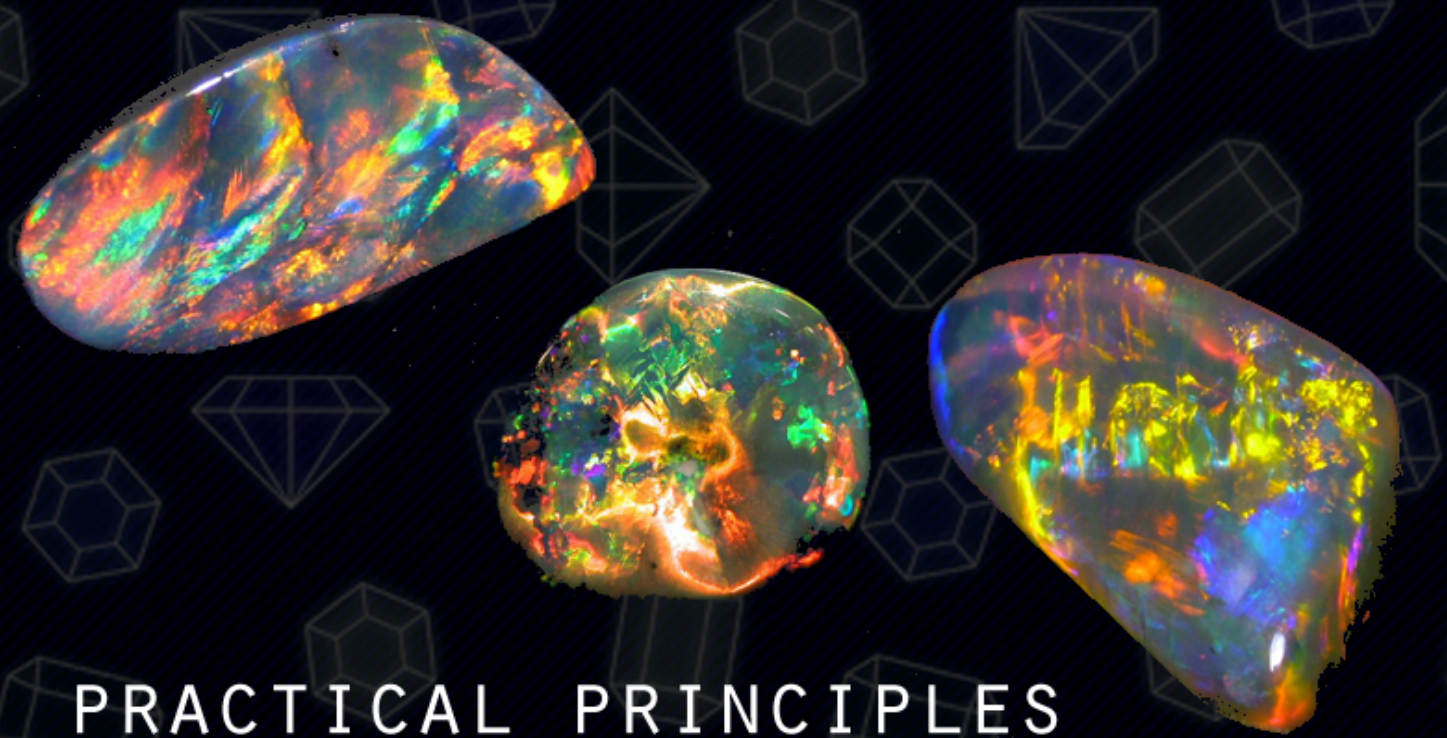
jQuery GEMS

A jQuery Tutorial Book



PRACTICAL PRINCIPLES

Designed with the intent to improve your understanding of one of the most popular JavaScript development libraries.

Written and Edited by Greg Sidelnikov greg.sidelnikov@gmail.com

jQuery **GEMS**

A jQuery Tutorial Book

All programming is connecting the dots. Therefore writing computer code is an art.

jQuery GEMS is meant to be read using the right side of your brain. In other words, ***intuitively***. For this reason, you will not find lists of functions here or dumps of unexplained "*it just works*" source code. Instead you are encouraged to exercise your creative mind.

This is a little different from the traditional approach to learning programming languages. There are already plenty of books written that way. The author of *this* book invites you to challenge your understanding of JavaScript.

Written and edited by Greg Sidelnikov

greg.sidelnikov@gmail.com

Revision 1 September 23th 2012
Revision 2 October 31st 2012
Revision 3 November 9th 2012 *First Release*

Table of Contents

jQuery Gems is a ***collection of practical jQuery tips***. Each page represents one *key* understanding about writing jQuery and JavaScript code. Each gem focuses on very specific knowledge or a *way* of doing something. But ***additional explanations*** essential to understanding each gem will not be skipped either. Some gems are longer than others.

Going through all of the tips included in this tutorial book is meant to widen and test your understanding of jQuery using simple examples.

If you are the owner of my first book: Understanding jQuery, which was a lengthy course in itself (covering jQuery *and* JavaScript,) then you should know that this volume takes an alternative approach to computer language education. Together, both books are designed to develop a deeper understanding of JavaScript and influence you to be more effective and efficient at writing jQuery code to ***solve real web-development problems***.

While this book assumes some basic JavaScript knowledge, remember that jQuery is nothing more than a library *written* in JavaScript. Essentially it is JavaScript, but it's called jQuery. They are one and the same.

This book is composed of various examples I have actually come across while writing code and learning *myself*: When I faced a problem and realized I needed a specific solution, I would require something concrete to fill in the blanks in my own knowledge. Once I solved a problem, I would try to understand what I was missing in order to come to the resolution. Therefore, a gem was born.

This book is a collection of such gems. They are simple and practical tips designed to expand and test your knowledge.

This book can be used as a reference for beginners to tighten understanding of jQuery quirks or read over a weekend by a professional developer to test the integrity of his or her web programming knowledge.

jQuery Quick Start

jQuery is a library that simplifies JavaScript programming.

Selecting Elements

```
$("body");                    // By tag name
$("#id");                     // By element id
$(".className");              // By element class name
$("div:nth-child(2)");        // Using a CSS pseudo-selector :nth-child
```

Executing jQuery methods on selected elements

```
$("table tr:nth-child(2)").hide();      // Hide 2nd table row
$("ul#list").sortable();                // Turn <ul id = "list"> into sortable list
```

Attaching events

```
$("#button").on("click",
   function(event) {
        // Do something with $(this) or this keyword
   });
```

jQuery css() and animate() methods

```
$("div#t").css("background-color", "white");      // Set background
$("div#t").animate( { left: "+=100px" }, 1500 );  // Move +=100 pixels from cur-
                                                  // rent position in 1.5 seconds.
```

Extending jQuery with your own methods

```
$.fn.extend( { red: function() {                  // Create $.fn.red();
                    $(this).css("color", "red");
               }
          });

// You can now use your method:
$("div").red();    // set font color to all divs red
```

Gem 1 Selecting HTML Elements.

The syntax for using the main jQuery object to select HTML elements is

```
$(selector);     // this will create a new jQuery object in memory.
```

Notice that the **selector** parameter accepts arguments as *objects*. Almost everything in JavaScript is an object. The **selector** parameter is ***not limited*** to CSS strings like "#id" or ".class" and can be one of the following:

1. A string specifying a CSS rule, such as "#id", ".class" or "div#id", etc.
2. A raw JavaScript object, like document.body
3. The object returned from document.getElementById("someid")
4. A jQuery object *itself* like $("div") as in: **var x** = $("div"); $(**x**);

Here are a few common examples:

```
$(".apple");              // Select all elements of class "apple"
$("#cancel");             // Select any element whose id is "cancel"
$("div#apple");           // Select div whose id is apple
$("table tr td")          // Select all tds (table tr is only a path)
$("ul li:nth-child(2)")   // Select only the 2nd LI in a UL list
$("div,img")              // Select all div and all img tags in the document
$("div").add("img")       // Same as above using method add()
```

The ways in which you can select elements are limited only by your *knowledge* of available CSS selectors. *jQuery follows the **standard CSS rules** you would use anywhere else - for example inside an HTML tag's style attribute.* Same rules apply:

style.css

```
div,img { color: blue; } // Equivalent of jQuery's $("div,img").css("color", "blue");
```

There is over a hundred ways to select elements using CSS commands. Most of them are the equivalents to standard CSS styles you are already familiar with. They are the same CSS stylesheet rules you've been using to design websites.

You can spend an entire week learning new CSS selectors. We will explore even

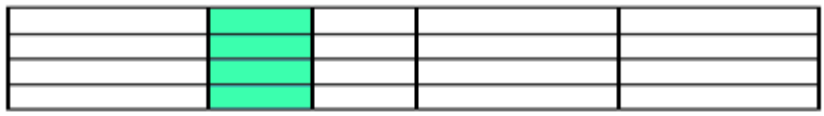
more of them in further sections of this book as we move forward. In fact, instead of studying lists of selectors, we will learn them spontaneously as you continue going through this book. It's more natural and doesn't overwhelm the reader.

After all when we learn new information, we don't memorize lists. We memorize associations. Visual diagrams throughout this book are designed to help you create those associations.

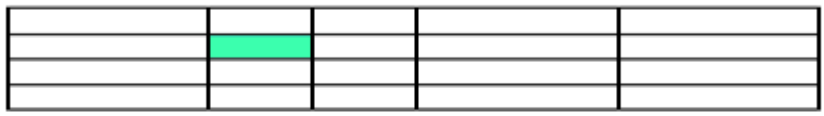
Aside from the dot (.) and number symbol (#) selectors,
we can use the ***:nth-child*** and ***:last*** selectors to navigate basic parent/child DOM in the manner explained in the following diagram.

Selectors that begin with a colon such as ***:first*** and ***:last*** are called *pseudo-selectors*.

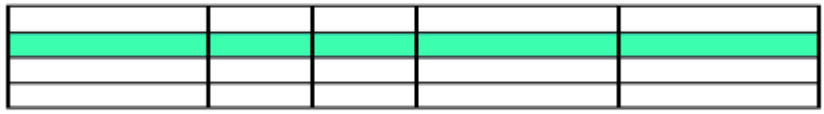
```
$("table tr td:nth-child(2)")
```



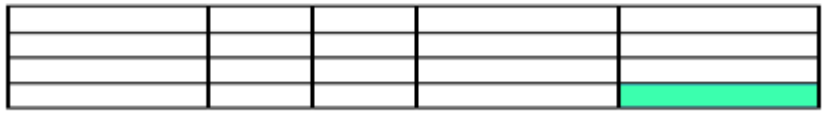
```
$("table tr:nth-child(2) td:nth-child(2)")
```



```
$("table tr:nth-child(2)")
```



```
$("table tr:last td:last")
```



Same ***nth-child*** rules apply to all other nested groups of elements, like ul and li, and even any arbitrary parent/child combinations.

Gem 2 Selecting all elements and using context.

The selector * selects absolutely all elements. The tags contained in the result will include the META tags and even the DOCTYPE if it was specified on the page. The * selector is not limited to elements within the BODY tag alone. It searches within the entire document. The DOM.

To find all HTML elements limited *only* to the body tag you will need to pass **document.body** object to the context parameter, so that elements like <head> and <script> (outside of <body>) are left out:

```
$("*", document.body);
```

In this case **document.body** is the context. This means * will be limited to searching within document.body only.

You can use any other JavaScript object as the context whether it's a CSS selector string or a raw DOM object. The context parameter can be ***any relevant JavaScript object*** related to an HTML element.

```
$("*", "body");    or    $("*", "div");
```

```
var context = $("body");
$("*", context);
```

or

```
$("*", document.getElementsByTagName("body"));
```

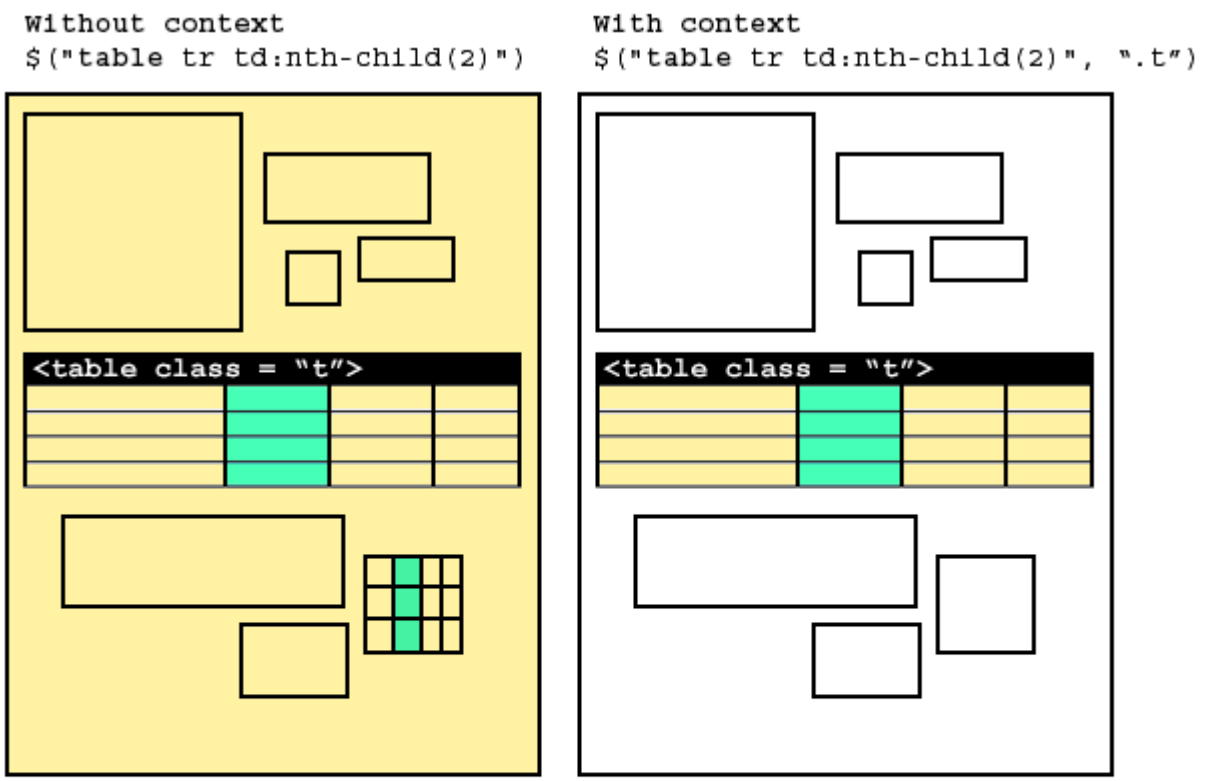
All of the above are legal examples of using jQuery context arguments.

Gem 2 (cont.) Why use context?

Context is a feature that has origin in ***visual thinking***. It has to do with the number of HTML elements being selected. And trying to cut them down to a minimum.

The developers of jQuery visualized the problem before they solved it. But we are only provided an argument called ***context*** and instructions on how to use it. In other words, we literally cannot see or visualize the problem that ***context*** was invented to solve. Yet, visualizing the problem is incredibly important if you wish to write high-performance JavaScript code.

The context parameter allows us to laser-target the area which will be scanned for desired elements. But context is not about ***focusing on one*** element. It's the oppsite - it is about ***cutting down the time it takes to search within the unneeded*** elements. The more we are able to ignore (see yellow areas below) the more performance we can get out of our script.



The entire DOM is **searched**, indicated in yellow. Without context the 2nd column in all existing tables is selected.

Using context the search is limited to the specified element. This saves memory and increases performance. Only the elements within context are **searched**. (In yellow.)

The point of context is not in being limited to **less elements**, but in spending **less time** to traverse the DOM and limiting the search time, because by default every single element is searched. Repeatedly calling a CSS selector without context (as in from within a repetitive function) can slow down your code.

Gem 3 Extending the jQuery object with your own functions.

You can name and create your own jQuery functions. You can add them to the main jQuery object using the ***extend*** method.

```
$.extend({
        paint_blue: function(obj)
        {
            $(obj).css("color", "blue"); // do something with the
                                         // selected object(s)
        }
    });
```

Then, you can use that function on *regular* jQuery selections:

```
$.paint_blue("div");    // "div" is passed as the obj parameter
```

Here is another simple example. Let's say we need a custom function that hides elements *our* way. Whichever way that will be is determined by the highlighted code contained within the ***custom anonymous function***:

```
// Create my own method "hide_my_way"
$.extend({ hide_my_way: function(obj) { $(obj).fadeOut(); } })

// Use the method I created on an arbitrary selection
$.hide_my_way("p");
```

The selected HTML elements are passed through ***obj*** parameter.

Here we simply use jQuery's ***fadeOut***() method. This makes our own function $.***hide_my_way***("div") synonymous with $("div").***fadeOut***(); Of course, we could have written any custom code within ***hide_my_way***, which is highlighted in yellow. The only limit is our imagination.

Gem 4 Using $.fn.extend

Gem 3 explained how to add functions to the main jQuery object using $.***extend***. These functions were methods belonging to the main jQuery object as in $.***my_method***(obj), where obj *could* be a selector.

But what if we wanted to create a selector-based method that executed a command directly on a jQuery CSS selector, as in $("div").***my_method***?

In order to accomplish this we use the $.***fn.extend*** method.

```
$.fn.extend({
        red: function()
        {
            $(this).css("color", "red");
        }
    });
```

Unlike $.extend, $fn.extend ***does not require*** a paremeter (like *obj* in the previous example) passed to the function. The ***this*** object will refer to the "p" argument itself:

```
$("p").red(); // Here, the this object is synonymous with the objects selected by the "p" selector (In this case, all HTML paragraph tags).
```

Oh, and one more thing:

$.***fn*** is exactly the same as $.***prototype***

The ***fn*** object was invented by jQuery developers to refer to ***prototype***. It's just an alias. But the name *prototype* itself is a reserved JavaScript keyword, it's part of JavaScript specification.

You have just discovered one way of using JavaScript prototype. It's used to extend functionality of JavaScript objects with our own methods.

This is one of the many numerous examples of how jQuery simplifies existing JavaScript syntax to help us do things in an easier and more intuitive way.

Gem 5 The distinction between the jQuery object and a JavaScript object.

We have mentioned the raw JavaScript object. The book also refers to it as the DOM object, or plain JavaScript object. We also make a clear distinction between such an object and a jQuery object. While both refer to the same object in memory it's important to understand the differences.

There is an important distinction between jQuery and JavaScript objects. Yet, they refer to one and the same thing. If you started to study jQuery *before* Javascript you might find this contradiction confusing.

First of all, let's make this clear. A ***plain JavaScript object*** is just a custom object. It's up to the programmer to determine its function:

```
var my_obj = { /* object core */ };
var my_func = function() { /* an object of type Function */ };
var my_cobj = new Object(); // a new arbitrary object
```

All these are examples of plain vanilla JavaScript objects of different kinds: represented as an ***object literal***, a ***function*** and as ***Object*** type.

Keep in mind, everything in JavaScript is an object. ***jQuery*** and ***$*** refer to the same object in memory, as defined by jQuery. They are equal:

```
var jQuery = $ = function() { /* jQuery library code */ }
```

Same thing here. jQuery is just an object of type Function.

jQuery, or the $ symbol is a *variable name* that refers to a JavaScript object ***defined as a JavaScript object of type Function***.

jQuery is a *custom* JavaScript object but what is a JavaScript object? From a programmer's point of view an object in any language (including

JavaScript) is a concept that allows us to use ***properties*** and ***methods***.

```
var obj = { myproperty: "Hello", mymethod: function(){ /* custom */ } };

var property = obj.myproperty;// grab a property, eg: string, integer, etc.
var method = obj.mymethod;    // grab a method (function, object, etc.)
method();                     // call the method
```

Do not treat the jQuery object as something different from a custom JavaScript object. They are based on one and the same principle. You can create your own object and name it ***myQuery***. It's possible to write your own JavaScript library using a custom JavaScript object.

The jQuery object is nothing more than that, it's just a JavaScript object created by the jQuery team. This jQuery object contains ***properties*** and ***methods*** that make already-existing functionality of JavaScript easier to use. It's also synonymous with $.

If there is a distinction between the ***jQuery*** object and an object like:

```
var obj = { };  // a custom object anyone can create
```

It is that *obj is an empty object*. We didn't add any code to it. But jQuery does. In a way you can think that both ***obj*** and the ***jQuery*** object ($) are made out of the same matter - the JavaScript object.

The jQuery object is just a JavaScript object. It's just not called obj. In order to really understand jQuery in depth we must understand what type of ***object functionality*** is allowed in JavaScript.

jQuery is a bunch of properties and methods inside a JavaScript object.

The jQuery object does not extend native functionality of JavaScript. It merely *makes* ***already-existing*** *JavaScript functionality easier to use.*

It does so ***by offering custom properties and methods***.

But there is one more thing. What is this JavaScript functionality? What is at the core of the function of JavaScript itself as defined by the specification of the JavaScript language?

It is JavaScript's ***script-based*** nature itself. It's the chainability. And the fact that a method can have a *method* or a *property*. And methods within those methods can also have their own methods and properties.

It's this concept of methods that can "extend" themselves into new methods that can extend into other methods, etc. is at the core of what JavaScript is, and therefore what jQuery is. As a simplified explanation:

```
// Define methods
var jQuery = $ = { method: { inner_method: { further: { /*etc*/ } } } };

// Call methods
$.method(...); // call a method
$.method.inner_method(...); //call a method within another method
```

If you understand this, you have already taken a significant step toward truly understanding the rest of JavaScript and therefore the jQuery library.

Gem 6 Using get(n) and [n] to return arrays of selected objects.

$("div").***get***(1) *is the same as* $("div")[1];

Remember that the jQuery object can return more than one element. $("div") command returns an ***array*** of jQuery objects. Even if only one DIV was returned - it is treated as a one-entry array.

You may already know what a JavaScript array is. Let's take a look:

```
var arr = new Array(1, 2, 3);    // A JavaScript array
```

The jQuery selector like $("div") returns a jQuery object, or a set of jQuery objects. Not a JavaScript array of raw DOM objects related to those elements. The following example where we use the ***[] operator*** will "dereference" the original jQuery obejct and turn it into an array:

```
$("div")[1]; // return the second DIV as a plain JavaScript DOM object.
```

We know that JavaScript array indexing starts with 0. Therefore, this example selects the ***second DIV element*** in the array of all returned elements. That is, if the page contains more than one DIV in the first place.

Both get() and [] return the raw JavaScript DOM object, ***not a jQuery object***. Therefore we cannot run jQuery operations on returned values.

Notice that get(1) and [1] return the second element (the second DIV, ***not first***) in the selection. This is because counting starts with 0. [0] and get(0) would return the very first DIV.

Because they do not return a jQuery object, you couldn't do something like this:

```
$("div").get(1).css("color", "red");    // wrong, css() does not exist here.
$("div")[1].css("color", "red");        // also wrong
```

The ***css***() method belongs to the jQuery object, but because ***get***(1) returns a plain vanilla JavaScript object, we cannot call jQuery functions on the result. Same goes for the [*n*] operator.

We know that (*almost*) everything in JavaScript is an object. And the jQuery object itself is aware of this principle. So, what would be a solution to the problem above?

We could pass the object returned from ***get***(1) as a selector to the jQuery object itself. Let's take a look:

```
var obj = $("div").get(1);    // this will return a regular JavaScript object
$(obj).css("color", "red");   // "wrap" obj in jQuery object to extend it
```

Now we can call ***css***() on the object returned from get(1). The same goes for the [] operator. Here is the equivalent example using [1]:

```
$( $("div")[1] ).css("color", "red"); // This works just fine
```

From the performance point of view, passing the jQuery object to a jQuery object as a selector is probably ***unwise***. But it's not impossible or illegal.

It's as if we "wrapped" jQuery around a plain JavaScript object. Doing this enabled us to call jQuery methods such as ***parent***() and all others on a regular JavaScript object. This is one of the ways jQuery extends existing functionality of JavaScript. By extending the functionality of objects.

Consider, once again, the following selector:

```
$("div");
```

What actually happens is that this statement will return a list of ***jQuery objects***. In other words a list of all ***jQuery*** objects referring to all existing DIV elements on the web page.

There are a few more minor distinctions between [] and get():

```
$("div")[]        // We can't use [] to return the entire list of DOM objects
$("div").get()    // However, we can do that using get() without parameters
```

Gem 7 Using get(), arrays, each() and reverse().

The ***.get***() method gives us access to the list of all matched elements as a JavaScript array containing plain JavaScript DOM objects:

```
var arr = $("div").get();          // grab elements as a JavaScript array
arr.reverse();                     // reverse() is a native JavaScript array
                                   // method
```

The example above reverses the order of the objects in the list which was returned by the $("div") selector.

An interesting way of doing things is to combine ***get*** with ***each*** to reverse the order of items returned by jQuery. Let's say we have this HTML:

```
<body>
   <p>1</p>
   <p>2</p>
   <p>3</p>
</body>
```

```
var arr = $("p").get().reverse();
$(arr).each(function(){ console.log( $(this).text() ); })
```

The console will output the following. Notice the reversed order:

```
3
2
1
```

Sometimes we need to go through each selected element in reverse. There are cases when this technique is invaluable.

Gem 8 Using gt() - "greater than."

Reduce the selection to start from the second selected element. This is a zero-based index, so 1 actually refers to element 2.

```
$("div").gt(1).css("color", "red");
```

HTML:

```
<div>1</div>
<div>2</div>
<div>3</div>
```

Result on the web page:

1
2
3

As you can see the ***gt*** method allows us to "trim" the selection from the top. Sometimes you want to skip the first few elements.

Gem 9 Determine the number of selected elements.

---

var n = $("div").length;

The variable n will contain the number of elements that were selected.

To check if any elements were selected at all, use the following code:

```
if ($("div").length > 0)
{
   // At least one element was selected
}
else
{
   // None were selected
}
```

Gem 10 Difference between $(this) and this inside a jQuery function.

Basically, there is no difference between ***$(this)*** and ***this***. They both refer to the same object and can be used interchangeably. But they do so in a slightly different way.

The ***this*** keyword is a way for an object to refer to itself from within itself. For example, it can be used within a jQuery method's callback function.

But wrapping ***this*** with the jQuery object as in ***$(this)*** allows us to refer to the this object ***and*** extend its functionality. In other words to enable jQuery methods on it.

Remember that ***this*** is often directly associated with an HTML element that was selected using jQuery's CSS selector.

But something interesting is also of importance here. When we extend the ***this*** object by wrapping it inside the jQuery object (or more correctly, passing it to the jQuery object) it loses its alias to the original JavaScript methods that natively exist in all JavaScript objects:

```
$("p").each(function()
{
   this.style.color = "red";        // Ok
   $(this).style.color = "red";          // Error. Cannot do this
   $(this)[0].style.color = "red";       // Ok; refer to Gem 6 to see why.
   $(this).css("color", "red");          // Ok to call jQuery's method .css()
});
```

Gem 11 Using each().

Use ***.each***() to run a function on each selected element.

```
$("p").each(function()
{
   this.style.color = "blue"; // Run on each selected element one by one
});
```

Notice that here we used the ***this*** object to refer to the element. This way we are dealing with the JavaScript DOM object, not the jQuery object. But we can use a jQuery object to accomplish the same effect:

```
$("p").each(function()
{
   $(this).css("color", "blue");
});
```

There isn't a right or wrong way. Think about this as using different techniques to accomplish the same thing. Working with the same objects in different ways.

Just for fun, consider this example:

```
$("p").each(function()
{
   $(this)[0].style.color = "blue";
});
```

Essentially we just converted the jQuery object into a JavaScript object. We use index [0] to work on the only available element within ***.each***().

Gem 12 Passing an object to jQuery as a selector.

You can pass the raw JavaScript element to jQuery.

$(***document.body***).css("background", "black");

Of course the alternative to this is just $("body") but it's not impossible to pass actual DOM objects instead of CSS selector strings.

This may not make much sense to you in an isolated example. But circumstances arise when we already have the document.body object stored in a variable. Using $("body") means to run complex selector code that ends up finding document.body anyway.

Passing the raw JavaScript object directly to the selector may increase performance of your application, because jQuery doesn't have to seek for it using a selector, which can usually tax performance.

Gem 13 Passing HTML as a selector.

---

jQuery has a way to draw a distinction between the type of arguments being passed to it. It can identify HTML statements. When this happens, the argument is no longer treated as a selector. In that case it will be treated as the value. Kind of in reverse order:

```
$("<p>hello</p>").appendTo("body");
```

Notice that hello is not a selector, "body" is. It's like we reversed the logic of the statement and mutually replaced the selector with the value being appended. The logic of the statement is changed when HTML is passed as a selector. The statement above is exactly the same as:

```
$("body").append("<p>hello</p>");
```

Notice, however, that we cannot append ***text*** using ***appendTo***() method, only ***HTML*** statements:

```
$("<p>hello</p>").appendTo("body");   // Append to bottom of "body"
$("hello").appendTo("body");          // Nothing happens, hello is not
                                      // an HTML statement, and there
                                      // is no HTML tag called <hello>
```

Unless you have a custom tag called <hello>, the second statement will produce no results whatsoever. The first example, however, was able to successfully identify ***valid HTML*** in the string passed to it. This changed the logic of the statement, flipping the selector and the value around.

Gem 14 Find all radio buttons within an HTML form.

$("input:radio", ***document.forms[0]***);

Or just use the direct alias to the form:

$("input:radio", ***"form#id"***);

Or even this:

$("input:radio", ***$("#id")***);

With regard to the last example, it is not recommended to use jQuery objects themselves as selectors or context. This example merely shows that it isn't impossible. But why is it possible? It's because the $ object is a function that takes parameters. JavaScript function parameters accept objects. This means that we can pretty much pass any JavaScript object to a function as its argument(s).

Even strings in JavaScript are objects. Or more precisely, they turn into objects when such use is demanded from them. This means "form#id" is essentially transformed to an object if a native property such as ***length*** is requested from it using the dot operator. For example:

var len = "form#id".***length***; // len will equal 7

The result is 7 because "form#id" is a string whose length is 7 characters.

The string becomes an object of type String. Objects of type String have a property ***.length***. That's exactly what happened here. A string turned into an object when we used the dot operator on it.

Gem 15 Different ways to refer to the same thing.

$("#***clock***")[0] is exactly the same as ***document.getElementById***("***clock***");

This is true because [0] will convert a jQuery object to a raw JavaScript DOM object. Notice that $("#clock") itself will refer to a jQuery object, not a plain JavaScript object.

```
$("#clock")[0].style.color = "red";                          // Ok
$("#clock").css("color", "red");                             // Ok
document.getElementById("clock").style.color = "red"         // Ok
$(document.getElementById("clock")).css("color", "red");     // Ok
$("#clock").style.color = "red";                          // Error
document.getElementById("clock").css("color", "red");        // Error
```

Why did the last two examples end up in an error? It's simple. A jQuery object cannot call plain JavaScript methods or access its properties. And a plain JavaScript DOM object cannot access jQuery's methods such as the ***.css***() method in the last example

Because jQuery is written in JavaScript, sometimes it's difficult to draw the line between where jQuery begins and where JavaScript ends. This line becomes thinner the more you understand about how JavaScript works.

In fact, once you understand everything about JavaScript, this line will disappear completely. jQuery and JavaScript are one and the same thing.

Gem 16 Difference between variables and objects.

To say that variables are objects is not entirely correct.

A variable is a handle to an object.

A variable can point to an object.

But it can also point to other data types.

Gem 17 What is a variable?

Variables are like tweezers used to pick functions, objects, integers, strings and arrays ***to work with***. A variable is just a name. It is not an object. Variables exist in many languages.

In JavaScript variables do not have a strict type. In other words, the variable is a pointer to an object of a particular type. This type is defined by the value that is being assigned to the variable itself.

But in strictly typed languages such as C++ a variable definition must include its type. For example you can't assign a string to a variable of type Integer. But in JavaScript it doesn't matter. The type just gets overwritten and the variable will hold an object of any type assigned to it.

Not all languages treat ***everything as an object***. But JavaScript is one of the languages that does. There are only a few things that are not objects in JavaScript, but hypothetically speaking 99% of things we refer to with variables are objects.

Gem 18 Prepending an element to itself removes the element.

Trying to prepend an element to itself will just remove it from the DOM:

$("#p2").***prepend***("#p2");

Oddly, and contrary to assumed function, this statement will not make a copy of "#p2" and prepend it to itself. This is because first jQuery removes the element and then it tries to prepend itself to itself after it has already been removed.

While we're on the subject, the proper way to remove an element or a set of elements from the DOM would be:

```
$("#p2").remove();  // Remove the element whose id is "p2"
$("p").remove();    // Remove all paragraphs from the page
```

Gem 19 Passing jQuery object to a jQuery object.

Though *unwise,* it's not impossible to pass the jQuery selector itself as arguments to a method:

```
$($("#p1")).prepend($("#p2"));
```

Though not recommended due to performance issues when working with thousands of elements, it's not impossible or illegal.

It's like multiplying the work that jQuery has to do because every time you call the $ function, jQuery constructs a brand new object.

Gem 20
Passing raw DOM objects as parameters to selectors and methods.

You can pass raw DOM as arguments to both selectors and methods:

```
var a = document.getElementById("p1");
var b = document.getElementById("p2")

$(a).prepend(b);
```

document.getElementById is the native JavaScript way to grab DOM objects related to the actual HTML elements.

Gem 21 jQuery UI documentation.

---

jQuery UI: online demos and documentation.

http://www.jqueryui.com/demos/

In the past I have gone through many documentation-style websites but I found them to be simply lists of functions that left the reader figuring things out on their own. jQuery UI website is different. It's an incredibly well-designed tutorial website.

In particular I found the following jQuery UI's methods useful when developing custom user interfaces:

Resizable
Selectable
Sortable
Slider

Gem 22 jQuery UI's .sortable() method.

With ***jQuery UI*** you can turn a UL/LI list into an interactive drag & drop list by calling .***sortable***() on the UL tag (the parent container tag).

.***sortable***() is not limited to just UL/LI tags. You can call it on any element that contains children. For example you can call it on a DIV which contains paragraph tags inside. Once the sortable method is applied to the parent DIV, all paragraphs inside become drag & droppable.

What is a Sortable List?

A sortable list is a list with a number of items. Applying the sortable functionality to a list allows us to drag and drop items in the list with the mouse.

Let's consider this HTML:

```
<ul id = "list">
   <li>Apples</li>
   <li>Oranges</li>
   <li>Bananas</li>
</ul>
```

And to make this list drag and droppable run this jQuery method on it:

```
$("#list").sortable();
```

Remember that jQuery UI source file must be included in your website in order to enable this method.

Gem 23 Getting new order from .sortable() after rearranging an item.

Once we make a list sortable like we did in Gem 22 we are likely to want to gain control over what happens on ***mouse events*** over such a list.

In order to do that we need to ***attach a custom event listener*** to the list.

Let's say we just drag and dropped an item to a new location on a list and we want to store this new order in a database. How do we get the order in which the items appear on the list after it has been rearranged?

We need to overwrite the sortable's **update** event.

```
$(document).ready(function()
{
   // 1. Enable sortable functionality on a list.
   // 2. Begin tracking update event and do something about it;
  //      the update event fires when an item has finished being re-arranged.
   $( "#list" ).sortable(
   {
      update: function(event, ui)
      {
            var serialized = $('#list').sortable('serialize');
            var newSt = serialized.split('&');
            var newAr = new Array();
            for (i = 0; i < newSt.length; i++)
                  newAr[i] = newSt[i].split('=')[1];
            var neworder = newAr; // This array contains the new order
            // Send this array to your PHP script to update the database
            $.ajax({ url: "updatedatabase.php", data: neworder });
      }
   }); // end $("#list").sortable
}); // end of $(document).ready
```

It's important to notice that code like this often goes inside ready's call-back function: $(**document**).***ready***(function(){ ...***here***... });

Why's that? Because that is the place where we initialize all of our JavaScript code. That's what document/ready function is for. It is the code that is executed when the DOM (the elements that make up your entire page) are finished loading.

Remember that a web page doesn't load in one step. It downloads from the server gradually, which is based on your Internet speed connection. Until this process is completed we cannot assume that all elements inside the DOM have been fully downloaded and ready for access by our script.

But this doesn't mean that document/ready is the only place we could or should ever call ***.sortable***() in.

Could you call .sortable() on a button click? Of course! It's up to you. Maybe your website requires that the ability to drag and drop items on a list should be toggled, or enabled only in certain circumstances. You are the designed of your own program. Try to avoid thinking that there exists only one single way of doing something.

Here are a few more ***sortable*** tips:

1. To temporarily disable sortable on a list, call -- $("#list").***sortable***("disable");
2. To enable the sortable again, call -- $("#list").***sortable***("enable");
3. To destroy the sortable ability completely call -- $("#list").***sortable***("destroy");
4. To serialize a sortable list call -- $("#list").***sortable***("serialize");

Important: And this has given me a lot of headache at first. When you ***serialize*** a sortable list, you will be returned the values of each item's ***id attribute***. Sortable's serialize method doesn't return the name or value of the LI or other HTML elements. It is therefore adviced to store the name/value pairs separated by (let's say # character) within the ID attribute itself:

```
<ul class = "ui-sortable" id = "list">
   <li id = "name1#val1">
   <li id = "name2#val2">
   <li id = "name3#val3">
</ul>
```

Gem 24 Making elements draggable with .draggable().

You can make an HTML element draggable using jQuery UI's method called *draggable*():

HTML:

```
<div id = "box"></div>
```

jQuery:

```
$(document).ready(function()
{
   $("#box").draggable();
});
```

Now you can drag #box around with your mouse on the screen.

Gem 25 Using .serialize() and .serializeArray() on a form element.

$("form#my_form").***serialize***() will return a string of all elements inside the form in the following format:

name=value&***name2***=value2&***name3***=value3...

In addition to ***.serialize***() we can use jQuery's ***.serializeArray***() which will encode a set of form elements as an array using name/value pairs:

```
$("form#my_form").serializeArray();
```

The results will be delivered in the following format:

```
[
   {
      name: "first_name",
      value: "John"
   },
   {
       name: "last_name",
       value: "Smith"
   }
];
```

Where name/value pairs are the values specified per each input element within the form.

Gem 26
Similarities between , and add(). Chainability and .end() method.

$("div, p"); is the same as $("div").***add***("p");

The ***.add***() method adds additional elements to the list. But why would we need a separate function if we can just specify a string like "div, p"?

The answer lies in JavaScript's ability to create "chained" methods. The chain is created using the ***dot operator*** (.) which allows us to access methods of an object. $("div, p"); selects both ***div*** and ***p*** elements into one single list. And so does $("div").add("p");

It's important to talk about one other thing to avoid confusion.

You will naturally wish that the ***.add***() method worked like this:

Example 1:

$("div").css("color", "red").***add***("p").css("color", "blue");

By breaking the statement up using the ***.add***() method it seems that it ***should*** allow us to ***apply other jQuery methods to individual selections separately***.

The order in which the methods are executed in Example 1 *suggests* that text inside ***div***s will turn red, and text inside ***p***s will turn blue. But that's not the case and ***this will not produce that result***.

Instead, all elements: the ***div***s and ***p***s ***will turn blue***. Example 1 is an exact equivalent compared to these different ways of doing the same thing:

Example 2:

```
$("div, p").css("color", "blue");          // Same result as Example 1.
$("div").add("p").css("color", "blue");    // Same result as Example 1.
```

That's because when we chained the ***.add***() method to the first ***.css***() method (in Example 1) the ***original list*** of "div" was retained and ***p***s were added ***to it***.

It appears that the inventors of this method gave more favor to simplicity over order awareness. In fact, the jQuery documentation states that method ***.add***() constructs a new jQuery object ***from the union*** of the previously selected elements and the ones passed into the method.

The ***.add***() method is designed to add up all previously selected elements, no matter the order in which it does so. It just keeps adding ***everything***.

Using chainability ***together*** with CSS ***selectors*** and jQuery ***methods*** is a powerful, yet elegant way of applying changes to HTML elements on your web page.

But if the most obvious and natural way of thinking about this function (Example 1) does not work as our intuitive perception suggests it should, what are other practical uses of ***.add***()?

Well, first of all, hold your horses. This sounds like we are painting a bad picture of the ***.add***() method. That's because we expect so much of it. But this doesn't make jQuery less versatile than we thought.

The key is in ***using jQuery methods in combination with each other***. Yes, alone the ***.add***() method is not as powerful as we'd like it to be.

But jQuery has another method called ***.end***() which clears all previously selected elements from the list of *currently selected elements* within the

chain order of the JavaScript statement:

$("div").css("color", "red").***end***().***add***("p").css("color", "blue");

Now, all ***div***s will be painted red, and all ***p***s will be painted blue.

Gem 27 Using .add(html).

Use $("div").***add***(***html***) to add an HTML fragment to all matched elements. However, this will not have a physical effect on the page right away. We still need to use a method such as .***appendTo***():

$("div").***add***("***Hello***").***appendTo***(document.body);

In combination with ***add***, ***appendTo*** appends HTML to the end of the content within the tag.

A close sibling to ***.add***().***appendTo***() combination is the ***html***() method:

$("div").***html***(***html***);

The actual HTML will be embedded ***within*** (in between) the matched tags, for example: ***here*** in this example.

The ***html*** variable passed as the only argument ***replaces all content within the selected tag(s)***.

Gem 29 Using JavaScript before learning it.

JavaScript is the only language people start using without actually learning it. -Douglas Crockford

This is a sound advice for programmers writing software in any computer language. We are often eager to start experimenting with a language not knowing the important details of its design.

What if you chose to learn 2 or 3 jQuery methods per day? To not just memorize their names, but to actually understand the practical situations in which they are used?

Sometimes learning something new requires discipline. It's true regardless of whether you are learning to write computer code or anything else.

Don't only rely on Google for looking up language documentation. Challenge yourself to improve your understanding of new concepts. Each time you look up a function name and its use, you are wasting valuable time that could be used on actually writing the program.

Once you really understand how something is meant to be used, writing software will become a lot more fun.

Gem 30 JSON.

JSON or ***JavaScript Object Notation*** can be a confusing concept to understand. This is because in different contexts it means different things.

In order to clear the fog of confusion, looking at JSON from a number of different perspectives may help you understand JSON better. This is especially useful if you've never heard of JSON before.

First let's recall what an ***object literal*** in JavaScript is. An object literal is a format that helps us create a custom JavaScript object. Let's take a look:

```
var obj_literal = { a: 1 };    // create an object using "object literal" format.
```

Now we can access the property a like this:

```
var num = obj_literal.a;    // num will equal 1
```

When working with ***object literals*** in JavaScript we have the right to use a valid JavaScript identifier such as ***a*** within the { } brackets as a property name. In addition we can use double quotes, if, for example we want to create a property name that shares a name with a reserved JavaScript keyword such as for, or while. As we know these are reserved keywords for creating for and while loops. We cannot use them as identifier names in an object literal. Unless we use double quotes:

```
var num = { for: 1 };       // error, we cannot use reserved keyword: for, as a name
var num = { "for": 1 };    // but this works ok
```

The first example will fail with a JavaScript error. However, in the second example we are free to refer to new property by ***num.for*** without creating a conflict with reserved keywords. And that's one of the purposes for using the double quotes.

This is an important lesson to know before going further into understanding JSON. In a moment, you will see why. Let's move on.

JSON: *The string format notation*

The acronym JSON itself is used in reference to a ***string format*** for representing objects using the familiar *object literal* format as a string. It's often used to represent objects as strings.

```
var json_string = '{"a": 1, "b": 2}';
```

It's easy to notice that this is actually an object literal stored as a string. And this is important. In this case we must use double quotes. But note, that the double quotes are not used to avoid clashes with ***reserved keywords***. When working with JSON specifically ***we are required to use double quotes*** around the property name regardless of the situation:

```
var json_string = '{a: 1, b: 2}';        // Not entirely correct JSON representation
var json_string = '{"a": 1, "b": 2}';    // A properly formatted JSON string
```

This can be further verified by calling JSON's stringify method:

```
// Convert an object literal to a correctly formatted JSON string
var obj_literal = { a: 1, b: 2 };
var json_result = JSON.stringify( obj_literal );
alert( json_result ); // will output { "a": 1, "b": 2 }; -- notice double quotes
```

JSON.stringify straightened the object literal into an acceptable JSON format, by actually forcing the names to be wrapped in double quotes where originally there were none. That's because it is a requirement. Stick to it when using JSON.

Treat everything else as an object literal. Though, ***just because it is possible*** to form a ***valid object literal*** using double quotes around names, that doesn't mean that this object literal will be used as a JSON. It doesn't have to. But ***when*** it is, we must include the double quotes around names.

JSON and Data-interchange

The most common use of JSON is data-interchange. Objects in different languages are composed in different ways. But they can be stored as a string in JSON format. This way a program written in one language can send an object over the network

as a string to a program written in another language. The receiver program will reconstruct the object into its native object form.

The JSON library usually contains methods for converting an object in memory to a string, and back into an object in the language JSON library is written for. JSON is a native extention to the JavaScript, so we're ready to use it out of the box.

* * *

Though the first two letters of the acronym "JS" refer to JavaScript, JSON is used in many other languages such as Java, PHP and C++. That's because one of the primary uses of JSON is to make it easy to "transfer" objects from one language to another over the network (Internet).

In other words it is used for saving and sending an object from one language into a program written in a different language. In JavaScript this is done by converting strings into objects and objects into strings:

parse:

```
// Convert a JSON string into a JavaScript object
var json_string = '{ "a": 1, "b": 2 }';
var x = JSON.parse( json_string );
alert(typeof(x));    // will display "object", it's no longer a string.
```

stringify:

```
// Convert an object into a JSON string
var obj_literal = { a: 1, b: 2 };
var x = JSON.stringify( obj_literal );
alert( typedef(x) );    // will display "string", we turned an object into a string.
```

In addition, note that JSON is not a natively-existing ***object*** in JavaScript. JSON is not an object like Integer, String or Date. Therefore, the following statement will produce a JavaScript error:

```
var string_object = new String();  // New string object created!
var json_object = new JSON();      // Error: no such thing as native JSON object.
```

Gem 31 .toJSON() on native JavaScript objects.

The ***.toJSON***() method exists natively on the Date, Number, String, and Boolean objects.

Gem 32 Using JSON.parse instead of eval(json_string).

Use JSON.***parse***(json_string) instead of ***eval***(json_string) to obtain the JSON object.

Gem 33 Dangers of eval().

---

eval is a function that evaluates a JavaScript statement read from a string (not from a direct statement) and executes it at runtime.

eval is very fast. However, it can compile and execute ***any*** JavaScript program so there can be security risks. A good word of advice is to avoid using it completely unless the server it's used on can be trusted.

Gem 34 Creating objects on the fly and accessing them.

This is a *JavaScript* Gem. But a gem nonetheless.

var x = { "type": "palm tree" }.***type***;

This statement will result in x = "palm tree."

In JavaScript (and therefore jQuery) we can create an object and start using it "on the fly" within the same statement.

Gem 35 Using ***$.extend()*** method to merge two or more objects into one.

In Gem 3 and 4 we explained how the $.***extend*** method can be used to add our own custom methods to the jQuery library. But there is one other use for this method. The secondary syntax of $.***extend*** using multiple arguments looks like this:

```
$.extend(target [, object1] [, objectN]);
```

Using $.***extend*** in such manner will take properties from all suggested objects and merge them into the target object. We can either do a ***simple*** or a ***deep*** merge. Let's take a look at the differences:

Simple Merge with Different Property Names

```
var obj1 = { banana: {color: "yellow"} };
var obj2 = { apple: {color: "red"} };
var obj3 = $.extend(obj1, obj2);

var result = JSON.stringify(obj3);
```

result=

```
{
   "banana":    {"color":"yellow"},
   "apple":       {"color":"red"}
}
```

But when merging two objects that share a property name, the results don't actually merge as you would expect, but rather one property will get completely overwritten by the other:

Simple Merge with the Same Property Name

A simple merge will take all values from obj1 and merge them with obj2 as seen in the next example. Of course, we can have many more properties than just banana. But the point of this example in particular is to demonstrate a simple merge using the same property names.

In a ***simple merge*** if both objects share a unique property name, the properties of the first object will be completely ***overwritten*** (not added up) with the properties of the second object without regard for properties originally stored in first object under the same name:

```
var obj1 = { banana: {color: "yellow"} };
var obj2 = { banana: {price: "$1"} };
var obj3 = $.extend(obj1, obj2);

var result = JSON.stringify(obj3);
```

result=

```
{
   "banana":    {"price":"$1"}
   // what happened to color yellow? it got overwritten
}
```

What happened to banana ***color***? It was overwritten and replaced by banana ***price***. That's because both objects share the same property name: ***banana***.

When object properties share the same name, one gets overwritten with the other. That's may not be what you're looking for.

And that's why we need a deep merge:

Deep Merge

```
var obj1 = { banana: {color: "yellow"} };
var obj2 = { banana: {price: "$1"} };
var obj3 = $.extend(true, obj1, obj2);

var result = JSON.stringify(obj3);
```

result=

```
{
   "banana":
   {
      "color":   "yellow",
      "price":   "$1"
   }
}
```

Now that looks better. A deep merge will merge sub-properties of objects too and add them up. I would imagine, if performance issues don't trouble you, use deep merge, especially when some objects you are merging are guaranteed to share property names.

But if you are merging two or more objects that are ***guaranteed*** to contain exclusively unique property names it would be wiser to use a simple merge because it's faster.

Gem 36

Using ***.css()*** and ***.attr("style")*** to set or get current style of an element.

```
// Returns "block" if the default value wasn't overwritten by a custom style.
$("p").css("display");
```

The .css() method can set and get the style of an HTML element. The

method .***css***() can set ***multiple*** css properties to an element. As in the following example:

$("div").***css***({ ***display***:"block", ***visibility***: "hidden" });

But .***css***() cannot ***get*** multiple properties of an element. Instead we should use .***attr***() to grab the style ***attribute*** of an element:

$("p").***attr***("style"); // Grab style attribute

However, this only works for styles defined within the style attribute. It will not work if you wish to grab all styles of an element.

In order to grab absolutely all styles, defined externally, as well as ones defined using the inline attribute, we could use this custom function which I found on StackOverflow.com/ website, but added my own comments that explain what the author did:

```
function css(a){
   var sheets = document.styleSheets, o = {};
   for (var i in sheets) {
      var rules = sheets[i].rules || sheets[i].cssRules;
      for (var r in rules) {
         if (a.is(rules[r].selectorText)) {
            // Here the author used the $.extend method discussed
            // in Gem 38, to merge two objects returned from css2json,
            // which is
            o = $.extend(o, css2json(rules[r].style), css2json(a.attr('style')));
         }
      }
   }
   return o;
}
```

You can find more details here:
http://stackoverflow.com/questions/754607/can-jquery-get-all-css-styles-associated-with-an-element

Gem 37 A string literal like "abc" is not always an object.

I know that we often say how everything (well, most things) in JavaScript are objects. But what things are not objects? Or more precisely ***when***?

A string literal "abc" is not an object. It's internally cast to an object when a method of String.prototype is called on it.

```
var a = "abc";              // still just a string
var len = "abc".length;     // now "abc" was cast to an object to get length
```

Gem 38 Objects before the prototype property.

If would be easier to study the prototype property once one becomes familiar with what ***objects*** are. An in-depth explanation of what objects are is beyond the scope of this book. However, if you are looking to learn more about objects, I explain them in Understanding jQuery.

Gem 39 Relationship between prototype property and the global object.

Notice that the prototype property refers to the original and global object composition. Not just the new object. This way the prototype method is not created more than once for each object, it is used in multiple objects.

Gem 40 Object blueprint composition using prototype.

The prototype property can be used to create a blueprint for all future objects derived from the original object.

```
// Define the function object obj and add prototype property shared:
function obj(){};
obj.prototype.shared = 1; // will be shared by all objects derived from obj.

// Now instantiate a new object n using obj "template" we just created.
var n = new obj();
alert(n.shared);       // Access variable "shared" stored in prototype.
```

Notice that when we defined a new object of type obj, we did not have to assign the property ***shared*** to it. It was inherited from the prototype property assigned to the "blueprint" object.

```
var m = new obj();  m.shared = 2;
alert(m.shared);
```

Note: Try to avoid thinking of prototype as merely the property used for creating shared object methods or properties. JavaScript is known as a Prototype-based programming language not without a reason. Prototype is an entire object model. JavaScript objects are associative arrays augmented with ***prototype*** functionality.

Gem 41 Prototype constructor.

```
var f = function(){ };
alert(f.prototype.constructor == f); // will return "true"
```

Gem 42
Only objects of type Function have a predefined prototype property.

(*Almost*) everything in JavaScript is an object. But only objects of type "function" have a predefined prototype property.

```
var obj = new Object(); // not a functional object
obj.prototype.test = function() { alert('Hello?'); }; // Error!

function MyObject() {} // a functional object
MyObject.prototype.test = function() { alert('OK'); } // OK
```

Gem 43 Prototype chain. The purpose of prototype.

Prototype is an important part of JavaScript object inheritance model. It plays a big role in the construction model of the jQuery object. jQuery allows us to extend the library itself using the prototype property. jQuery renames prototype property to ***fn***. This is why sometimes you will come across something that looks like this: jQuery.***fn*** or $.***fn***. In reality jQuery assigns the new name ***fn*** to the prototype property which already exists on the custom jQuery/$ object (as part of JavaScript specification). This means that when you say $.***fn*** you are actually saying $.***prototype***.

If a member (a property or a method) of an object cannot be found in the

object it looks for it in the prototype chain. The chain consists of other objects. The prototype of a given instance can be accessed with the variable with a special keyword __proto__. Every object has one, as there is no difference between classes and instances in javascript.

The advantage of adding a function / variable to the prototype is that it has to be in the memory only once, not for every instance.

It's also useful for inheritance, because the prototype chain can consist of many other objects.

What is the exact purpose of this ".prototype" property?

The interface to standard classes become extensible. For example, you are using the ***String*** class and you would like to add a custom method for all of your string objects. Such method doesn't exist natively on the JavaScript's String class/object. So, you write your own to fit your needs.

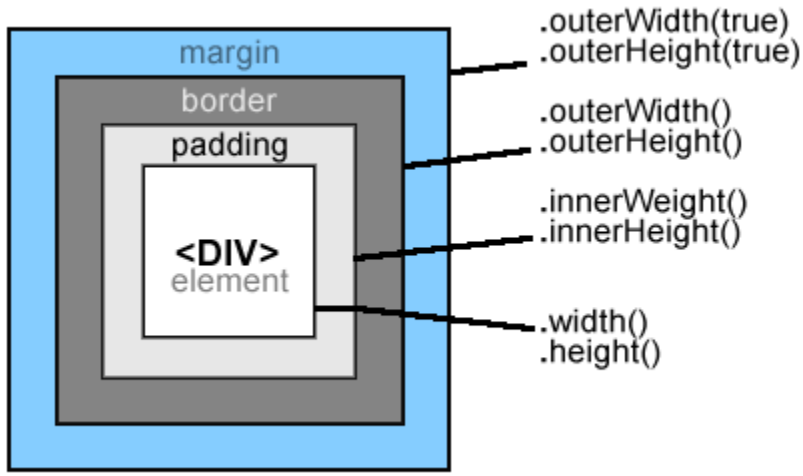
In other languages you would spend time creating a new function, a new subclass, or use object composition. In JavaScript, the prototype property solves this problem by letting the user add his or her own methods available to a class. Even to a native, pre-existing class defined by JavaScript.

This is how the entire jQuery library was created. Most methods are simply prototype extentions to the custom JavaScript object named ***jQuery*** which is the same as the dollar sign, ***$***. You can even add your own methods to the existing jQuery library using fn/prototype interface. That's what creating custom plugins is all about.

Think of prototypes as an extra vtable-pointer. When some members are missing from the original class, the prototype is looked up at runtime.

When working with website or UI layouts, one of the most frequently desired CSS properties are those of HTML element dimensions. Using plain JavaScript functions we can easily obtain simple width and height of an element. But what about ***padding***, ***border*** and ***margin***?

jQuery makes the task much easier. The diagram below demonstrates how the 8 functions from jQuery's CSS API help us get all the right width and height dimensions, including or excluding margin, border and padding:



It's a good idea to start getting in the habit of using these functions. Notice that .outerWidth and .outerHeight functions can be used to get the width including ***padding*** + ***border***, and margin as an option. If you provide these functions with the boolean ***true*** parameter, the margin is included. Otherwise it is not. Use *width*() and *height*() to get the absolute width and height of the element (not counting padding, border or margin.)

Note: These functions will return values without the trailing "px," which is helpful when using width/height as part of a mathematical equation. We no longer need to use *parseInt*, e.g.: ***var*** num = ***parseInt***("10px");

Gem 45 Using the data- attribute with ***.attr()*** method

HTML5 introduced us to data- atrributes. They are custom data attributes that can be assigned to any HTML element.

The ***.attr***("data-var"); method can refer to a custom HTML5 attribute.

For example: <div ***data-var***="xyz">.

The data- attributes may be used when no other attribute is available to store a value. For example, using ***data-title*** on an <a href> element would make little sense because the <a> tag already has a ***title*** attribute.

If you wished to give an element (which doesn't have a default title attribute according to CSS specification) a title it would serve a practical purpose. In this case the custom data-title attribute would serve a new purpose determined by the programmer, not an HTML specification for predefined attributes. But this is like crossing over into the domain of writing custom code. It's where you decide the architecture of your program. Try to avoid overusing data-vars or your code will quickly become cluttered.

Using data- attributes to store numeric values is also incredibly useful. Imagine being able to get x, y, width and height values of a element styled with position: absolute; This can be accomplished by using a jQuery command such as:

```
var width = $("div#id").outerWidth(true);     // get the calculated width
$("div#id").attr("data-w", width);   // store the value in a data- attribute
```

Assuming that the original div#id will not change its dimensions throughout the lifetime of the webpage, we no longer have to recalculate with ***.outerWidth()*** every time we need to grab its width.

Simply access it as a string, using the ***.attr()*** method again, this time without the second parameter:

```
var width  = $("div#id").attr("data-w");   // get the value stored in data-w
```

Gem 46 Using *$.data()*

In the previous gem we have explored the method ***.attr()*** which helped us access and store HTML attribute values.

In a similar way jQuery allows attaching entire data structures to an element. Each data set has a name which is referred to as ***key***. This is the name of the data.

The ***$.data*** method comes in two variations:

```
$.data(element, key, value);          // Set value
$.data(element, key);                 // Get value
```

Data can be attached to elements using CSS selecors or directly via the familiar JavaScript objects such as ***document.body*** related to that element (in this case the <***body***> tag.)

Using these methods let's see how we can attach data or entire datasets to HTML elements or objects related to those elements:

```
// Set a simple variable to document.body
$.data(document.body, "hello", 52);

// Retrieve value, example:
var str1 = $.data(document.body, "hello");
var str2 = $.data(document.body).hello;      // same as above
```

Now, let's take a look at how we can attach an entire object containing multiple properties to an HTML element:

```
var body = $("body");
$.data(body, "dataset", { browser: "Chrome", os: "Snow Leopard" });
```

Notice that we stored data in an object $("body") assigned to a variable of the same name. This way we don't have to create a new object $("body") just because we want to access data associated with this element.

And now let's retrieve the values *browser* and *os*:

```
var browser = $.data(body, "dataset", "browser");   // ok
var browser = $.data(body, "dataset").browser;      // ok
var browser = $.data(body).dataset.browser;         // ok
```

All these are valid ways to access the properties that were set.

Let's now take a look at how to set and access deep-nested objects. The values will be assigned recursively. The rules are quite similar to the previous example:

```
var div = $("div#lion");

$.data(div, "core",
{
   animal:
   {
      name: "lion",
      legs: 4,
      color: "#C19A6B" /* The first recorded use of lion as a color name in English was in 1551. It's HTML code is #C19A6B. */
   }
});

// Now let's retrieve the data:

var name = $.data(div).core.animal.name;     // "lion"
var legs = $.data(div).core.animal.legs;     // 4
var color = $.data(div).core.animal.color;   // "color"
```

Thank You

Thank you for buying a copy of my book - ***jQuery Gems***. As you may know, your purchase includes free life-time book updates.

You will continue receiving book updates ***including new chapters*** automatically. An updated version of this book will be sent to the PayPal e-mail address you used to buy this edition. New updates happen every 1-2 months or upon availability (a quality update takes time to make) as I gather more educational material.

Free updates are already included with your purchase and you don't have to do anything. The same goes for Understanding jQuery my other book about JavaScript programming.

Greg Sidelnikov
greg.sidelnikov@gmail.com