Diseño gráfico orientado a la Web

Modulo: Dise¶o de P§gina Web

Tema 3: Iniciando con jQuery

Ing. Jorge Centeno Borge.
Correo-e: georgejzcb@yahoo.es

Universidad Nacional Autónoma de Nicaragua, León

Colaboran:

Universidad de Alcalá

aecid

TEC
Tecnológico de Costa Rica

TEMA 3: INICIANDO CON JQUERY

CONTENIDO

CAPÍTULO 1: CONCEPTOS BÁSICOS DE JAVASCRIPT

Introducción

Sintaxis Básica

Operadores

Código Condicional

Bucles

Palabras Reservadas

Vectores

Objetos

Funciones

Determinación del Tipo de Variable

La palabra clave this

Alcance

Clausuras

CAPÍTULO 2: CONCEPTOS BÁSICOS DE JQUERY

$(document).ready()

Selección de Elementos

Comprobar Selecciones

Trabajar con selecciones

CSS, Estilos, & Dimensiones

Atributos

Recorrer el DOM

Manipulación de Elementos

BIBLIOGRAFÍA

Murphey, R. **Fundamentos de jQuery.**

0.2. Conceptos Básicos de JavaScript

0.2.1. Introducción

jQuery se encuentra escrito en JavaScript, un lenguaje de programación muy rico y expresivo.

El capítulo está orientado a personas sin experiencia en el lenguaje, abarcando conceptos básicos y problemas frecuentes que pueden presentarse al trabajar con el mismo. Por otro lado, la sección puede ser beneficiosa para quienes utilicen otros lenguajes de programación para entender las peculiaridades de JavaScript.

Si usted esta interesado en aprender el lenguaje más en profundidad, puede leer el libro *JavaScript: The Good Parts* escrito por Douglas Crockford.

0.2.2. Sintaxis Básica

Comprensión de declaraciones, nombres de variables, espacios en blanco, y otras sintaxis básicas de JavaScript.

Declaración simple de variable

```
var foo = 'hola mundo';
```

Los espacios en blanco no tienen valor fuera de las comillas

```
var foo =         'hola mundo';
```

Los paréntesis indican prioridad

```
2 * 3 + 5;    // es igual a 11, la multiplicación ocurre primero
2 * (3 + 5);  // es igual a 16, por los paréntesis, la suma ocurre primero
```

La tabulación mejora la lectura del código, pero no posee ningún significado especial

```
var foo = function() {
  console.log('hola');
};
```

0.2.3. Operadores

Operadores Básicos

Los operadores básicos permiten manipular valores.

Concatenación

```
var foo = 'hola';
var bar = 'mundo';

console.log(foo + ' ' + bar); // la consola de depuración muestra 'hola mundo'
```

Multiplicación y división

```
2 * 3;
2 / 3;
```

Incrementación y decrementación

```
var i = 1;

var j = ++i;  // incrementación previa:  j es igual a 2; i es igual a 2
var k = i++;  // incrementación posterior: k es igual a 2; i es igual a 3
```

Operaciones con Números y Cadenas de Caracteres

En JavaScript, las operaciones con números y cadenas de caracteres (en inglés *strings*) pueden ocasionar resultados no esperados.

Suma vs. concatenación

```
var foo = 1;
var bar = '2';

console.log(foo + bar);  // error: La consola de depuración muestra 12
```

Forzar a una cadena de caracteres actuar como un número

```
var foo = 1;
var bar = '2';

// el constructor 'Number' obliga a la cadena comportarse como un número
console.log(foo + Number(bar));  // la consola de depuración muestra 3
```

El constructor *Number*, cuando es llamado como una función (como se muestra en el ejemplo) obliga a su argumento a comportarse como un número. También es posible utilizar el operador de *suma unaria*, entregando el mismo resultado:

Forzar a una cadena de caracteres actuar como un número (utilizando el operador de suma unaria)

```
console.log(foo + +bar);
```

Operadores Lógicos

Los operadores lógicos permiten evaluar una serie de operandos utilizando operaciones AND y OR.

Operadores lógicos AND y OR

```
var foo = 1;
var bar = 0;
var baz = 2;

foo || bar;   // devuelve 1, el cual es verdadero (true)
bar || foo;   // devuelve 1, el cual es verdadero (true)

foo && bar;   // devuelve 0, el cual es falso (false)
foo && baz;   // devuelve 2, el cual es verdadero (true)
baz && foo;   // devuelve 1, el cual es verdadero (true)
```

El operador `||` (OR lógico) devuelve el valor del primer operando, si éste es verdadero; caso contrario devuelve el segundo operando. Si ambos operandos son falsos devuelve falso (*false*). El operador `&&` (AND lógico) devuelve el valor del primer operando si éste es falso; caso contrario devuelve el segundo operando. Cuando ambos valores son verdaderos devuelve verdadero (*true*), sino devuelve falso.

Puede consultar la sección `Elementos Verdaderos y Falsos` para más detalles sobre que valores se evalúan como `true` y cuales se evalúan como `false`.

> ***Nota***
>
> *Puede que a veces note que algunos desarrolladores utilizan esta lógica en flujos de control en lugar de utilizar la declaración* `if`*. Por ejemplo:*

```
// realizar algo con foo si foo es verdadero
foo && doSomething(foo);

// establecer bar igual a baz si baz es verdadero;
// caso contrario, establecer a bar igual al
// valor de createBar()
var bar = baz || createBar();
```

Este estilo de declaración es muy elegante y conciso; pero puede ser difícil para leer (sobretodo para principiantes). Por eso se explícita, para reconocerlo cuando este leyendo código. Sin embargo su utilización no es recomendable a menos que esté cómodo con el concepto y su comportamiento.

Operadores de Comparación

Los operadores de comparación permiten comprobar si determinados valores son equivalentes o idénticos.

Operadores de Comparación

```
var foo = 1;
var bar = 0;
var baz = '1';
var bim = 2;

foo == bar;   // devuelve falso (false)
foo != bar;   // devuelve verdadero (true)
foo == baz;   // devuelve verdadero (true); tenga cuidado
```

```
foo === baz;             // devuelve falso (false)
foo !== baz;             // devuelve verdadero (true)
foo === parseInt(baz);   // devuelve verdadero (true)

foo > bim;    // devuelve falso (false)
bim > baz;    // devuelve verdadero (true)
foo <= baz;   // devuelve verdadero (true)
```

0.2.4. Código Condicional

A veces se desea ejecutar un bloque de código bajo ciertas condiciones. Las estructuras de control de flujo — a través de la utilización de las declaraciones `if` y `else` permiten hacerlo.

Control del flujo

```
var foo = true;
var bar = false;

if (bar) {
  // este código nunca se ejecutará
  console.log('hola!');
}

if (bar) {
    // este código no se ejecutará
} else {
  if (foo) {
      // este código se ejecutará
  } else {
      // este código se ejecutará si foo y bar son falsos (false)
  }
}
```

> ***Nota***
>
> *En una línea singular, cuando se escribe una declaración* ***`if`****, las llaves no son estrictamente necesarias; sin embargo es recomendable su utilización, ya que hace que el código sea mucho más legible.*

Debe tener en cuenta de no definir funciones con el mismo nombre múltiples veces dentro de declaraciones `if`/`else`, ya que puede obtener resultados no esperados.

Elementos Verdaderos y Falsos

Para controlar el flujo adecuadamente, es importante entender qué tipos de valores son "verdaderos" y cuales "falsos". A veces, algunos valores pueden parecer una cosa pero al final terminan siendo otra.

Valores que devuelven `verdadero (true)`

```
'0';
'any string'; // cualquier cadena
```

```
[];  // un vector vacío
{};  // un objeto vacío
1;   // cualquier número distinto a cero
```

Valores que devuelven `falso (false)`

```
0;
'';  // una cadena vacía
NaN; // la variable JavaScript "not-a-number" (No es un número)
null; // un valor nulo
undefined;  // tenga cuidado -- indefinido (undefined) puede ser redefinido
```

Variables Condicionales Utilizando el Operador Ternario

A veces se desea establecer el valor de una variable dependiendo de cierta condición. Para hacerlo se puede utilizar una declaración `if/else`, sin embargo en muchos casos es más conveniente utilizar el operador ternario. [Definición: El *operador ternario* evalúa una condición; si la condición es verdadera, devuelve cierto valor, caso contrario devuelve un valor diferente.]

El operador ternario

```
// establecer a foo igual a 1 si bar es verdadero;
// caso contrario, establecer a foo igual a 0
var foo = bar ? 1 : 0;
```

El operador ternario puede ser utilizado sin devolver un valor a la variable, sin embargo este uso generalmente es desaprobado.

Declaración Switch

En lugar de utilizar una serie de declaraciones if/else/else if/else, a veces puede ser útil la utilización de la declaración `switch`. [Definición: La declaración `Switch` evalúa el valor de una variable o expresión, y ejecuta diferentes bloques de código dependiendo de ese valor.]

Una declaración Switch

```
switch (foo) {

  case 'bar':
      alert('el valor es bar');
  break;

  case 'baz':
      alert('el valor es baz');
  break;

  default:
      alert('de forma predeterminada se ejecutará este código');
  break;

}
```

Las declaraciones `switch` son poco utilizadas en JavaScript, debido a que el mismo comportamiento es posible obtenerlo creando un objeto, el cual posee más potencial ya que es posible reutilizarlo, usarlo para realizar pruebas, etc. Por ejemplo:

```
var stuffToDo = {
  'bar' : function() {
      alert('el valor es bar');
  },

  'baz' : function() {
      alert('el valor es baz');
  },

  'default' : function() {
      alert('de forma predeterminada se ejecutará este código');
  }
};

if (stuffToDo[foo]) {
  stuffToDo[foo]();
} else {
  stuffToDo['default']();
}
```

Más adelante se abarcará el concepto de objetos.

0.2.5. Bucles

Los bucles (en inglés *loops*) permiten ejecutar un bloque de código un determinado número de veces.

Bucles

```
// muestra en la consola 'intento 0', 'intento 1', ..., 'intento 4'
for (var i=0; i<5; i++) {
  console.log('intento ' + i);
}
```

*Note que en el ejemplo se utiliza la palabra var antes de la variable `i`, esto hace que dicha variable quede dentro del "alcance" (en inglés *scope*) del bucle. Más adelante en este capítulo se examinará en profundidad el concepto de alcance.*

Bucles Utilizando For

Un bucle utilizando `for` se compone de cuatro estados y posee la siguiente estructura:

```
for ([expresiónInicial]; [condición]; [incrementoDeLaExpresión])
 [cuerpo]
```

El estado *expresiónInicial* es ejecutado una sola vez, antes que el bucle comience. éste otorga la oportunidad de preparar o declarar variables.

El estado *condición* es ejecutado antes de cada repetición, y retorna un valor que decide si el bucle debe continuar ejecutándose o no. Si el estado condicional evalúa un valor falso el bucle se detiene.

El estado *incrementoDeLaExpresión* es ejecutado al final de cada repetición y otorga la oportunidad de cambiar el estado de importantes variables. Por lo general, este estado implica la incrementación o decrementación de un contador.

El *cuerpo* es el código a ejecutar en cada repetición del bucle.

Un típico bucle utilizando `for`

```
for (var i = 0, limit = 100; i < limit; i++) {
  // Este bloque de código será ejecutado 100 veces
  console.log('Actualmente en ' + i);
  // Nota: el último registro que se mostrará
  // en la consola será "Actualmente en 99"
}
```

Bucles Utilizando While

Un bucle utilizando `while` es similar a una declaración condicional `if`, excepto que el cuerpo va a continuar ejecutándose hasta que la condición a evaluar sea falsa.

```
while ([condición]) [cuerpo]
```

Un típico bucle utilizando `while`

```
var i = 0;
while (i < 100) {
  // Este bloque de código se ejecutará 100 veces
  console.log('Actualmente en ' + i);
  i++; // incrementa la variable i
}
```

Puede notar que en el ejemplo se incrementa el contador dentro del cuerpo del bucle, pero también es posible combinar la condición y la incrementación, como se muestra a continuación:

Bucle utilizando `while` con la combinación de la condición y la incrementación

```
var i = -1;
while (++i < 100) {
  // Este bloque de código se ejecutará 100 veces
  console.log('Actualmente en ' + i);
}
```

Se comienza en `-1` y luego se utiliza la incrementación previa (`++i`).

Bucles Utilizando Do-while

Este bucle es exactamente igual que el bucle utilizando `while` excepto que el cuerpo es ejecutado al menos una vez antes que la condición sea evaluada.

```
do [cuerpo] while ([condición])
```

Un bucle utilizando `do-while`

```
do {

  // Incluso cuando la condición sea falsa
  // el cuerpo del bucle se ejecutará al menos una vez.

  alert('Hola');

} while (false);
```

Este tipo de bucles son bastantes atípicos ya que en pocas ocasiones se necesita un bucle que se ejecute al menos una vez. De cualquier forma debe estar al tanto de ellos.

Break y Continue

Usualmente, el fin de la ejecución de un bucle resultará cuando la condición no siga evaluando un valor verdadero, sin embargo también es posible parar un bucle utilizando la declaración `break` dentro del cuerpo.

Detener un bucle con break

```
for (var i = 0; i < 10; i++) {
  if (something) {
      break;
  }
}
```

También puede suceder que quiera continuar con el bucle sin tener que ejecutar más sentencias del cuerpo del mismo bucle. Esto puede realizarse utilizando la declaración `continue`.

Saltar a la siguiente iteración de un bucle

```
for (var i = 0; i < 10; i++) {

  if (something) {
      continue;
  }

  // La siguiente declaración será ejecutada
  // si la condición 'something' no se cumple
  console.log('Hola');

}
```

0.2.6. Palabras Reservadas

JavaScript posee un número de "palabras reservadas", o palabras que son especiales dentro del mismo lenguaje. Debe utilizar estas palabras cuando las necesite para su uso específico.

- `abstract`
- `boolean`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `class`
- `const`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `double`
- `else`
- `enum`
- `export`
- `extends`
- `final`
- `finally`
- `float`
- `for`
- `function`
- `goto`
- `if`
- `implements`
- `import`
- `in`
- `instanceof`
- `int`
- `interface`
- `long`
- `native`
- `new`
- `package`
- `private`
- `protected`
- `public`
- `return`
- `short`
- `static`
- `super`

- `switch`
- `synchronized`
- `this`
- `throw`
- `throws`
- `transient`
- `try`
- `typeof`
- `var`
- `void`
- `volatile`
- `while`
- `with`

0.2.7. Vectores

Los vectores (en español también llamados *matrices* o *arreglos* y en inglés *arrays*) son listas de valores con índice-cero (en inglés *zero-index*), es decir, que el primer elemento del vector está en el índice 0. Éstos son una forma práctica de almacenar un conjunto de datos relacionados (como cadenas de caracteres), aunque en realidad, un vector puede incluir múltiples tipos de datos, incluso otros vectores.

Un vector simple

```
var myArray = [ 'hola', 'mundo' ];
```

Acceder a los ítems del vector a través de su índice

```
var myArray = [ 'hola', 'mundo', 'foo', 'bar' ];
console.log(myArray[3]);   // muestra en la consola 'bar'
```

Obtener la cantidad de ítems del vector

```
var myArray = [ 'hola', 'mundo' ];
console.log(myArray.length);   // muestra en la consola 2
```

Cambiar el valor de un ítem de un vector

```
var myArray = [ 'hola', 'mundo' ];
myArray[1] = 'changed';
```

Como se muestra en el ejemplo "Cambiar el valor de un ítem de un vector" es posible cambiar el valor de un ítem de un vector, sin embargo, por lo general, no es aconsejable.

Añadir elementos a un vector

```
var myArray = [ 'hola', 'mundo' ];
myArray.push('new');
```

Trabajar con vectores

```
var myArray = [ 'h', 'o', 'l', 'a' ];
var myString = myArray.join('');   // 'hola'
var mySplit = myString.split('');  // [ 'h', 'o', 'l', 'a' ]
```

0.2.8. Objetos

Los objetos son elementos que pueden contener cero o más conjuntos de pares de nombres claves y valores asociados a dicho objeto. Los nombres claves pueden ser cualquier palabra o número válido. El valor puede ser cualquier tipo de valor: un número, una cadena, un vector, una función, incluso otro objeto.

[Definición: Cuando uno de los valores de un objeto es una función, ésta es nombrada como un *método* del objeto.] De lo contrario, se los llama *propiedades*.

Curiosamente, en JavaScript, casi todo es un objeto — vectores, funciones, números, incluso cadenas — y todos poseen propiedades y métodos.

Creación de un "objeto literal"

```
var myObject = {
  sayHello: function() {
    console.log('hola');
  },

  myName: 'Rebecca'
};

myObject.sayHello();   // se llama al método sayHello,
                       // el cual muestra en la consola 'hola'

console.log(myObject.myName);    // se llama a la propiedad myName,
                                 // la cual muestra en la consola 'Rebecca'
```

> ***Nota***
>
> *Notar que cuando se crean objetos literales, el nombre de la propiedad puede ser cualquier identificador JavaScript, una cadena de caracteres (encerrada entre comillas) o un número:*

```
var myObject = {
    validIdentifier: 123,
    'some string': 456,
    99999: 789
};
```

Los objetos literales pueden ser muy útiles para la organización del código, para más información puede leer el artículo (en inglés) Using Objects to Organize Your Code por Rebecca Murphey.

0.2.9. Funciones

Las funciones contienen bloques de código que se ejecutaran repetidamente. A las mismas se le pueden pasar argumentos, y opcionalmente la función puede devolver un valor.

Las funciones pueden ser creadas de varias formas:

Declaración de una función

```
function foo() { /* hacer algo */ }
```

Declaración de una función nombrada

```
var foo = function() { /* hacer algo */ }
```

Es preferible el método de función nombrada debido a algunas profundas razones técnicas. Igualmente, es probable encontrar a los dos métodos cuando se revise código JavaScript.

Utilización de Funciones

Una función simple

```
var greet = function(person, greeting) {
  var text = greeting + ', ' + person;
  console.log(text);
};


greet('Rebecca', 'Hola');  // muestra en la consola 'Hola, Rebecca'
```

Una función que devuelve un valor

```
var greet = function(person, greeting) {
  var text = greeting + ', ' + person;
  return text;
};

console.log(greet('Rebecca','Hola'));   // la función devuelve 'Hola, Rebecca',
                                         // la cual se muestra en la consola
```

Una función que devuelve otra función

```
var greet = function(person, greeting) {
  var text = greeting + ', ' + person;
  return function() { console.log(text); };
};


var greeting = greet('Rebecca', 'Hola');
greeting();  // se muestra en la consola 'Hola, Rebecca'
```

Funciones Anónimas Autoejecutables

Un patrón común en JavaScript son las funciones anónimas autoejecutables. Este patrón consiste en crear una expresión de función e inmediatamente ejecutarla. El mismo es muy útil para casos en que no se desea intervenir espacios de nombres globales, debido a que ninguna variable declarada dentro de la función es visible desde afuera.

Función anónima autoejecutable

```
(function(){
  var foo = 'Hola mundo';
})();


console.log(foo);   // indefinido (undefined)
```

Funciones como Argumentos

En JavaScript, las funciones son "ciudadanos de primera clase" — pueden ser asignadas a variables o pasadas a otras funciones como argumentos. En jQuery, pasar funciones como argumentos es una práctica muy común.

Pasar una función anónima como un argumento

```
var myFn = function(fn) {
  var result = fn();
  console.log(result);
};

myFn(function() { return 'hola mundo'; });   // muestra en la consola 'hola mundo'
```

Pasar una función nombrada como un argumento

```
var myFn = function(fn) {
  var result = fn();
  console.log(result);
};

var myOtherFn = function() {
    return 'hola mundo';
};

myFn(myOtherFn);   // muestra en la consola 'hola mundo'
```

0.2.10. Determinación del Tipo de Variable

JavaScript ofrece una manera de poder comprobar el "tipo" (en inglés *type*) de una variable. Sin embargo, el resultado puede ser confuso — por ejemplo, el tipo de un vector es "object".

Por eso, es una práctica común utilizar el operador `typeof` cuando se trata de determinar el tipo de un valor específico.

Determinar el tipo en diferentes variables

```
var myFunction = function() {
  console.log('hola');
};

var myObject = {
    foo : 'bar'
```

```
};

var myArray = [ 'a', 'b', 'c' ];

var myString = 'hola';

var myNumber = 3;

typeof myFunction;   // devuelve 'function'
typeof myObject;     // devuelve 'object'
typeof myArray;      // devuelve 'object' -- tenga cuidado
typeof myString;     // devuelve 'string'
typeof myNumber;     // devuelve 'number'

typeof null;         // devuelve 'object' -- tenga cuidado


if (myArray.push && myArray.slice && myArray.join) {
    // probablemente sea un vector
    // (este estilo es llamado, en inglés, "duck typing")
}

if (Object.prototype.toString.call(myArray) === '[object Array]') {
    // definitivamente es un vector;
    // esta es considerada la forma más robusta
    // de determinar si un valor es un vector.
}
```

jQuery ofrece métodos para ayudar a determinar el tipo de un determinado valor. Estos métodos serán vistos más adelante.

0.2.11. La palabra clave `this`

En JavaScript, así como en la mayoría de los lenguajes de programación orientados a objetos, `this` es una palabra clave especial que hace referencia al objeto en donde el método está siendo invocado. El valor de `this` es determinado utilizando una serie de simples pasos:

1. Si la función es invocada utilizando Function.call o Function.apply, `this` tendrá el valor del primer argumento pasado al método. Si el argumento es nulo (*null*) o indefinido (*undefined*), `this` hará referencia el objeto global (el objeto `window`);
2. Si la función a invocar es creada utilizando Function.bind, `this` será el primer argumento que es pasado a la función en el momento en que se la crea;
3. Si la función es invocada como un método de un objeto, `this` referenciará a dicho objeto;
4. De lo contrario, si la función es invocada como una función independiente, no unida a algún objeto, `this` referenciará al objeto global.

Una función invocada utilizando Function.call

```
var myObject = {
  sayHello : function() {
```

```
      console.log('Hola, mi nombre es ' + this.myName);
  },

  myName : 'Rebecca'
};

var secondObject = {
  myName : 'Colin'
};

myObject.sayHello();                  // registra 'Hola, mi nombre es Rebecca'
myObject.sayHello.call(secondObject); // registra 'Hola, mi nombre es Colin'
```

Una función creada utilizando Function.bind

```
var myName = 'el objeto global',

  sayHello = function () {
      console.log('Hola, mi nombre es ' + this.myName);
  },

  myObject = {
      myName : 'Rebecca'
  };

var myObjectHello = sayHello.bind(myObject);

sayHello();       // registra 'Hola, mi nombre es el objeto global'
myObjectHello();  // registra 'Hola, mi nombre es Rebecca'
```

Una función vinculada a un objeto

```
var myName = 'el objeto global',

  sayHello = function() {
      console.log('Hola, mi nombre es ' + this.myName);
  },

  myObject = {
      myName : 'Rebecca'
  },

  secondObject = {
      myName : 'Colin'
  };

myObject.sayHello = sayHello;
secondObject.sayHello = sayHello;

sayHello();               // registra 'Hola, mi nombre es el objeto global'
myObject.sayHello();      // registra 'Hola, mi nombre es Rebecca'
secondObject.sayHello();  // registra 'Hola, mi nombre es Colin'
```

Nota

En algunas oportunidades, cuando se invoca una función que se encuentra dentro de un espacio de nombres (en inglés namespace*) amplio, puede ser una tentación guardar la referencia a la función actual en una variable más corta y accesible. Sin embargo, es importante no realizarlo en instancias de métodos, ya que puede llevar a la ejecución de código incorrecto. Por ejemplo:*

```
var myNamespace = {
  myObject: {
    sayHello: function() {
      console.log('Hola, mi nombre es ' + this.myName);
    },

    myName: 'Rebecca'
  }
};

var hello = myNamespace.myObject.sayHello;

hello();  // registra 'Hola, mi nombre es undefined'
```

Para que no ocurran estos errores, es necesario hacer referencia al objeto en donde el método es invocado:

```
var myNamespace = {
  myObject : {
    sayHello : function() {
        console.log('Hola, mi nombre es ' + this.myName);
    },

    myName : 'Rebecca'
  }
};

var obj = myNamespace.myObject;

obj.sayHello();  // registra 'Hola, mi nombre es Rebecca'
```

0.2.12. Alcance

El "alcance" (en inglés *scope*) se refiere a las variables que están disponibles en un bloque de código en un tiempo determinado. La falta de comprensión de este concepto puede llevar a una frustrante experiencia de depuración.

Cuando una variable es declarada dentro de una función utilizando la palabra clave `var`, ésta únicamente esta disponible para el código dentro de la función — todo el código fuera de dicha función no puede acceder a la variable. Por otro lado, las funciones definidas *dentro* de la función *podrán* acceder a la variable declarada.

Las variables que son declaradas dentro de la función sin la palabra clave `var` no quedan dentro del ámbito de la misma función — JavaScript buscará el lugar en donde la variable fue previamente

declarada, y en caso de no haber sido declarada, es definida dentro del alcance global, lo cual puede ocasionar consecuencias inesperadas;

Funciones tienen acceso a variables definidas dentro del mismo alcance

```
var foo = 'hola';

var sayHello = function() {
  console.log(foo);
};

sayHello();         // muestra en la consola 'hola'
console.log(foo);   // también muestra en la consola 'hola'
```

El código de afuera no tiene acceso a la variable definida dentro de la función

```
var sayHello = function() {
  var foo = 'hola';
  console.log(foo);
};

sayHello();         // muestra en la consola 'hola'
console.log(foo);   // no muestra nada en la consola
```

Variables con nombres iguales pero valores diferentes pueden existir en diferentes alcances

```
var foo = 'mundo';

var sayHello = function() {
  var foo = 'hola';
  console.log(foo);
};

sayHello();         // muestra en la consola 'hola'
console.log(foo);   // muestra en la consola 'mundo'
```

Las funciones pueden "ver" los cambios en las variables antes de que la función sea definida

```
var myFunction = function() {
  var foo = 'hola';

  var myFn = function() {
      console.log(foo);
  };

  foo = 'mundo';

  return myFn;
};
```

```
var f = myFunction();
f();  // registra 'mundo' -- error
```

Alcance

```
// una función anónima autoejecutable
(function() {
  var baz = 1;
  var bim = function() { alert(baz); };
  bar = function() { alert(baz); };
})();

console.log(baz);  // La consola no muestra nada, ya que baz
                   // esta definida dentro del alcance de la función anónima

bar();  // bar esta definido fuera de la función anónima
        // ya que fue declarada sin la palabra clave var; además,
        // como fue definida dentro del mismo alcance que baz,
        // se puede consultar el valor de baz a pesar que
        // ésta este definida dentro del alcance de la función anónima

bim();  // bim no esta definida para ser accesible fuera de la función anónima,
        // por lo cual se mostrará un error
```

0.2.13. Clausuras

Las clausuras (en inglés *closures*) son una extensión del concepto de alcance (*scope*) — funciones que tienen acceso a las variables que están disponibles dentro del ámbito en donde se creó la función. Si este concepto es confuso, no debe preocuparse: se entiende mejor a través de ejemplos.

En el ejemplo 2.47 se muestra la forma en que funciones tienen acceso para cambiar el valor de las variables. El mismo comportamiento sucede en funciones creadas dentro de bucles — la función "observa" el cambio en la variable, incluso después de que la función sea definida, resultando que en todos los clicks aparezca una ventana de alerta mostrando el valor 5.

¿Cómo establecer el valor de `i`?

```
/* esto no se comporta como se desea; */
/* cada click mostrará una ventana de alerta con el valor 5 */
for (var i=0; i<5; i++) {
  $('<p>hacer click</p>').appendTo('body').click(function() {
      alert(i);
  });
}
```

Establecer el valor de `i` utilizando una clausura

```
/* solución: "clausurar" el valor de i dentro de createFunction */
var createFunction = function(i) {
  return function() {
```

```
    alert(i);
  };
};

for (var i = 0; i < 5; i++) {
  $('<p>hacer click</p>').appendTo('body').click(createFunction(i));
}
```

Las clausuras también pueden ser utilizadas para resolver problemas con la palabra clave `this`, la cual es única en cada alcance.

Utilizar una clausura para acceder simultáneamente a instancias de objetos internos y externos.

```
var outerObj = {
  myName: 'externo',
  outerFunction: function() {

    // provee una referencia al mismo objeto outerObj
    // para utilizar dentro de innerFunction
    var self = this;

    var innerObj = {
      myName: 'interno',
      innerFunction: function() {
        console.log(self.myName, this.myName); // registra 'externo interno'
      }
    };

    innerObj.innerFunction();

    console.log(this.myName); // registra 'externo'
  }
};

outerObj.outerFunction();
```

Este mecanismo puede ser útil cuando trabaje con funciones de devolución de llamadas (en inglés *callbacks*). Sin embargo, en estos casos, es preferible que utilice Function.bind ya que evitará cualquier sobrecarga asociada con el alcance (*scope*).

0.3. Conceptos Básicos de jQuery

0.3.1. $(document).ready()

No es posible interactuar de forma segura con el contenido de una página hasta que el documento no se encuentre preparado para su manipulación. jQuery permite detectar dicho estado a través de la declaración `$(document).ready()` de forma tal que el bloque se ejecutará sólo una vez que la página este disponible.

El bloque $(document).ready()

```
$(document).ready(function() {
    console.log('el documento está preparado');
});
```

Existe una forma abreviada para `$(document).ready()` la cual podrá encontrar algunas veces; sin embargo, es recomendable no utilizarla en caso que este escribiendo código para gente que no conoce jQuery.

Forma abreviada para $(document).ready()

```
$(function() {
    console.log('el documento está preparado');
});
```

Además es posible pasarle a `$(document).ready()` una función nombrada en lugar de una anónima:

Pasar una función nombrada en lugar de una función anónima

```
function readyFn() {
    // código a ejecutar cuando el documento este listo
}


$(document).ready(readyFn);
```

0.3.2. Selección de Elementos

El concepto más básico de jQuery es el de "seleccionar algunos elementos y realizar acciones con ellos". La biblioteca soporta gran parte de los selectores CSS3 y varios más no estandarizados. En http://api.jquery.com/category/selectors/ se puede encontrar una completa referencia sobre los selectores de la biblioteca.

A continuación se muestran algunas técnicas comunes para la selección de elementos:

Selección de elementos en base a su ID

```
$('#myId'); // notar que los IDs deben ser únicos por página
```

Selección de elementos en base al nombre de clase

```
$('div.myClass'); // si se especifica el tipo de elemento,
                  // se mejora el rendimiento de la selección
```

Selección de elementos por su atributo

```
$('input[name=first_name]'); // tenga cuidado, que puede ser muy lento
```

Selección de elementos en forma de selector CSS

```
$('#contents ul.people li');
```

Pseudo-selectores

```
$('a.external:first');  // selecciona el primer elemento <a>
                        // con la clase 'external'
$('tr:odd');            // selecciona todos los elementos <tr>
                        // impares de una tabla
$('#myForm :input');    // selecciona todos los elementos del tipo input
                        // dentro del formulario #myForm
$('div:visible');       // selecciona todos los divs visibles
$('div:gt(2)');         // selecciona todos los divs excepto los tres primeros
$('div:animated');      // selecciona todos los divs actualmente animados
```

> ***Nota***
>
> *Cuando se utilizan los pseudo-selectores* `:visible` *y* `:hidden`*, jQuery comprueba la visibilidad actual del elemento pero no si éste posee asignados los estilos CSS* `visibility` *o* `display` *— en otras palabras, verifica si* el alto y ancho físico del elemento *es mayor a cero. Sin embargo, esta comprobación no funciona con los elementos* `<tr>`*; en este caso, jQuery comprueba si se está aplicando el estilo* `display` *y va a considerar al elemento como oculto si posee asignado el valor* `none`*. Además, los elementos que aún no fueron añadidos al DOM serán tratados como ocultos, incluso si tienen aplicados estilos indicando que deben ser visibles (En la sección Manipulación de este manual, se explica como crear y añadir elementos al DOM).*

Como referencia, este es el fragmento de código que utiliza jQuery para determinar cuando un elemento es visible o no. Se incorporaron los comentarios para que quede más claro su entendimiento:

```
jQuery.expr.filters.hidden = function( elem ) {
    var width = elem.offsetWidth, height = elem.offsetHeight,
        skip = elem.nodeName.toLowerCase() === "tr";

    // ¿el elemento posee alto 0, ancho 0 y no es un <tr>?
    return width === 0 && height === 0 && !skip ?

        // entonces debe estar oculto (hidden)
        true :

        // pero si posee ancho y alto
        // y no es un <tr>
        width > 0 && height > 0 && !skip ?

            // entonces debe estar visible
            false :

            // si nos encontramos aquí, es porque el elemento posee ancho
            // y alto, pero además es un <tr>,
            // entonces se verifica el valor del estilo display
            // aplicado a través de CSS
            // para decidir si está oculto o no
            jQuery.curCSS(elem, "display") === "none";
};
```

```
jQuery.expr.filters.visible = function( elem ) {
    return !jQuery.expr.filters.hidden( elem );
};
```

Elección de Selectores

La elección de buenos selectores es un punto importante cuando se desea mejorar el rendimiento del código. Una pequeña especificidad — por ejemplo, incluir el tipo de elemento (como `div`) cuando se realiza una selección por el nombre de clase — puede ayudar bastante. Por eso, es recomendable darle algunas "pistas" a jQuery sobre en que lugar del documento puede encontrar lo que desea seleccionar. Por otro lado, demasiada especificidad puede ser perjudicial. Un selector como `#miTabla thead tr th.especial` es un exceso, lo mejor sería utilizar `#miTabla th.especial`.

jQuery ofrece muchos selectores basados en atributos, que permiten realizar selecciones basadas en el contenido de los atributos utilizando simplificaciones de expresiones regulares.

```
// encontrar todos los <a> cuyo atributo rel terminan en "thinger"
$("a[rel$='thinger']");
```

Estos tipos de selectores pueden resultar útiles pero también ser muy lentos. Cuando sea posible, es recomendable realizar la selección utilizando IDs, nombres de clases y nombres de etiquetas.

Si desea conocer más sobre este asunto, Paul Irish realizó una gran presentación sobre mejoras de rendimiento en JavaScript (en ingles), la cual posee varias diapositivas centradas en selectores.

Comprobar Selecciones

Una vez realizada la selección de los elementos, querrá conocer si dicha selección entregó algún resultado. Para ello, pueda que escriba algo así:

```
if ($('div.foo')) { ... }
```

Sin embargo esta forma no funcionará. Cuando se realiza una selección utilizando `$()`, siempre es devuelto un objeto, y si se lo evalúa, éste siempre devolverá `true`. Incluso si la selección no contiene ningún elemento, el código dentro del bloque `if` se ejecutará.

En lugar de utilizar el código mostrado, lo que se debe hacer es preguntar por la cantidad de elementos que posee la selección que se ejecutó. Esto es posible realizarlo utilizando la propiedad JavaScript `length`. Si la respuesta es 0, la condición evaluará falso, caso contrario (más de 0 elementos), la condición será verdadera.

Evaluar si una selección posee elementos

```
if ($('div.foo').length) { ... }
```

Guardar Selecciones

Cada vez que se hace una selección, una gran cantidad de código es ejecutado. jQuery no guarda el resultado por si solo, por lo tanto, si va a realizar una selección que luego se hará de nuevo, deberá salvar la selección en una variable.

Guardar selecciones en una variable

```
var $divs = $('div');
```

> ***Nota***
>
> *En el ejemplo "Guardar selecciones en una variable", la variable comienza con el signo de dólar. Contrariamente a otros lenguajes de programación, en JavaScript este signo no posee ningún significado especial — es solamente otro carácter. Sin embargo aquí se utilizará para indicar que dicha variable posee un objeto jQuery. Esta práctica — una especie de Notación Húngara — es solo una convención y no es obligatoria.*

Una vez que la selección es guardada en la variable, se la puede utilizar en conjunto con los métodos de jQuery y el resultado será igual que utilizando la selección original.

> ***Nota***
>
> *La selección obtiene sólo los elementos que están en la página cuando se realizó dicha acción. Si luego se añaden elementos al documento, será necesario repetir la selección o añadir los elementos nuevos a la selección guardada en la variable. En otras palabras, las selecciones guardadas no se actualizan "mágicamente" cuando el DOM de modifica.*

Refinamiento y Filtrado de Selecciones

A veces, puede obtener una selección que contiene más de lo que necesita; en este caso, es necesario refinar dicha selección. jQuery ofrece varios métodos para poder obtener exactamente lo que desea.

Refinamiento de selecciones

```
$('div.foo').has('p');          // el elemento div.foo contiene elementos <p>
$('h1').not('.bar');            // el elemento h1 no posse la clase 'bar'
$('ul li').filter('.current');  // un item de una lista desordenada
                                // que posse la clase 'current'
$('ul li').first();             // el primer item de una lista desordenada
$('ul li').eq(5);               // el sexto item de una lista desordenada
```

Selección de Elementos de un Formulario

jQuery ofrece varios pseudo-selectores que ayudan a encontrar elementos dentro de los formularios, éstos son especialmente útiles ya que dependiendo de los estados de cada elemento o su tipo, puede ser difícil distinguirlos utilizando selectores CSS estándar.

:button Selecciona elementos `<button>` y con el atributo `type='button'`

:checkbox Selecciona elementos `<input>` con el atributo `type='checkbox'`

:checked Selecciona elementos `<input>` del tipo `checkbox` seleccionados

:disabled Selecciona elementos del formulario que están deshabitados

:enabled Selecciona elementos del formulario que están habilitados

:file Selecciona elementos `<input>` con el atributo `type='file'`

:image Selecciona elementos `<input>` con el atributo `type='image'`

:input Selecciona elementos `<input>`, `<textarea>` y `<select>`

:password Selecciona elementos `<input>` con el atributo `type='password'`

:radio Selecciona elementos `<input>` con el atributo `type='radio'`

:reset Selecciona elementos `<input>` con el atributo `type='reset'`

:selected Selecciona elementos `<options>` que están seleccionados

:submit Selecciona elementos `<input>` con el atributo `type='submit'`

:text Selecciona elementos `<input>` con el atributo `type='text'`

Utilizando pseudo-selectores en elementos de formularios

```
$('#myForm :input'); // obtiene todos los elementos inputs
                     // dentro del formulario #myForm
```

0.3.3. Trabajar con Selecciones

Una vez realizada la selección de los elementos, es posible utilizarlos en conjunto con diferentes métodos. éstos, generalmente, son de dos tipos: obtenedores (en inglés *getters*) y establecedores (en inglés *setters*). Los métodos obtenedores devuelven una propiedad del elemento seleccionado; mientras que los métodos establecedores fijan una propiedad a todos los elementos seleccionados.

Encadenamiento

Si en una selección se realiza una llamada a un método, y éste devuelve un objeto jQuery, es posible seguir un "encadenado" de métodos en el objeto.

Encadenamiento

```
$('#content').find('h3').eq(2).html('nuevo texto para el tercer elemento h3');
```

Por otro lado, si se está escribiendo un encadenamiento de métodos que incluyen muchos pasos, es posible escribirlos línea por línea, haciendo que el código luzca más agradable para leer.

Formateo de código encadenado

```
$('#content')
    .find('h3')
    .eq(2)
    .html('nuevo texto para el tercer elemento h3');
```

Si desea volver a la selección original en el medio del encadenado, jQuery ofrece el método `$.fn.end` para poder hacerlo.

Restablecer la selección original utilizando el método `$.fn.end`

```
$('#content')
    .find('h3')
    .eq(2)
        .html('nuevo texto para el tercer elemento h3')
    .end() // reestablece la selección a todos los elementos h3 en #content
    .eq(0)
        .html('nuevo texto para el primer elemento h3');
```

Nota

El encadenamiento es muy poderoso y es una característica que muchas bibliotecas JavaScript han adoptado desde que jQuery se hizo popular. Sin embargo, debe ser utilizado con cuidado. Un encadenamiento de métodos extensivo pueden hacer un código extremadamente difícil de modificar y depurar. No existe una regla que indique que tan largo o corto debe ser el encadenado — pero es recomendable que tenga en cuenta este consejo.

Obtenedores (Getters) & Establecedores (Setters)

jQuery "sobrecarga" sus métodos, en otras palabras, el método para establecer un valor posee el mismo nombre que el método para obtener un valor. Cuando un método es utilizado para establecer un valor, es llamado método establecedor (en inglés *setter*). En cambio, cuando un método es utilizado para obtener (o leer) un valor, es llamado obtenedor (en inglés *getter*).

El método `$.fn.html` utilizado como establecedor

```
$('h1').html('hello world');
```

El método html utilizado como obtenedor

```
$('h1').html();
```

Los métodos establecedores devuelven un objeto jQuery, permitiendo continuar con la llamada de más métodos en la misma selección, mientras que los métodos obtenedores devuelven el valor por el cual se consultó, pero no permiten seguir llamando a más métodos en dicho valor.

0.3.4. CSS, Estilos, & Dimensiones

jQuery incluye una manera útil de obtener y establecer propiedades CSS a los elementos.

Nota

Las propiedades CSS que incluyen como separador un guión del medio, en JavaScript deben ser transformadas a su estilo CamelCase*. Por ejemplo, cuando se la utiliza como propiedad de un método, el estilo CSS `font-size` deberá ser expresado como `fontSize`. Sin embargo, esta regla no es aplicada cuando se pasa el nombre de la propiedad CSS al método `$.fn.css` — en este caso, los dos formatos (en* CamelCase *o con el guión del medio) funcionarán.*

Obtener propiedades CSS

```
$('h1').css('fontSize'); // devuelve una cadena de caracteres como "19px"
$('h1').css('font-size'); // también funciona
```

Establecer propiedades CSS

```
$('h1').css('fontSize', '100px'); // establece una propiedad individual CSS
$('h1').css({
    'fontSize' : '100px',
    'color' : 'red'
}); // establece múltiples propiedades CSS
```

Notar que el estilo del argumento utilizado en la segunda línea del ejemplo — es un objeto que contiene múltiples propiedades. Esta es una forma común de pasar múltiples argumentos a una función, y muchos métodos establecedores de la biblioteca aceptan objetos para fijar varias propiedades de una sola vez.

A partir de la versión 1.6 de la biblioteca, utilizando `$.fn.css` también es posible establecer valores relativos en las propiedades CSS de un elemento determinado:

Establecer valores CSS relativos

```
$('h1').css({
    'fontSize' : '+=15px', // suma 15px al tamaño original del elemento
    'paddingTop' : '+=20px' // suma 20px al padding superior original del elemento
});
```

Utilizar Clases para Aplicar Estilos CSS

Para obtener valores de los estilos aplicados a un elemento, el método `$.fn.css` es muy útil, sin embargo, su utilización como método establecedor se debe evitar (ya que, para aplicar estilos a un elemento, se puede hacer directamente desde CSS). En su lugar, lo ideal, es escribir reglas CSS que se apliquen a clases que describan los diferentes estados visuales de los elementos y luego cambiar la clase del elemento para aplicar el estilo que se desea mostrar.

Trabajar con clases

```
var $h1 = $('h1');

$h1.addClass('big');
$h1.removeClass('big');
$h1.toggleClass('big');

if ($h1.hasClass('big')) { ... }
```

Las clases también pueden ser útiles para guardar información del estado de un elemento, por ejemplo, para indicar que un elemento fue seleccionado.

Dimensiones

jQuery ofrece una variedad de métodos para obtener y modificar valores de dimensiones y posición de un elemento.

El código mostrado en el ejemplo “Métodos básicos sobre Dimensiones” es solo un breve resumen de las funcionalidades relaciones a dimensiones en jQuery; para un completo detalle puede consultar http://api.jquery.com/category/dimensions/.

Métodos básicos sobre Dimensiones

```
$('h1').width('50px');   // establece el ancho de todos los elementos H1
$('h1').width();         // obtiene el ancho del primer elemento H1

$('h1').height('50px');  // establece el alto de todos los elementos H1
$('h1').height();        // obtiene el alto del primer elemento H1

$('h1').position();      // devuelve un objeto conteniendo
                         // información sobre la posición
                         // del primer elemento relativo al
                         // "offset" (posición) de su elemento padre
```

0.3.5. Atributos

Los atributos de los elementos HTML que conforman una aplicación pueden contener información útil, por eso es importante poder establecer y obtener esa información.

El método `$.fn.attr` actúa tanto como método establecedor como obtenedor. Además, al igual que el método `$.fn.css`, cuando se lo utiliza como método establecedor, puede aceptar un conjunto de palabra clave-valor o un objeto conteniendo más conjuntos.

Establecer atributos

```
$('a').attr('href', 'allMyHrefsAreTheSameNow.html');
$('a').attr({
    'title' : 'all titles are the same too',
    'href' : 'somethingNew.html'
});
```

En el ejemplo, el objeto pasado como argumento está escrito en varias líneas. Como se explicó anteriormente, los espacios en blanco no importan en JavaScript, por lo cual, es libre de utilizarlos para hacer el código más legible. En entornos de producción, se pueden utilizar herramientas de minificación, los cuales quitan los espacios en blanco (entre otras cosas) y comprimen el archivo final.

Obtener atributos

```
$('a').attr('href');  // devuelve el atributo href perteneciente
                      // al primer elemento <a> del documento
```

0.3.6. Recorrer el DOM

Una vez obtenida la selección, es posible encontrar otros elementos utilizando a la misma selección.

En http://api.jquery.com/category/traversing/ puede encontrar una completa documentación sobre los métodos de recorrido de DOM (en inglés *traversing*) que posee jQuery.

> ***Nota***
>
> *Debe ser cuidadoso en recorrer largas distancias en un documento — recorridos complejos obligan que la estructura del documento sea siempre la misma, algo que es difícil de garantizar. Uno -o dos- pasos para el recorrido esta bien, pero generalmente hay que evitar atravesar desde un contenedor a otro.*

Moverse a través del DOM utilizando métodos de recorrido

```
$('h1').next('p');              // seleccionar el inmediato y próximo
                                // elemento <p> con respecto a H1
$('div:visible').parent();      // seleccionar el elemento contenedor
                                // a un div visible
$('input[name=first_name]').closest('form');  // seleccionar el elemento
                                              // <form> más cercano a un input
$('#myList').children();        // seleccionar todos los elementos
                                // hijos de #myList
$('li.selected').siblings();    // seleccionar todos los items
                                // hermanos del elemento <li>
```

También es posible interactuar con la selección utilizando el método `$.fn.each`. Dicho método interactúa con todos los elementos obtenidos en la selección y ejecuta una función por cada uno. La función recibe como argumento el índice del elemento actual y al mismo elemento. De forma predeterminada, dentro de la función, se puede hacer referencia al elemento DOM a través de la declaración `this`.

Interactuar en una selección

```
$('#myList li').each(function(idx, el) {
    console.log(
        'El elemento ' + idx +
        'contiene el siguiente HTML: ' +
        $(el).html()
    );
});
```

0.3.7. Manipulación de Elementos

Una vez realizada la selección de los elementos que desea utilizar, "la diversión comienza". Es posible cambiar, mover, remover y duplicar elementos. También crear nuevos a través de una sintaxis simple.

La documentación completa sobre los métodos de manipulación puede encontrarla en la sección `Manipulation`: http://api.jquery.com/category/manipulation/.

Obtener y Establecer Información en Elementos

Existen muchas formas por las cuales de puede modificar un elemento. Entre las tareas más comunes están las de cambiar el HTML interno o algún atributo del mismo. Para este tipo de tareas, jQuery ofrece métodos simples, funcionales en todos los navegadores modernos. Incluso es posible obtener información sobre los elementos utilizando los mismos métodos pero en su forma de método obtenedor.

> ***Nota***
>
> *Realizar cambios en los elementos, es un trabajo trivial, pero hay debe recordar que el cambio afectará a* todos *los elementos en la selección, por lo que, si desea modificar un sólo elemento, tiene que estar seguro de especificarlo en la selección antes de llamar al método establecedor.*

Nota

Cuando los métodos actúan como obtenedores, por lo general, solamente trabajan con el primer elemento de la selección. Además no devuelven un objeto jQuery, por lo cual no es posible encadenar más métodos en el mismo. Una excepción es el método `$.fn.text`*, el cual permite obtener el texto de los elementos de la selección.*

$.fn.html Obtiene o establece el contenido HTML de un elemento.

$.fn.text Obtiene o establece el contenido en texto del elemento; en caso se pasarle como argumento código HTML, este es despojado.

$.fn.attr Obtiene o establece el valor de un determinado atributo.

$.fn.width Obtiene o establece el ancho en pixeles del primer elemento de la selección como un entero.

$.fn.height Obtiene o establece el alto en pixeles del primer elemento de la selección como un entero.

$.fn.position Obtiene un objeto con información sobre la posición del primer elemento de la selección, relativo al primer elemento padre posicionado. *Este método es solo obtenedor.*

$.fn.val Obtiene o establece el valor (*value*) en elementos de formularios.

Cambiar el HTML de un elemento

```
$('#myDiv p:first')
    .html('Nuevo <strong>primer</strong> párrafo');
```

Mover, Copiar y Remover Elementos

Existen varias maneras para mover elementos a través del DOM; las cuales se pueden separar en dos enfoques:

- querer colocar el/los elementos seleccionados de forma relativa a otro elemento;
- querer colocar un elemento relativo a el/los elementos seleccionados.

Por ejemplo, jQuery provee los métodos `$.fn.insertAfter` y `$.fn.after`. El método `$.fn.insertAfter` coloca a el/los elementos seleccionados después del elemento que se haya pasado como argumento; mientras que el método `$.fn.after` coloca al elemento pasado como argumento después del elemento seleccionado. Otros métodos también siguen este patrón: `$.fn.insertBefore` y `$.fn.before`; `$.fn.appendTo` y `$.fn.append`; y `$.fn.prependTo` y `$.fn.prepend`.

La utilización de uno u otro método dependerá de los elementos que tenga seleccionados y el tipo de referencia que se quiera guardar con respecto al elemento que se esta moviendo.

Mover elementos utilizando diferentes enfoques

```
// hacer que el primer item de la lista sea el último
var $li = $('#myList li:first').appendTo('#myList');

// otro enfoque para el mismo problema
$('#myList').append($('#myList li:first'));

// debe tener en cuenta que no hay forma de acceder a la
// lista de items que se ha movido, ya que devuelve
// la lista en sí
```

Clonar Elementos Cuando se utiliza un método como `$.fn.appendTo`, lo que se está haciendo es mover al elemento; pero a veces en lugar de eso, se necesita mover un duplicado del mismo elemento. En este caso, es posible utilizar el método `$.fn.clone`.

Obtener una copia del elemento

```
// copiar el primer elemento de la lista y moverlo al final de la misma
$('#myList li:first').clone().appendTo('#myList');
```

> ***Nota***
>
> *Si se necesita copiar información y eventos relacionados al elemento, se debe pasar `true` como argumento de `$.fn.clone`.*

Remover elementos Existen dos formas de remover elementos de una página: Utilizando `$.fn.remove` o `$.fn.detach`. Cuando desee remover de forma permanente al elemento, utilize el método `$.fn.remove`. Por otro lado, el método $.fn.detach también remueve el elemento, pero mantiene la información y eventos asociados al mismo, siendo útil en el caso que necesite reinsertar el elemento en el documento.

> ***Nota***
>
> *El método `$.fn.detach` es muy útil cuando se esta manipulando de forma severa un elemento, ya que es posible eliminar al elemento, trabajarlo en el código y luego restaurarlo en la página nuevamente. Esta forma tiene como beneficio no tocar el DOM mientras se está modificando la información y eventos del elemento.*

Por otro lado, si se desea mantener al elemento pero se necesita eliminar su contenido, es posible utiliza el método `$.fn.empty`, el cual "vaciará" el contenido HTML del elemento.

Crear Nuevos Elementos

jQuery provee una forma fácil y elegante para crear nuevos elementos a través del mismo método `$()` que se utiliza para realizar selecciones.

Crear nuevos elementos

```
$('<p>Un nuevo párrafo</p>');
$('<li class="new">nuevo item de la lista</li>');
```

Crear un nuevo elemento con atributos utilizando un objeto

```
$('<a/>', {
    html : 'Un <strong>nuevo</strong> enlace',
    'class' : 'new',
    href : 'foo.html'
});
```

Note que en el objeto que se pasa como argumento, la propiedad class está entre comillas, mientras que la propiedad href y html no lo están. Por lo general, los nombres de propiedades no deben estar

entre comillas, excepto en el caso que se utilice como nombre una palabra reservada (como es el caso de class).

Cuando se crea un elemento, éste no es añadido inmediatamente a la página, sino que se debe hacerlo en conjunto con un método.

Crear un nuevo elemento en la página

```
var $myNewElement = $('<p>Nuevo elemento</p>');
$myNewElement.appendTo('#content');

$myNewElement.insertAfter('ul:last'); // eliminará al elemento <p>
                                      // existente en #content
$('ul').last().after($myNewElement.clone());  // clonar al elemento <p>
                                               // para tener las dos versiones
```

Estrictamente hablando, no es necesario guardar al elemento creado en una variable — es posible llamar al método para añadir el elemento directamente después de $(). Sin embargo, la mayoría de las veces se deseará hacer referencia al elemento añadido, por lo cual, si se guarda en una variable no es necesario seleccionarlo después.

Crear y añadir al mismo tiempo un elemento a la página

```
$('ul').append('<li>item de la lista</li>');
```

> ***Nota***
>
> *La sintaxis para añadir nuevos elementos a la página es muy fácil de utilizar, pero es tentador olvidar que hay un costo enorme de rendimiento al agregar elementos al DOM de forma repetida. Si esta añadiendo muchos elementos al mismo contenedor, en lugar de añadir cada elemento uno por vez, lo mejor es concatenar todo el HTML en una única cadena de caracteres para luego anexarla al contenedor. Una posible solución es utilizar un vector que posea todos los elementos, luego reunirlos utilizando* `join` *y finalmente anexarla.*

```
var myItems = [], $myList = $('#myList');

for (var i=0; i<100; i++) {
    myItems.push('<li>item ' + i + '</li>');
}

$myList.append(myItems.join(''));
```

Manipulación de Atributos

Las capacidades para la manipulación de atributos que ofrece la biblioteca son extensos. La realización de cambios básicos son simples, sin embargo el método `$.fn.attr` permite manipulaciones más complejas.

Manipular un simple atributo

```
$('#myDiv a:first').attr('href', 'newDestination.html');
```

Manipular múltiples atributos

```
$('#myDiv a:first').attr({
    href : 'newDestination.html',
    rel : 'super-special'
});
```

Utilizar una función para determinar el valor del nuevo atributo

```
$('#myDiv a:first').attr({
    rel : 'super-special',
    href : function(idx, href) {
        return '/new/' + href;
    }
});

$('#myDiv a:first').attr('href', function(idx, href) {
    return '/new/' + href;
});
```