Fundamentos de jQuery

Textos originales:
Rebecca Murphey

Traducción, adaptación y textos adicionales:
Leandro D'Onofrio

Correcciones:
Gustavo Raúl Aragón, Pablo Maronna, Denis Ciccale y otras personas

Con contribuciones de James Padolsey, Paul Irish y otros.

Agosto 2013

Índice general

0.1. Bienvenido/a

jQuery se está convirtiendo rápidamente en una herramienta que todo desarrollador de interfaces web debería de conocer. El propósito de este libro es proveer un resumen de la biblioteca, de tal forma que para cuando lo haya terminado de leer, será capaz de realizar tareas básicas utilizando jQuery y tendrá una sólida base para continuar el aprendizaje. El libro fue diseñado para ser utilizado como material en un salón de clases, pero también puede ser útil para estudiarlo de forma individual.

La modalidad de trabajo es la siguiente: En primer lugar se dedicará tiempo a comprender un concepto para luego realizar un ejercicio relacionado. Algunos de los ejercicios pueden llegar a ser triviales, mientras que otros no tanto. El objetivo es aprender a resolver de manera fácil lo que normalmente se resolvería con jQuery. Las soluciones a todos los ejercicios están incluidas en el mismo material de aprendizaje.

0.1.1. Obtener el Material de Aprendizaje

El material de aprendizaje y el código fuente de los ejemplos que se utilizan en el libro están hospedados en un repositorio de Github. Desde allí es posible descargar un archivo .zip o .tar con el código para utilizar en un servidor web.

Si usted suele utilizar Git, es bienvenido de clonar o modificar el repositorio.

0.1.2. Software

Para trabajar con los contenidos del libro, necesitará las siguientes herramientas:

- navegador web Firefox;
- la extensión Firebug, para Firefox;
- un editor de textos planos (como Notepad++/Sublime Text 2 para Windows, gedit/Kate para Linux o TextMate para Mac OS X);
- para las secciones dedicadas a Ajax: Un servidor local (como WAMP o MAMP) o un cliente FTP/SSH (como FileZilla) para acceder a un servidor remoto.

0.1.3. Añadir JavaScript a una Página

Existen dos formas de insertar código JavaScript dentro de una página: escribiendo código en la misma (en ingles inline) o a través de un archivo externo utilizando la etiqueta script. El orden en el cual se incluye el código es importante: un código que depende de otro debe ser incluido después del que referencia (Ejemplo: Si la función B depende de A, el orden debe ser A,B y no B,A).

Para mejorar el rendimiento de la página, el código JavaScript debe ser incluido al final del HTML. Además, cuando se trabaja en un ambiente de producción con múltiples archivos JavaScript, éstos deben ser combinados en un solo archivo.

Ejemplo de código JavaScript en línea

```
<script>
console.log('hello');
</script>
```

Ejemplo de inclusión de un archivo externo JavaScript

```
<script src='/js/jquery.js'></script>
```

0.1.4. Depuración del Código JavaScript

La utilización de una herramienta de depuración es esencial para trabajar con JavaScript. Firefox provee un depurador a través de la extensión Firebug; mientras que Safari y Chrome ya traen uno integrado.

Cada depurador ofrece:

- un editor multi-linea para experimentar con JavaScript;
- un inspector para revisar el código generado en la página;
- un visualizador de red o recursos, para examinar las peticiones que se realizan.

Cuando usted este escribiendo código JavaScript, podrá utilizar alguno de los siguientes métodos para enviar mensajes a la consola del depurador:

- `console.log()` para enviar y registrar mensajes generales;
- `console.dir()` para registrar un objeto y visualizar sus propiedades;
- `console.warn()` para registrar mensajes de alerta;
- `console.error()` para registrar mensajes de error.

Existen otros métodos para utilizar desde la consola, pero estos pueden variar según el navegador. La consola además provee la posibilidad de establecer puntos de interrupción y observar expresiones en el código con el fin de facilitar su depuración.

0.1.5. Ejercicios

La mayoría de los capítulos concluyen con uno o más ejercicios. En algunos, podrá trabajar directamente con Firebug; en otros deberá escribir código JavaScript luego de incluir la biblioteca jQuery en el documento.

Aún así, para completar ciertos ejercicios, necesitará consultar la documentación oficial de jQuery. Aprender a encontrar respuestas, es una parte importante del proceso de aprendizaje.

Estas son algunas sugerencias para hacer frente a los problemas:

- en primer lugar, asegúrese de entender bien el problema que está tratando de resolver;
- luego, averigüe a qué elementos tendrá que acceder con el fin de resolver el problema, y determine cómo accederlos. Puede utilizar Firebug para verificar que esta obteniendo el resultado esperado;
- finalmente, averigüe qué necesita hacer con esos elementos para resolver el problema. Puede ser útil, antes de comenzar, escribir comentarios explicando lo que va a realizar.

No tenga miedo de cometer errores. Tampoco trate en el primer intento escribir de forma perfecta su código. Cometer errores y experimentar con soluciones es parte del proceso de aprendizaje y le ayudará a que sea un mejor desarrollador.

Podrá encontrar en la carpeta `/ejercicios/soluciones` ejemplos de soluciones a los ejercicios del libro.

0.1.6. Convenciones Utilizadas en el Libro

Existen una serie de convenciones utilizadas en el libro:

Los métodos que pueden ser llamados desde el objeto jQuery, serán referenciados como `$.fn.nombreDelMetodo`. Los métodos que existen en el espacio de nombres (en inglés *namespace*) de jQuery pero que no pueden ser llamados desde el objeto jQuery serán referenciados como `$.nombreDelMetodo`. Si esto no significa mucho para usted, no se preocupe — será más claro a medida que vaya progresando en el libro.

Ejemplo de un código

```
// el código de ejemplo aparecerá de esta forma
```

Las remarcaciones aparecerán de esta forma.

> ***Nota***
>
> *Las notas sobre algún tema aparecerán de esta forma.*

0.1.7. Notas de la Traducción

- Debido a que el material tiene como fin el aprendizaje y la enseñanza, el mismo se encuentra traducido a español formal (usted).
- Muchos conceptos técnicos son nombrados en su versión traducida a español. Sin embargo, para tener de referencia, también se explica como es llamado en inglés.
- Los ejemplos y soluciones a ejercicios no están completamente traducidos. Esto es debido a que, cuando esté trabajando en un proyecto real, el código que encuentre en otros sitios probablemente esté en inglés. Aún así, se han traducido los comentarios incorporados en los códigos de ejemplos y algunos textos particulares para facilitar la comprensión.

0.1.8. Material de Referencia

Existe una gran cantidad de artículos que se ocupan de algún aspecto de jQuery. Algunos son excelentes pero otros, francamente, son erróneos. Cuando lea un artículo sobre jQuery, este seguro que se está abarcando la misma versión de la biblioteca que está utilizando, y resístase a la tentación de copiar y pegar el código — tómese un tiempo para poder entenderlo.

A continuación se listan una serie de excelentes recursos para utilizar durante el aprendizaje. El más importante de todos es el código fuente de jQuery, el cual contiene (en su formato sin comprimir) una completa documentación a través de comentarios. La biblioteca no es una caja negra — el entendimiento de ella irá incrementándose exponencialmente si la revisa de vez en cuando — y es muy recomendable que la guarde en los favoritos de su navegador para tenerla como guía de referencia.

- El código fuente de jQuery
- Documentación de jQuery
- Foro de jQuery
- Favoritos en Delicious
- Canal IRC #jquery en Freenode

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 se 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;
});
```

0.3.8. Ejercicios

Selecciones

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/sandbox.js` o trabaje directamente con Firebug para cumplir los siguientes puntos:

1. Seleccionar todos los elementos `div` que poseen la clase "module".
2. Especificar tres selecciones que puedan seleccionar el tercer ítem de la lista desordenada #myList. ¿Cuál es el mejor para utilizar? ¿Porqué?
3. Seleccionar el elemento `label` del elemento `input` utilizando un selector de atributo.
4. Averiguar cuantos elementos en la página están ocultos (ayuda: `.length`).
5. Averiguar cuantas imágenes en la página poseen el atributo `alt`.
6. Seleccionar todas las filas impares del cuerpo de la tabla.

Recorrer el DOM

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/sandbox.js` o trabaje directamente con Firebug para cumplir los siguientes puntos:

1. Seleccionar todas las imágenes en la página; registrar en la consola el atributo `alt` de cada imagen.
2. Seleccionar el elemento `input`, luego dirigirse hacia el formulario y añadirle una clase al mismo.
3. Seleccionar el ítem que posee la clase "current" dentro de la lista #myList y remover dicha clase en el elemento; luego añadir la clase "current" al siguiente ítem de la lista.
4. Seleccionar el elemento `select` dentro de #specials; luego dirigirse hacia el botón `submit`.
5. Seleccionar el primer ítem de la lista en el elemento #slideshow; añadirle la clase "current" al mismo y luego añadir la clase "disabled" a los elementos hermanos.

Manipulación

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/sandbox.js` o trabaje directamente con Firebug para cumplir los siguientes puntos:

1. Añadir 5 nuevos ítems al final de la lista desordenada #myList. Ayuda:

```
for (var i = 0; i<5; i++) { ... }
```

2. Remover los ítems impares de la lista.
3. Añadir otro elemento `h2` y otro párrafo al último `div.module`.
4. Añadir otra opción al elemento `select`; darle a la opción añadida el valor *"Wednesday"*.
5. Añadir un nuevo `div.module` a la página después del último; luego añadir una copia de una de las imágenes existentes dentro del nuevo `div`.

0.4. El núcleo de jQuery

0.4.1. `$` vs `$()`

Hasta ahora, se ha tratado completamente con métodos que se llaman desde el objeto jQuery. Por ejemplo:

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

Dichos métodos son parte del espacio de nombres (en inglés *namespace*) `$.fn`, o del prototipo (en inglés *prototype*) de jQuery, y son considerados como métodos del objeto jQuery.

Sin embargo, existen métodos que son parte del espacio de nombres de $ y se consideran como métodos del núcleo de jQuery.

Estas distinciones pueden ser bastantes confusas para usuarios nuevos. Para evitar la confusión, debe recordar estos dos puntos:

- los métodos utilizados en selecciones se encuentran dentro del espacio de nombres `$.fn`, y automáticamente reciben y devuelven una selección en sí;
- métodos en el espacio de nombres `$` son generalmente métodos para diferentes utilidades, no trabajan con selecciones, no se les pasa ningún argumento y el valor que devuelven puede variar.

Existen algunos casos en donde métodos del objeto y del núcleo poseen los mismos nombres, como sucede con `$.each` y `$.fn.each`. En estos casos, debe ser cuidadoso de leer bien la documentación para saber que objeto utilizar correctamente.

0.4.2. Métodos Utilitarios

jQuery ofrece varios métodos utilitarios dentro del espacio de nombres `$`. Estos métodos son de gran ayuda para llevar a cabo tareas rutinarias de programación. A continuación se muestran algunos ejemplos, para una completa documentación sobre ellos, visite http://api.jquery.com/category/utilities/.

$.trim Remueve los espacios en blanco del principio y final.

```
$.trim('    varios espacios en blanco   ');
// devuelve 'varios espacios en blanco'
```

$.each Interactúa en vectores y objetos.

```
$.each([ 'foo', 'bar', 'baz' ], function(idx, val) {
    console.log('elemento ' + idx + 'es ' + val);
});

$.each({ foo : 'bar', baz : 'bim' }, function(k, v) {
    console.log(k + ' : ' + v);
});
```

> ***Nota***
>
> *Como se dijo antes, existe un método llamado* `$.fn.each`*, el cual interactúa en una selección de elementos.*

$.inArray Devuelve el índice de un valor en un vector, o -1 si el valor no se encuentra en el vector.

```
var myArray = [ 1, 2, 3, 5 ];

if ($.inArray(4, myArray) !== -1) {
    console.log('valor encontrado');
}
```

$.extend Cambia la propiedades del primer objeto utilizando las propiedades de los subsecuentes objetos.

```
var firstObject = { foo : 'bar', a : 'b' };
var secondObject = { foo : 'baz' };

var newObject = $.extend(firstObject, secondObject);
console.log(firstObject.foo); // 'baz'
console.log(newObject.foo);   // 'baz'
```

Si no se desea cambiar las propiedades de ninguno de los objetos que se utilizan en `$.extend`, se debe incluir un objeto vacío como primer argumento.

```
var firstObject = { foo : 'bar', a : 'b' };
var secondObject = { foo : 'baz' };

var newObject = $.extend({}, firstObject, secondObject);
console.log(firstObject.foo); // 'bar'
console.log(newObject.foo);   // 'baz'
```

$.proxy Devuelve una función que siempre se ejecutará en el alcance (*scope*) provisto — en otras palabras, establece el significado de *this* (incluido dentro de la función) como el segundo argumento.

```
var myFunction = function() { console.log(this); };
var myObject = { foo : 'bar' };

myFunction(); // devuelve el objeto window

var myProxyFunction = $.proxy(myFunction, myObject);
myProxyFunction(); // devuelve el objeto myObject
```

Si se posee un objeto con métodos, es posible pasar dicho objeto y el nombre de un método para devolver una función que siempre se ejecuta en el alcance de dicho objeto.

```
var myObject = {
    myFn : function() {
        console.log(this);
    }
};

$('#foo').click(myObject.myFn); // registra el elemento DOM #foo
$('#foo').click($.proxy(myObject, 'myFn')); // registra myObject
```

0.4.3. Comprobación de Tipos

Como se mencionó en el capítulo "Conceptos Básicos de JavaScript", jQuery ofrece varios métodos útiles para determinar el tipo de un valor específico.

Comprobar el tipo de un determinado valor

```
var myValue = [1, 2, 3];

// Utilizar el operador typeof de JavaScript para comprobar tipos primitivos
typeof myValue == 'string'; // falso (false)
typeof myValue == 'number'; // falso (false)
typeof myValue == 'undefined'; // falso (false)
typeof myValue == 'boolean'; // falso (false)

// Utilizar el operador de igualdad estricta para comprobar valores nulos (null)
myValue === null; // falso (false)

// Utilizar los métodos jQuery para comprobar tipos no primitivos
```

```
jQuery.isFunction(myValue); // falso (false)
jQuery.isPlainObject(myValue); // falso (false)
jQuery.isArray(myValue); // verdadero (true)
jQuery.isNumeric(16); // verdadero (true). No disponible en versiones inferiores a jQuery 1.7
```

0.4.4. El Método Data

A menudo encontrará que existe información acerca de un elemento que necesita guardar. En JavaScript es posible hacerlo añadiendo propiedades al DOM del elemento, pero esta práctica conlleva enfrentarse a pérdidas de memoria (en inglés *memory leaks*) en algunos navegadores. jQuery ofrece una manera sencilla para poder guardar información relacionada a un elemento, y la misma biblioteca se ocupa de manejar los problemas que pueden surgir por falta de memoria.

Guardar y recuperar información relacionada a un elemento

```
$('#myDiv').data('keyName', { foo : 'bar' });
$('#myDiv').data('keyName'); // { foo : 'bar' }
```

A través del método `$.fn.data` es posible guardar cualquier tipo de información sobre un elemento. Es difícil exagerar la importancia de este concepto cuando se está desarrollando una aplicación compleja.

Por ejemplo, si desea establecer una relación entre el ítem de una lista y el div que hay dentro de este ítem, es posible hacerlo cada vez que se interactúa con el ítem, pero una mejor solución es hacerlo una sola vez, guardando un puntero al div utilizando el método `$.fn.data`:

Establecer una relación entre elementos utilizando el método `$.fn.data`

```
$('#myList li').each(function() {
    var $li = $(this), $div = $li.find('div.content');
    $li.data('contentDiv', $div);
});

// luego, no se debe volver a buscar al div;
// es posible leerlo desde la información asociada al item de la lista
var $firstLi = $('#myList li:first');
$firstLi.data('contentDiv').html('nuevo contenido');
```

Además es posible pasarle al método un objeto conteniendo uno o más pares de conjuntos palabra clave-valor.

A partir de la versión 1.5 de la biblioteca, jQuery permite utilizar al método `$.fn.data` para obtener la información asociada a un elemento que posea el atributo HTML5 `data-*`:

Elementos con el atributo `data-*`

```
<a id='foo' data-foo='baz' href='#'>Foo</a>

<a id='foobar' data-foo-bar='fol' href='#'>Foo Bar</a>
```

Obtener los valores asociados a los atributos `data-*` con `$.fn.data`

```
// obtiene el valor del atributo data-foo
// utilizando el método $.fn.data
console.log($('#foo').data('foo')); // registra 'baz'

// obtiene el valor del segundo elemento
console.log($('#foobar').data('fooBar')); // registra 'fol'
```

Nota

A partir de la versión 1.6 de la biblioteca, para obtener el valor del atributo `data-foo-bar` *del segundo elemento, el argumento en* `$.fn.data` *se debe pasar en estilo* CamelCase.

Nota

Para más información sobre el atributo HTML5 `data-*` *visite http://www.w3.org/TR/html5/global-attributes.html#embedding-custom-non-visible-data-with-the-data-attributes.*

0.4.5. Detección de Navegadores y Características

Más allá que jQuery elimine la mayoría de las peculiaridades de JavaScript entre cada navegador, existen ocasiones en que se necesita ejecutar código en un navegador específico.

Para este tipo de situaciones, jQuery ofrece el objeto `$.support` y `$.browser` (este último en desuso). Una completa documentación sobre estos objetos puede encontrarla en http://api.jquery.com/jQuery.support/ y http://api.jquery.com/jQuery.browser/

El objetivo de `$.support` es determinar qué características soporta el navegador web.

El objeto `$.browser` permite detectar el tipo de navegador y su versión. Dicho objeto está en desuso (aunque en el corto plazo no está planificada su eliminación del núcleo de la biblioteca) y se recomienda utilizar al objeto `$.support` para estos propósitos.

0.4.6. Evitar Conflictos con Otras Bibliotecas JavaScript

Si esta utilizando jQuery en conjunto con otras bibliotecas JavaScript, las cuales también utilizan la variable `$`, pueden llegar a ocurrir una serie de errores. Para poder solucionarlos, es necesario poner a jQuery en su modo "no-conflicto". Esto se debe realizar inmediatamente después que jQuery se cargue en la página y antes del código que se va a ejecutar.

Cuando se pone a jQuery en modo "no-conflicto", la biblioteca ofrece la opción de asignar un nombre para reemplazar a la variable `$`.

Poner a jQuery en modo no-conflicto

```
<script src="prototype.js"></script>            // la biblioteca prototype
                                                // también utiliza $
<script src="jquery.js"></script>               // se carga jquery
                                                // en la página
<script>var $j = jQuery.noConflict();</script>  // se inicializa
                                                // el modo "no-conflicto"
```

También es posible seguir utilizando `$` conteniendo el código en una función anónima autoejecutable. Éste es un patrón estándar para la creación de extensiones para la biblioteca, ya que `$` queda encerrada dentro del alcance de la misma función anónima.

Utilizar $ dentro de una función anónima autoejecutable

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
jQuery.noConflict();

(function($) {
   // el código va aquí, pudiendo utilizar $
})(jQuery);
</script>
```

0.5. Eventos

0.5.1. Introducción

jQuery provee métodos para asociar controladores de eventos (en inglés *event handlers*) a selectores. Cuando un evento ocurre, la función provista es ejecutada. Dentro de la función, la palabra clave `this` hace referencia al elemento en que el evento ocurre.

Para más detalles sobre los eventos en jQuery, puede consultar http://api.jquery.com/category/events/.

La función del controlador de eventos puede recibir un objeto. Este objeto puede ser utilizado para determinar la naturaleza del evento o, por ejemplo, prevenir el comportamiento predeterminado de éste. Para más detalles sobre el objeto del evento, visite http://api.jquery.com/category/events/event-object/.

0.5.2. Vincular Eventos a Elementos

jQuery ofrece métodos para la mayoría de los eventos — entre ellos `$.fn.click`, `$.fn.focus`, `$.fn.blur`, `$.fn.change`, etc. Estos últimos son formas reducidas del método `$.fn.on` de jQuery (`$.fn.bind` en versiones anteriores a jQuery 1.7). El método `$.fn.on` es útil para vincular (en inglés *binding*) la misma función de controlador a múltiples eventos, para cuando se desea proveer información al controlador de evento, cuando se está trabajando con eventos personalizados o cuando se desea pasar un objeto a múltiples eventos y controladores.

Vincular un evento utilizando un método reducido

```
$('p').click(function() {
    console.log('click');
});
```

Vincular un evento utilizando el método `$.fn.on`

```
$('p').on('click', function() {
    console.log('click');
});
```

Vincular un evento utilizando el método `$.fn.on` con información asociada

```
$('input').on(
    'click blur',  // es posible vincular múltiples eventos al elemento
    { foo : 'bar' }, // se debe pasar la información asociada como argumento

    function(eventObject) {
        console.log(eventObject.type, eventObject.data);
        // registra el tipo de evento y la información asociada { foo : 'bar' }
    }
);
```

Vincular Eventos para Ejecutar una vez

A veces puede necesitar que un controlador particular se ejecute solo una vez — y después de eso, necesite que ninguno más se ejecute, o que se ejecute otro diferente. Para este propósito jQuery provee el método `$.fn.one`.

Cambiar controladores utilizando el método `$.fn.one`

```
$('p').one('click', function() {
    console.log('Se clickeó al elemento por primera vez');
    $(this).click(function() { console.log('Se ha clickeado nuevamente'); });
});
```

El método `$.fn.one` es útil para situaciones en que necesita ejecutar cierto código la primera vez que ocurre un evento en un elemento, pero no en los eventos sucesivos.

Desvincular Eventos

Para desvincular (en ingles *unbind*) un controlador de evento, puede utilizar el método `$.fn.off` pasándole el tipo de evento a desconectar. Si se pasó como adjunto al evento una función nombrada, es posible aislar la desconexión de dicha función pasándola como segundo argumento.

Desvincular todos los controladores del evento click en una selección

```
$('p').off('click');
```

Desvincular un controlador particular del evento click

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

$('p').on('click', foo).on('click', bar);
$('p').off('click', bar); // foo esta atado aún al evento click
```

Espacios de Nombres para Eventos

Cuando se esta desarrollando aplicaciones complejas o extensiones de jQuery, puede ser útil utilizar espacios de nombres para los eventos, y de esta forma evitar que se desvinculen eventos cuando no lo desea.

Asignar espacios de nombres a eventos

```
$('p').on('click.myNamespace', function() { /* ... */ });
$('p').off('click.myNamespace');
$('p').off('.myNamespace'); // desvincula todos los eventos con
                            // el espacio de nombre 'myNamespace'
```

Vinculación de Múltiples Eventos

Muy a menudo, elementos en una aplicación estarán vinculados a múltiples eventos, cada uno con una función diferente. En estos casos, es posible pasar un objeto dentro de `$.fn.on` con uno o más pares de nombres claves/valores. Cada clave será el nombre del evento mientras que cada valor será la función a ejecutar cuando ocurra el evento.

Vincular múltiples eventos a un elemento

```
$('p').on({
    'click': function() {
        console.log('clickeado');
    },
    'mouseover': function() {
        console.log('sobrepasado');
    }
});
```

0.5.3. El Objeto del Evento

Como se menciona en la introducción, la función controladora de eventos recibe un objeto del evento, el cual contiene varios métodos y propiedades. El objeto es comúnmente utilizado para prevenir la acción predeterminada del evento a través del método *preventDefault*. Sin embargo, también contiene varias propiedades y métodos útiles:

pageX, pageY La posición del puntero del ratón en el momento que el evento ocurrió, relativo a las zonas superiores e izquierda de la página.
type El tipo de evento (por ejemplo "click").
which El botón o tecla presionada.
data Alguna información pasada cuando el evento es ejecutado.
target El elemento DOM que inicializó el evento.
preventDefault() Cancela la acción predeterminada del evento (por ejemplo: seguir un enlace).
stopPropagation() Detiene la propagación del evento sobre otros elementos.

Por otro lado, la función controladora también tiene acceso al elemento DOM que inicializó el evento a través de la palabra clave `this`. Para convertir a dicho elemento DOM en un objeto jQuery (y poder utilizar los métodos de la biblioteca) es necesario escribir `$(this)`, como se muestra a continuación:

```
var $this = $(this);
```

Cancelar que al hacer click en un enlace, éste se siga

```
$('a').click(function(e) {
    var $this = $(this);
```

```
    if ($this.attr('href').match('evil')) {
        e.preventDefault();
        $this.addClass('evil');
    }
});
```

0.5.4. Ejecución automática de Controladores de Eventos

A través del método `$.fn.trigger`, jQuery provee una manera de disparar controladores de eventos sobre algún elemento sin requerir la acción del usuario. Si bien este método tiene sus usos, no debería ser utilizado para simplemente llamar a una función que pueda ser ejecutada con un click del usuario. En su lugar, debería guardar la función que se necesita llamar en una variable, y luego pasar el nombre de la variable cuando realiza el vinculo (*binding*). De esta forma, podrá llamar a la función cuando lo desee en lugar de ejecutar `$.fn.trigger`.

Disparar un controlador de eventos de la forma correcta

```
var foo = function(e) {
    if (e) {
        console.log(e);
    } else {
        console.log('esta ejecucción no provino desde un evento');
    }
};


$('p').click(foo);

foo(); // en lugar de realizar $('p').trigger('click')
```

0.5.5. Incrementar el Rendimiento con la Delegación de Eventos

Cuando trabaje con jQuery, frecuentemente añadirá nuevos elementos a la página, y cuando lo haga, necesitará vincular eventos a dichos elementos. En lugar de repetir la tarea cada vez que se añade un elemento, es posible utilizar la delegación de eventos para hacerlo. Con ella, podrá enlazar un evento a un elemento contenedor, y luego, cuando el evento ocurra, podrá ver en que elemento sucede.

La delegación de eventos posee algunos beneficios, incluso si no se tiene pensando añadir más elementos a la página. El tiempo requerido para enlazar controladores de eventos a cientos de elementos no es un trabajo trivial; si posee un gran conjunto de elementos, debería considerar utilizar la delegación de eventos a un elemento contenedor.

> ***Nota***
>
> *A partir de la versión 1.4.2, se introdujo `$.fn.delegate`, sin embargo a partir de la versión 1.7 es preferible utilizar el evento `$.fn.on` para la delegación de eventos.*

Delegar un evento utilizando `$.fn.on`

```
$('#myUnorderedList').on('click', 'li', function(e) {
    var $myListItem = $(this);
    // ...
});
```

Delegar un evento utilizando `$.fn.delegate`

```
$('#myUnorderedList').delegate('li', 'click', function(e) {
    var $myListItem = $(this);
    // ...
});
```

Desvincular Eventos Delegados

Si necesita remover eventos delegados, no puede hacerlo simplemente desvinculándolos. Para eso, utilice el método `$.fn.off` para eventos conectados con `$.fn.on`, y `$.fn.undelegate` para eventos conectados con `$.fn.delegate`. Al igual que cuando se realiza un vinculo, opcionalmente, se puede pasar el nombre de una función vinculada.

Desvincular eventos delegados

```
$('#myUnorderedList').off('click', 'li');
$('#myUnorderedList').undelegate('li', 'click');
```

0.5.6. Funciones Auxiliares de Eventos

jQuery ofrece dos funciones auxiliares para el trabajo con eventos:

`$.fn.hover`

El método `$.fn.hover` permite pasar una o dos funciones que se ejecutarán cuando los eventos `mouseenter` y `mouseleave` ocurran en el elemento seleccionado. Si se pasa una sola función, está será ejecutada en ambos eventos; en cambio si se pasan dos, la primera será ejecutada cuando ocurra el evento `mouseenter`, mientras que la segunda será ejecutada cuando ocurra `mouseleave`.

> ***Nota***
> *A partir de la versión 1.4 de jQuery, el método requiere obligatoriamente dos funciones.*

La función auxiliar hover

```
$('#menu li').hover(function() {
    $(this).toggleClass('hover');
});
```

`$.fn.toggle`

Al igual que el método anterior, `$.fn.toggle` recibe dos o más funciones; cada vez que un evento ocurre, la función siguiente en la lista se ejecutará. Generalmente, `$.fn.toggle` es utilizada con solo dos funciones. En caso que utiliza más de dos funciones, tenga cuidado, ya que puede ser dificultar la depuración del código.

La función auxiliar toggle

```
$('p.expander').toggle(
    function() {
        $(this).prev().addClass('open');
    },
    function() {
        $(this).prev().removeClass('open');
    }
);
```

0.5.7. Ejercicios

Crear una "Sugerencia" para una Caja de Ingreso de Texto

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/inputHint.js` o trabaje directamente con Firebug. La tarea a realizar es utilizar el texto del elemento label y aplicar una "sugerencia" en la caja de ingreso de texto. Los pasos ha seguir son los siguientes:

1. Establecer el valor del elemento *input* igual al valor del elemento *label*.
2. Añadir la clase "hint" al elemento *input*.
3. Remover el elemento *label*.
4. Vincular un evento *focus* en el *input* para remover el texto de sugerencia y la clase "hint".
5. Vincular un evento *blur* en el *input* para restaurar el texto de sugerencia y la clase "hint" en caso que no se haya ingresado algún texto.

¿Qué otras consideraciones debe considerar si se desea aplicar esta funcionalidad a un sitio real?

Añadir una Navegación por Pestañas

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/tabs.js` o trabaje directamente con Firebug. La tarea a realizar es crear una navegación por pestañas para los dos elementos *div.module*. Los pasos ha seguir son los siguientes:

1. Ocultar todos los elementos *div.module*.
2. Crear una lista desordenada antes del primer *div.module* para utilizar como pestañas.
3. Interactuar con cada div utilizando `$.fn.each`. Por cada uno, utilizar el texto del elemento *h2* como el texto para el ítem de la lista desordenada.
4. Vincular un evento *click* a cada ítem de la lista de forma que:
    - muestre el *div* correspondiente y oculte el otro;
    - añada la clase "current" al ítem seleccionado;
    - remueva la clase "current" del otro ítem de la lista.
5. Finalmente, mostrar la primera pestaña.

0.6. Efectos

0.6.1. Introducción

Con jQuery, agregar efectos a una página es muy fácil. Estos efectos poseen una configuración predeterminada pero también es posible proveerles parámetros personalizados. Además es posible crear animaciones particulares estableciendo valores de propiedades CSS.

Para una completa documentación sobre los diferentes tipos de efectos puede visitar la sección `effects`: http://api.jquery.com/category/effects/.

0.6.2. Efectos Incorporados en la Biblioteca

Los efectos más utilizado ya vienen incorporados dentro de la biblioteca en forma de métodos:

$.fn.show Muestra el elemento seleccionado.
$.fn.hide Oculta el elemento seleccionado.
$.fn.fadeIn De forma animada, cambia la opacidad del elemento seleccionado al 100 %.
$.fn.fadeOut De forma animada, cambia la opacidad del elemento seleccionado al 0
$.fn.slideDown Muestra el elemento seleccionado con un movimiento de deslizamiento vertical.
$.fn.slideUp Oculta el elemento seleccionado con un movimiento de deslizamiento vertical.
$.fn.slideToggle Muestra o oculta el elemento seleccionado con un movimiento de deslizamiento vertical, dependiendo si actualmente el elemento está visible o no.

Uso básico de un efecto incorporado

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

Cambiar la Duración de los Efectos

Con la excepción de `$.fn.show` y `$.fn.hide`, todos los métodos tienen una duración predeterminada de la animación en 400ms. Este valor es posible cambiarlo.

Configurar la duración de un efecto

```
$('h1').fadeIn(300);      // desvanecimiento en 300ms
$('h1').fadeOut('slow');  // utilizar una definición de velocidad interna
```

jQuery.fx.speeds jQuery posee un objeto en `jQuery.fx.speeds` el cual contiene la velocidad predeterminada para la duración de un efecto, así como también los valores para las definiciones *"slow"* y *"fast"*.

```
speeds: {
    slow: 600,
    fast: 200,
    // velocidad predeterminada
    _default: 400
}
```

Por lo tanto, es posible sobrescribir o añadir nuevos valores al objeto. Por ejemplo, puede que quiera cambiar el valor predeterminado del efecto o añadir una velocidad personalizada.

Añadir velocidades personalizadas a `jQuery.fx.speeds`

```
jQuery.fx.speeds.muyRapido = 100;
jQuery.fx.speeds.muyLento = 2000;
```

Realizar una Acción Cuando un Efecto fue Ejecutado

A menudo, querrá ejecutar una acción una vez que la animación haya terminado — ya que si ejecuta la acción antes que la animación haya acabado, puede llegar a alterar la calidad del efecto o afectar a los elementos que forman parte de la misma. [Definición: *Las funciones de devolución de llamada* (en inglés *callback functions*) proveen una forma para ejecutar código una vez que un evento haya terminado.] En este caso, el evento que responderá a la función será la conclusión de la animación. Dentro de la función de devolución, la palabra clave `this` hace referencia al elemento en donde el efecto fue ejecutado y al igual que sucede con los eventos, es posible transformarlo a un objeto jQuery utilizando `$(this)`.

Ejecutar cierto código cuando una animación haya concluido

```
$('div.old').fadeOut(300, function() { $(this).remove(); });
```

Note que si la selección no retorna ningún elemento, la función nunca se ejecutará. Este problema lo puede resolver comprobando si la selección devuelve algún elemento; y en caso que no lo haga, ejecutar la función de devolución inmediatamente.

Ejecutar una función de devolución incluso si no hay elementos para animar

```
var $thing = $('#nonexistent');

var cb = function() {
    console.log('realizado');
};

if ($thing.length) {
    $thing.fadeIn(300, cb);
} else {
    cb();
}
```

0.6.3. Efectos Personalizados con `$.fn.animate`

Es posible realizar animaciones en propiedades CSS utilizando el método `$.fn.animate`. Dicho método permite realizar una animación estableciendo valores a propiedades CSS o cambiando sus valores actuales.

Efectos personalizados con `$.fn.animate`

```
$('div.funtimes').animate(
    {
```

```
        left : "+=50",
        opacity : 0.25
    },
    300, // duration
    function() { console.log('realizado'); // función de devolución de llamada
});
```

> ***Nota***
>
> *Las propiedades relacionadas al color no pueden ser animadas utilizando el método* ***`$.fn.animate`****, pero es posible hacerlo a través de la extensión color plugin. Más adelante en el libro de discutirá la utilización de extensiones.*

Easing

[Definición: El concepto de *Easing* describe la manera en que un efecto ocurre — es decir, si la velocidad durante la animación es constante o no.] jQuery incluye solamente dos métodos de easing: *swing* y *linear*. Si desea transiciones más naturales en las animaciones, existen varias extensiones que lo permiten.

A partir de la versión 1.4 de la biblioteca, es posible establecer el tipo de transición por cada propiedad utilizando el método `$.fn.animate`.

Transición de easing por cada propiedad

```
$('div.funtimes').animate(
    {
        left : [ "+=50", "swing" ],
        opacity : [ 0.25, "linear" ]
    },
    300
);
```

Para más detalles sobre las opciones de easing, consulte http://api.jquery.com/animate/.

0.6.4. Control de los Efectos

jQuery provee varias herramientas para el manejo de animaciones.

$.fn.stop Detiene las animaciones que se están ejecutando en el elemento seleccionado.
$.fn.delay Espera un tiempo determinado antes de ejecutar la próxima animación.

```
$('h1').show(300).delay(1000).hide(300);
```

jQuery.fx.off Si el valor es verdadero (*true*), no existirán transiciones para las animaciones; y a los elementos se le establecerá el estado final de la animación. Este método puede ser especialmente útil cuando se esta trabajando con navegadores antiguos.

0.6.5. Ejercicios

Mostrar Texto Oculto

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/blog.js`. La tarea es añadir alguna interactividad a la sección blog de la página:

- al hacer click en alguno de los titulares del *div #blog*, se debe mostrar el párrafo correspondiente con un efecto de deslizamiento;
- al hacer click en otro titular, se debe ocultar el párrafo mostrado con un efecto de deslizamiento y mostrar nuevamente el párrafo correspondiente también con un efecto de deslizamiento. Ayuda: No se olvide de utilizar el selector `:visible`.

Crear un Menú Desplegable

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/navigation.js`. La tarea es poder desplegar los ítems del menú superior de la página:

- al pasar el puntero del ratón por encima de un ítem del menú, se debe mostrar su submenú en caso que exista;
- al no estar más encima de un ítem, el submenú se debe ocultar.

Para poder realizarlo, utilice el método `$.fn.hover` para añadir o remover una clase en el submenú para poder controlar si debe estar oculto o visible (El archivo `/ejercicios/css/styles.css` incluye una clase "hover" para este propósito)

Crear un Slideshow

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/slideshow.js`. La tarea es añadir un slideshow a la página con JavaScript.

1. Mover el elemento *#slideshow* a la parte superior de la página.
2. Escribir un código que permita mostrar los ítems de forma cíclica, mostrando un ítem por unos segundos, luego ocultándolo con un efecto *fade out* y mostrando el siguiente con un efecto *fade in.
3. Una vez llegado al último ítem de la lista, comenzar de nuevo con el primero.

Para un desafío mayor, realice un área de navegación por debajo del slideshow que muestre cuantas imágenes existen y en cual se encuentra (ayuda: $.fn.prevAll‘ puede resultar útil).

0.7. Ajax

0.7.1. Introducción

El método *XMLHttpRequest* (XHR) permite a los navegadores comunicarse con el servidor sin la necesidad de recargar la página. Este método, también conocido como Ajax (*Asynchronous JavaScript and XML*), permite la creación de aplicaciones ricas en interactividad.

Las peticiones Ajax son ejecutadas por el código JavaScript, el cual envía una petición a una URL y cuando recibe una respuesta, una función de devolución puede ser ejecutada la cual recibe como argumento la respuesta del servidor y realiza algo con ella. Debido a que la respuesta es asíncrona, el resto del código de la aplicación continua ejecutándose, por lo cual, es imperativo que una función de devolución sea ejecutada para manejar la respuesta.

A través de varios métodos, jQuery provee soporte para Ajax, permitiendo abstraer las diferencias que pueden existir entre navegadores. Los métodos en cuestión son `$.get()`, `$.getScript()`, `$.getJSON()`, `$.post()` y `$().load()`.

A pesar que la definición de Ajax posee la palabra "XML", la mayoría de las aplicaciones no utilizan dicho formato para el transporte de datos, sino que en su lugar se utiliza HTML plano o información en formato JSON (*JavaScript Object Notation*).

En general, Ajax no trabaja a través de dominios diferentes. Sin embargo, existen excepciones, como los servicios que proveen información en formato JSONP (*JSON with Padding*), los cuales permiten una funcionalidad limitada a través de diferentes dominios.

0.7.2. Conceptos Clave

La utilización correcta de los métodos Ajax requiere primero la comprensión de algunos conceptos clave.

GET vs. POST

Los dos métodos HTTP más comunes para enviar una petición a un servidor son GET y POST. Es importante entender la utilización de cada uno.

El método GET debe ser utilizado para operaciones no-destructivas — es decir, operaciones en donde se esta "obteniendo" datos del servidor, pero no modificando. Por ejemplo, una consulta a un servicio de búsqueda podría ser una petición GET. Por otro lado, las solicitudes GET pueden ser almacenadas en la cache del navegador, pudiendo conducir a un comportamiento impredecible si no se lo espera. Generalmente, la información enviada al servidor, es enviada en una cadena de datos (en inglés *query string*).

El método POST debe ser utilizado para operaciones destructivas — es decir, operaciones en donde se está incorporando información al servidor. Por ejemplo, cuando un usuario guarda un artículo en un blog, esta acción debería utilizar POST. Por otro lado, este tipo de método no se guarda en la cache del navegador. Además, una cadena de datos puede ser parte de la URL, pero la información tiende a ser enviada de forma separada.

Tipos de Datos

Generalmente, jQuery necesita algunas instrucciones sobre el tipo de información que se espera recibir cuando se realiza una petición Ajax. En algunos casos, el tipo de dato es especificado por el nombre

del método, pero en otros casos se lo debe detallar como parte de la configuración del método:

text Para el transporte de cadenas de caracteres simples.
html Para el transporte de bloques de código HTML que serán ubicados en la página.
script Para añadir un nuevo *script* con código JavaScript a la página.
json Para transportar información en formato JSON, el cual puede incluir cadenas de caracteres, vectores y objetos.

> ***Nota***
>
> *A partir de la versión 1.4 de la biblioteca, si la información JSON no está correctamente formateada, la petición podría fallar. Visite* `http://json.org` *para obtener detalles sobre un correcto formateo de datos en JSON.*

Es recomendable utilizar los mecanismos que posea el lenguaje del lado de servidor para la generación de información en formato JSON.

jsonp Para transportar información JSON de un dominio a otro.
xml Para transportar información en formato XML.

A pesar de los diferentes tipos de datos de que se puede utilizar, es recomendable utilizar el formato JSON, ya que es muy flexible, permitiendo por ejemplo, enviar al mismo tiempo información plana y HTML.

Asincronismo

Debido a que, de forma predeterminada, las llamadas Ajax son asíncronas, la respuesta del servidor no esta disponible de forma inmediata. Por ejemplo, el siguiente código no debería funcionar:

```
var response;
$.get('foo.php', function(r) { response = r; });
console.log(response); // indefinido (undefined)
```

En su lugar, es necesario especificar una función de devolución de llamada; dicha función se ejecutará cuando la petición se haya realizado de forma correcta ya que es en ese momento cuando la respuesta del servidor esta lista.

```
$.get('foo.php', function(response) { console.log(response); });
```

Políticas de Mismo Origen y JSONP

En general, las peticiones Ajax están limitadas a utilizar el mismo protocolo (*http* o *https*), el mismo puerto y el mismo dominio de origen. Esta limitación no se aplica a los scripts cargados a través del método Ajax de jQuery.

La otra excepción es cuando se hace una petición que recibirá una respuesta en formato JSONP. En este caso, el proveedor de la respuesta debe responder la petición con un `script` que puede ser cargado utilizando la etiqueta `<script>`, evitando así la limitación de realizar peticiones desde el mismo dominio. Dicha respuesta contendrá la información solicitada, contenida en una función

Ajax y Firebug

Firebug (o el inspector WebKit que viene incluido en Chrome o Safari) son herramientas imprescindibles para trabajar con peticiones Ajax, ya que es posible observarlas desde la pestaña Consola de Firebug (o yendo a Recursos > Panel XHR desde el inspector de Webkit) y revisar los detalles de dichas peticiones. Si algo esta fallando cuando trabaja con Ajax, este es el primer lugar en donde debe dirigirse para saber cual es el problema.

0.7.3. Métodos Ajax de jQuery

Como se indicó anteriormente, jQuery posee varios métodos para trabajar con Ajax. Sin embargo, todos están basados en el método `$.ajax`, por lo tanto, su comprensión es obligatoria. A continuación se abarcará dicho método y luego se indicará un breve resumen sobre los demás métodos.

Generalmente, es preferible utilizar el método $.ajax en lugar de los otros, ya que ofrece más características y su configuración es muy comprensible.

$.ajax

El método `$.ajax` es configurado a través de un objeto, el cual contiene todas las instrucciones que necesita jQuery para completar la petición. Dicho método es particularmente útil debido a que ofrece la posibilidad de especificar acciones en caso que la petición haya fallado o no. Además, al estar configurado a través de un objeto, es posible definir sus propiedades de forma separada, haciendo que sea más fácil la reutilización del código. Puede visitar http://api.jquery.com/jQuery.ajax/ para consultar la documentación sobre las opciones disponibles en el método.

Utilizar el método $.ajax

```
$.ajax({
    // la URL para la petición
    url : 'post.php',

    // la información a enviar
    // (también es posible utilizar una cadena de datos)
    data : { id : 123 },

    // especifica si será una petición POST o GET
    type : 'GET',

    // el tipo de información que se espera de respuesta
    dataType : 'json',

    // código a ejecutar si la petición es satisfactoria;
    // la respuesta es pasada como argumento a la función
    success : function(json) {
        $('<h1/>').text(json.title).appendTo('body');
        $('<div class="content"/>')
            .html(json.html).appendTo('body');
    },

    // código a ejecutar si la petición falla;
```

```
    // son pasados como argumentos a la función
    // el objeto jqXHR (extensión de XMLHttpRequest), un texto con el estatus
   // de la petición y un texto con la descripción del error que haya dado el servidor
    error : function(jqXHR, status, error) {
        alert('Disculpe, existió un problema');
    },

    // código a ejecutar sin importar si la petición falló o no
    complete : function(jqXHR, status) {
        alert('Petición realizada');
    }
});
```

> ***Nota***
>
> *Una aclaración sobre el parámetro* `dataType`*: Si el servidor devuelve información que es diferente al formato especificado, el código fallará, y la razón de porque lo hace no siempre quedará clara debido a que la respuesta HTTP no mostrará ningún tipo de error. Cuando esté trabajando con peticiones Ajax, debe estar seguro que el servidor esta enviando el tipo de información que esta solicitando y verifique que la cabecera* `Content-type` *es exacta al tipo de dato. Por ejemplo, para información en formato JSON, la cabecera* `Content-type` *debería ser* `application/json`*.*

Opciones del método `$.ajax` El método $.ajax posee muchas opciones de configuración, y es justamente esta característica la que hace que sea un método muy útil. Para una lista completa de las opciones disponibles, puede consultar http://api.jquery.com/jQuery.ajax/; a continuación se muestran las más comunes:

async Establece si la petición será asíncrona o no. De forma predeterminada el valor es `true`. Debe tener en cuenta que si la opción se establece en `false`, la petición bloqueará la ejecución de otros códigos hasta que dicha petición haya finalizado.

cache Establece si la petición será guardada en la cache del navegador. De forma predeterminada es `true` para todos los *dataType* excepto para "*script*" y "*jsonp*". Cuando posee el valor `false`, se agrega una cadena de caracteres anti-cache al final de la URL de la petición.

complete Establece una función de devolución de llamada que se ejecuta cuando la petición esta completa, aunque haya fallado o no. La función recibe como argumentos el objeto jqXHR (en versiones anteriores o iguales a jQuery 1.4, recibe en su lugar el objeto de la petición en crudo `XMLHTTPRequest`) y un texto especificando el estatus de la misma petición (`success`, `notmodified`, `error`, `timeout`, `abort`, o `parsererror`).

context Establece el alcance en que la/las funciones de devolución de llamada se ejecutaran (por ejemplo, define el significado de `this` dentro de las funciones). De manera predeterminada `this` hace referencia al objeto originalmente pasado al método `$.ajax`.

data Establece la información que se enviará al servidor. Esta puede ser tanto un objeto como una cadena de datos (por ejemplo `foo=bar&baz=bim`.)

dataType Establece el tipo de información que se espera recibir como respuesta del servidor. Si no se especifica ningún valor, de forma predeterminada, jQuery revisa el tipo de *MIME* que posee la respuesta.

error Establece una función de devolución de llamada a ejecutar si resulta algún error en la petición. Dicha función recibe como argumentos el objeto jqXHR (en versiones anteriores o iguales a jQuery 1.4, recibe en su lugar el objeto de la petición en crudo `XMLHTTPRequest`), un texto especificando el estatus de la misma petición (`timeout`, `error`, `abort`, o `parsererror`) y un texto

con la descripción del error que haya enviado el servidor (por ejemplo `Not Found` o `Internal Server Error`).

jsonp Establece el nombre de la función de devolución de llamada a enviar cuando se realiza una petición *JSONP*. De forma predeterminada el nombre es "*callback*

success Establece una función a ejecutar si la petición ha sido satisfactoria. Dicha función recibe como argumentos el objeto jqXHR (en versiones anteriores o iguales a jQuery 1.4, recibe en su lugar el objeto de la petición en crudo `XMLHTTPRequest`), un texto especificando el estatus de la misma petición y la información de la petición (convertida a objeto JavaScript en el caso que *dataType* sea *JSON*), el estatus de la misma.

timeout Establece un tiempo en milisegundos para considerar a una petición como fallada.

traditional Si su valor es true, se utiliza el estilo de serialización de datos utilizado antes de jQuery 1.4. Para más detalles puede visitar http://api.jquery.com/jQuery.param/.

type De forma predeterminada su valor es "GET". Otros tipos de peticiones también pueden ser utilizadas (como PUT y DELETE), sin embargo pueden no estar soportados por todos los navegadores.

url Establece la URL en donde se realiza la petición.

La opción `url` es obligatoria para el método `$.ajax`;

Como se comentó anteriormente, para una lista completa de las opciones disponibles, puede consultar http://api.jquery.com/jQuery.ajax/.

> ***Nota***
>
> *A partir de la versión 1.5 de jQuery, las opciones* `beforeSend`, `success`, `error` *y* `complete` *reciben como uno de sus argumentos el objeto* `jqXHR` *siendo este una extensión del objeto nativo* `XMLHTTPRequest`*. El objeto* `jqXHR` *posee una serie de métodos y propiedades que permiten modificar u obtener información particular de la petición a realizar, como por ejemplo sobrescribir el tipo de* MIME *que posee la respuesta que se espera por parte del servidor. Para información sobre el objeto* `jqXHR` *puede consultar http://api.jquery.com/jQuery.ajax/#jqXHR.*

> ***Nota***
>
> *A partir de la versión 1.5 de jQuery, las opciones* `success`, `error` *y* `complete` *pueden recibir un vector con varias funciones de devolución, las cuales serán ejecutadas en turnos.*

Métodos Convenientes

En caso que no quiera utilizar el método `$.ajax`, y no necesite los controladores de errores, existen otros métodos más convenientes para realizar peticiones Ajax (aunque, como se indicó antes, estos están basados el método `$.ajax` con valores preestablecidos de configuración).

Los métodos que provee la biblioteca son:

$.get Realiza una petición GET a una URL provista.

$.post Realiza una petición POST a una URL provista.

$.getScript Añade un script a la página.

$.getJSON Realiza una petición GET a una URL provista y espera que un dato JSON sea devuelto.

Los métodos deben tener los siguientes argumentos, en orden:

url La URL en donde se realizará la petición. Su valor es obligatorio.

data La información que se enviará al servidor. Su valor es opcional y puede ser tanto un objeto como una cadena de datos (como `foo=bar&baz=bim`).

> ***Nota***
>
> *Esta opción no es valida para el método `$.getScript`.*

success callback Una función opcional que se ejecuta en caso que petición haya sido satisfactoria. Dicha función recibe como argumentos la información de la petición y el objeto en bruto de dicha petición.

data type El tipo de dato que se espera recibir desde el servidor. Su valor es opcional.

> ***Nota***
>
> *Esta opción es solo aplicable para métodos en que no está especificado el tipo de dato en el nombre del mismo método.*

Utilizar métodos convenientes para peticiones Ajax

```
// obtiene texto plano o html
$.get('/users.php', { userId : 1234 }, function(resp) {
    console.log(resp);
});

// añade un script a la página y luego ejecuta la función especificada
$.getScript('/static/js/myScript.js', function() {
    functionFromMyScript();
});

// obtiene información en formato JSON desde el servidor
$.getJSON('/details.php', function(resp) {
    $.each(resp, function(k, v) {
        console.log(k + ' : ' + v);
    });
});
```

`$.fn.load`

El método `$.fn.load` es el único que se puede llamar desde una selección. Dicho método obtiene el código HTML de una URL y rellena a los elementos seleccionados con la información obtenida. En conjunto con la URL, es posible especificar opcionalmente un selector, el cual obtendrá el código especificado en dicha selección.

Utilizar el método `$.fn.load` para rellenar un elemento

```
$('#newContent').load('/foo.html');
```

Utilizar el método `$.fn.load` para rellenar un elemento basado en un selector

```
$('#newContent').load('/foo.html #myDiv h1:first', function(html) {
  alert('Contenido actualizado');
});
```

0.7.4. Ajax y Formularios

Las capacidades de jQuery con Ajax pueden ser especialmente útiles para el trabajo con formularios. Por ejemplo, la extensión jQuery Form Plugin es una extensión para añadir capacidades Ajax a formularios. Existen dos métodos que debe conocer para cuando este realizando este tipo de trabajos: `$.fn.serialize` y `$.fn.serializeArray`.

Transformar información de un formulario a una cadena de datos

```
$('#myForm').serialize();
```

Crear un vector de objetos conteniendo información de un formulario

```
$('#myForm').serializeArray();

// crea una estructura como esta:
[
    { name : 'field1', value : 123 },
    { name : 'field2', value : 'hello world' }
]
```

0.7.5. Trabajar con JSONP

En los últimos tiempos, la introducción de JSONP, ha permitido la creación de aplicaciones híbridas de contenidos. Muchos sitios importantes ofrecen JSONP como servicio de información, el cual se accede a través de una API (en inglés *Application programming interface*) predefinida. Un servicio particular que permite obtener información en formato JSONP es Yahoo! Query Language, el cual se utiliza a continuación para obtener, por ejemplo, noticias sobre gatos:

Utilizar YQL y JSONP

```
$.ajax({
    url : 'http://query.yahooapis.com/v1/public/yql',

    // se agrega como parámetro el nombre de la función de devolución,
    // según se especifica en el servicio de YQL
    jsonp : 'callback',

    // se le indica a jQuery que se espera información en formato JSONP
    dataType : 'jsonp',

    // se le indica al servicio de YQL cual es la información
    // que se desea y que se la quiere en formato JSON
    data : {
        q : 'select title,abstract,url from search.news where query="cat"',
        format : 'json'
    },

    // se ejecuta una función al ser satisfactoria la petición
    success : function(response) {
        console.log(response);
    }
});
```

jQuery se encarga de solucionar todos los aspectos complejos de la petición JSONP. Lo único que debe hacer es especificar el nombre de la función de devolución (en este caso "*callback*", según lo especifica YQL) y el resultado final será como una petición Ajax normal.

0.7.6. Eventos Ajax

A menudo, querrá ejecutar una función cuando una petición haya comenzado o terminado, como por ejemplo, mostrar o ocultar un indicador. En lugar de definir estas funciones dentro de cada petición, jQuery provee la posibilidad de vincular eventos Ajax a elementos seleccionados. Para una lista completa de eventos Ajax, puede consultar http://docs.jquery.com/Ajax_Events.

Mostrar/Ocultar un indicador utilizando Eventos Ajax

```
$('#loading_indicator')
    .ajaxStart(function() { $(this).show(); })
    .ajaxStop(function() { $(this).hide(); });
```

0.7.7. Ejercicios

Cargar Contenido Externo

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/load.js`. La tarea es cargar el contenido de un artículo de blog cuando el usuario haga click en el título del ítem.

1. Crear un elementos *div* después del titulo de cada titulo de artículo de blog y guardar una referencia hacia ellos en el elemento de titulo utilizando `$.fn.data`.
2. Vincular un evento click al titulo, el cual utilizará el método `$.fn.load` para cargar en cada *div* creado el contenido apropiado desde el archivo `/ejercicios/data/blog.html`. No olvide de deshabilitar el comportamiento predeterminado del evento *click*.

Notar que cada titulo de artículo de blog en `index.html` incluye un enlace hacia el artículo. Necesitará aprovechar el atributo href de cada enlace para obtener el contenido propio de blog.html. Una vez obtenida el valor del atributo, puede utilizar la siguiente forma para procesar la información y convertirla en un selector para utilizar en conjunto con `$.fn.load`:

```
var href = 'blog.html#post1';
var tempArray = href.split('#');
var id = '#' + tempArray[1];
```

Recuerde utilizar `console.log` para asegurarse que esta realizando lo correcto.

Cargar Contenido Utilizando JSON

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/specials.js`. La tarea es mostrar los detalles del usuario para un día determinado cuando se selecciona desde la lista desplegable.

1. Añadir un elemento *div* después del formulario que se encuentra dentro del elemento *#specials*; allí será el lugar en donde se colocará la información a obtener.
2. Vincular el evento *change* en el elemento *select*; cuando se realiza un cambio en la selección, enviar una petición Ajax a `/ejercicios/data/specials.json`.
3. Cuando la petición devuelve una respuesta, utilizar el valor seleccionado en el select (ayuda: `$.fn.val`) para buscar la información correspondiente en la respuesta JSON.
4. Añadir algún HTML con la información obtenida en el *div* creado anteriormente.
5. Finalmente remover el botón *submit* del formulario.

Notar que cada vez que la selección cambia, se realiza una petición Ajax. ¿Cómo cambiaría el código para realizar solo una petición y guardar la información para aprovecharla cuando se vuelve a cambiar la opción seleccionada?

0.8. Extensiones

0.8.1. ¿Qué es una Extensión?

Una extensión de jQuery es simplemente un nuevo método que se utilizará para extender el prototipo (*prototype*) del objeto jQuery. Cuando se extiende el prototipo, todos los objetos jQuery hereden los métodos añadidos. Por lo tanto, cuando se realiza una llamada `jQuery()`, es creado un nuevo objeto jQuery con todos los métodos heredados.

El objetivo de una extensión es realizar una acción utilizando una colección de elementos, de la misma forma que lo hacen, por ejemplo, los métodos `fadeOut` o `addClass` de la biblioteca.

Usted puede realizar sus propias extensiones y utilizarlas de forma privada en su proyecto o también puede publicarlas para que otras personas le saquen provecho.

0.8.2. Crear una Extensión Básica

El código para realizar una extensión básica es la siguiente:

```
(function($){
    $.fn.myNewPlugin = function() {
        return this.each(function(){
            // realizar algo
        });
    };
})(jQuery);
```

La extensión del prototipo del objeto jQuery ocurre en la siguiente línea:

```
$.fn.myNewPlugin = function() { //...
```

La cual es encerrada en una función autoejecutable:

```
(function($){
    //...
})(jQuery);
```

Esta posee la ventaja de crear un alcance "privado", permitiendo utilizar el signo dolar sin tener la preocupación de que otra biblioteca también este utilizando dicho signo.

Por ahora, internamente la extensión queda:

```
$.fn.myNewPlugin = function() {
    return this.each(function(){
        // realizar algo
    });
};
```

Dentro de ella, la palabra clave `this` hace referencia al objeto jQuery en donde la extensión es llamada.

```
var somejQueryObject = $('#something');

$.fn.myNewPlugin = function() {
    alert(this === somejQueryObject);
};

somejQueryObject.myNewPlugin(); // muestra un alerta con 'true'
```

El objeto jQuery, normalmente, contendrá referencias a varios elementos DOM, es por ello que a menudo se los refiere como una colección.

Para interactuar con la colección de elementos, es necesario realizar un bucle, el cual se logra fácilmente con el método `each()`:

```
$.fn.myNewPlugin = function() {
    return this.each(function(){

    });
};
```

Al igual que otros métodos, `each()` devuelve un objeto jQuery, permitiendo utilizar el encadenado de métodos (`$(...).css().attr()...`). Para no romper esta convención, la extensión a crear deberá devolver el objeto `this`, para permitir seguir con el encadenamiento. A continuación se muestra un pequeño ejemplo:

```
(function($){
    $.fn.showLinkLocation = function() {
        return this.filter('a').each(function(){
            $(this).append(
                ' (' + $(this).attr('href') + ')'
            );
        });
    };
})(jQuery);

// Ejemplo de utilización:
$('a').showLinkLocation();
```

La extensión modificará todos los enlaces dentro de la colección de elementos y les añadirá el valor de su atributo `href` entre paréntesis.

```
<!-- Antes que la extensión sea llamada: -->
<a href="page.html">Foo</a>

<!-- Después que la extensión es llamada: -->
<a href="page.html">Foo (page.html)</a>
```

También es posible optimizar la extensión:

```
(function($){
    $.fn.showLinkLocation = function() {
        return this.filter('a').append(function(){
              return ' (' + this.href + ')';
        });
    };
})(jQuery);
```

El método `append` permite especificar una función de devolución de llamada, y el valor devuelto determinará que es lo que se añadirá a cada elemento. Note también que no se utiliza el método `attr`, debido a que la API nativa del DOM permite un fácil acceso a la propiedad `href`.

A continuación se muestra otro ejemplo de extensión. En este caso, no se requiere realizar un bucle en cada elemento ya que se delega la funcionalidad directamente en otro método jQuery:

```
(function($){
    $.fn.fadeInAndAddClass = function(duration, className) {
        return this.fadeIn(duration, function(){
            $(this).addClass(className);
        });
    };
})(jQuery);

// Ejemplo de utilización:
$('a').fadeInAndAddClass(400, 'finishedFading');
```

0.8.3. Encontrar y Evaluar Extensiones

Uno de los aspectos más populares de jQuery es la diversidad de extensiones que existen.

Sin embargo, la calidad entre extensiones puede variar enormemente. Muchas son intensivamente probadas y bien mantenidas, pero otras son creadas de forma apresurada y luego ignoradas, sin seguir buenas prácticas.

Google es la mejor herramienta para encontrar extensiones (aunque el equipo de jQuery este trabajando para mejorar su repositorio de extensiones). Una vez encontrada la extensión, posiblemente quiera consultar la lista de correos de jQuery o el canal IRC #jquery para obtener la opinión de otras personas sobre dicha extensión.

Asegúrese que la extensión este bien documentada, y que se ofrecen ejemplos de su utilización. También tenga cuidado con las extensiones que realizan más de lo que necesita, estas pueden llegar a sobrecargar

su página. Para más consejos sobre como detectar una extensión mediocre, puede leer el artículo (en inglés) Signs of a poorly written jQuery plugin por Remy Sharp.

Una vez seleccionada la extensión, necesitará añadirla a su página. Primero, descargue la extensión, descomprímala (si es necesario) y muévala a la carpeta de su aplicación. Finalmente insértela utilizando el elemento script (luego de la inclusión de jQuery).

0.8.4. Escribir Extensiones

A veces, desee realizar una funcionalidad disponible en todo el código, por ejemplo, un método que pueda ser llamado desde una selección el cual realice una serie de operaciones.

La mayoría de las extensiones son métodos creados dentro del espacio de nombres `$.fn`. jQuery garantiza que un método llamado sobre el objeto jQuery sea capaz de acceder a dicho objeto a través de `this`. En contrapartida, la extensión debe garantizar de devolver el mismo objeto recibido (a menos que se especifique lo contrario).

A continuación se muestra un ejemplo:

Crear una extensión para añadir y remover una clase en un elemento al suceder el evento hover

```
// definición de la extensión
(function($){
    $.fn.hoverClass = function(c) {
        return this.hover(
            function() { $(this).toggleClass(c); }
        );
    };
})(jQuery);

// utilizar la extensión
$('li').hoverClass('hover');
```

Para más información sobre el desarrollo de extensiones, puede consultar el artículo (en inglés) A Plugin Development Pattern de Mike Alsup. En dicho artículo, se desarrolla una extensión llamada `$.fn.hilight`, la cual provee soporte para la extensión metadata (en caso de estar presente) y provee un método descentralizado para establecer opciones globales o de instancias de la extensión.

El patrón de desarrollo de extensiones para jQuery explicado por Mike Alsup

```
//
// crear una clausura
//
(function($) {
  //
  // definición de la extensión
  //
  $.fn.hilight = function(options) {
    debug(this);
    // generación de las opciones principales antes de interactuar
    var opts = $.extend({}, $.fn.hilight.defaults, options);
    // se iteractua y formatea cada elemento
```

```
    return this.each(function() {
      $this = $(this);
      // generación de las opciones especificas de cada elemento
      var o = $.meta ? $.extend({}, opts, $this.data()) : opts;
      // actualización de los estilos de cada elemento
      $this.css({
        backgroundColor: o.background,
        color: o.foreground
      });
      var markup = $this.html();
      // se llama a la función de formateo
      markup = $.fn.hilight.format(markup);
      $this.html(markup);
    });
  };
  //
  // función privada para realizar depuración
  //
  function debug($obj) {
    if (window.console && window.console.log)
      window.console.log('hilight selection count: ' + $obj.size());
  };
  //
  // definir y exponer la función de formateo
  //
  $.fn.hilight.format = function(txt) {
    return '<strong>' + txt + '</strong>';
  };
  //
  // opciones predeterminadas
  //
  $.fn.hilight.defaults = {
    foreground: 'red',
    background: 'yellow'
  };
//
// fin de la clausura
//
})(jQuery);
```

0.8.5. Escribir Extensiones con Mantenimiento de Estado Utilizando Widget Factory de jQuery UI

> ***Nota***
>
> *Esta sección esta basada, con permiso del autor, en el artículo Building Stateful jQuery Plugins de Scott Gonzalez.*

Mientras que la mayoría de las extensiones para jQuery son sin mantenimiento de estado (en inglés *stateless*) — es decir, extensiones que se ejecutan solamente sobre un elemento, siendo esa su única interacción — existe un gran conjunto de funcionalidades que no se aprovechan en el patrón básico con que se desarrollan las extensiones.

Con el fin de llenar ese vacío, jQuery UI (jQuery User Interface) ha implementado un sistema más avanzado de extensiones. Este sistema permite manejar estados y admite múltiples funciones para ser expuestas en una única extensión. Dicho sistema es llamado *widget factory* y forma parte de la versión 1.8 de jQuery UI a través de `jQuery.widget`, aunque también puede ser utilizado sin depender de jQuery UI.

Para demostrar las capacidades de *widget factory*, se creará una extensión que tendrá como funcionalidad ser una barra de progreso.

Por ahora, la extensión solo permitirá establecer el valor de la barra de progreso una sola vez. Esto se realizará llamando a `jQuery.widget` con dos parámetros: el nombre de la extensión a crear y un objeto literal que contendrá las funciones soportadas por la extensión. Cuando la extensión es llamada, una instancia de ella es creada y todas las funciones se ejecutaran en el contexto de esa instancia.

Existen dos importantes diferencias en comparación con una extensión estándar de jQuery: En primer lugar, el contexto es un objeto, no un elemento DOM. En segundo lugar, el contexto siempre es un único objeto, nunca una colección.

Una simple extensión con mantenimiento de estado utilizando widget factory de jQuery UI

```
$.widget("nmk.progressbar", {
    _create: function() {
        var progress = this.options.value + "%";
        this.element
            .addClass("progressbar")
            .text(progress);
    }
});
```

El nombre de la extensión debe contener un espacio de nombres, en este caso se utiliza `nmk`. Los espacios de nombres tienen una limitación de un solo nivel de profundidad — es decir que por ejemplo, no es posible utilizar `nmk.foo`. Como se puede ver en el ejemplo, *widget factory* provee dos propiedades para ser utilizadas. La primera, `this.element` es un objeto jQuery que contiene exactamente un elemento. En caso que la extensión sea ejecutada en más de un elemento, una instancia separada de la extensión será creada por cada elemento y cada una tendrá su propio `this.element`. La segunda propiedad, `this.options`, es un conjunto de pares clave/valor con todas las opciones de la extensión. Estas opciones pueden pasarse a la extensión como se muestra a continuación:

> ***Nota***
>
> *Cuando esté realizando sus propias extensiones es recomendable utilizar su propio espacio de nombres, ya que deja en claro de donde proviene la extensión y si es parte de una colección mayor. Por otro lado, el espacio de nombres* `ui` *está reservado para las extensiones oficiales de jQuery UI.*

Pasar opciones al widget

```
$("<div></div>")
    .appendTo( "body" )
    .progressbar({ value: 20 });
```

Cuando se llama a `jQuery.widget` se extiende a jQuery añadiendo el método a `jQuery.fn` (de la misma forma que cuando se crea una extensión estándar). El nombre de la función que se añade esta basado en el nombre que se pasa a `jQuery.widget`, sin el espacio de nombres (en este caso el nombre será `jQuery.fn.progressbar`).

Como se muestra a continuación, es posible especificar valores predeterminados para cualquier opción. Estos valores deberían basarse en la utilización más común de la extensión.

Establecer opciones predeterminadas para un widget

```
$.widget("nmk.progressbar", {
    // opciones predeterminadas
    options: {
        value: 0
    },

    _create: function() {
        var progress = this.options.value + "%";
        this.element
            .addClass( "progressbar" )
            .text( progress );
    }
});
```

Añadir Métodos a un Widget

Ahora que es posible inicializar la extensión, es necesario añadir la habilidad de realizar acciones a través de métodos definidos en la extensión. Para definir un método en la extensión es necesario incluir la función en el objeto literal que se pasa a `jQuery.widget`. También es posible definir métodos "privados" anteponiendo un guión bajo al nombre de la función.

Crear métodos en el Widget

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        var progress = this.options.value + "%";
        this.element
            .addClass("progressbar")
            .text(progress);
    },

    // crear un método público
    value: function(value) {
        // no se pasa ningún valor, entonces actúa como método obtenedor
        if (value === undefined) {
            return this.options.value;
        // se pasa un valor, entonces actúa como método establecedor
        } else {
```

```
            this.options.value = this._constrain(value);
            var progress = this.options.value + "%";
            this.element.text(progress);
        }
    },

    // crear un método privado
    _constrain: function(value) {
        if (value > 100) {
            value = 100;
        }
        if (value < 0) {
            value = 0;
        }
        return value;
    }
});
```

Para llamar a un método en una instancia de la extensión, se debe pasar el nombre de dicho método a la extensión. En caso que se llame a un método que acepta parámetros, estos se deben pasar a continuación del nombre del método.

Llamar a métodos en una instancia de extensión

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar({ value: 20 });

// obtiene el valor actual
alert(bar.progressbar("value"));

// actualiza el valor
bar.progressbar("value", 50);

// obtiene el valor nuevamente
alert(bar.progressbar("value"));
```

> ***Nota***
>
> *Ejecutar métodos pasando el nombre del método a la misma función jQuery que se utiliza para inicializar la extensión puede parecer extraño, sin embargo es realizado así para prevenir la "contaminación" del espacio de nombres de jQuery manteniendo al mismo tiempo la capacidad de llamar a métodos en cadena.*

Trabajar con las Opciones del Widget

Uno de los métodos disponibles automáticamente para la extensión es `option`. Este método permite obtener y establecer opciones después de la inicialización y funciona exactamente igual que los métodos *attr* y *css* de jQuery: pasando únicamente un nombre como argumento el método funciona como obtenedor, mientras que pasando uno o más conjuntos de nombres y valores el método funciona como establecedor. Cuando es utilizado como método obtenedor, la extensión devolverá el valor actual de la opción correspondiente al nombre pasado como argumento. Por otro lado, cuando es utilizado como

un método establecedor, el método `_setOption` de la extensión será llamado por cada opción que se desea establecer.

Responder cuando una opción es establecida

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value + "%";
        this.element.text(progress);
    }
});
```

Añadir Funciones de Devolución de Llamada

Uno de las maneras más fáciles de extender una extensión es añadir funciones de devolución de llamada, para que de esta forma el usuario puede reaccionar cuando el estado de la extensión cambie. A continuación se mostrará como añadir una función de devolución de llamada a la extensión creada para indicar cuando la barra de progreso haya alcanzado el 100 %. El método `_trigger` obtiene tres parámetros: el nombre de la función de devolución, el objeto de evento nativo que inicializa la función de devolución y un conjunto de información relevante al evento. El nombre de la función de devolución es el único parámetro obligatorio, pero los otros pueden ser muy útiles si el usuario desea implementar funcionalidades personalizadas.

Proveer funciones de devolución de llamada

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
```

```
    },

    _update: function() {
        var progress = this.options.value + "%";
        this.element.text(progress);
        if (this.options.value == 100) {
            this._trigger("complete", null, { value: 100 });
        }
    }
});
```

Las funciones de devolución son esencialmente sólo opciones adicionales, por lo cual, pueden ser establecidas como cualquier otra opción. Cada vez que una función de devolución es ejecutada, un evento correspondiente se activa también. El tipo de evento se determina mediante la concatenación del nombre de la extensión y el nombre de la función de devolución. Dicha función y evento reciben dos mismos parámetros: un objeto de evento y un conjunto de información relevante al evento.

Si la extensión tendrá alguna funcionalidad que podrá ser cancelada por el usuario, la mejor manera de hacerlo es creando funciones de devolución cancelables. El usuario podrá cancelar una función de devolución o su evento asociado de la misma manera que se cancela cualquier evento nativo: llamando a `event.preventDefault()` o utilizando `return false`.

Vincular a eventos del widget

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar({
        complete: function(event, data) {
            alert( "Función de devolución" );
        }
    })
    .on("progressbarcomplete", function(event, data) {
        alert("El valor de la barra de progreso es " + data.value);
    });

bar.progressbar("option", "value", 100);
```

En profundidad: Widget Factory

Cuando se llama a `jQuery.widget`, ésta crea una función constructora para la extensión y establece el objeto literal que se pasa como el prototipo para todas las instancias de la extensión. Todas las funcionalidades que automáticamente se añaden a la extensión provienen del prototipo base del widget, el cual es definido como `jQuery.Widget.prototype`. Cuando una instancia de la extensión es creada, es guardada en el elemento DOM original utilizando `jQuery.data`, con el nombre de la extensión como palabra clave.

Debido a que la instancia de la extensión esta directamente vinculada al elemento DOM, es posible acceder a la instancia de la extensión de forma directa. Esto permite llamar a métodos directamente en la instancia de la extensión en lugar de pasar el nombre del método como una cadena de caracteres, dando la posibilidad de acceder a las propiedades de la extensión.

```
var bar = $("<div></div>")
    .appendTo("body")
```

```
    .progressbar()
    .data("progressbar" );

// llamar a un método directamente en la instancia de la extensión
bar.option("value", 50);

// acceder a propiedades en la instancia de la extensión
alert(bar.options.value);
```

Uno de los mayores beneficios de tener un constructor y un prototipo para una extensión es la facilidad de extender la extensión. El hecho de añadir o cambiar métodos en el prototipo de la extensión, permite también modificarlos en todas las instancias de la extensión. Por ejemplo, si deseamos añadir un método a la extensión de barra de progreso para permitir restablecer el progreso a 0 %, es posible hacerlo añadiendo este método al prototipo y automáticamente estará disponible para ser llamada desde cualquier instancia de la extensión.

```
$.nmk.progressbar.prototype.reset = function() {
    this._setOption("value", 0);
};
```

Limpieza

En algunos casos, tendrá sentido permitir a los usuarios aplicar y desaplicar la extensión. Esto es posible hacerlo a través del método `destroy`. Con dicho método, es posible deshacer todo lo realizado con la extensión. También éste es llamado automáticamente si el elemento vinculado a la extensión es eliminado del DOM (por lo cual también es posible utilizarlo para la "recolección de basura"). El método `destroy` predeterminado remueve el vínculo entre el elemento DOM y la instancia de la extensión

Añadir un método destroy al widget

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value + "%";
        this.element.text(progress);
        if (this.options.value == 100 ) {
            this._trigger("complete", null, { value: 100 });
```

```
        }
    },

    destroy: function() {
        this.element
            .removeClass("progressbar")
            .text("");

        // llama a la función base destroy
        $.Widget.prototype.destroy.call(this);
    }
});
```

Conclusión

La utilización de *Widget factory* es solo una manera de crear extensiones con mantenimiento de estado. Existen algunos modelos diferentes que pueden ser utilizados y cada uno posee sus ventajas y desventajas. *Widget factory* resuelve muchos problemas comunes, mejora significativamente la productividad y la reutilización de código.

0.8.6. Ejercicios

Realizar una Tabla Ordenable

Para este ejercicio, la tarea es identificar, descargar e implementar una extensión que permita ordenar la tabla existente en la página index.html. Cuando esté listo, todas las columnas de la tabla deben poder ser ordenables.

Escribir una Extensión Para Cambiar el Color de Fondo en Tablas

Abra el archivo `/ejercicios/index.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/stripe.js`. La tarea es escribir una extensión llamada "stripe" la cual podrá ser llamada desde cualquier elemento table y deberá cambiar el color de fondo de las filas impares en el cuerpo de la tabla. El color podrá ser especificado como parámetro de la extensión.

```
$('#myTable').stripe('#cccccc');
```

No olvide de devolver la tabla para que otros métodos puedan ser encadenados luego de la llamada a la extensión.

0.9. Mejores Prácticas para Aumentar el Rendimiento

Este capítulo cubre numerosas mejores prácticas de JavaScript y jQuery,sin un orden en particular. Muchas de estas prácticas están basadas en la presentación jQuery Anti-Patterns for Performance (en inglés) de Paul Irish.

0.9.1. Guardar la Longitud en Bucles

En un bucle, no es necesario acceder a la longitud de un vector cada vez que se evalúa la condición; dicho valor se puede guardar previamente en una variable.

```
var myLength = myArray.length;

for (var i = 0; i < myLength; i++) {
    // do stuff
}
```

0.9.2. Añadir Nuevo Contenido por Fuera de un Bucle

Si va a insertar muchos elementos en el DOM, hágalo todo de una sola vez, no de una por vez.

```
// mal
$.each(myArray, function(i, item) {
   var newListItem = '<li>' + item + '</li>';
   $('#ballers').append(newListItem);
});

// mejor: realizar esto
var frag = document.createDocumentFragment();

$.each(myArray, function(i, item) {
    var newListItem = '<li>' + item + '</li>';
    frag.appendChild(newListItem);
});
$('#ballers')[0].appendChild(frag);

// o esto:
var myHtml = '';

$.each(myArray, function(i, item) {
    myHtml += '<li>' + item + '</li>';
});
$('#ballers').html(myHtml);
```

0.9.3. No Repetirse

No se repita; realice las cosas una vez y sólo una, caso contrario lo estará haciendo mal.

```
// MAL
if ($eventfade.data('currently') != 'showing') {
    $eventfade.stop();
}

if ($eventhover.data('currently') != 'showing') {
    $eventhover.stop();
}
```

```
if ($spans.data('currently') != 'showing') {
    $spans.stop();
}

// BIEN
var $elems = [$eventfade, $eventhover, $spans];
$.each($elems, function(i,elem) {
    if (elem.data('currently') != 'showing') {
        elem.stop();
    }
});
```

0.9.4. Cuidado con las Funciones Anónimas

No es aconsejable utilizar de sobremanera las funciones anónimas. Estas son difíciles de depurar, mantener, probar o reutilizar. En su lugar,utilice un objeto literal para organizar y nombrar sus controladores y funciones de devolución de llamada.

```
// MAL
$(document).ready(function() {
    $('#magic').click(function(e) {
        $('#yayeffects').slideUp(function() {
            // ...
        });
    });

    $('#happiness').load(url + ' #unicorns', function() {
        // ...
    });
});

// MEJOR
var PI = {
    onReady : function() {
        $('#magic').click(PI.candyMtn);
        $('#happiness').load(PI.url + ' #unicorns', PI.unicornCb);
    },

    candyMtn : function(e) {
        $('#yayeffects').slideUp(PI.slideCb);
    },

    slideCb : function() { ... },

    unicornCb : function() { ... }
};

$(document).ready(PI.onReady);
```

0.9.5. Optimización de Selectores

La optimización de selectores es menos importante de lo que solía ser, debido a la implementación en algunos navegadores de`document.querySelectorAll()`, pasando la carga de jQuery hacia el navegador. Sin embargo, existen algunos consejos que debe tener en cuenta.

Selectores basados en ID

Siempre es mejor comenzar las selecciones con un ID.

```
// rápido
$('#container div.robotarm');

// super-rápido
$('#container').find('div.robotarm');
```

El ejemplo que utiliza `$.fn.find` es más rápido debido a que la primera selección utiliza el motor de selección interno Sizzle — mientras que la selección realizada únicamente por ID utiliza `document.getElementById()`, el cual es extremadamente rápido debido a que es una función nativa del navegador.

Especificidad

Trate de ser especifico para el lado derecho de la selección y menos específico para el izquierdo.

```
// no optimizado
$('div.data .gonzalez');

// optimizado
$('.data td.gonzalez');
```

Use en lo posible `etiqueta.clase` del lado derecho de la selección, y solo `etiqueta` o `.clase` en la parte izquierda.

```
$('.data table.attendees td.gonzalez');

// mucho mejor: eliminar la parte media de ser posible
$('.data td.gonzalez');
```

La segunda selección tiene mejor rendimiento debido a que atraviesa menos capas para buscar el elemento.

Evitar el Selector Universal

Selecciones en donde se especifica de forma implícita o explícita una selección universal puede resultar muy lento.

```
$('.buttons > *');      // muy lento
$('.buttons').children();  // mucho mejor

$('.gender :radio');       // selección universal implícita
$('.gender *:radio');      // misma forma, pero de forma explícita
$('.gender input:radio');  // mucho mejor
```

0.9.6. Utilizar la Delegación de Eventos

La delegación de eventos permite vincular un controlador de evento a un elemento contenedor (por ejemplo, una lista desordenada) en lugar de múltiples elementos contenidos (por ejemplo, los ítems de una lista). jQuery hace fácil este trabajo a través de `$.fn.live` y`$.fn.delegate`. En lo posible, es recomendable utilizar `$.fn.delegate` en lugar de `$.fn.live`, ya que elimina la necesidad de una selección y su contexto explícito reduce la carga en aproximadamente un 80 %.

Además, la delegación de eventos permite añadir nuevos elementos contenedores a la página sin tener que volver a vincular sus controladores de eventos.

```
// mal (si existen muchos items en la lista)
$('li.trigger').click(handlerFn);

// mejor: delegación de eventos con $.fn.live
$('li.trigger').live('click', handlerFn);

// mucho mejor: delegación de eventos con $.fn.delegate
// permite especificar un contexto de forma fácil
$('#myList').delegate('li.trigger', 'click', handlerFn);
```

0.9.7. Separar Elementos para Trabajar con Ellos

En lo posible, hay que evitar la manipulación del DOM. Para ayudar con este propósito, a partir de la versión 1.4, jQuery introduce `$.fn.detach` el cual permite trabajar elementos de forma separada del DOM para luego insertarlos.

```
var $table = $('#myTable');
var $parent = $table.parent();

$table.detach();
// ... se añaden muchas celdas a la tabla
$parent.append(table);
```

0.9.8. Utilizar Estilos en Cascada para Cambios de CSS en Varios Elementos

Si va a cambiar el CSS en más de 20 elementos utilizando `$.fn.css`, considere realizar los cambios de estilos añadiéndolos en una etiqueta *style*. De esta forma se incrementa un 60 % el rendimiento.

```
// correcto hasta 20 elementos, lento en más elementos
$('a.swedberg').css('color', '#asd123');
$('<style type="text/css">a.swedberg { color : #asd123 }</style>')
    .appendTo('head');
```

0.9.9. Utilizar `$.data` en Lugar de `$.fn.data`

Utilizar `$.data` en un elemento del DOM en lugar de `$.fn.data` en una selección puede ser hasta 10 veces más rápido. Antes de realizarlo, este seguro de comprender la diferencia entre un elemento DOM y una selección jQuery.

```
// regular
$(elem).data(key,value);

// 10 veces más rápido
$.data(elem,key,value);
```

0.9.10. No Actuar en Elementos no Existentes

jQuery no le dirá si esta tratando de ejecutar código en una selección vacía — esta se ejecutará como si nada estuviera mal. Dependerá de usted comprobar si la selección contiene elementos.

```
// MAL: el código a continuación ejecuta tres funciones
// sin comprobar si existen elementos
// en la selección
$('#nosuchthing').slideUp();

// Mejor
var $mySelection = $('#nosuchthing');
if ($mySelection.length) { $mySelection.slideUp(); }

// MUCHO MEJOR: añadir una extensión doOnce
jQuery.fn.doOnce = function(func){
    this.length && func.apply(this);
    return this;

}


$('li.cartitems').doOnce(function(){

    // realizar algo

});
```

Este consejo es especialmente aplicable para widgets de jQuery UI, los cuales poseen mucha carga incluso cuando la selección no contiene elementos.

0.9.11. Definición de Variables

Las variables pueden ser definidas en una sola declaración en lugar de varias.

```
// antiguo
var test = 1;
```

```
var test2 = function() { ... };
var test3 = test2(test);

// mejor forma
var test = 1,
    test2 = function() { ... },
    test3 = test2(test);
```

En funciones autoejecutables, las definiciones de variables pueden pasarse todas juntas.

```
(function(foo, bar) { ... })(1, 2);
```

0.9.12. Condicionales

```
// antiguo
if (type == 'foo' || type == 'bar') { ... }

// mejor
if (/^(foo|bar)$/.test(type)) { ... }

// búsqueda en objeto literal
if (({ foo : 1, bar : 1 })[type]) { ... }
```

0.9.13. No Tratar a jQuery como si fuera una Caja Negra

Utilice el código fuente de la biblioteca como si fuera su documentación — guarde el enlace http://bit.ly/jqsource como marcador para tener de referencia.

0.10. Organización del Código

0.10.1. Introducción

Cuando se emprende la tarea de realizar aplicaciones complejas del lado del cliente, es necesario considerar la forma en que se organizará el código. Este capitulo está dedicado a analizar algunos patrones de organización de código para utilizar en una aplicación realizada con jQuery. Además se explorará el sistema de gestión de dependencias de RequireJS.

Conceptos Clave

Antes de comenzar con los patrones de organización de código, es importante entender algunos conceptos clave:

- el código debe estar divido en unidades funcionales — módulos, servicios, etc. Y se debe evitar la tentación de tener todo en un único bloque `$(document).ready()`. Este concepto se conoce como encapsulación;
- no repetir código. Identificar piezas similares y utilizar técnicas de herencia;

- a pesar de la naturaleza de jQuery, no todas las aplicaciones JavaScript trabajan (o tienen la necesidad de poseer una representación) en el DOM;
- las unidades de funcionalidad deben tener una articulación flexible (en inglés loosely coupled) — es decir, una unidad de funcionalidad debe ser capaz de existir por si misma y la comunicación con otras unidades debe ser a través de un sistema de mensajes como los eventos personalizados o pub/sub. Por otro lado, siempre que sea posible, de debe mantener alejada la comunicación directa entre unidades funcionales.

El concepto de articulación flexible puede ser especialmente problemático para desarrolladores que hacen su primera incursión en aplicaciones complejas. Por lo tanto, si usted esta empezando a crear aplicaciones, solamente sea consciente de este concepto.

0.10.2. Encapsulación

El primer paso para la organización del código es separar la aplicación en distintas piezas.

Muchas veces, este esfuerzo suele ser suficiente para mantener al código en orden.

El Objeto Literal

Un objeto literal es tal vez la manera más simple de encapsular código relacionado. Este no ofrece ninguna privacidad para propiedades o métodos, pero es útil para eliminar funciones anónimas, centralizar opciones de configuración, y facilitar el camino para la reutilización y refactorización.

Un objeto literal

```
var myFeature = {
    myProperty : 'hello',

    myMethod : function() {
        console.log(myFeature.myProperty);
    },

    init : function(settings) {
        myFeature.settings = settings;
    },

    readSettings : function() {
        console.log(myFeature.settings);
    }
};

myFeature.myProperty; // 'hello'
myFeature.myMethod(); // registra 'hello'
myFeature.init({ foo : 'bar' });
myFeature.readSettings(); // registra { foo : 'bar' }
```

El objeto posee una propiedad y varios métodos, los cuales son públicos (es decir, cualquier parte de la aplicación puede verlos). ¿Cómo se puede aplicar este patrón con jQuery? Por ejemplo, en el siguiente código escrito en el estilo tradicional:

```
// haciendo click en un item de la lista se carga cierto contenido,
// luego utilizando el ID de dicho item se ocultan
// los items aledaños
$(document).ready(function() {
  $('#myFeature li')
    .append('<div/>')
    .click(function() {
      var $this = $(this);
      var $div = $this.find('div');
      $div.load('foo.php?item=' +
        $this.attr('id'),
        function() {
          $div.show();
          $this.siblings()
            .find('div').hide();
        }
      );
    });
});
```

Si el ejemplo mostrado representa el 100 % de la aplicación, es conveniente dejarlo como esta, ya que no amerita hacer una reestructuración. En cambio, si la pieza es parte de una aplicación más grande, estaría bien separar dicha funcionalidad de otras no relacionadas. Por ejemplo, es conveniente mover la URL a la cual se hace la petición fuera del código y pasarla al área de configuración. También romper la cadena de métodos para hacer luego más fácil la modificación.

Utilizar un objeto literal para una funcionalidad jQuery

```
var myFeature = {
    init : function(settings) {
        myFeature.config = {
            $items : $('#myFeature li'),
            $container : $('<div class="container"></div>'),
            urlBase : '/foo.php?item='
        };

        // permite sobrescribir la configuración predeterminada
        $.extend(myFeature.config, settings);

        myFeature.setup();
    },

    setup : function() {
        myFeature.config.$items
            .each(myFeature.createContainer)
            .click(myFeature.showItem);
    },

    createContainer : function() {
        var $i = $(this),
            $c = myFeature.config.$container.clone()
                      .appendTo($i);
```

```
        $i.data('container', $c);
    },

    buildUrl : function() {
        return myFeature.config.urlBase +
               myFeature.$currentItem.attr('id');
    },

    showItem : function() {
        var myFeature.$currentItem = $(this);
        myFeature.getContent(myFeature.showContent);
    },

    getContent : function(callback) {
        var url = myFeature.buildUrl();
        myFeature.$currentItem
            .data('container').load(url, callback);
    },

    showContent : function() {
        myFeature.$currentItem
            .data('container').show();
        myFeature.hideContent();
    },

    hideContent : function() {
        myFeature.$currentItem.siblings()
            .each(function() {
                $(this).data('container').hide();
            });
    }
};

$(document).ready(myFeature.init);
```

La primera característica a notar es que el código es más largo que el original — como se dijo anteriormente, si este fuera el alcance de la aplicación, utilizar un objeto literal seria probablemente una exageración.

Con la nueva organización, las ventajas obtenidas son:

- separación de cada funcionalidad en pequeños métodos. En un futuro, si se quiere cambiar la forma en que el contenido se muestra, será claro en donde habrá que hacerlo. En el código original, este paso es mucho más difícil de localizar;
- se eliminaron los usos de funciones anónimas;
- las opciones de configuración se movieron a una ubicación central;
- se eliminaron las limitaciones que poseen las cadenas de métodos, haciendo que el código sea más fácil para refactorizar, mezclar y reorganizar.

Por sus características, la utilización de objetos literales permiten una clara mejora para tramos largos de código insertados en un bloque `$(document).ready()`. Sin embargo, no son más avanzados que tener varias declaraciones de funciones dentro de un bloque `$(document).ready()`.

El Patrón Modular

El patrón modular supera algunas limitaciones del objeto literal, ofreciendo privacidad para variables y funciones, exponiendo a su vez (si se lo desea) una API pública.

El patrón modular

```
var feature =(function() {

    // variables y funciones privadas
    var privateThing = 'secret',
        publicThing = 'not secret',

        changePrivateThing = function() {
            privateThing = 'super secret';
        },

        sayPrivateThing = function() {
            console.log(privateThing);
            changePrivateThing();
        };

    // API publica
    return {
        publicThing : publicThing,
        sayPrivateThing : sayPrivateThing
    }

})();

feature.publicThing; // registra 'not secret'

feature.sayPrivateThing();
// registra 'secret' y cambia el valor
// de privateThing
```

En el ejemplo, se autoejecuta una función anónima la cual devuelve un objeto. Dentro de la función, se definen algunas variables. Debido a que ellas son definidas dentro de la función, desde afuera no se tiene acceso a menos que se pongan dentro del objeto que se devuelve. Esto implica que ningún código fuera de la función tiene acceso a la variable `privateThing` o a la función `sayPrivateThing`. Sin embargo, `sayPrivateThing` posee acceso a `privateThing` y `changePrivateThing` debido a estar definidos en el mismo alcance.

El patrón es poderoso debido a que permite tener variables y funciones privadas, exponiendo una API limitada consistente en devolver propiedades y métodos de un objeto.

A continuación se muestra una revisión del ejemplo visto anteriormente, con las mismas características, pero exponiendo un único método público del modulo, `showItemByIndex()`.

Utilizar el patrón modular para una funcionalidad jQuery

```
$(document).ready(function() {
    var feature = (function() {

        var $items = $('#myFeature li'),
            $container = $('<div class="container"></div>'),
            $currentItem,

            urlBase = '/foo.php?item=',

            createContainer = function() {
                var $i = $(this),
                    $c = $container.clone().appendTo($i);

                $i.data('container', $c);
            },

            buildUrl = function() {
                return urlBase + $currentItem.attr('id');
            },

            showItem = function() {
                var $currentItem = $(this);
                getContent(showContent);
            },

            showItemByIndex = function(idx) {
                $.proxy(showItem, $items.get(idx));
            },

            getContent = function(callback) {
                $currentItem.data('container').load(buildUrl(), callback);
            },

            showContent = function() {
                $currentItem.data('container').show();
                hideContent();
            },

            hideContent = function() {
                $currentItem.siblings()
                    .each(function() {
                        $(this).data('container').hide();
                });
            };

        $items
            .each(createContainer)
            .click(showItem);

        return { showItemByIndex : showItemByIndex };
    })();
```

```
    feature.showItemByIndex(0);
});
```

0.10.3. Gestión de Dependencias

> ***Nota***
>
> *Esta sección esta basada en la excelente documentación de RequireJS y es utilizada con el permiso de James Burke, autor de RequireJS.*

Cuando un proyecto alcanza cierto tamaño, comienza a ser difícil el manejo de los módulos de una aplicación, ya que es necesario saber ordenarlos de forma correcta, y comenzar a combinarlos en un único archivo para lograr la menor cantidad de peticiones. También es posible que se quiera cargar código "al vuelo" luego de la carga de la página.

RequireJS es una herramienta de gestión de dependencias creada por James Burke, la cual ayuda a manejar los módulos, cargarlos en un orden correcto y combinarlos de forma fácil sin tener que realizar ningún cambio. A su vez, otorga una manera fácil de cargar código una vez cargada la página, permitiendo minimizar el tiempo de descarga.

RequireJS posee un sistema modular, que sin embargo, no es necesario seguirlo para obtener sus beneficios. El formato modular de RequireJS permite la escritura de código encapsulado, incorporación de internacionalización (i18n) a los paquetes (para permitir utilizarlos en diferentes lenguajes) e incluso la utilización de servicios JSONP como dependencias.

Obtener RequireJS

La manera más fácil de utilizar RequireJS con jQuery es descargando el paquete de jQuery con RequireJS ya incorporado en él. Este paquete excluye porciones de código que duplican funciones de jQuery. También es útil descargar un ejemplo de proyecto jQuery que utiliza RequireJS.

Utilizar RequireJS con jQuery

Utilizar RequireJS es simple, tan solo es necesario incorporar en la página la versión de jQuery que posee RequireJS incorporado y a continuación solicitar los archivos de la aplicación. El siguiente ejemplo asume que tanto jQuery como los otros archivos están dentro de la carpeta `scripts/`.

Utilizar RequireJS: Un ejemplo simple

```
<!DOCTYPE html>
<html>
    <head>
        <title>jQuery+RequireJS Sample Page</title>
        <script src="scripts/require-jquery.js"></script>
        <script>require(["app"]);</script>
    </head>
    <body>
        <h1>jQuery+RequireJS Sample Page</h1>
    </body>
</html>
```

La llamada a `require([.ªpp"])` le dice a RequireJS que cargue el archivo `scripts/app.js`. RequireJS cargará cualquier dependencia pasada a `require()` sin la extensión `.js` desde el mismo directorio que en que se encuentra el archivo `require-jquery.js`, aunque también es posible especificar la ruta de la siguiente forma:

```
<script>require(["scripts/app.js"]);</script>
```

El archivo `app.js` es otra llamada a `require.js` para cargar todos los archivos necesarios para la aplicación. En el siguiente ejemplo, `app.js` solicita dos extensiones `jquery.alpha.js` y `jquery.beta.js` (no son extensiones reales, solo ejemplos). Estas extensiones están en la misma carpeta que `require-jquery.js`:

Un simple archivo JavaScript con dependencias

```
require(["jquery.alpha", "jquery.beta"], function() {
    //las extensiones jquery.alpha.js y jquery.beta.js han sido cargadas.
    $(function() {
        $('body').alpha().beta();
    });
});
```

Crear Módulos Reusables con RequireJS

RequireJS hace que sea fácil definir módulos reusables a través de `require.def()`. Un modulo RequireJS puede tener dependencias que pueden ser utilizadas para definir un módulo, además de poder devolver un valor — un objeto, una función, u otra cosa — que puede ser incluso utilizado otros módulos.

Si el módulo no posee ninguna dependencia, tan solo se debe especificar el nombre como primer argumento de `require.def()`. El segundo argumento es un objeto literal que define las propiedades del módulo. Por ejemplo:

Definición de un módulo RequireJS que no posee dependencias

```
require.def("my/simpleshirt",
    {
        color: "black",
        size: "unisize"
    }
);
```

El ejemplo debe ser guardado en el archivo `my/simpleshirt.js`.

Si el modulo posee dependencias, es posible especificarlas en el segundo argumento de `require.def()` a través de un vector) y luego pasar una función como tercer argumento. Esta función será llamada para definir el módulo una vez cargadas todos las dependencias. Dicha función recibe los valores devueltos por las dependencias como un argumento (en el mismo orden en que son requeridas en el vector) y luego la misma debe devolver un objeto que defina el módulo.

Definición de un módulo RequireJS con dependencias

```
require.def("my/shirt",
    ["my/cart", "my/inventory"],
    function(cart, inventory) {
        //devuelve un objeto que define a "my/shirt"
        return {
            color: "blue",
            size: "large"
            addToCart: function() {
                inventory.decrement(this);
                cart.add(this);
            }
        }
    }
);
```

En este ejemplo, el modulo `my/shirt` es creado. Este depende de `my/cart` y `my/inventory`. En el disco, los archivos están estructurados de la siguiente forma:

```
my/cart.js
my/inventory.js
my/shirt.js
```

La función que define `my/shirt` no es llamada hasta que `my/cart` y `my/inventory` hayan sido cargadas, y dicha función recibe como argumentos a los módulos como `cart` y `inventory`. El orden de los argumentos de la función debe coincidir con el orden en que las dependencias se requieren en el vector. El objeto devuelto define el módulo `my/shirt`. Definiendo los módulos de esta forma, `my/shirt` no existe como un objeto global, ya que múltiples módulos pueden existir en la página al mismo tiempo.

Los módulos no tienen que devolver un objeto; cualquier tipo de valor es permitido.

Definición de un módulo RequireJS que devuelve una función

```
require.def("my/title",
    ["my/dependency1", "my/dependency2"],
    function(dep1, dep2) {
        // devuelve una función para definir "my/title".
        // Este devuelve o establece
        // el titulo de la ventana
        return function(title) {
            return title ? (window.title = title) : window.title;
        }
    }
);
```

Solo un módulo debe ser requerido por archivo JavaScript.

Optimizar el Código con las Herramientas de RequireJS

Una vez incorporado RequireJS para el manejo de dependencias, la optimización del código es muy fácil. Descargue el paquete de RequireJS y colóquelo en cualquier lugar, preferentemente fuera del área de desarrollo web. Para los propósitos de este ejemplo, el paquete de RequireJS esta ubicado en una carpeta paralela al directorio `webapp` (la cual contiene la página HTML y todos los archivos JavaScript de la aplicación). La estructura de directorios es:

```
requirejs/ (utilizado para ejecutar las herramientas)
webapp/app.html
webapp/scripts/app.js
webapp/scripts/require-jquery.js
webapp/scripts/jquery.alpha.js
webapp/scripts/jquery.beta.js
```

Luego, en la carpeta en donde se encuentran `require-jquery.js` y `app.js`, crear un archivo llamado `app.build.js` con el siguiente contenido:

Archivo de configuración para las herramientas de optimización de RequireJS

```
{
    appDir: "../",
    baseUrl: "scripts/",
    dir: "../../webapp-build",
    //Comentar la siguiente línea si se desea
    //minificar el código por el compilador
    //en su modo "simple"
    optimize: "none",

    modules: [
        {
            name: "app"
        }
    ]
}
```

Para utilizar la herramienta, es necesario tener instalado Java 6. Closure Compiler es utilizado para la minificación del código (en caso que `optimize: "none"` esté comentado).

Para comenzar a procesar los archivos, abrir una ventana de comandos, dirigirse al directorio `webapp/scripts` y ejecutar:

```
# para sistemas que no son windows
../../requirejs/build/build.sh app.build.js

# para sistemas windows
..\..\requirejs\build\build.bat app.build.js
```

Una vez ejecutado, el archivo `app.js` de la carpeta `webapp-build` contendrá todo el código de `app.js` más el de `jquery.alpha.js` y `jquery.beta.js`. Si se abre el archivo `app.html` (también en la carpeta `webapp-build`) podrá notar que ninguna petición se realiza para cargar `jquery.alpha.js` y `jquery.beta.js`.

0.10.4. Ejercicios

Crear un Módulo Portlet

Abra el archivo `/ejercicios/portlets.html` en el navegador. Realice el ejercicio utilizando el archivo `/ejercicios/js/portlets.js`. El ejercicio consiste en crear una función creadora de portlet que utilice el patrón modular, de tal manera que el siguiente código funcione:

```
var myPortlet = Portlet({
    title : 'Curry',
    source : 'data/html/curry.html',
    initialState : 'open' // or 'closed'
});

myPortlet.$element.appendTo('body');
```

Cada portlet deberá ser un `div` con un título, un área de contenido, un botón para abrir/cerrar el portlet, un botón para removerlo y otro para actualizarlo. El portlet devuelto por la función deberá tener la siguiente API pública:

```
myPortlet.open(); // fuerza a abrir
myPortlet.close(); // fuerza a cerrar
myPortlet.toggle(); // alterna entre los estados abierto y cerrado
myPortlet.refresh(); // actualiza el contenido
myPortlet.destroy(); // remueve el portlet de la página
myPortlet.setSource('data/html/onions.html'); // cambia el código
```

0.11. Eventos Personalizados

0.11.1. Introducción a los Eventos Personalizados

Todos estamos familiarizados con los eventos básicos — `click`, `mouseover`, `focus`, `blur`, `submit`, etc. — que surgen a partir de la interacción del usuario con el navegador.

Los eventos personalizados permiten conocer el mundo de la programación orientada a eventos (en inglés *event-driven programming*). En este capítulo, se utilizará el sistema de eventos personalizados de jQuery para crear una simple aplicación de búsqueda en *Twitter*.

En un primer momento puede ser difícil entender el requisito de utilizar eventos personalizados, ya que los eventos convencionales permiten satisfacer todas las necesidades. Sin embargo, los eventos personalizados ofrecen una nueva forma de pensar la programación en JavaScript. En lugar de enfocarse en el elemento que ejecuta una acción, los eventos personalizados ponen la atención en el elemento en donde la acción va a ocurrir. Este concepto brinda varios beneficios:

- los comportamientos del elemento objetivo pueden ser ejecutados por diferentes elementos utilizando el mismo código;
- los comportamientos pueden ser ejecutados en múltiples, similares elementos objetivos a la vez;
- los comportamientos son asociados de forma más clara con el elemento objetivo, haciendo que el código sea más fácil de leer y mantener.

Un ejemplo es la mejor forma de explicar el asunto. Suponga que posee una lámpara incandescente en una habitación de una casa. La lámpara actualmente esta encendida. La misma es controlada por dos interruptores de tres posiciones y un *clapper* (interruptor activado por aplausos):

```
<div class="room" id="kitchen">
    <div class="lightbulb on"></div>
    <div class="switch"></div>
```

```
    <div class="switch"></div>
    <div class="clapper"></div>
</div>
```

Ejecutando el *clapper* o alguno de los interruptores, el estado de la lampara cambia. A los interruptores o al *clapper* no le interesan si la lámpara esta prendida o apagada, tan solo quieren cambiar su estado

Sin la utilización de eventos personalizados, es posible escribir la rutina de la siguiente manera:

```
$('.switch, .clapper').click(function() {
    var $light = $(this).parent().find('.lightbulb');
    if ($light.hasClass('on')) {
        $light.removeClass('on').addClass('off');
    } else {
        $light.removeClass('off').addClass('on');
    }
});
```

Por otro lado, utilizando eventos personalizados, el código queda así:

```
$('.lightbulb').on('changeState', function(e) {
    var $light = $(this);
    if ($light.hasClass('on')) {
        $light.removeClass('on').addClass('off');
    } else {
        $light.removeClass('off').addClass('on');
    }
});

$('.switch, .clapper').click(function() {
    $(this).parent().find('.lightbulb').trigger('changeState');
});
```

Algo importante ha sucedido: el comportamiento de la lámpara se ha movido, antes estaba en los interruptores y en el *clapper*, ahora se encuentra en la misma lámpara.

También es posible hacer el ejemplo un poco más interesante. Suponga que se ha añadido otra habitación a la casa, junto con un interruptor general, como se muestra a continuación:

```
<div class="room" id="kitchen">
    <div class="lightbulb on"></div>
    <div class="switch"></div>
    <div class="switch"></div>
    <div class="clapper"></div>
</div>
<div class="room" id="bedroom">
    <div class="lightbulb on"></div>
    <div class="switch"></div>
    <div class="switch"></div>
    <div class="clapper"></div>
</div>
<div id="master_switch"></div>
```

Si existe alguna lámpara prendida en la casa, es posible apagarlas a través del interruptor general, de igual forma si existen luces apagadas, es posible prenderlas con dicho interruptor. Para realizar esta tarea, se agregan dos eventos personalizados más a la lámpara: `turnOn` y `turnOff`. A través de una lógica en el evento `changeState` se decide qué evento personalizado utilizar:

```
$('.lightbulb')
    .on('changeState', function(e) {
        var $light = $(this);
        if ($light.hasClass('on')) {
            $light.trigger('turnOff');
        } else {
            $light.trigger('turnOn');
        }
    })
    .on('turnOn', function(e) {
        $(this).removeClass('off').addClass('on');
    })
    .on('turnOff', function(e) {
        $(this).removeClass('off').addClass('on');
    });

$('.switch, .clapper').click(function() {
    $(this).parent().find('.lightbulb').trigger('changeState');
});

$('#master_switch').click(function() {
    if ($('.lightbulb.on').length) {
        $('.lightbulb').trigger('turnOff');
    } else {
        $('.lightbulb').trigger('turnOn');
    }
});
```

Note como el comportamiento del interruptor general se ha vinculado al interruptor general mientras que el comportamiento de las lámparas pertenece a las lámparas.

> ***Nota***
>
> *Si esta acostumbrado a la programación orientada a objetos, puede resultar útil pensar de los eventos personalizados como métodos de objetos. En términos generales, el objeto al que pertenece el método se crea a partir del selector jQuery. Vincular el evento personalizado `changeState` a todos los elementos `$('.light')` es similar a tener una clase llamada `Light` con un método `changeState`, y luego instanciar nuevos objetos `Light` por cada elemento.*

Recapitulación: $.fn.on y $.fn.trigger

En el mundo de los eventos personalizados, existen dos métodos importantes de jQuery: `$.fn.on` y `$.fn.trigger`. En el capítulo dedicado a eventos se explicó la utilización de estos dos métodos para trabajar con eventos del usuario; en este capítulo es importante recordar 2 puntos:

- el método `$.fn.on` toma como argumentos un tipo de evento y una función controladora de evento. Opcionalmente, puede recibir información asociada al evento como segundo argumento,

desplazando como tercer argumento a la función controladora de evento. Cualquier información pasada estará disponible a la función controladora a través de la propiedad `data` del objeto del evento. A su vez, la función controladora recibe el objeto del evento como primer argumento;

- el método `$.fn.trigger` toma como argumentos el tipo de evento y opcionalmente, puede tomar un vector con valores. Estos valores serán pasados a la función controladora de eventos como argumentos luego del objeto del evento.

A continuación se muestra un ejemplo de utilización de `$.fn.on` y `$.fn.trigger` en donde se utiliza información personalizada en ambos casos:

```
$(document).on('myCustomEvent', { foo : 'bar' }, function(e, arg1, arg2) {
    console.log(e.data.foo); // 'bar'
    console.log(arg1); // 'bim'
    console.log(arg2); // 'baz'
});

$(document).trigger('myCustomEvent', [ 'bim', 'baz' ]);
```

Un Ejemplo de Aplicación

Para demostrar el poder de los eventos personalizados, se desarrollará una simple herramienta para buscar en *Twitter*. Dicha herramienta ofrecerá varias maneras para que el usuario realice una búsqueda: ingresando el término a buscar en una caja de texto o consultando los "temas de moda" de *Twitter*.

Los resultados de cada término se mostrarán en un contenedor de resultados; dichos resultados podrán expandirse, colapsarse, refrescarse y removerse, ya sea de forma individual o conjunta.

El resultado final de la aplicación será el siguiente:

Figura 11.1. La aplicación finalizada

Iniciación Se empieza con un HTML básico:

```
<h1>Twitter Search</h1>
<input type="button" id="get_trends"
    value="Load Trending Terms" />

<form>
    <input type="text" class="input_text"
        id="search_term" />
    <input type="submit" class="input_submit"
        value="Add Search Term" />
</form>

<div id="twitter">
    <div class="template results">
        <h2>Search Results for
        <span class="search_term"></span></h2>
    </div>
</div>
```

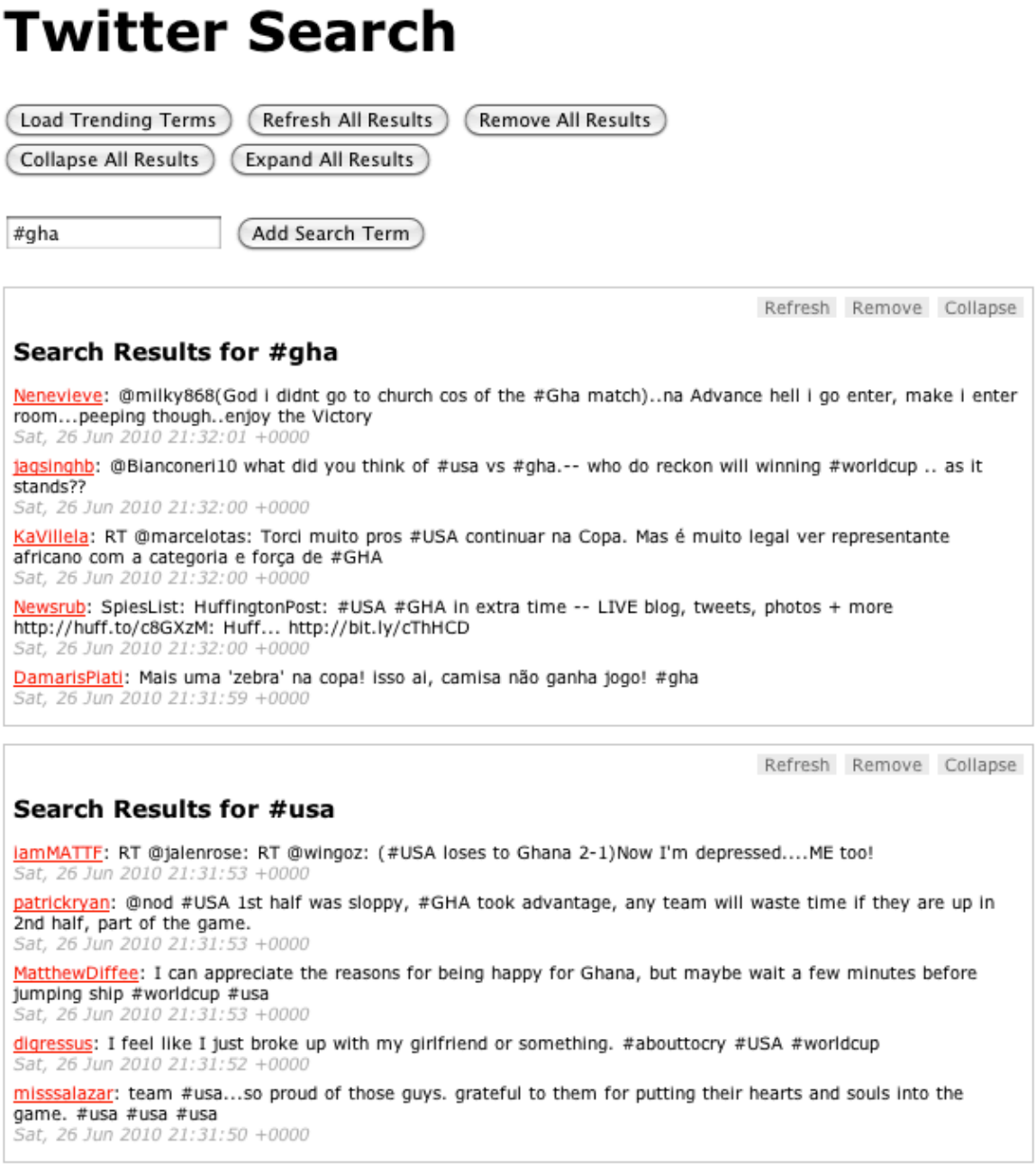

Figura 1: *La aplicación finalizada*

El HTML posee un contenedor (#twitter) para el widget, una plantilla para los resultados (oculto con CSS) y un simple formulario en donde el usuario puede escribir el término a buscar.

Existen dos tipos de elementos en los cuales actuar: los contenedores de resultados y el contenedor *Twitter*.

Los contenedores de resultados son el corazón de la aplicación. Se creará una extensión para preparar cada contenedor una vez que éste se agrega al contenedor *Twitter*. Además, entre otras cosas, la extensión vinculará los eventos personalizados por cada contenedor y añadirá en la parte superior derecha de cada contenedor botones que ejecutarán acciones. Cada contenedor de resultados tendrá los siguientes eventos personalizados:

refresh Señala que la información del contenedor se esta actualizando y dispara la petición que busca los datos para el término de búsqueda.

populate Recibe la información JSON y la utiliza para rellenar el contenedor.

remove Remueve el contenedor de la página luego de que el usuario confirme la acción. Dicha confirmación puede omitirse si se pasa `true` como segundo argumento del controlador de evento. El evento además remueve el término asociado con el contenedor de resultados del objeto global que contiene los términos de búsqueda.

collapse Añade una clase al contenedor, la cual ocultará el resultado a través de CSS. Además cambiará el botón de "Colapsar" a "Expandir".

expand Remueve la clase del contenedor que añade el evento *collapse*. Además cambiará el botón de "Expandir" a "Colapsar".

Además, la extensión es responsable de añadir los botones de acciones al contenedor, vinculando un evento `click` a cada botón y utilizando la clase de cada ítem para determinar qué evento personalizado será ejecutado en cada contenedor de resultados.

```
$.fn.twitterResult = function(settings) {
    return this.each(function() {
        var $results = $(this),
            $actions = $.fn.twitterResult.actions =
                $.fn.twitterResult.actions ||
                $.fn.twitterResult.createActions(),
            $a = $actions.clone().prependTo($results),
            term = settings.term;

        $results.find('span.search_term').text(term);

        $.each(
            ['refresh', 'populate', 'remove', 'collapse', 'expand'],
            function(i, ev) {
                $results.bind(
                    ev,
                    { term : term },
                    $.fn.twitterResult.events[ev]
                );
            }
        );

        // utiliza la clase de cada acción para determinar
        // que evento se ejecutará en el panel de resultados
```

```
        $a.find('li').click(function() {
            // pasa el elemento <li> clickeado en la función
            // para que se pueda manipular en caso de ser necesario
            $results.trigger($(this).attr('class'), [ $(this) ]);
        });
    });
};

$.fn.twitterResult.createActions = function() {
    return $('<ul class="actions" />').append(
        '<li class="refresh">Refresh</li>' +
        '<li class="remove">Remove</li>' +
        '<li class="collapse">Collapse</li>'
    );
};

$.fn.twitterResult.events = {
    refresh : function(e) {
           // indica que los resultados se estan actualizando
        var $this = $(this).addClass('refreshing');

        $this.find('p.tweet').remove();
        $results.append('<p class="loading">Loading ...</p>');

        // obtiene la información de Twitter en formato jsonp
        $.getJSON(
            'http://search.twitter.com/search.json?q=' +
                escape(e.data.term) + '&rpp=5&callback=?',
            function(json) {
                $this.trigger('populate', [ json ]);
            }
        );
    },

    populate : function(e, json) {
        var results = json.results;
        var $this = $(this);

        $this.find('p.loading').remove();

        $.each(results, function(i,result) {
            var tweet = '<p class="tweet">' +
                '<a href="http://twitter.com/' +
                result.from_user +
                '">' +
                result.from_user +
                '</a>: ' +
                result.text +
                ' <span class="date">' +
                result.created_at +
                '</span>' +
            '</p>';
```

```
            $this.append(tweet);
        });

        // indica que los resultados
        // ya se han actualizado
        $this.removeClass('refreshing');
    },

    remove : function(e, force) {
        if (
            !force &&
            !confirm('Remove panel for term ' + e.data.term + '?')
        ) {
            return;
        }
        $(this).remove();

        // indica que ya no se tendrá
        // un panel para el término
        search_terms[e.data.term] = 0;
    },

    collapse : function(e) {
        $(this).find('li.collapse').removeClass('collapse')
            .addClass('expand').text('Expand');

        $(this).addClass('collapsed');
    },

    expand : function(e) {
        $(this).find('li.expand').removeClass('expand')
            .addClass('collapse').text('Collapse');

        $(this).removeClass('collapsed');
    }
};
```

El contenedor *Twitter*, posee solo dos eventos personalizados:

getResults Recibe un término de búsqueda y comprueba si ya no existe un contenedor de resultados para dicho término. En caso de no existir, añade un contenedor utilizando la plantilla de resultados, lo configura utilizando la extensión `$.fn.twitterResult` (mostrada anteriormente) y luego ejecuta el evento `refresh` con el fin de cargar correctamente los resultados. Finalmente, guarda el término buscado para no tener volver a pedir los datos sobre la búsqueda.

getTrends Consulta a *Twitter* el listado de los 10 primeros "términos de moda", interactúa con ellos y ejecuta el evento `getResults` por cada uno, de tal modo que añade un contenedor de resultados por cada término.

Vinculaciones en el contenedor *Twitter*:

```
$('#twitter')
```

```
    .on('getResults', function(e, term) {
        // se comprueba que ya no exista una caja para el término
        if (!search_terms[term]) {
            var $this = $(this);
            var $template = $this.find('div.template');

            // realiza una copia de la plantilla
            // y la inserta como la primera caja de resultados
            $results = $template.clone().
                removeClass('template').
                insertBefore($this.find('div:first')).
                twitterResult({
                    'term' : term
                });

            // carga el contenido utilizando el evento personalizado "refresh"
            // vinculado al contenedor de resultados
            $results.trigger('refresh');
            search_terms[term] = 1;
        }
    })
    .on('getTrends', function(e) {
        var $this = $(this);
      $.getJSON('http://api.twitter.com/1/trends/1.json?callback=?', function(json) {
                var trends = json[0].trends;
                $.each(trends, function(i, trend) {
                    $this.trigger('getResults', [ trend.name ]);
                });
            });
    });
```

Hasta ahora, se ha escrito una gran cantidad de código que no realiza nada, lo cual no esta mal. Se han especificado todos los comportamientos que se desean para los elementos núcleos y se ha creado un sólido marco para la creación rápida de la interfaz.

A continuación, se conecta la caja de búsqueda y el botón para cargar los "Temas de moda". En la caja de texto, se captura el término ingresado y se pasa al mismo tiempo que se ejecuta el evento `getResults`. Por otro lado, haciendo click en el botón para cargar los "Temas de moda", se ejecuta el evento `getTrends`:

```
$('form').submit(function(e) {
    e.preventDefault();
    var term = $('#search_term').val();
    $('#twitter').trigger('getResults', [ term ]);
});

$('#get_trends').click(function() {
    $('#twitter').trigger('getTrends');
});
```

Añadiendo botones con un ID apropiado, es posible remover, colapsar, expandir y refrescar todos los contenedores de resultados al mismo tiempo. Para el botón que remueve el contenedor, notar que se

esta pasando `true` al controlador del evento como segundo argumento, indicando que no se desea una confirmación del usuario para remover el contenedor.

```
$.each(['refresh', 'expand', 'collapse'], function(i, ev) {
    $('#' + ev).click(function(e) { $('#twitter div.results').trigger(ev); });
});

$('#remove').click(function(e) {
    if (confirm('Remove all results?')) {
        $('#twitter div.results').trigger('remove', [ true ]);
    }
});
```

Conclusión Los eventos personalizados ofrecen una nueva manera de pensar el código: ellos ponen el énfasis en el objetivo de un comportamiento, no en el elemento que lo activa. Si se toma el tiempo desde el principio para explicar las piezas de su aplicación, así como los comportamientos que esas piezas necesitan exhibir, los eventos personalizados proveen una manera poderosa para "hablar" con esas piezas, ya sea de una en una o en masa.

Una vez que los comportamientos se han descripto, se convierte en algo trivial ejecutarlos desde cualquier lugar, lo que permite la rápida creación y experimentación de opciones de interfaz. Finalmente, los eventos personalizados también permiten mejorar la lectura del código y su mantenimiento, haciendo clara la relación entre un elemento y su comportamiento.

Puede ver la aplicación completa en los archivos `demos/custom-events/custom-events.html` y `demos/custom-events/js/custom-events.js` del material que componen este libro.

0.12. Funciones y ejecuciones diferidas a través del objeto `$.Deferred`

0.12.1. Introducción

A partir de la versión 1.5 de jQuery, la biblioteca introdujo una nueva utilidad: El objeto diferido `$.Deferred` (en inglés *Deferred Object*). Este objeto introduce nuevas formas para la invocación y ejecución de las funciones de devolución (*callbacks*), permitiendo crear aplicaciones más robustas y flexibles. Para más detalles sobre `$.Deferred`, puede consultar http://api.jquery.com/category/deferred-object/.

0.12.2. El objeto diferido y Ajax

El caso más común en donde se puede apreciar la utilidad del objeto diferido es en el manejo de las funciones de devolución en peticiones Ajax.

Según se pudo apreciar en el capítulo dedicado, una manera de invocar una petición Ajax es:

Manera tradicional de utilizar el método `$.ajax`

```
$.ajax({
   // la URL para la petición
   url : 'post.php',
```

```
    // funciónes de devolución a ejecutar
    // en caso que la petición haya sido
    // satisfactoria, con error y/o completada
    success : function(data) {
        alert('Petición realizada satisfactoriamente');
    },
    error : function(jqXHR, status, error) {
        alert('Disculpe, existió un problema');
    },
    complete : function(jqXHR, status) {
        alert('Petición realizada');
    }
});
```

Como se puede observar, las funciones de devolución son configuradas dentro del mismo objeto `$.ajax`. Esta manera es incomoda y poco flexible ya que **no permite desacoplar las funciones de devolución de la misma petición Ajax**. Y en grandes aplicaciones esto puede llegar a ser un problema.

El objeto diferido nos permite reescribir el código anterior de la siguiente manera:

El objeto diferido en una petición Ajax

```
// dentro de una variable se define
// la configuración de la petición ajax
var ajax = $.ajax({
    url : 'post.php'
});

// a través del método done() ejecutamos
// la función de devolución satisfactoria (sucess)
ajax.done(function(){
    alert('Petición realizada satisfactoriamente');
});

// a través del método fail() ejecutamos
// la función de devolución de error (error)
ajax.fail(function(){
    alert('Disculpe, existió un problema');
});

// a través del método always() ejecutamos
// la función de devolución de petición completada (complete)
ajax.always(function(){
    alert('Petición realizada');
});
```

A través de los métodos `deferred.done`, `deferred.fail` y `deferred.always` es posible desacoplar las funciones de devolución de la misma petición Ajax, permitiendo un manejo más cómodo de las mismas.

Notar que en en ningún momento se llama al objeto diferido `$.Deferred`. Esto es porque jQuery ya lo incorpora implícitamente dentro del manejo del objeto `$.ajax`. Más adelante se explicará como utilizar al objeto `$.Deferred` de manera explícita.

De la misma forma es posible crear colas de funciones de devolución o atarlas a diferentes lógicas/acciones:

Colas de funciones de devolución en una petición Ajax

```
// definición de la petición Ajax
var ajax = $.ajax({
    url : 'post.php'
});

// primera función de devolución a ejecutar
ajax.done(function(){
    alert('Primera función de devolución en caso satisfactorio');
});

// segunda función de devolución a ejecutar
// inmediatamente después de la primera
ajax.done(function(){
    alert('Segunda función de devolución en caso satisfactorio');
});

// si el usuario hace click en #element se
// agrega una tercera función de devolución
$('#element').click(function(){

    ajax.done(function(){
        alert('Tercera función de devolución si el usuario hace click');
    });

});

// en caso que exista un error se define otra
// función de devolución
ajax.fail(function(){
    alert('Disculpe, existió un problema');
});
```

Al ejecutarse la petición Ajax, y en caso de que ésta haya sido satisfactoria, se ejecutan dos funciones de devolución, una detrás de la otra. Sin embargo si el usuario hace click en `#element` se agrega una tercera función de devolución, la cual también se ejecuta inmediatamente, sin volver a realizar la petición Ajax. Esto es porque el objeto diferido (que se encuentra implícitamente en la variable `ajax`) ya tiene información asociada sobre que la petición Ajax se realizó correctamente.

`deferred.then`

Otra manera de utilizar los métodos `deferred.done` y `deferred.fail` es a través de `deferred.then`, el cual permite definir en un mismo bloque de código las funciones de devolución a suceder en los casos satisfactorios y erróneos.

Utilización del método `deferred.then`

```
// definición de la petición Ajax
```

```
var ajax = $.ajax({
    url : 'post.php'
});

// el método espera dos funciones de devolución
ajax.then(

    // la primera es la función de devolución satisfactoria
    function(){
        alert('Petición realizada satisfactoriamente');
    },

    // la segunda es la función de devolución errónea
    function(){
        alert('Disculpe, existió un problema');
    }

);
```

0.12.3. Creación de objetos diferidos con `$.Deferred`

Así como es posible desacoplar las funciones de devolución en una petición Ajax, también es posible realizarlo en otras funciones utilizando de manera explícita el objeto `$.Deferred`.

Por ejemplo, una función que verifica si un número es par, de la manera tradicional puede escribirse de la siguiente manera:

Función sin utilizar el objeto `$.Deferred`

```
// función que calcula si un número entero es par o impar
var isEven = function(number) {

    if (number%2 == 0){
        return true;
    } else {
        return false;
    }

}

// si es par registra un mensaje,
// en caso contrario registra otro
if (isEven(2)){
    console.log('Es par');
} else {
    console.log('Es impar');
}
```

Utilizando el objeto `$.Deferred`, el mismo ejemplo puede reescribirse de la siguiente forma:

Función utilizando el objeto `$.Deferred`

```
// función que calcula si un número entero es par o impar
var isEven = function(number) {

    // guarda en una variable al objeto $.Deferred()
    var dfd = $.Deferred();

    // si es par, resuelve al objeto utilizando deferred.resolve,
    // caso contrario, lo rechaza utilizando deferred.reject
    if (number%2 == 0){
        dfd.resolve();
    } else {
        dfd.reject();
    }

    // devuelve al objeto diferido con su estado definido
    return dfd.promise();

}

// con deferred.then se manejan las funciones de devolución
// en los casos que el numero sea par o impar
isEven(2).then(

    // la primera es la función de devolución satisfactoria
    function(){
        console.log('Es par');
    },

    // la segunda es la función de devolución errónea
    function(){
        console.log('Es impar');
    }

);
```

Los métodos `deferred.resolve` y `deferred.reject` permiten **definir el estado interno** del objeto `$.Deferred()`. Esta definición **es permanente, es decir, no es posible modificarla después** y es lo que permite manejar el comportamiento y ejecución de las funciones de devolución posteriores para cada uno de los casos.

Notar que la función `isEven` devuelve el método `deferred.promise`. El mismo es una versión del objeto diferido, pero que sólo permite leer su estado o añadir nuevas funciones de devolución.

> ***Nota***
>
> *En los ejemplos que utilizaban Ajax mostrados anteriormente, los métodos `deferred.resolve` y `deferred.reject` son llamados de manera interna por jQuery dentro de la configuración `sucess` y `error` de la petición. Por eso mismos se decía que el objeto diferido estaba incorporado implícitamente dentro del objeto `$.ajax`.*

Los métodos `deferred.resolve` y `deferred.reject` además permiten devolver valores para ser utilizados por las funciones de devolución.

Función con `deferred.resolve` y `deferred.reject` devolviendo valores reutilizables

```
// función que calcula si un numero entero es par o impar
var isEven = function(number) {

    var dfd = $.Deferred();

    // resuelve o rechaza al objeto utilizando
    // y devuelve un texto con el resultado
    if (number%2 == 0){
        dfd.resolve('El número ' + number + ' es par');
    } else {
        dfd.reject('El número ' + number + ' es impar');
    }

    // devuelve al objeto diferido con su estado definido
    return dfd.promise();

}

isEven(2).then(

    function(result){
        console.log(result); // Registra 'El número 2 es par'
    },

    function(result){
        console.log(result);
    }

);
```

Nota

Es posible determinar el estado de un objeto diferido a través del método `deferred.state`*. El mismo devuelve un string con alguno de estos tres valores:* `pending`*,* `resolved` *o* `rejected`*. Para más detalles sobre* `deferred.state`*, puede consultar http://api.jquery.com/deferred.state/.*

`deferred.pipe`

Existen casos en que se necesita modificar el estado de un objeto diferido o filtrar la información que viene asociada. Para estos casos existe `deferred.pipe`. Su funcionamiento es similar a `deferred.then`, con la diferencia que `deferred.pipe` devuelve un nuevo objeto diferido modificado a través de una función interna.

Función filtrando valores utilizando `deferred.pipe`

```
// función que calcula si un número entero es par o impar
var isEven = function(number) {

    var dfd = $.Deferred();

    if (number%2 == 0){
```

```
        dfd.resolve(number);
    } else {
        dfd.reject(number);
    }

    return dfd.promise();

}

// vector con una serie de números pares e impares
var numbers = [0, 2, 9, 10, 5, 8, 12];

// a través de deferred.pipe se pregunta si número se encuentra
// dentro del vector numbers
isEven(2).pipe(

    function(number){

        // crea un nuevo objeto diferido
        var dfd = $.Deferred();

        if($.inArray(number, numbers) !== -1){
            dfd.resolve();
        } else {
            dfd.reject();
        }

        // devuelve un nuevo objeto diferido
        return dfd.promise();

    }

).then(

    function(){
        // al estar dentro del vector numbers y ser par,
        // se registra este mensaje
        console.log('El número es par y se encuentra dentro de numbers');
    },

    function(){
        console.log('El número es impar o no se encuentra dentro de numbers');
    }

);
```

Para más detalles sobre `deferred.pipe`, puede consultar http://api.jquery.com/deferred.pipe/.

`$.when`

El método `$.when` permite ejecutar funciones de devolución, cuando uno o más objetos diferidos posean algún estado definido.

Un caso común de utilización de `$.when` es cuando se quiere verificar que dos peticiones Ajax separadas se han realizado.

Utilización de `$.when`

```
// primera petición ajax
var comments = $.ajax({
    url : '/echo/json/'
});

// segunda petición ajax
var validation = $.ajax({
    url : '/echo/json/'
});

// cuando las dos peticiones sean realizadas
// ejecuta alguna función de devolución definida
// dentro de deferred.then
$.when(comments, validation).then(
    function(){
        alert('Peticiones realizadas');
    },
    function(){
        alert('Disculpe, existió un problema');
    }
);
```

Para más detalles sobre `$.when`, puede consultar http://api.jquery.com/jQuery.when/.

Derechos de autor