- "Object-Oriented JavaScript" (page 15) discusses implementing an object in JavaScript.
- "JavaScript Best Practices" (page 21) shows you best practices to consider when coding in JavaScript.

Getting WebKit Builds

Safari is powered by the WebKit framework, an open source project found at WebKit.org. To try your web-based JavaScript code with the latest JavaScript language features, download a nightly build from http://nightly.webkit.org/.

See Also

The Reference Library > Apple Applications > Safari provides useful information on WebKit, the technology that provides Apple's JavaScript runtime. Read *WebKit DOM Programming Topics* to learn more about the WebKit DOM interfaces, useful when coding a webpage or Dashboard widget.

Read *Dashboard Programming Topics* for information on the technologies available to you when creating a Dashboard widget. Additional Dashboard documents and sample code can be found in the Reference Library > Apple Applications > Dashboard.

Installer Tiger Examples includes information on using the JavaScript runtime included with Apple's Installer application.

JavaScript Basics

JavaScript is a programming language used by web browsers to provide scripting access to webpages. It's also being used more and more by desktop applications to provide an embedded scripting environment. If you're already familiar with programming, JavaScript is easy to learn and provides many of the conventions you're already used to.

This chapter introduces JavaScript language basics, including variables, iteration, and functions. Read it if you're familiar with programming principles and want to learn specifics about the JavaScript language.

Variables and Assignments

The basic building block in JavaScript is the variable. You declare variables using the `var` keyword, like this:

```
var foo;
```

Important: Although you can use variables without declaring them explicitly, this is strongly discouraged. If you assign a value to an undeclared variable within a function, you are effectively creating and using a global variable instead of a variable local to that function. If the function ever gets called recursively, or if it is simultaneously called by two different user actions, the two executing instances of the function will clobber each other's storage, resulting in very hard-to-find bugs.

You should get in the habit of always declaring variables with the `var` keyword prior to use unless you are intentionally storing data globally.

JavaScript variables are type agnostic—they can hold data of any type, like a number or an object. Once you declare a variable, you assign it a value like this:

```
foo = 5;
```

The assignment operator, `=`, gives the variable to its left the value of the expression to its right. As shown, a variable can be given a literal value like a number. It can also be given the value of another variable, like this:

```
var foo, bar;
foo = 5;
bar = foo;
```

Notice that `foo` and `bar` are declared in the same statement. This is a convenience used to declare many variables at once. Also note that each statement ends with a semicolon; JavaScript doesn't require this behavior but it's a helpful convention that prevents ambiguity since it clearly marks the end of a statement. As statements get more complex, leaving out a semicolon may cause unintended consequences, so it's best to include one at the end of each statement.

Data Types

There are five primitive types in JavaScript, as shown in Table 1.

Table 1 Primitive types in JavaScript

Type	Description
`number`	A numerical value, capable of handling integer and floating-point values
`boolean`	A logical value, either `true` or `false`
`string`	A sequence of characters
`null`	No value
`undefined`	An unknown value

JavaScript freely converts between data types as needed. Consider this statement:

```
var x = 1 + "2";
```

JavaScript first promotes the number `1` to the string `"1"` and then assigns the newly concatenated string `"12"` to `x`.

If you want to do a numeric addition, you must use the parseInt function. For example:

```
var x = 1 + parseInt("2", 10);
```

The second parameter is a radix value, and is guessed based on the number if omitted following the same rules as a number in C. (Numbers starting with 0x are hex, other numbers starting with 0 are octal, and all other numbers are base 10.)

Anything that is not a primitive type is an object. Primitive types are promoted to objects as needed, like this:

```
var message = "Hello World";
var firstCharacter = message.charAt(0);
```

JavaScript promotes the string `"Hello World"` to an object in order to call the `charAt` member function.

Conditional Statements

JavaScript includes conditional statements where enclosed code runs only if its preceding statement evaluates to a Boolean `true` value. A typical conditional statement starts with the `if` keyword and is followed by a statement to test, like this:

```
if ( variable == true)
{
   // code to execute when this statement is true
}
```

As in most C-like languages, `true` is equivalent to the number one (`1`) and `false` is equivalent to the number zero (`0`).

Within the brackets `{ ... }`, you place the code to execute if the preceding statement is true. The statement itself needs to evaluate to a Boolean value, either `true` or `false`. Various operators are used to assemble conditional statements, as shown in Table 2.

Table 2 Conditional Operators

Conditional Operator	Description
==	Returns `true` if both values are equal. Coerces values of different types before evaluating.
===	Returns `true` if both values are equal and of the same type. Returns `false` otherwise.
!=	Returns `true` if values are not equal. Coerces values of different types before evaluating
!==	Returns `true` if both values are not equal or not of the same type. Returns `false` otherwise.
<	Returns `true` if the left value is less than the right value. Returns `false` otherwise.
<=	Returns `true` if the left value is less than or equal to the right value. Returns `false` otherwise.
>	Returns `true` if the left value is greater than the right value. Returns `false` otherwise.
>=	Returns `true` if the left value is greater than or equal to the right value. Returns `false` otherwise.

Each if statement can also have a corresponding else statement, where the enclosed code is executed when the if statement evaluates to false, like this:

```
if ( variable == true)
{
    // code to execute when this statement is true
}
else {
    // code to execute when this statement is false
}
```

When you want to check a value versus a number of values, a `switch` statement is useful. In a `switch` statement, you provide a conditional statement or variable and cases to match the statement's outcome or variable's value, like this:

```
switch ( variable ){
    case 1: // code to execute when variable equals 1
            break;
    case 2: // code to execute when variable equals 2
            break;
    default: // code to execute when variable equals something other than the
cases
             break;
}
```

Note that each case ends with a break statement and that the `default` case, if provided, is called if none of the cases are matched.

Iteration Using Loops

JavaScript provides structures for looping code while a condition is true. The most basic looping structure is the `while` loop, which executes code while a conditional statement equals `true`:

```
while ( variable == true )
{
     // code to execute while variable is true
}
```

A variation on the while loop is the `do-while` loop, which executes its code at least once before checking its conditional statement, like this:

```
do
{
    // code to execute while variable is true
}
while ( variable == true );
```

The final loop type is the `for` loop. It uses three statements: an initialization statement, a conditional statement, and a iterative statement, called every time the loop completes. The most common pattern of use calls for the first statement to initialize a counter variable, the second to check the counter's values, and the third to increment it, like this:

```
for ( var i = 0; i < variable; ++i )
{
    // code to execute within this loop
}
```

Functions

JavaScript allows you to wrap code into a function that you can reuse anywhere. To create a function, use the `function` keyword, provide a name, and include a list of any variables you want to pass in:

```
function myFunction ( variableOne, variableTwo )
{
    // code to execute within this function
}
```

Functions can be declared and used anywhere within a block of JavaScript code. Variables that contain primitive values are passed into a function by value, whereas objects are passed in by reference. Functions can optionally return a value using the `return` keyword:

```
function myFunction ( variableOne )
{
    // code to execute within this function
    return variableOne;
}
```

Functions implicitly return `undefined` if you don't supply a return value.

Functions can also be assigned to a variable, like this:

```
var myFunctionVariable = function ( variable )
{
    // code to execute within this function
}
```

Once a function is defined within a variable, it can be passed as an argument into any other function. Within the receiving function, you can call that function by placing parentheses after the variable's name and optionally including arguments, like this:

```
function myFunction ( myFunctionVariable )
{
    myFunctionVariable( "aString" );
}
```

Other Special Characters

As in most C-like languages, JavaScript has various characters that allow you to connect multiple statements in useful ways. This section describes a few of them.

The Question Mark and Colon Connectors

The question mark and colon can be used in a comparison or calculation in the same way you would use an if-then-else statement in code. For example, suppose you want to increment or decrement the value of variable `Q`, depending on whether the value of the variable `increment` is true or false. You could write that code in either of two ways:

```
if (increment) { Q++; } else { Q--; }
// or
Q=increment ? Q+1 : Q-1;
```

These two statements are equivalent in every way. Overuse of this syntax can make code less readable, but moderate use can sometimes significantly clean up what would otherwise be deeply nested control statements or unnecessary duplication of large amounts of code.

For example, this syntax is often used for very simple conversion of numerical values to strings for output. The snippet below prints "yes" or "no" depending on whether the `increment` variable is set or not:

```
alert('Increment: '+(increment ? "yes" : "no"));
```

The Comma Separator

The comma separator is used to separate items in lists, such as function parameters, items in an array, and so on, and under some circumstances, can also be used to separate expressions.

When used to separate expressions, the comma separator instructs the interpreter to execute them in sequence. The result of the expression is the rightmost expression. This use of the comma is generally reserved for locations in your code where a semicolon has special meaning beyond separating statements.

For example, you use a semicolon to separate the parts of a C-style `for` loop as shown in this snippet:

```
for ( var i = 0; i < variable; ++i )
{
    // code to execute within this loop
}
```

If you need to increment multiple values at each stage of the loop, you could make your code harder to read by indexing one value off of the other (maybe) or by placing additional assignment statements prior to the start of the loop and as the last statement in the loop. However, this practice tends to make it harder to spot the loop iterator variables and thus makes the code harder to follow.

Instead, you should write the `for` loop as shown in this snippet, which displays an alert with five characters each, taken from different places in two strings:

```
var string = "The quick brown fox jumped over the lazy dog.";
var stringb = "This is a test.  This is only a test.";

for (var i=0, j=4; i<5; i++, j++) {
        alert('string['+i+'] = '+string[i]);
        alert('stringb['+j+'] = '+stringb[j]);
}
```

Note: The comma separator behaves subtly differently in the first part of a `for` statement. This code creates two local variables, `i` and `j`. If you must mix local and global variables, you must declare the local variables outside the loop and omit the `var` keyword entirely.

Within the middle part of a for loop, the comma separator is nearly useless except for code obfuscation purposes. The rightmost comparison statement in such a list is used for comparison purposes.

Object-Oriented JavaScript

JavaScript code doesn't necessarily follow object-oriented design patterns. However, the JavaScript language provides support for common object-oriented conventions, like constructors and inheritance, and benefits from using these principles by being easier to maintain and reuse.

Read this chapter if you're already familiar with object-oriented programming and want to see how to create object-oriented code with JavaScript.

Declaring an Object

JavaScript provides the ability to create objects that contain member variables and functions. To define a new object, first create a new function with the desired object's name, like this:

```
function myObject()
{
    // constructor code goes here
}
```

This function works as the object's constructor, setting up any variables needed by the object and handling any initializer values passed in as arguments. Within the constructor or any member function, persistent instance variables are created by adding them to the `this` keyword and providing the variable with a value, like this:

```
this.myVariable = 5;
```

To create a new member function, create a new anonymous function and assign it to the prototype property of the object. For example:

```
myObject.prototype.memberMethod = function()
{
    // code to execute within this method
}
```

Important: You must do this assignment outside the constructor. Assigning a member function to a prototype should not be confused with assigning an anonymous function to a member variable of a class, which only adds the function to a particular instance of the class, and is probably not what you want.

After you have created the constructor for your object, you can create new object instances by using the `new` keyword followed by a call to the constructor function. For example:

```
var myObjectVariable = new myObject();
```

This particular constructor takes no parameters. However, you can also create more complex constructors for classes that allow you to specify commonly used or mandatory initial values. For example:

```
function myObject(bar)
{
        this.bar = bar;
}

myobj = new myObject(3);
alert('myobj.bar is '+myobj.bar); // prints 3.
```

One common use of this technique is to create initializers that duplicate the contents of an existing object. This is described further in "Copying an Object" (page 18).

Freeing Objects

Just as you used the `new` operator to create an object, you should delete objects when you are finished with them, like this:

```
delete myObjectVariable;
```

The JavaScript runtime automatically garbage collects objects when their value is set to `null`. However, setting an object to `null` doesn't remove the variable that references the object from memory. Using `delete` ensures that this memory is reclaimed in addition to the memory used by the object itself.

Important: Do not call `delete` if you are keeping references to the object in other variables. Calling `delete` on the object can cause a performance hit when it is accessed from those other variables.

Arrays and Objects

JavaScript provides arrays for collecting data. Since an array is an object, you need to create a new instance of an array before using it, like this:

```
var myArray = new Array();
```

Alternatively, you can use this simpler syntax:

```
var myArray = [];
```

Once you create an array, you reference elements using integer values inside of brackets, like this:

```
myArray[0] = "first value";
myArray[1] = 5;
```

The previous example shows that arrays, like variables, can hold data of any type.

Generic objects and arrays can be used with associative properties, where strings replace the index number:

```
myObject["indexOne"] = "first value";
myObject["indexTwo"] = 5;
```

When you use a string as an index, you can use a period instead of brackets to access and assign data to that property:

```
myObject.name = "Apple Inc.";
myObject.city = "Cupertino";
```

You can declare an object and its contents inline, as well:

```
myObject = {
    name: "Apple Inc.",
    city: "Cupertino"
}
```

A variation of the `for` loop is available for iterating within the properties of an array or object. Called a `for-in` loop, it looks like this:

```
for ( var index in myArray )
{
    // code to execute within this loop
}
```

Getters and Setters

Recent WebKit builds (available from webkit.org) contain support for getter and setter functions. These functions behave like a variable but really call a function to change any underlying values that make a property. Declaring a getter looks like this:

```
myObject.__defineGetter__( "myGetter", function() { return this.myVariable; }
);
```

It is essential that a getter function return a value because, when used, the returned value is used in an assignment operation like this:

```
var someVariable = myObject.myGetter;
```

Like a getter, a setter is defined on an object and provides a keyword and a function:

```
myObject.__defineSetter__( "mySetter", function(aValue) { this.myVariable =
aValue; } );
```

Note that setter functions need to take an argument that's used in an assignment statement, like this:

```
myObject.mySetter = someVariable;
```

Inheritance

JavaScript allows an object to inherit from other objects via the `prototype` keyword. To inherit from another object, set your object's prototype equal to the prototype of the parent object, like this:

```
MyChildObject.prototype = MyParentObject.prototype;
```

Note: Because the prototype keyword is undefined on instances of a class, this action must be performed on the *constructor function* names, not on variables containing instances of the class.

This copies all of the parent's functions and variables to your object. It does *not*, however, copy the default values stored in that object. If your purpose is to subclass an object completely, you must also execute code equivalent to the original constructor or construct an instance of the original type and copy its values as described in "Copying an Object" (page 18).

Recent WebKit builds also include the ability to extend an existing DOM object via the `prototype` keyword. The advantage of this over inheritance is a smaller memory footprint and easier function overloading.

Copying an Object

In object-oriented programming, you often need to make a copy of an object. The most straightforward (but painful) way to do this is to methodically copy each value by hand. For example:

```
function myObjectCopy(oldobj)
{
        // copy values one at a time
        this.bar = oldobj.bar;
}
```

For very simple objects, explicit copying is fine. However, as objects grow larger, this technique can become unmanageable, particularly if additional variables are added to an object outside the constructor. Fortunately, because a JavaScript object is essentially an associative array, you can manipulate it just as you would an array. For example:

```
function myObjectCopy(oldobj)
{
        // copy values with an iterator
        for (myvar in oldobj) {
                this[myvar] = oldobj[myvar];
        }
}
```

You should be aware, however, that assigning an object to a variable merely stores a reference to that object, not a copy of that object, and thus any objects stored within this object will not be copied by the above code.

If you need to perform a deep copy of a structured tree of data, you should explicitly call a function to copy the nested objects. The easiest way to do this is to include a `clone` function in each class, test each variable to see if it contains an object with a clone function, and call that function if it exists. For example:

```
// Create an inner object with a variable x whose default
// value is 3.
function innerObj()
{
        this.x = 3;
}
innerObj.prototype.clone = function() {
    var temp = new innerObj();
    for (myvar in this) {
```

```
        // this object does not contain any objects, so
        // use the lightweight copy code.
        temp[myvar] = this[myvar];
    }
    return temp;
}

// Create an outer object with a variable y whose default
// value is 77.
function outerObj()
{
        // The outer object contains an inner object.  Allocate it here.
        this.inner = new innerObj();
        this.y = 77;
}
outerObj.prototype.clone = function() {
    var temp = new outerObj();
    for (myvar in this) {
        if (this[myvar].clone) {
            // This variable contains an object with a
            // clone operator.  Call it to create a copy.
            temp[myvar] = this[myvar].clone();
        } else {
            // This variable contains a scalar value,
            // a string value, or an object with no
            // clone function.  Assign it directly.
            temp[myvar] = this[myvar];
        }
    }
    return temp;
}

// Allocate an outer object and assign non-default values to variables in
// both the outer and inner objects.
outer = new outerObj;
outer.inner.x = 4;
outer.y = 16;

// Clone the outer object (which, in turn, clones the inner object).
newouter = outer.clone();

// Verify that both values were copied.
alert('inner x is '+newouter.inner.x); // prints 4
alert('y is '+newouter.y); // prints 16
```

JavaScript Best Practices

There are a number of considerations that you should take into account when writing JavaScript code. Whether you're trying to tune your code for better performance or test it for compatibility, these best practices can help your code perform better and be more compatible.

If you're looking for performance and testing tips for JavaScript coding, read this chapter.

Memory and Performance Considerations

This section includes a number of tips for minimizing your JavaScript application's memory footprint and helping it perform better.

- **Release initialization functions.** Code that's called once and never used again can be deleted after its execution. For instance, deleting a window's `onload` handler function releases any memory associated with the function, like this:

```
var foo = function()
{
    // code that makes this function work
    delete foo;
}
window.addEventListener('load', foo, false);
```

- **Use delete statements.** Whenever you create an object using a `new` statement, pair it with a `delete` statement. This ensures that all of the memory associated with the object, including its property name, is available for garbage collection. The `delete` statement is discussed more in "Freeing Objects" (page 16).
- **Test for an element's existence.** Before using nonstandard elements, check for their existence like this:

```
if ( "innerHTML" in document.getElementById("someDiv") )
{
    // code that works with innerHTML
}
```

- **Avoid evaluated statements.** Using the `eval` function disables performance and memory optimizations in the JavaScript runtime. Consider using function variables, as discussed in "Functions" (page 12), in cases where you are passing in lines of code to another function, like `setTimeout` or `setInterval`.

Testing Your Code

When creating a JavaScript application, consider these tips for testing your code.

- **Try multiple browsers.** It's a good practice to try any web-based JavaScript code you write in many browsers. In addition to the version of Safari included with Mac OS X, consider trying your code on these browsers:
  - WebKit nightly - http://nightly.webkit.org/
  - Mozilla Firefox - http://www.mozilla.com/firefox/
  - Opera - http://www.opera.com/
- **Try other toolkit versions.** If you're using a third-party JavaScript toolkit such as Dojo, MochiKit, or prototype, try different versions of the library to see if your code uses the library properly.
- **Try a verifier.** JavaScript verifiers such as JSLint are useful at pinpointing compatibility problems and cases where the JavaScript code you wrote doesn't conform to standards.

Global Scope Considerations

When writing JavaScript code, it's good practice to avoid placing code and variables within the global scope of the file. Here are some considerations to consider regarding global scope:

- **Store data in the DOM tree, in cookies, or in HTML 5 client-side storage objects.** The DOM tree, cookies, and HTML 5 client-side storage are good ways to conceal data to avoid conflicts with pages outside your domain. For performance reasons, you should generally limit use of these techniques to infrequently-used data, but any of these three techniques is better than using global variables.

  You can store data in the DOM tree like this:

```
/* Store data in the DOM tree */
var DOMobj = document.getElementById('page_globals');
DOMobj.setAttribute('foo', myIntegerVar);

/* for large text content, consider using this: */
DOMobj.style.display = "none";
DOMobj.innerHTML = myStringVar;

/* or this: */

DOMobj.style.display = "none";
if (typeof(DOMobj.innerText) != 'undefined') {
    # Internet Explorer workaround
    DOMobj.innerText = myStringVar;
} else {
    # Per the W3C standard
    DOMobj.textContent = myStringVar;
}

/* Recall the data later */
var DOMobj = document.getElementById('page_globals');
var myIntegerVar = parseInt(DOMobj.getAttribute('foo'));

/* Recall the large text data with this: */
var myStringVar = DOMobj.innerHTML;

/* or this: */
```

```
var myStringVar;
if (typeof(DOMobj.innerText) != 'undefined') {
    # Internet Explorer workaround
    myStringVar = DOMobj.innerText;
} else {
    # Per the W3C standard
    myStringVar = DOMobj.textContent;
}
```

Cross-Browser Compatibility Note: Internet Explorer does not support the `textContent` attribute on elements. For maximum compatibility, as shown in the previous example, you should support not only the W3C-standard `textContent` attribute, but also the IE-specific `innerText` attribute.

You can also store data in cookies using the `cookie` attribute of the `document` object. The syntax for creating and expiring cookies is somewhat complex and easy to get wrong. Thus, this method is discouraged as a global variable alternative unless you already have code for creating and modifying cookies. To learn more about cookies in JavaScript, read http://www.quirksmode.org/js/cookies.html.

Finally, for data that should persist across page loads, you can use HTML 5 client-side storage as described in *Safari Client-Side Storage and Offline Applications Programming Guide*. Safari 3.1 and later provides both SQL database and local key-value storage.

- **Use the var keyword.** Any variable created without the `var` keyword is created at the global scope and is not garbage collected when the function returns (because it doesn't go out of scope), presenting the opportunity for a memory leak.
- **Use a global array, global object, or namespace prefix.** If you need global variables, use a global object that contains all of the global variables, like this:

```
var myBulletinBoardAppGlobals = {
    foo: "some value",
    bar: null
};
```

Or consider using a namespace prefix:

```
CPWFoo = "some value";
CPWBar = null;
```

These practices prevent your global variables from colliding with global DOM objects present in the runtime.

- **Use an initialization function.** If you have code that's run immediately when the JavaScript is loaded, place it in a function that's called when the window loads, like this:

```
myBulletinBoardAppGlobals.foo = function()
{
    // code that makes this function work
    delete myBulletinBoardAppGlobals.foo;
}
window.addEventListener('load', globals.foo, false);
```

JavaScript Toolkits

JavaScript toolkits enhance productivity but often carry a significant memory footprint that may slow down your JavaScript application. When using a toolkit, try to reduce the toolkit's footprint to just the APIs that you use in your application.

Also, if you're developing a Dashboard widget, try to avoid bundling the Apple-provided Apple Classes within your widget; instead, follow the instructions provided in Introduction to the Apple Classes.

Document Revision History

This table describes the changes to *Apple JavaScript Coding Guidelines*.

Date	Notes
2009-05-05	Added additional clarification on use of delete, storage objects.
2008-11-19	Removed a related documents link to a document that no longer exists.
2008-10-15	Added additional content about using the DOM tree as an alternative to global variables. Made several typographical fixes.
2008-01-15	Changed the title from "JavaScript Coding Guidelines for Mac OS X."
2007-09-11	Added some additional tips and techniques.
2007-07-10	Fixed typographical errors.
2007-06-11	New document that provides an overview of the JavaScript language, its object-oriented features, and coding best practices.