CHAPTER 2
Data Access

One of the classic computer science problems is determining where data should be stored for optimal reading and writing. Where data is stored is related to how quickly it can be retrieved during code execution. This problem in JavaScript is somewhat simplified because of the small number of options for data storage. Similar to other languages, though, where data is stored can greatly affect how quickly it can be accessed later. There are four basic places from which data can be accessed in JavaScript:

Literal values
: Any value that represents just itself and isn't stored in a particular location. JavaScript can represent strings, numbers, Booleans, objects, arrays, functions, regular expressions, and the special values `null` and `undefined` as literals.

Variables
: Any developer-defined location for storing data created by using the `var` keyword.

Array items
: A numerically indexed location within a JavaScript `Array` object.

Object members
: A string-indexed location within a JavaScript object.

Each of these data storage locations has a particular cost associated with reading and writing operations involving the data. In most cases, the performance difference between accessing information from a literal value versus a local variable is trivial. Accessing information from array items and object members is more expensive, though exactly which is more expensive depends heavily on the browser. Figure 2-1 shows the relative speed of accessing 200,000 values from each of these four locations in various browsers.

Older browsers using more traditional JavaScript engines, such as Firefox 3, Internet Explorer, and Safari 3.2, show a much larger amount of time taken to access values versus browsers that use optimizing JavaScript engines. The general trends, however, remain the same across all browsers: literal value and local variable access tend to be faster than array item and object member access. The one exception, Firefox 3,

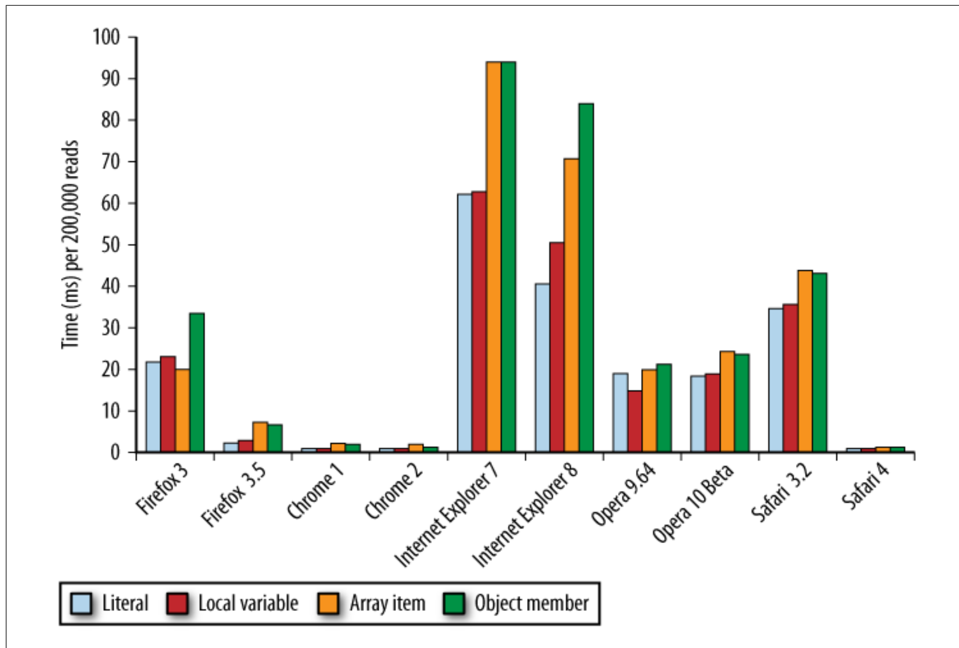


Figure 2-1. Time per 200,000 reads from various data locations

optimized array item access to be much faster. Even so, the general advice is to use literal values and local variables whenever possible and limit use of array items and object members where speed of execution is a concern. To that end, there are several patterns to look for, avoid, and optimize in your code.

Managing Scope

The concept of scope is key to understanding JavaScript not just from a performance perspective, but also from a functional perspective. Scope has many effects in JavaScript, from determining what variables a function can access to assigning the value of `this`. There are also performance considerations when dealing with JavaScript scopes, but to understand how speed relates to scope, it's necessary to understand exactly how scope works.

Scope Chains and Identifier Resolution

Every function in JavaScript is represented as an object—more specifically, as an instance of `Function`. Function objects have properties just like any other object, and these include both the properties that you can access programmatically and a series of internal properties that are used by the JavaScript engine but are not accessible through code. One of these properties is `[[Scope]]`, as defined by ECMA-262, Third Edition.

The internal [[Scope]] property contains a collection of objects representing the scope in which the function was created. This collection is called the function's *scope chain* and it determines the data that a function can access. Each object in the function's scope chain is called a *variable object*, and each of these contains entries for variables in the form of key-value pairs. When a function is created, its scope chain is populated with objects representing the data that is accessible in the scope in which the function was created. For example, consider the following global function:

```
function add(num1, num2){
    var sum = num1 + num2;
    return sum;
}
```

When the add() function is created, its scope chain is populated with a single variable object: the global object representing all of the variables that are globally defined. This global object contains entries for window, navigator, and document, to name a few. Figure 2-2 shows this relationship (note the global object in this figure shows only a few of the global variables as an example; there are many others).

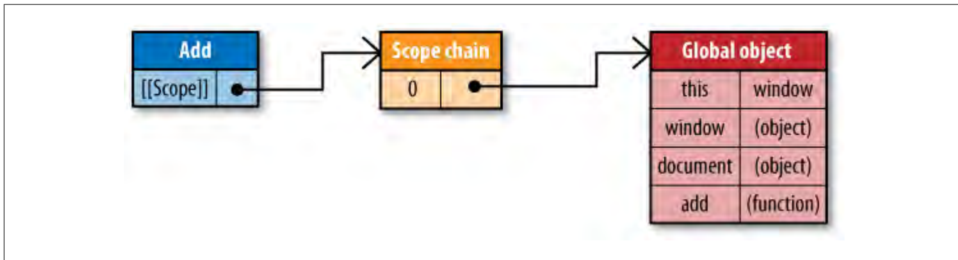


Figure 2-2. Scope chain for the add() function

The add function's scope chain is later used when the function is executed. Suppose that the following code is executed:

```
var total = add(5, 10);
```

Executing the add function triggers the creation of an internal object called an *execution context*. An execution context defines the environment in which a function is being executed. Each execution context is unique to one particular execution of the function, and so multiple calls to the same function result in multiple execution contexts being created. The execution context is destroyed once the function has been completely executed.

An execution context has its own scope chain that is used for identifier resolution. When the execution context is created, its scope chain is initialized with the objects contained in the executing function's `[[Scope]]` property. These values are copied over into the execution context scope chain in the order in which they appear in the function. Once this is complete, a new object called the *activation object* is created for the execution context. The activation object acts as the variable object for this execution and contains entries for all local variables, named arguments, the `arguments` collection, and `this`. This object is then pushed to the front of the scope chain. When the execution context is destroyed, so is the activation object. Figure 2-3 shows the execution context and its scope chain for the previous example code.

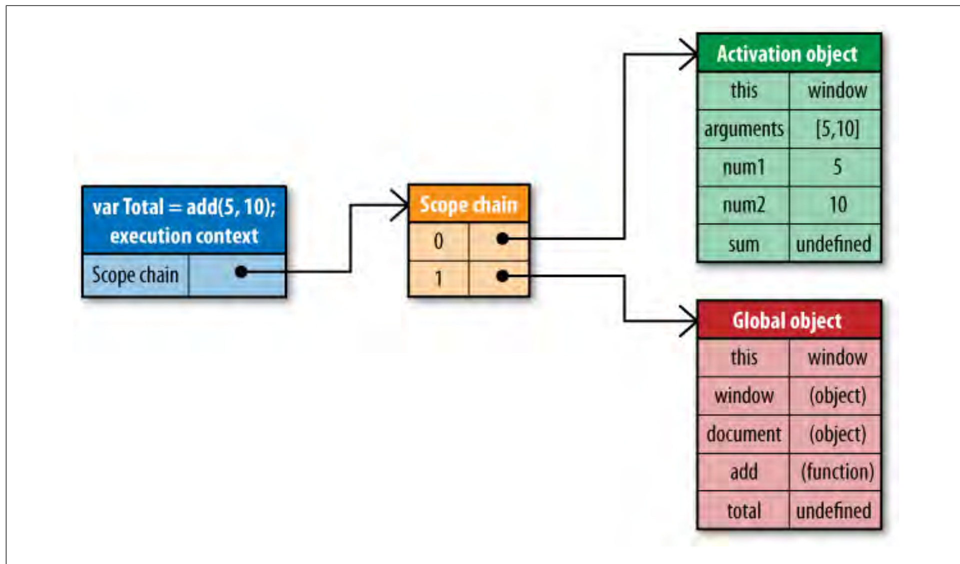
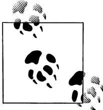


Figure 2-3. Scope chain while executing add()

Each time a variable is encountered during the function's execution, the process of identifier resolution takes place to determine where to retrieve or store the data. During this process, the execution context's scope chain is searched for an identifier with the same name. The search begins at the front of the scope chain, in the execution function's activation object. If found, the variable with the specified identifier is used; if not, the search continues on to the next object in the scope chain. This process continues until either the identifier is found or there are no more variable objects to search, in which case the identifier is deemed to be undefined. The same approach is taken for each identifier found during the function execution, so in the previous example, this would happen for `sum`, `num1`, and `num2`. It is this search process that affects performance.



Note that two variables with the same name may exist in different parts of the scope chain. In that case, the identifier is bound to the variable that is found first in the scope chain traversal, and the first variable is said to *shadow* the second.

Identifier Resolution Performance

Identifier resolution isn't free, as in fact no computer operation really is without some sort of performance overhead. The deeper into the execution context's scope chain an identifier exists, the slower it is to access for both reads and writes. Consequently, local variables are always the fastest to access inside of a function, whereas global variables will generally be the slowest (optimizing JavaScript engines are capable of tuning this in certain situations). Keep in mind that global variables always exist in the last variable object of the execution context's scope chain, so they are always the furthest away to resolve. Figures 2-4 and 2-5 show the speed of identifier resolution based on their depth in the scope chain. A depth of 1 indicates a local variable.

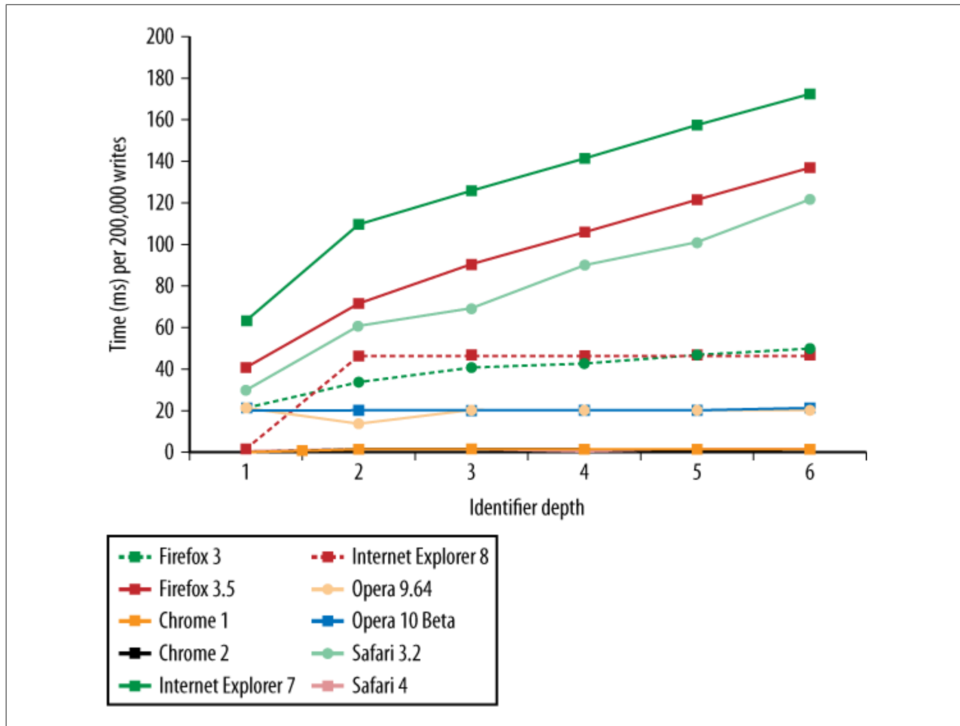


Figure 2-4. Identifier resolution for write operations

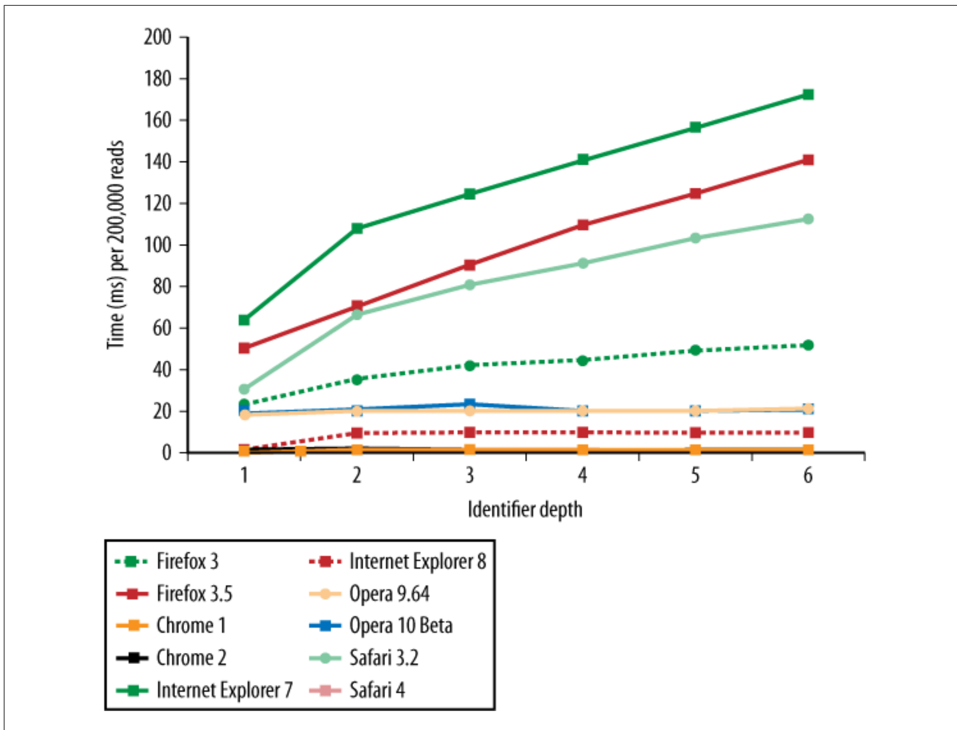


Figure 2-5. Identifier resolution for read operations

The general trend across all browsers is that the deeper into the scope chain an identifier exists, the slower it will be read from or written to. Browsers with optimizing JavaScript engines, such as Chrome and Safari 4, don't have this sort of performance penalty for accessing out-of-scope identifiers, whereas Internet Explorer, Safari 3.2, and others show a more drastic effect. It's worth noting that earlier browsers, such as Internet Explorer 6 and Firefox 2, had incredibly steep slopes and would not even appear within the bounds of this graph at the high point if their data had been included.

Given this information, it's advisable to use local variables whenever possible to improve performance in browsers without optimizing JavaScript engines. A good rule of thumb is to always store out-of-scope values in local variables if they are used more than once within a function. Consider the following example:

```
function initUI(){
    var bd = document.body,
        links = document.getElementsByTagName("a"),
        i= 0,
        len = links.length;

    while(i < len){
        update(links[i++]);
    }
```

```
    document.getElementById("go-btn").onclick = function(){
        start();
    };

    bd.className = "active";
}
```

This function contains three references to `document`, which is a global object. The search for this variable must go all the way through the scope chain before finally being resolved in the global variable object. You can mitigate the performance impact of repeated global variable access by first storing the reference in a local variable and then using the local variable instead of the global. For example, the previous code can be rewritten as follows:

```
function initUI(){
    var doc = document,
        bd = doc.body,
        links = doc.getElementsByTagName("a"),
        i= 0,
        len = links.length;

    while(i < len){
        update(links[i++]);
    }

    doc.getElementById("go-btn").onclick = function(){
        start();
    };

    bd.className = "active";
}
```

The updated version of `initUI()` first stores a reference to `document` in the local `doc` variable. Instead of accessing a global variables three times, that number is cut down to one. Accessing `doc` instead of `document` is faster because it's a local variable. Of course, this simplistic function won't show a huge performance improvement, because it's not doing that much, but imagine larger functions with dozens of global variables being accessed repeatedly; that is where the more impressive performance improvements will be found.

Scope Chain Augmentation

Generally speaking, an execution context's scope chain doesn't change. There are, however, two statements that temporarily augment the execution context's scope chain while it is being executed. The first of these is `with`.

The `with` statement is used to create variables for all of an object's properties. This mimics other languages with similar features and is usually seen as a convenience to avoid writing the same code repeatedly. The `initUI()` function can be written as the following:

```
function initUI(){
    with (document){     //avoid!
        var bd = body,
            links = getElementsByTagName("a"),
            i= 0,
            len = links.length;

        while(i < len){
            update(links[i++]);
        }

        getElementById("go-btn").onclick = function(){
            start();
        };

        bd.className = "active";
    }
}
```

This rewritten version of `initUI()` uses a `with` statement to avoid writing `document` elsewhere. Though this may seem more efficient, it actually creates a performance problem.

When code execution flows into a `with` statement, the execution context's scope chain is temporarily augmented. A new variable object is created containing all of the properties of the specified object. That object is then pushed to the front of the scope chain, meaning that all of the function's local variables are now in the second scope chain object and are therefore more expensive to access (see Figure 2-6).

By passing the `document` object into the `with` statement, a new variable object containing all of the `document` object's properties is pushed to the front of the scope chain. This makes it very fast to access `document` properties but slower to access the local variables such as `bd`. For this reason, it's best to avoid using the `with` statement. As shown previously, it's just as easy to store `document` in a local variable and get the performance improvement that way.

The `with` statement isn't the only part of JavaScript that artificially augments the execution context's scope chain; the `catch` clause of the `try-catch` statement has the same effect. When an error occurs in the `try` block, execution automatically flows to the `catch` and the exception object is pushed into a variable object that is then placed at the front of the scope chain. Inside of the `catch` block, all variables local to the function are now in the second scope chain object. For example:

```
try {
    methodThatMightCauseAnError();
} catch (ex){
    alert(ex.message);  //scope chain is augmented here
}
```

Note that as soon as the `catch` clause is finished executing, the scope chain returns to its previous state.

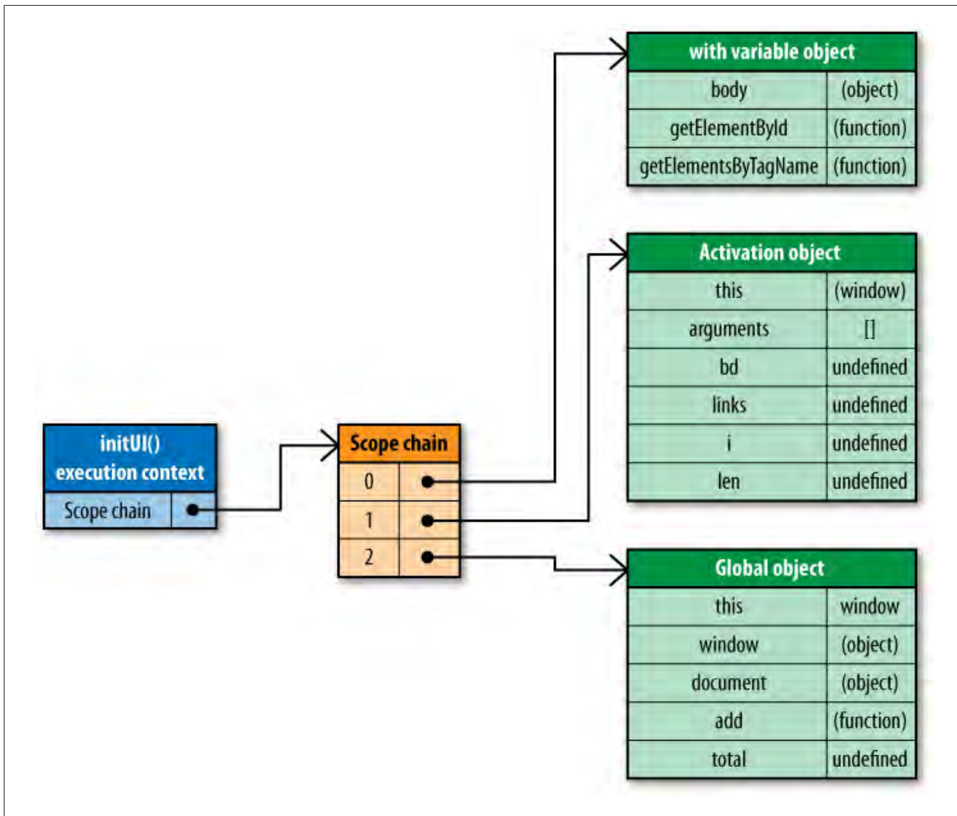


Figure 2-6. Augmented scope chain in a with statement

The `try-catch` statement is very useful when applied appropriately, and so it doesn't make sense to suggest complete avoidance. If you do plan on using a `try-catch`, make sure that you understand the likelihood of error. A `try-catch` should never be used as the solution to a JavaScript error. If you know an error will occur frequently, then that indicates a problem with the code itself that should be fixed.

You can minimize the performance impact of the `catch` clause by executing as little code as necessary within it. A good pattern is to have a method for handling errors that the `catch` clause can delegate to, as in this example:

```
try {
    methodThatMightCauseAnError();
} catch (ex){
    handleError(ex);  //delegate to handler method
}
```

Here a `handleError()` method is the only code that is executed in the `catch` clause. This method is free to handle the error in an appropriate way and is passed the exception object generated from the error. Since there is just one statement executed and no local

variables accessed, the temporary scope chain augmentation does not affect the performance of the code.

Dynamic Scopes

Both the `with` statement and the `catch` clause of a `try-catch` statement, as well as a function containing `eval()`, are all considered to be *dynamic scopes*. A dynamic scope is one that exists only through execution of code and therefore cannot be determined simply by static analysis (looking at the code structure). For example:

```
function execute(code) {
  eval(code);

  function subroutine(){
    return window;
  }

  var w = subroutine();

  //what value is w?
};
```

The `execute()` function represents a dynamic scope due to the use of `eval()`. The value of `w` can change based on the value of `code`. In most cases, `w` will be equal to the global `window` object, but consider the following:

```
execute("var window = {};")
```

In this case, `eval()` creates a local `window` variable in `execute()`, so `w` ends up equal to the local `window` instead of the global. There is no way to know if this is the case until the code is executed, which means the value of the `window` identifier cannot be predetermined.

Optimizing JavaScript engines such as Safari's Nitro try to speed up identifier resolution by analyzing the code to determine which variables should be accessible at any given time. These engines try to avoid the traditional scope chain lookup by indexing identifiers for faster resolution. When a dynamic scope is involved, however, this optimization is no longer valid. The engines need to switch back to a slower hash-based approach for identifier resolution that more closely mirrors traditional scope chain lookup.

For this reason, it's recommended to use dynamic scopes only when absolutely necessary.

Closures, Scope, and Memory

Closures are one of the most powerful aspects of JavaScript, allowing a function to access data that is outside of its local scope. The use of closures has been popularized through the writings of Douglas Crockford and is now ubiquitous in most complex

web applications. There is, however, a performance impact associated with using closures.

To understand the performance issues with closures, consider the following:

```
function assignEvents(){

    var id = "xdi9592";

    document.getElementById("save-btn").onclick = function(event){
        saveDocument(id);
    };
}
```

The `assignEvents()` function assigns an event handler to a single DOM element. This event handler is a closure, as it is created when the `assignEvents()` is executed and can access the `id` variable from the containing scope. In order for this closure to access `id`, a specific scope chain must be created.

When `assignEvents()` is executed, an activation object is created that contains, among other things, the `id` variable. This becomes the first object in the execution context's scope chain, with the global object coming second. When the closure is created, its `[[Scope]]` property is initialized with both of these objects (see Figure 2-7).

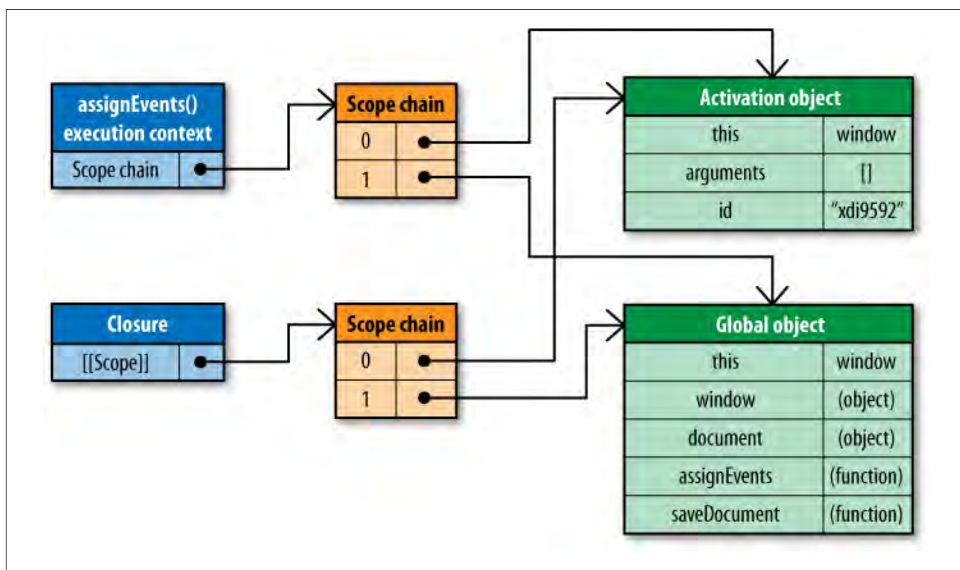


Figure 2-7. Scope chains of the assignEvents() execution context and closure

Since the closure's `[[Scope]]` property contains references to the same objects as the execution context's scope chain, there is a side effect. Typically, a function's activation object is destroyed when the execution context is destroyed. When there's a closure involved, though, the activation object isn't destroyed, because a reference still exists

in the closure's `[[Scope]]` property. This means that closures require more memory overhead in a script than a nonclosure function. In large web applications, this might become a problem, especially where Internet Explorer is concerned. IE implements DOM objects as nonnative JavaScript objects, and as such, closures can cause memory leaks (see Chapter 3 for more information).

When the closure is executed, an execution context is created whose scope chain is initialized with the same two scope chain objects referenced in `[[Scope]]`, and then a new activation object is created for the closure itself (see Figure 2-8).

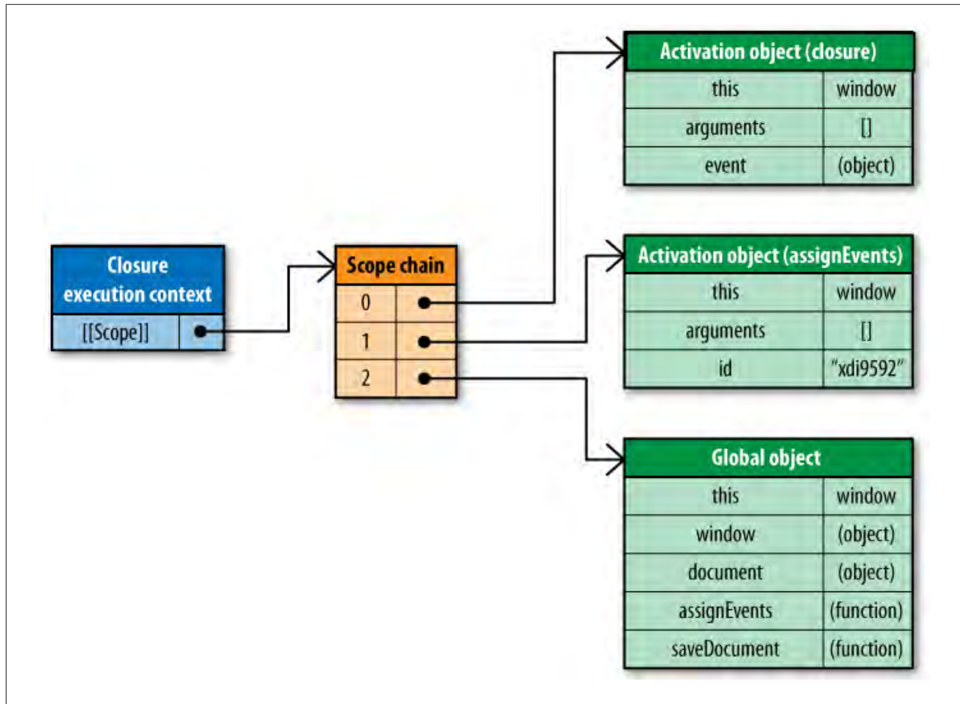


Figure 2-8. Executing the closure

Note that both identifiers used in the closure, `id` and `saveDocument`, exist past the first object in the scope chain. This is the primary performance concern with closures: you're often accessing a lot of out-of-scope identifiers and therefore are incurring a performance penalty with each access.

It's best to exercise caution when using closures in your scripts, as they have both memory and execution speed concerns. However, you can mitigate the execution speed impact by following the advice from earlier in this chapter regarding out-of-scope variables: store any frequently used out-of-scope variables in local variables, and then access the local variables directly.

Object Members

Most JavaScript is written in an object-oriented manner, either through the creation of custom objects or the use of built-in objects such as those in the Document Object Model (DOM) and Browser Object Model (BOM). As such, there tends to be a lot of object member access.

Object members are both properties and methods, and there is little difference between the two in JavaScript. A named member of an object may contain any data type. Since functions are represented as objects, a member may contain a function in addition to the more traditional data types. When a named member references a function, it's considered a method, whereas a member referencing a nonfunction data type is considered a property.

As discussed earlier in this chapter, object member access tends to be slower than accessing data in literals or variables, and in some browsers slower than accessing array items. To understand why this is the case, it's necessary to understand the nature of objects in JavaScript.

Prototypes

Objects in JavaScript are based on *prototypes*. A prototype is an object that serves as the base of another object, defining and implementing members that a new object must have. This is a completely different concept than the traditional object-oriented programming concept of classes, which define the process for creating a new object. Prototype objects are shared amongst all instances of a given object type, and so all instances also share the prototype object's members.

An object is tied to its prototype by an internal property. Firefox, Safari, and Chrome expose this property to developers as `__proto__`; other browsers do not allow script access to this property. Any time you create a new instance of a built-in type, such as `Object` or `Array`, these instances automatically have an instance of `Object` as their prototype.

Consequently, objects can have two types of members: instance members (also called "own" members) and prototype members. Instance members exist directly on the object instance itself, whereas prototype members are inherited from the object prototype. Consider the following example:

```
var book = {
    title: "High Performance JavaScript",
    publisher: "Yahoo! Press"
};

alert(book.toString());   //"[object Object]"
```

In this code, the book object has two instance members: title and publisher. Note that there is no definition for the method toString() but that the method is called and behaves appropriately without throwing an error. The toString() method is a prototype member that the book object is inheriting. Figure 2-9 shows this relationship.

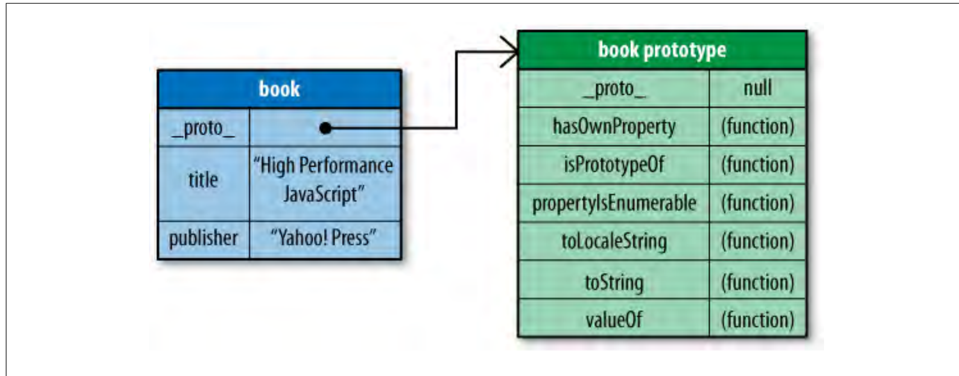


Figure 2-9. Relationship between an instance and prototype

The process of resolving an object member is very similar to resolving a variable. When book.toString() is called, the search for a member named "toString" begins on the object instance. Since book doesn't have a member named toString, the search then flows to the prototype object, where the toString() method is found and executed. In this way, book has access to every property or method on its prototype.

You can determine whether an object has an instance member with a given name by using the hasOwnProperty() method and passing in the name of the member. To determine whether an object has access to a property with a given name, you can use the in operator. For example:

```
var book = {
    title: "High Performance JavaScript",
    publisher: "Yahoo! Press"
};

alert(book.hasOwnProperty("title"));       //true
alert(book.hasOwnProperty("toString"));    //false

alert("title" in book);       //true
alert("toString" in book);    //true
```

In this code, hasOwnProperty() returns true when "title" is passed in because title is an object instance; the method returns false when "toString" is passed in because it doesn't exist on the instance. When each property name is used with the in operator, the result is true both times because it searches the instance and prototype.

Prototype Chains

The prototype of an object determines the type or types of which it is an instance. By default, all objects are instances of `Object` and inherit all of the basic methods, such as `toString()`. You can create a prototype of another type by defining and using a constructor. For example:

```
function Book(title, publisher){
    this.title = title;
    this.publisher = publisher;
}

Book.prototype.sayTitle = function(){
    alert(this.title);
};

var book1 = new Book("High Performance JavaScript", "Yahoo! Press");
var book2 = new Book("JavaScript: The Good Parts", "Yahoo! Press");

alert(book1 instanceof Book);   //true
alert(book1 instanceof Object); //true

book1.sayTitle();          //"High Performance JavaScript"
alert(book1.toString());  //"[object Object]"
```

The `Book` constructor is used to create a new instance of `Book`. The `book1` instance's prototype (`__proto__`) is `Book.prototype`, and `Book.prototype`'s prototype is `Object`. This creates a prototype chain from which both `book1` and `book2` inherit their members. Figure 2-10 shows this relationship.

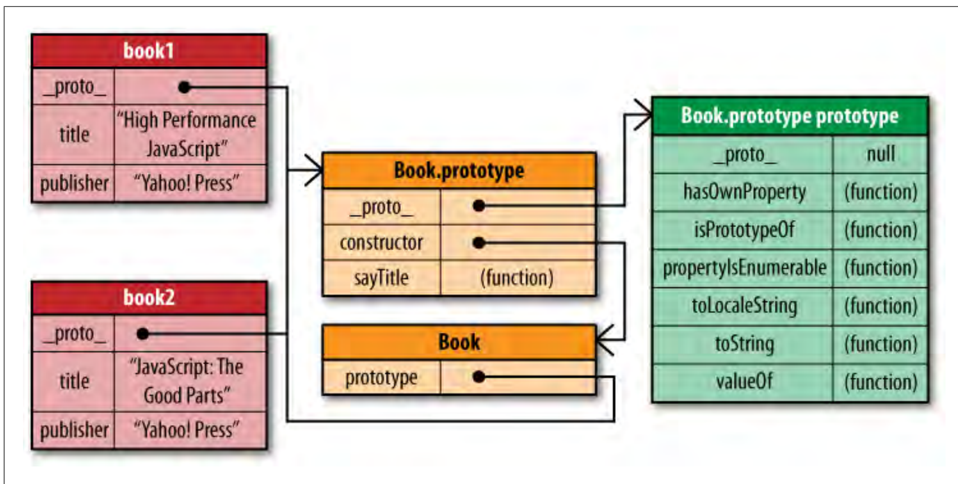


Figure 2-10. Prototype chains

Note that both instances of `Book` share the same prototype chain. Each instance has its own title and publisher properties, but everything else is inherited through prototypes.

Now when `book1.toString()` is called, the search must go deeper into the prototype chain to resolve the object member "toString". As you might suspect, the deeper into the prototype chain that a member exists, the slower it is to retrieve. Figure 2-11 shows the relationship between member depth in the prototype and time to access the member.

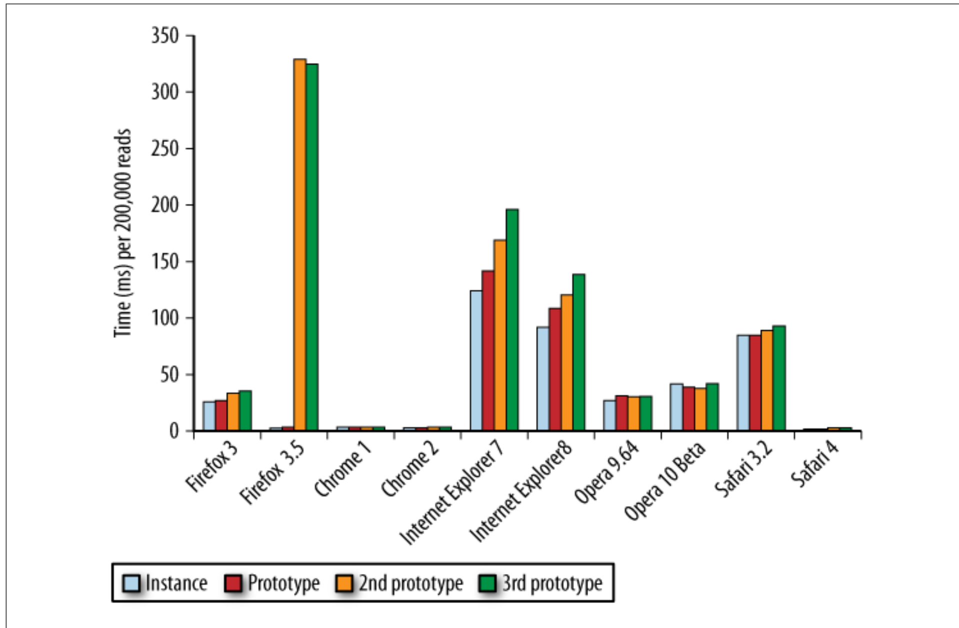


Figure 2-11. Data access going deeper into the prototype chain

Although newer browsers with optimizing JavaScript engines perform this task well, older browsers—especially Internet Explorer and Firefox 3.5—incur a performance penalty with each additional step into the prototype chain. Keep in mind that the process of looking up an instance member is still more expensive than accessing data from a literal or a local variable, so adding more overhead to traverse the prototype chain just amplifies this effect.

Nested Members

Since object members may contain other members, it's not uncommon to see patterns such as `window.location.href` in JavaScript code. These nested members cause the JavaScript engine to go through the object member resolution process each time a dot is encountered. Figure 2-12 shows the relationship between object member depth and time to access.

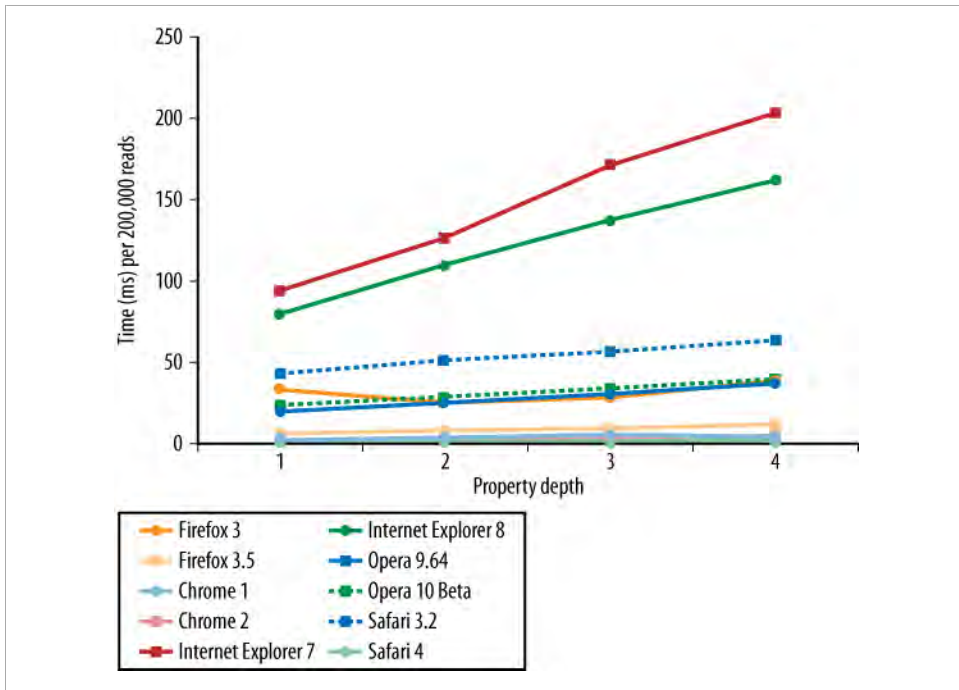


Figure 2-12. Access time related to property depth

It should come as no surprise, then, that the deeper the nested member, the slower the data is accessed. Evaluating `location.href` is always faster than `window.location.href`, which is faster than `window.location.href.toString()`. If these properties aren't on the object instances, then member resolution will take longer as the prototype chain is searched at each point.

In most browsers, there is no discernible difference between accessing an object member using dot notation (`object.name`) versus bracket notation (`object["name"]`). Safari is the only browser in which dot notation is consistently faster, but not by enough to suggest not using bracket notation.

Caching Object Member Values

With all of the performance issues related to object members, it's easy to believe that they should be avoided whenever possible. To be more accurate, you should be careful to use object member only when necessary. For instance, there's no reason to read the value of an object member more than once in a single function:

```
function hasEitherClass(element, className1, className2){
    return element.className == className1 || element.className == className2;
}
```

In this code, `element.className` is accessed twice. Clearly this value isn't going to change during the course of the function, yet there are still two object member lookups performed. You can eliminate one property lookup by storing the value in a local variable and using that instead:

```
function hasEitherClass(element, className1, className2){
    var currentClassName = element.className;
    return currentClassName == className1 || currentClassName == className2;
}
```

This rewritten version of the function limits the number of member lookups to one. Since both member lookups were reading the property's value, it makes sense to read the value once and store it in a local variable. That local variable then is much faster to access.

Generally speaking, if you're going to read an object property more than one time in a function, it's best to store that property value in a local variable. The local variable can then be used in place of the property to avoid the performance overhead of another property lookup. This is especially important when dealing with nested object members that have a more dramatic effect on execution speed.

JavaScript namespacing, such as the technique used in YUI, is a source of frequently accessed nested properties. For example:

```
function toggle(element){
    if (YAHOO.util.Dom.hasClass(element, "selected")){
        YAHOO.util.Dom.removeClass(element, "selected");
        return false;
    } else {
        YAHOO.util.Dom.addClass(element, "selected");
        return true;
    }
}
```

This code repeats `YAHOO.util.Dom` three times to access three different methods. For each method there are three member lookups, for a total of nine, making this code quite inefficient. A better approach is to store `YAHOO.util.Dom` in a local variable and then access that local variable:

```
function toggle(element){
    var Dom = YAHOO.util.Dom;
    if (Dom.hasClass(element, "selected")){
        Dom.removeClass(element, "selected");
        return false;
    } else {
        Dom.addClass(element, "selected");
        return true;
    }
}
```

The total number of member lookups in this code has been reduced from nine to five. You should never look up an object member more than once within a single function, unless the value may have changed.

One word of caution: it is not recommended to use this technique for object methods. Many object methods use `this` to determine the context in which they are being called, and storing a method in a local variable causes `this` to be bound to `window`. Changing the value of `this` leads to programmatic errors, as the JavaScript engine won't be able to resolve the appropriate object members it may depend on.

Summary

Where you store and access data in JavaScript can have a measurable impact on the overall performance of your code. There are four places to access data from: literal values, variables, array items, and object members. These locations all have different performance considerations.

- Literal values and local variables can be accessed very quickly, whereas array items and object members take longer.
- Local variables are faster to access than out-of-scope variables because they exist in the first variable object of the scope chain. The further into the scope chain a variable is, the longer it takes to access. Global variables are always the slowest to access because they are always last in the scope chain.
- Avoid the `with` statement because it augments the execution context scope chain. Also, be careful with the `catch` clause of a `try-catch` statement because it has the same effect.
- Nested object members incur significant performance impact and should be minimized.
- The deeper into the prototype chain that a property or method exists, the slower it is to access.
- Generally speaking, you can improve the performance of JavaScript code by storing frequently used object members, array items, and out-of-scope variables in local variables. You can then access the local variables faster than the originals.

By using these strategies, you can greatly improve the perceived performance of a web application that requires a large amount of JavaScript code.