JavaScript

Sesión 4 - **JavaScript Avanzado**

Índice

4.1 *Closures*

- JavaScript permite definir funciones dentro de otras funciones.
  - Una función interna tiene acceso a sus parámetros y variables, y también puede acceder a los parámetros y variables de la función a la que está anidada (la externa).
  - La función interna contiene un enlace al contexto exterior, el cual se conoce como **closure**
- La función definida en el *closure* "recuerda" el entorno en el que se ha creado.
- Ofrecen un enorme poder expresivo.
- Es un tipo especial de objeto que combina dos cosas: una **función**, y el **entorno** en que se creó esa función.
  - El entorno está formado por las variables locales que estaban dentro del alcance en el momento que se creó el closure.

Ejemplo *closure* I

```
function inicia() {
  var nombre = "Batman";
  function muestraNombre() {
    console.log(nombre);
  }
  muestraNombre();
}
inicia();
```

- La función interna `muestraNombre()` sólo está disponible en el cuerpo de la función `inicia()`, y en vez de tener una variable propia, reutiliza la variable `nombre` declarada en la función externa

Ejemplo *closure* II

```
function inicia() {
  var nombre = "Batman";
  function muestraNombre() {
    console.log(nombre);
  }
  muestraNombre();
}
inicia();
```

```
function creaFunc() {
  var nombre = "Batman";
  function muestraNombre() {
    console.log(nombre);
  }
  return muestraNombre;
}

var miFunc = creaFunc();
miFunc();
```

- La función externa devuelve la función interna `muestraNombre()` antes de ejecutarla.
- miFunc se ha convertido en un *closure* que incorpora tanto la función `muestraNombre` como la cadena `"Batman"` que existían cuando se creó el *closure*.
- Hemos creado un *closure* haciendo que la función padre devuelva una función interna.

Closures con parámetros

- La función externa devuelve una función interna que recibe parámetros

```
function creaSumador(x) {
  return function(y) {
    return x + y;
  };
}

var suma5 = creaSumador(5);
var suma10 = creaSumador(10);

console.log(suma5(2));  // 7
console.log(suma10(2)); // 12
```

- Tanto `suma5` como `suma10` son *closures*: comparten la misma definición de cuerpo de función, pero almacenan diferentes entornos.

Alcance en *closures* I

- Las funciones internas puede tener sus propias variables, cuyo alcance se restringe a la propia función

```
function funcExterna() {
  function funcInterna() {
    var varInterna = 0;
    varInterna++;
    console.log('varInterna = ' + varInterna);
  }
  return funcInterna;
}
var ref = funcExterna();
ref();
ref();
var ref2 = funcExterna();
ref2();
ref2();
```

```
varInterna = 1
varInterna = 1
varInterna = 1
varInterna = 1
```

Alcance en *closures* II

- Las funciones internas pueden referenciar a las variables globales del mismo modo que cualquier otro tipo de función

```
var varGlobal = 0;
function funcExterna() {
  function funcInterna() {
    varGlobal++;
    console.log('varGlobal = ' + varGlobal);
  }
  return funcInterna;
}
var ref = funcExterna();
ref();
ref();
var ref2 = funcExterna();
ref2();
ref2();
```

```
varGlobal = 1
varGlobal = 2
varGlobal = 3
varGlobal = 4
```

Alcance en *closures* III

- ¿Qué ocurre si la variable es local a la función externa?
- La función interna hereda el alcance del padre → podemos referenciar a dicha variable.
  - La variable externa permanece atada a la función interna.
  - Cuando la función interna finaliza, la memoria no se libera, ya que todavía la necesita el *closure*.

```
function funcExterna() {
  var varExterna = 0;
  function funcInterna() {
    varExterna++;
    console.log('varExterna = ' + varExterna);
  }
  return funcInterna;
}
var ref = funcExterna();
ref();
ref();
var ref2 = funcExterna();
ref2();
ref2();
```

```
varExterna = 1
varExterna = 2
varExterna = 1
varExterna = 2
```

Interacciones entre *closures*

```
function funcExterna() {
  var varExterna = 0;
  function funcInterna1() {
    varExterna++;
    console.log('(1) varExterna = ' + varExterna);
  }
  function funcInterna2() {
    varExterna += 2;
    console.log('(2) varExterna = ' + varExterna);
  }
  return {'func1': funcInterna1, 'func2': funcInterna2};
}
var ref = funcExterna();
ref.func1();
ref.func2();
ref.func1();
var ref2 = funcExterna();
ref2.func1();
ref2.func2();
ref2.func1();
```

```
(1) varExterna = 1
(2) varExterna = 3
(1) varExterna = 4
(1) varExterna = 1
(2) varExterna = 3
(1) varExterna = 4
```

- Las dos funciones internas referencian a la misma variable local
- Comparten el mismo entorno de closure

Uso de *closures*

- Un *closure* permite asociar algunos datos (el entorno) con una función que opera sobre esos datos.
  - Paralelismos con la programación orientada a objetos
- Podemos utilizar un *closure* en cualquier lugar en el que normalmente usaríamos **un objeto con sólo un método**.
- Se emplean en gran medida en la **gestión de eventos**, al asociar el código del manejador como un *callback*

Métodos privados

- JavaScript no soporta los métodos privados
- Se pueden emular mediante *closures*.
- Patrón **Módulo**

- Elementos privados
  - `contadorPriv`
  - `cambiar()`
- Las funciones públicas pueden acceder a los elementos privados
- Similar a métodos estáticos

```
var Contador = (function() {
  var contadorPriv = 0;
  function cambiar(val) {
    contadorPriv += val;
  }
  return {
    incrementar: function() {
      cambiar(1);
    },
    decrementar: function() {
      cambiar(-1);
    },
    valor: function() { return contadorPriv;}
  }
})();

console.log(Contador.valor()); // 0
Contador.incrementar();
Contador.incrementar();
console.log(Contador.valor()); // 2
Contador.decrementar();
console.log(Contador.valor()); // 1
```

Ejemplo factoría contador

- Si en vez de utilizar una IIFE, incluimos el código dentro de una función, ésta se convierte en una factoría
- Cada llamada a `crearContador()` crea un nuevo entorno

```
var crearContador = function() {
  var contadorPriv = 0;
  function cambiar(val) { contadorPriv += val; }
  return {
    incrementar: function() { cambiar(1); },
    decrementar: function() { cambiar(-1); },
    valor: function() { return contadorPriv; }
  }
};

var Contador1 = crearContador();
var Contador2 = crearContador();
alert(Contador1.valor()); // 0
Contador1.incrementar();
Contador1.incrementar();
console.log(Contador1.valor()); // 2
Contador1.decrementar();
console.log(Contador1.valor()); // 1
console.log(Contador2.valor()); // 0
```

Closures dentro de bucles

```
function muestraAyuda(textoAyuda) {
  document.getElementById('ayuda').innerHTML = textoAyuda;
}

function setupAyuda() {
  var textosAyuda = [
    {'id': 'email', 'ayuda': 'Dirección de correo electrónico'},
    {'id': 'nombre', 'ayuda': 'Nombre completo'},
    {'id': 'edad', 'ayuda': 'Edad (debes tener más de 16 años)'}
  ];

  for (var i = 0; i < textosAyuda.length; i++) {
    var elem = textosAyuda[i];
    document.getElementById(elem.id).onfocus = function() {
      muestraAyuda(elem.ayuda);
    }
  }
}

setupAyuda();
```

```
<p id="ayuda">La ayuda aparecerá aquí</p>
<p>Correo electrónico: <input type="email" id="email" name="email"></p>
<p>Nombre: <input type="text" id="nombre" name="nombre"></p>
<p>Edad: <input type="number" id="edad" name="edad"></p>
```

Solución a Closures en Bucles

```
function muestraAyuda(textoAyuda) {
  document.getElementById('ayuda').innerHTML = textoAyuda;
}

function crearCallbackAyuda(ayuda) {
  return function() { muestraAyuda(ayuda); };
}


function setupAyuda() {
  var textosAyuda = [
    {'id': 'email', 'ayuda': 'Dirección de correo electrónico'},
    {'id': 'nombre', 'ayuda': 'Nombre completo'},
    {'id': 'edad', 'ayuda': 'Edad (debes tener más de 16 años)'}
  ];

  for (var i = 0; i < textosAyuda.length; i++) {
    var elem = textosAyuda[i];
    document.getElementById(elem.id).onfocus = crearCallbackAyuda(elem.ayuda);
  }
}
```

Autoevaluación: Closures

```
var nodos = document.getElementsByTagName('button');
for (var i=0, len = nodos.length; i<len; i++) {
  nodos[i].addEventListener('click', function() {
    console.log("Click en elemento número " + i);
  });
}
```

```
var nodos = document.getElementsByTagName('button');
for (var i=0, len = nodos.length; i<len; i++) {
  nodos[i].addEventListener('click', (function(i) {
    return function() {
      console.log("Click en elemento número " + i);
    }
  })(i));
}
```

4.2 Módulos

- Permiten reutilizar código entre diferentes aplicaciones

- Reservar las **variables globales** para los objetos que tienen relevancia a nivel de sistema
  - Sin nombres ambiguos

    Minimizar el riesgo de colisión con otros objetos.
  - Evitar la creación de objetos globales, a no ser que sea estrictamente necesarios.

- Excepción → crear un pequeño conjunto de objetos globales que harán de **espacios de nombre** para los módulos y subsistemas existentes.

Espacio de nombres estático

- Se fija *hard coded*
- Mediante asignación directa

```
var miApp = {};

miApp.id = 0;

miApp.siguiente = function() {
  miApp.id++;
  console.log(miApp.id);
  return miApp.id;
};

miApp.reset = function() {
  miApp.id = 0;
};

miApp.siguiente();  // 1
miApp.siguiente();  // 2
miApp.reset();
miApp.siguiente();  // 1
```

```
var miApp = {};

miApp.id = 0;

miApp.siguiente = function() {
  this.id++;
  console.log(this.id);
  return this.id;
};

miApp.reset = function() {
  this.id = 0;
};

miApp.siguiente(); // 1
miApp.siguiente(); // 2
var getNextId = miApp.siguiente;
getNextId(); // NaN
```

¡¡¡ cuidado con `this` !!!

Espacio de nombres mediante objeto literal

- Sigue existiendo el riesgo de obtener un valor inesperado si se asigna un método a una variable
  - Asumimos que los objetos definidos dentro de un objeto literal no se van a reasignar

```
var miApp = {
  id: 0,

  siguiente: function() {
    this.id++;
    console.log(this.id);
    return this.id;
  },

  reset: function() {
    this.id = 0;
  }
};

miApp.siguiente();  // 1
miApp.siguiente();  // 2
miApp.reset();
miApp.siguiente();  // 1
```

4.2.2 Patrón módulo

- La lógica se protege del alcance global mediante una IIFE, la cual devuelve un objeto que representa el interfaz público del módulo.

```
var miModulo = (function() {
  var privado;
})();
```

- Al invocar inmediatamente la función y asignar el resultado a una variable que define el espacio de nombre, el API del módulo se restringe a dicho namespace.

```
var miApp = (function() {
  var id = 0;

  return {
    siguiente: function() {
      id++;
      console.log(id);
      return id;
    },

    reset: function() {
      id = 0;
    }
  };
})();

miApp.siguiente();  // 1
miApp.siguiente();  // 2
miApp.reset();
miApp.siguiente();  // 1
```

Paso de Parámetros

- Se pasan como un objeto literal

```
miApp.siguiente({incremento: 5});
```

- En el módulo comprobamos si viene algún parámetro mediante `||`

```
siguiente: function() {
  var misArgs = arguments[0] || '';
  var miIncremento = misArgs.incremento || 1;

  id = id + miIncremento;
  console.log(id);
  return id;
}
```

Valores de Configuración

- Se centralizan y agrupan en un objeto privado → `CONF`

```
var miApp = (function() {
  var id = 0;

  var CONF = {
    incremento: 1,
    decremento: 1
  };

  return {
    siguiente: function() {
      var misArgs = arguments[0] || '';
      var miIncremento = misArgs.incremento || CONF.incremento;
      id = id + miIncremento;
      console.log(id);
      return id;
    },
    reset: function() {
      id = 0;
    }
  };
})();

miApp.siguiente();  // 1
miApp.siguiente({incremento: 5});  // 6
miApp.reset();
miApp.siguiente();  // 1
```

Encadenando Llamadas *(Method chaining)*

```
var miApp = (function() {
  var id = 0;

  var CONF = {
    incremento: 1, decremento: 1
  };

  return {
    siguiente: function() {
      var misArgs = arguments[0] || '';
      var miIncremento = misArgs.incremento || CONF.incremento;
      id = id + miIncremento;
      console.log(id);
      return this;
    },
    anterior: function() {
      var misArgs = arguments[0] || '';
      var miDecremento = misArgs.decremento || CONF.decremento;

      id = id - miDecremento;
      console.log(id);
      return this;
    },
    reset: function() { id = 0; },
  };
})();
```

- Encadenar la salida de un método como la entrada de otro
- Devolver `this` como resultado de cada método

```
miApp.siguiente();  // 1
miApp.siguiente({incremento: 5});  // 6
miApp.anterior();  // 5
miApp.reset();
miApp.siguiente().siguiente().anterior({decremento: 3});  // 1 2 -1
```

Espacio de nombres dinámico

- Inyección del espacio de nombres
- Se emplea un *proxy* referenciado por la IIFE
  - Se pasa como parámetro

```
var miApp = {};
(function(contexto) {
  var id = 0;

  contexto.siguiente = function() {
    id++;
    console.log(id);
    return id;
  };

  contexto.reset = function() {
    id = 0;
  };
})(miApp);

miApp.siguiente();  // 1
miApp.siguiente();  // 2
miApp.reset();
miApp.siguiente();  // 1
```

```
var miApp = {};
(function() {
  var id = 0;

  this.siguiente = function() {
    id++;
    console.log(id);
    return id;
  };

  this.reset = function() {
    id = 0;
  };
}).apply(miApp);

miApp.siguiente();  // 1
miApp.siguiente();  // 2
miApp.reset();
miApp.siguiente();  // 1
```

4.3 Expresiones Regulares

- Modo de describir un patrón en una cadena de datos.

- La expresión regular se define mediante el objeto `RegExp` o entre barras `/`
- `regexp.test(cadena)` → comprueba si la expresión regular se encuentra en la cadena
- `cadena.search(regexp)` → similar a `indexOf` pero con expresiones regulares
  - obtendremos -1 si no la encuentra, o la posición comenzando por 0.

```
var exReBatman = /Batman/;
var exReBatman2 = new RegExp("Batman");

var cadena = "¿Sabías que Batman es mejor que Joker, y el mejor amigo de Batman es Robin?";
if (exReBatman.test(cadena)) {
  console.log("la cadena contiene a Batman");
  var pos = cadena.search(exReBatman);
  console.log("en la posición " + pos);
}
```

Patrones de Conjunto de Caracteres

Elemento	Uso	RegExp	Ejemplo
`^`	Comienza por	`/^Batman/`	Batman es el mejor
`$`	Acaba por	`/Batman$/`	El mejor es Batman
`[abc]`	dentro del rango (*a*, *b* o *c*)	`/B[aei]tman/`	Batman, Betman, Bitman
`[^abc]`	fuera del rango	`/B[^aei]tman/`	Botman, Bbtman, …
`[a-z]`	entre un rango (de *a* a *z*)	`/B[a-e]tman/`	Batman, Bbtman, Bctman, … Betman
`.`	cualquier caracter	`/B.tman/`	Batman, Bbtman, B-tman, B(tman, …
`\d`	dígito	`/B\dtman/`	B1tman, B2tman, …
`\w`	alfanumérico o _	`/B\wtman/`	Batman, B_tman, B1tman, …
`\s`	espacio (tab, nueva línea)	`/Batman\s/`	Batman ,
`\b`	límite de palabra	`/\bBatman/`	Batman con espacio delante o tras un salto de línea

```
/[0123456789]/.test("en 2015");
/[0-9]/.test("en 2015");
/\d\d-\d\d-\d\d\d\d/.test("31-01-2015");
/[01]/.test("111101111");
/[^01]/.test("2015");
```

Repeticiones de Patrón

Elemento	Uso	RegExp	Ejemplo
a?	cero o uno de *a*	`/Ba?tman/`	Btman, Batman
a*	cero o más de *a*	`/Ba*tman/`	Btman, Batman, Baatman, …
a+	uno o más de *a*	`/Ba+tman/`	Batman, Baatman, Baaatman, …
a{num}	exactamente *num* unidades de *a*	`/Ba{3}tman/`	Baaatman
a{num,}	*num* o más unidades de *a*	`/Ba{3,}tman/`	Baaatman, Baaaaatman, …
a{,num}	hasta *num* unidades de *a*	`/Ba{,3}tman/`	Batman, Baatman, Baaatman
a{num1,num2}	de *num1* a *num2* unidades de *a*	`/Ba{2,4}tman/`	Baatman, Baaatman, Baaaatman

```
/\d+/.test("2015"); // falso para ""
/\d*/.test("2015");
/\d*/.test("");
/selfie?/.test("selfie");
/selfie?/.test("selfi");
/\d{1,2}-\d{1,2}-\d{4}/.test("31-01-2015");
```

Agrupando y/o Eligiendo Expresiones

http://www.regexper.com

- Agrupar
  - Uso de paréntesis `( )`
  - Permiten utilizar un patrón de repetición sobre una expresión
- Elegir
  - Uso de tuberias `|`
  - Elección entre el patrón de su izquierda y el de su derecha

```
var bebeLlorando = /buu+(juu+)+/;
bebeLlorando.test("buujuuuujuujuuu");
```

```
var periodoTemporal = /\b\d+ ((dia|semana|año)s?|mes(es)?)\b/;
console.log("periodoTemporal".test("3 semanas"));
```

- `/[0-9]{8}([-]?)[A-Za-z]/`
- `/(0[1-9]|[12][0-9]|3[01])[- /.](0[1-9]|1[012])[- /.](19|20)\d\d/`
- `/[-0-9a-zA-Z.+_]+@[-0-9a-zA-Z.+_]+\.[a-zA-Z]{2,4}/`
- `/^(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([^?#]*))?(?:\?([^#]*))?(?:#(.*))?$/`

Flags

- Se incluyen tras cerrar la expresión
- Modifican las restricciones que se aplican

- **g** (*global matching*) → realiza la búsqueda en la cadena completa, no se detiene en la primera ocurrencia
- **i** (*case insensitive*) → la expresión deja de ser sensible a las mayúsculas
- **m** (*multiple lines*) → aplica los caracteres de inicio y fin de linea (`^` y `$` respectivamente) a cada línea de una cadena que contiene varias lineas.

Ocurrencias

- `RegExp.exec(cadena)` → método que devuelve un objeto con información de las ocurrencias encontradas o `null` en caso contrario.

```
var ocurrencia = /\d+/.exec("uno dos 100");
console.log(ocurrencia); // ["100"]
console.log(ocurrencia.index); // 8
```

- Las cadenas también tienen un método `match` que se comporta del mismo modo
  - `String.match(regexp)`

```
console.log("uno dos 100".match(/\d+/));
```

Subexpresiones de Ocurrencias

- Si la expresión regular contiene subexpresiones agrupadas mediante **paréntesis**, la ocurrencia que cumple esos grupos aparecerá en el array.
  - La ocurrencia completa siempre es el primer elemento.
  - El siguiente elemento es la parte que cumple el primer grupo (aquel cuyo paréntesis de apertura venga primero), después el segundo grupo, etc..

```
var patronDNI = /([0-9]{8})([-]?)([A-Za-z])/;
var ocurrencia = patronDNI.exec("12345678A"); // ["12345678A", "12345678", "", "A"]
var numero = ocurrencia[1];
var letra = ocurrencia[3];

var txtEntreComillasSimples = /'([^']*)'/;
console.log(txtEntreComillasSimples.exec("Yo soy 'Batman'")); // ["'Batman'","Batman"]
```

- Cuando un grupo se cumple en múltiples ocasiones, sólo se añade la última ocurrencia.

```
console.log(/(\d)+/.exec("2015"));  // ["2015","5"];
```

RegExp y Fechas

- Mediante expresiones regulares podemos parsear una cadena que contiene una fecha y construir un objeto `Date`.

```
var regExFecha = /(\d{1,2})-(\d{1,2})-(\d{4})/;
var ocurrencia = regExFecha.exec("31-01-2015");
var hoy = new Date(ocurrencia[3],ocurrencia[2]-1,ocurrencia[1]);
```

Reemplazar

- `String.replace(subcadena, nuevaSubcadena)`

```
console.log("papa".replace("p","m")); // "mapa"
```

- `String.replace(regexp, nuevaSubcadena)`

```
console.log("papa".replace(/p/g,"m")); // "mama"
```

- En la `nuevaSubcadena` podemos volver a las ocurrencias y trabajar con ellas

```
var personas = "Medrano, Aitor\nGallardo, Domingo\nSuch, Alejandro";
console.log(personas.replace(/([\w]+), ([\w]+)/g, "$2 $1"));
```

Cadenas de Substitución

- `$`*n* referencia al bloque *n* de la expresión regular.
- `$1` referencia al primer patrón, `$2` para el segundo, … hasta `$9`.
- La ocurrencia completa se referencia mediante `$&`.

```
var personas = "Medrano, Aitor\nGallardo, Domingo\nSuch, Alejandro";
console.log(personas.replace(/([\w]+), ([\w]+)/g, "$2 $1"));
```

Reemplazar con función

- `String.replace(regexp, función)`
  - Se invoca para todas las ocurrencias (y para la ocurrencia completa también)

```
var cadena = "Los mejores lenguajes son Java y JavaScript";
console.log(cadena.replace(/\b(java|javascript)\b/ig, function(str) {
  return str.toUpperCase();
}))
```

4.4 Librerías en *JavaScript*

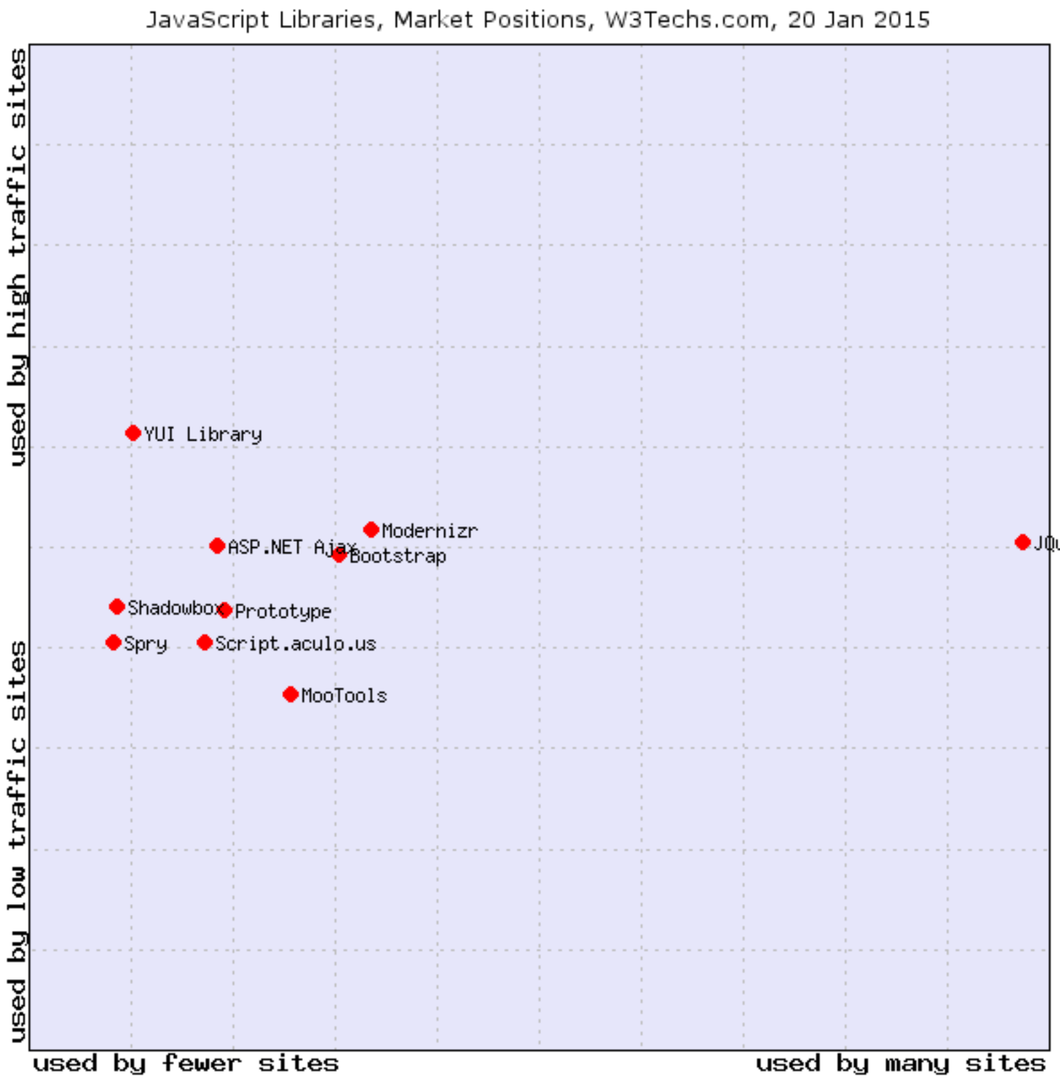
- Rico ecosistema de librerías
- Generalistas

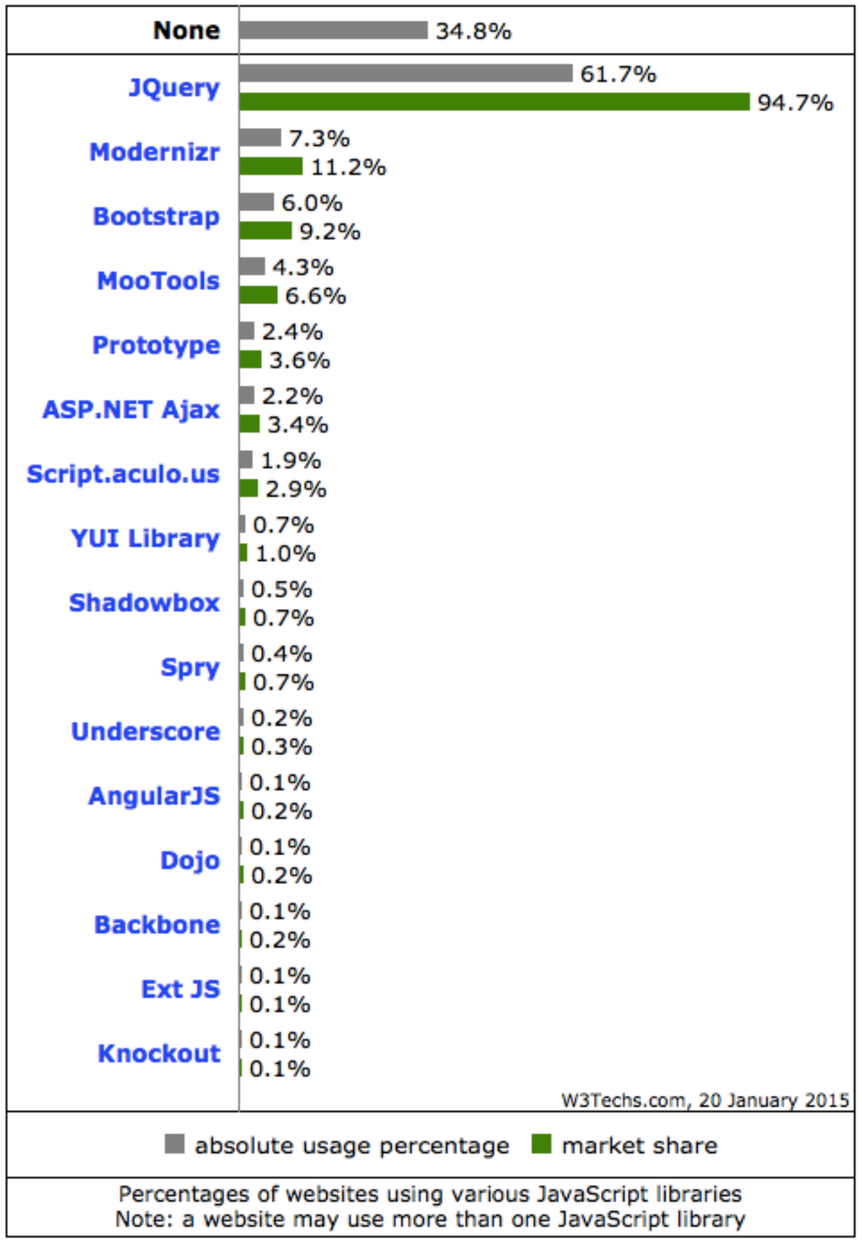
jQuery	estándar de factor
Prototype	+ scriptaculous
Mootools	soporte herencia
YUI	Yahoo
Dojo Toolkit	nucleo ligero (4KB)
Closure Library	Google
Underscore	completa HTML5

- Específicas

Bootstrap	web responsive
Handlebars	uso de plantillas
YepNope	carga condicional de scripts
LightBox	carrusel de fotos
ShadowBox	contenido multimedia en ventanas emergentes
Parsley	validación de formularios
Datejs	gestión de fechas

Situación actual





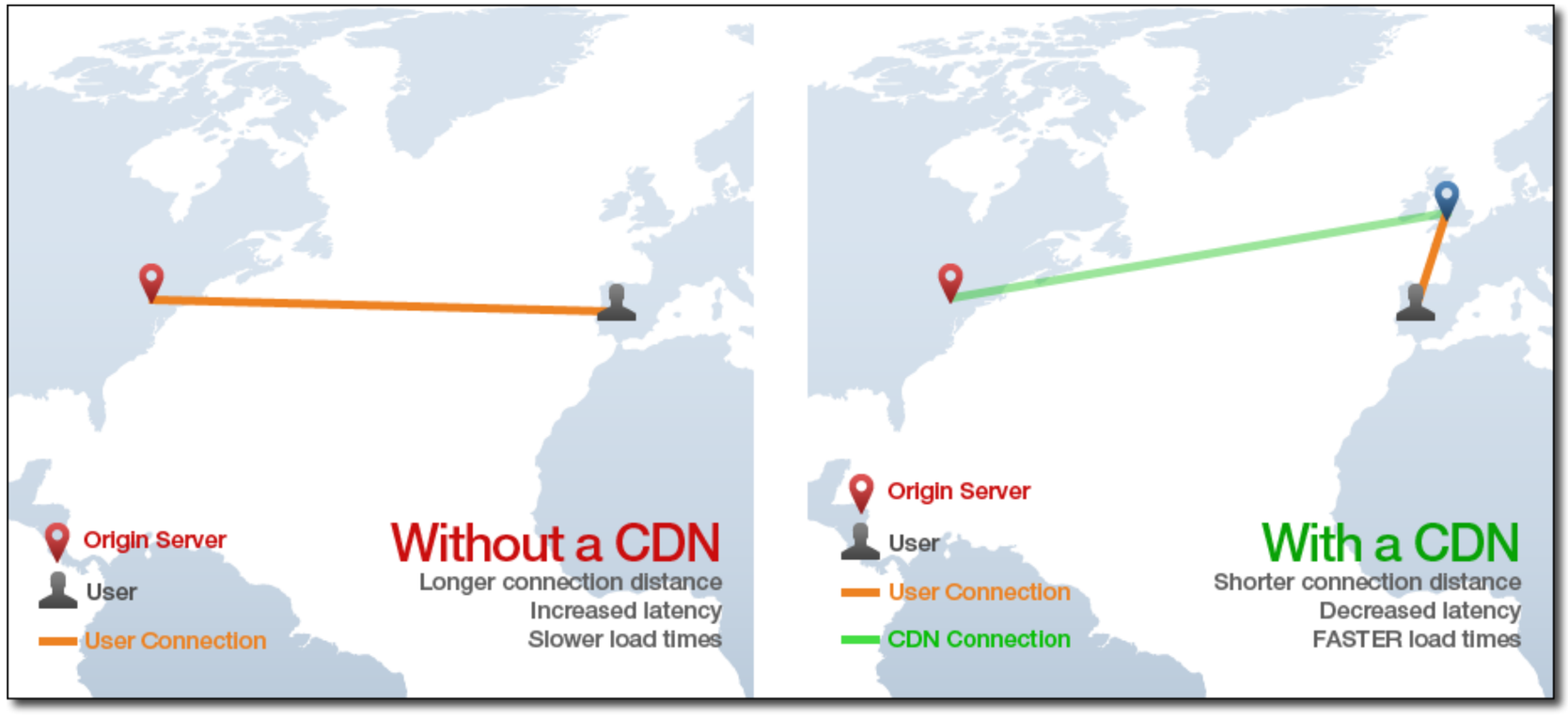
Inclusión de librerías

- Etiqueta `<script>`
- El orden de inclusión es importante
- Las librerías que dependen de otras deben incluirse después.

```
...
  <script src="jquery.js" />
  <script src="libQueUsaJQuery.js" />
</body>
```

CDN *(Content Delivery Network)*

- Servidores que guardan copias de las librerías de manera transparente al desarrollador y redirigen la petición al servidor más cercano



Google CDN

- Google ofrece enlaces a la gran mayoría de librerías existentes
- https://developers.google.com/speed/libraries/
- http://code.google.com/apis/libraries
- Al usarse por múltiples desarrolladores
  - El navegador ya la tenga cacheada
  - Si no, se encontrará hospedada en un servidor más cercano que el *hosting* de nuestra aplicación.

```
  ...
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.1/jquery.min.js"></script>
</body>
</html>
```

4.5 Testing

- Las pruebas son importantes
- Muy importantes

- Una **aserción** es una sentencia que predice el resultado del código con el valor esperado

- `console.assert(bool, mensaje)`
  - Comprueba si se cumple la aserción. Si no, muestra el mensaje

```
console.assert(1 == "1", "Conversión de tipos");
console.assert(1 === "1", "Sin conversión de tipos”);
// Assertion failed: Sin conversión de tipos
```

  - No se pueden automatizar → El desarrollador debe comprobar la consola

QUnit

- Framework de pruebas unitarias
- Desarrollado por el equipo de jQuery
  - Ampliamente probado e implantado

- http://qunitjs.com/
- Descargar o enlazar vía CDN

- Soportado por JSBin

Lanzador QUnit

```
<!DOCTYPE html>
<html lang="es">
<head>
    <title>QUnit Test Suite</title>
    <meta charset="utf-8" />
    <link rel="stylesheet" href="//code.jquery.com/qunit/qunit-1.22.0.css"
          type="text/css" media="screen">
</head>
<body>

<div id="qunit"></div>
<div id="qunit-fixture"></div>

<script type="text/javascript" src="miProyecto.js"></script>
<script src="//code.jquery.com/qunit/qunit-1.22.0.js"></script>
<script src="misPruebas.js"></script>

</body>
</html>
```

Resultado

- ***Hide passed tests***: permite ocultar las pruebas exitosas y mostrar sólo los fallos.
- ***Check for Globals***: crea una lista de todas las propiedades del objeto `window`, antes y después de cada caso de prueba y posteriormente comprueba las diferencias.
  - Si se añaden o modifican propiedades el test fallará, mostrando las diferencias.
  - Así podemos comprobar que ni nuestro código ni nuestras pruebas exportan accidentalmente alguna variable global.
- ***No try-catch***: la prueba se ejecuta fuera de un bloque try-catch.
  - Si una prueba lanza una excepción, el hilo morirá → no permite continuar y relanza una excepción nativa
  - Útil en navegadores antiguos con escaso soporte para *debug*, como IE6.

Caso de Prueba

- `QUnit.test()` / `QUnit.asyncTest()`
  - Crean un caso de prueba
- Parámetros:
  1. Cadena que identifica la suite de pruebas
  2. Función que contiene las aserciones que el framework ejecutará.

```
QUnit.test('Hola QUnit', function(assert) {
  // Aserciones con las pruebas
});
```

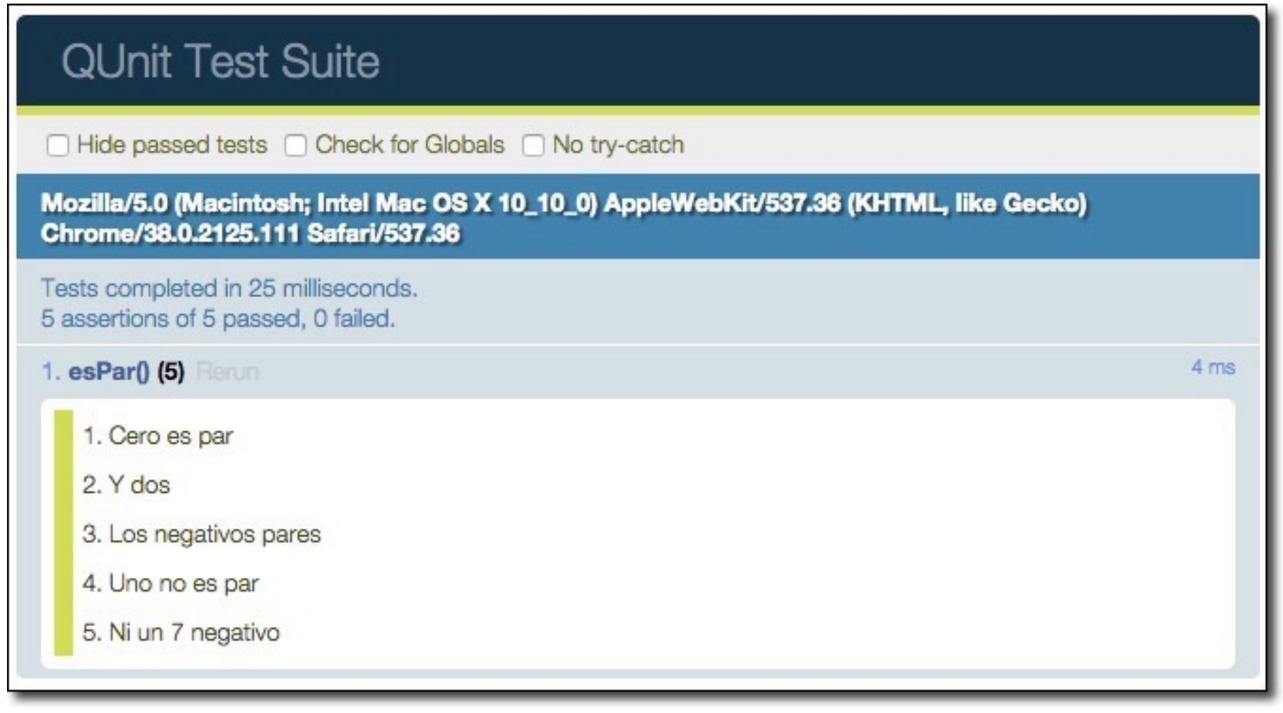
- Vamos a probar la función `esPar()`

```
function esPar(num) {
  return num % 2 === 0;
}
```

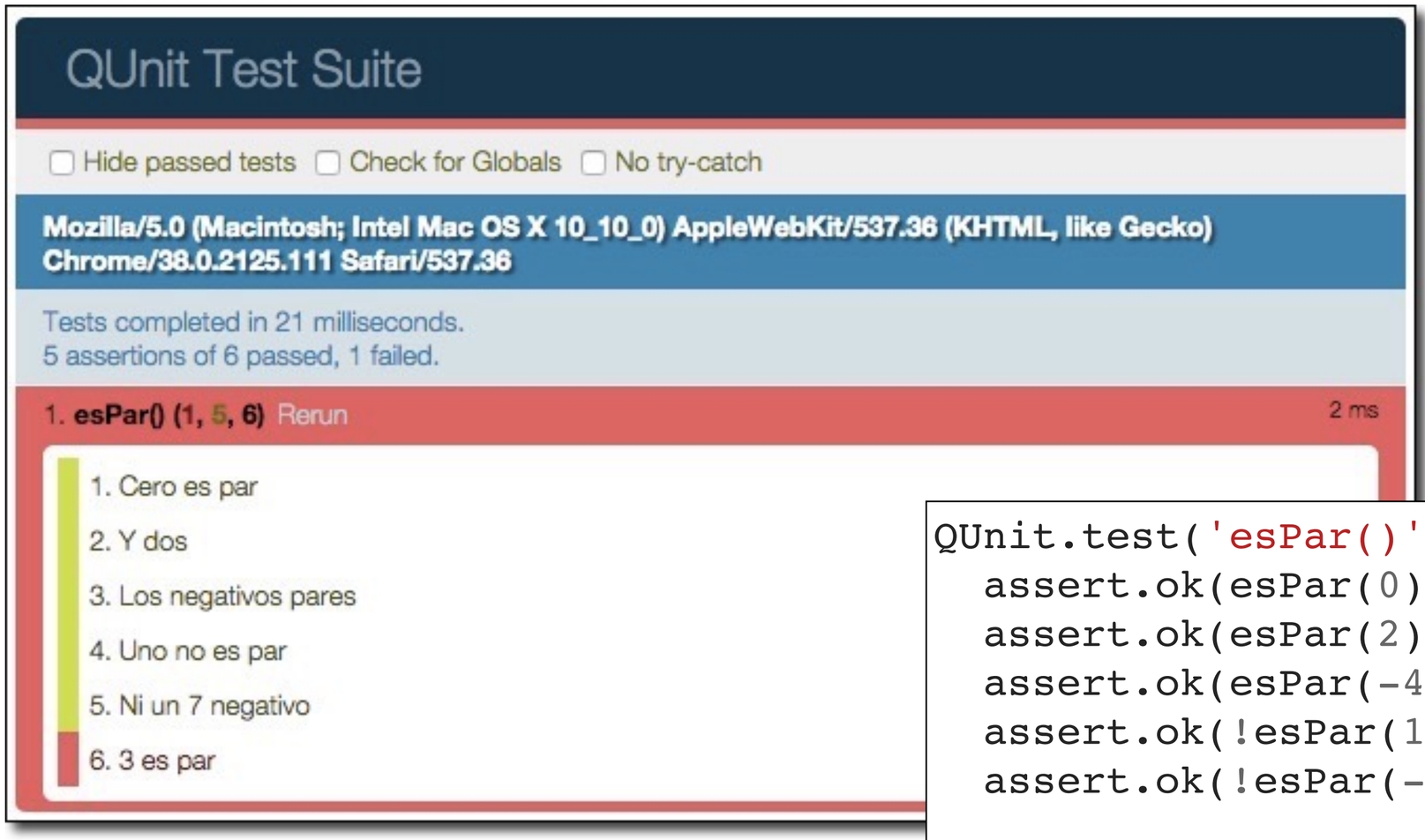
Aserciones

- Pertenecen al objeto `assert`
- **`ok`**`(asercionBool, mensaje)`
- `equal`, `notEqual`
- `strictEqual`, `notStrictEqual`
- `deepEqual`, `notDeepEqual`
- `throws`

```
QUnit.test('esPar()', function(assert) {
  assert.ok(esPar(0), 'Cero es par');
  assert.ok(esPar(2), 'Y dos');
  assert.ok(esPar(-4), 'Los negativos pares');
  assert.ok(!esPar(1), 'Uno no es par');
  assert.ok(!esPar(-7), 'Ni un 7 negativo');
});
```



Error



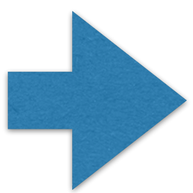

```
QUnit.test('esPar()', function(assert) {
  assert.ok(esPar(0), 'Cero es par');
  assert.ok(esPar(2), 'Y dos');
  assert.ok(esPar(-4), 'Los negativos pares');
  assert.ok(!esPar(1), 'Uno no es par');
  assert.ok(!esPar(-7), 'Ni un 7 negativo');

  assert.ok(esPar(3), '3 es par');
});
```

Comparaciones

- **equal**(*valorReal*, *valorEsperado*, *mensaje*)
- notEqual(*valorReal*, *valorEsperado*, *mensaje*)
- Operador ==, !=

```
QUnit.test('comparaciones', function(assert) {
  assert.ok( 1 == 1, 'uno es igual que uno');
});
```



```
QUnit.test('comparaciones', function(assert) {
  assert.equal( 1, 1, 'uno es igual que uno');
  assert.equal( 2, 1, ‘falla porque 2 != 1');
});
```

Comparación Estricta

- **`strictEqual`**`(`*`valorReal, valorEsperado, mensaje`*`)`
- `notStrictEqual(`*`valorReal, valorEsperado, mensaje`*`)`
- Operador `===`, `!==`

```
QUnit.test('comparacionesEstrictas', function(assert) {
  assert.equal( 0, false, 'pasa la prueba');
  assert.strictEqual( 0, false, 'falla');
  assert.equal( null, undefined, 'pasa la prueba');
  assert.strictEqual( null, undefined, 'falla');
});
```

- Ni las comparaciones estrictas ni las normales funcionan con arrays u objetos.

```
QUnit.test('comparacionesArrays', function(assert) {
  assert.equal( {}, {}, 'falla, objetos diferentes');
  assert.equal( {a: 1}, {a: 1} , 'falla');
  assert.equal( [], [], 'falla, diferentes arrays');
  assert.equal( [1], [1], 'falla');
});
```

Identidad

- **`deepEqual`**`(`*`valorReal, valorEsperado, mensaje`*`)`
- `notDeepEqual(`*`valorReal, valorEsperado, mensaje`*`)`
- Comparación recursiva
- Funciona tanto con tipos primitivos como con arrays, objetos y expresiones regulares

```
QUnit.test('comparacionesArraysRecursiva', function(assert) {
  assert.deepEqual( {}, {}, 'correcto, los objetos tienen el mismo contenido');
  assert.deepEqual( {a: 1}, {a: 1} , 'correcto');
  assert.deepEqual( [], [], 'correcto, los arrays tienen el mismo contenido');
  assert.deepEqual( [1], [1], 'correcto');
})
```

Propiedades

- **`propEqual`**`(`*`valorReal, valorEsperado, mensaje`*`)`
- Sólo compara las propiedades y valores de un objeto

```
QUnit.test('comparacionesPropiedades', function(assert) {
  assert.propEqual( {}, {}, 'correcto, los objetos tienen el mismo contenido');
  assert.propEqual( {a: 1}, {a: 1} , 'correcto');
  assert.propEqual( [], [], 'correcto, los arrays tienen el mismo contenido');
  assert.propEqual( [1], [1], 'correcto');
});
```

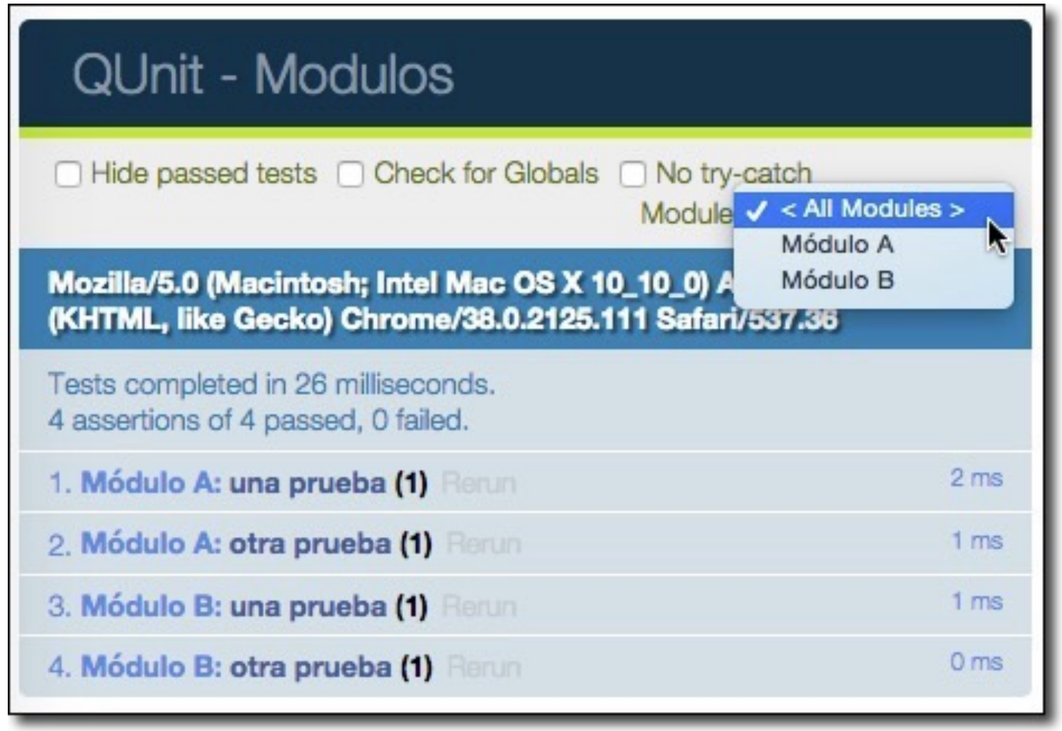
Excepciones

- **`throws`**`(`*`función [, excepcionEsperada] [,mensaje]`*`)`
  - Podemos indicarle la `excepcionEsperada`
- Permiten comprobar si nuestro código lanza `Error`


```
QUnit.test('excepciones', function(assert) {
  assert.throws( function() { throw Error("Hola, soy un Error"); },
    'pasa al lanzar el Error');
  assert.throws( function() { x; }, // ReferenceError
    'pasa al no definir x, ');
  assert.throws( function() { esPar(2); },
    'falla porque no se lanza ningun Error');
});
```

Módulos

- Permiten agrupar casos de prueba
- `QUnit.module('nombreModulo')`




```
QUnit.module('Módulo A');
QUnit.test('una prueba', function(assert) { assert.ok(true, "ok"); });
QUnit.test('otra prueba', function(assert) { assert.ok(true, "ok"); });

QUnit.module('Módulo B');
QUnit.test('una prueba', function(assert) { assert.ok(true, "ok"); });
QUnit.test('otra prueba', function(assert) { assert.ok(true, "ok"); });
```

Módulos: `setup` y `teardown`

- Podemos extraer código común dentro del módulo.
- 2º parámetro → un objeto que contiene las propiedades:
  - **`setup`**: código que se ejecuta antes de cada prueba
  - **`teardown`**: código que se ejecuta después de cada prueba

- Si volvemos a llamar a `QUnit.module()` sin ningún parámetro adicional, las funciones `setup` y `teardown` se resetearán.

```
QUnit.module("Módulo C", {
  setup: function( assert ) {
    assert.ok( true, "una aserción extra antes de cada test" );
  }, teardown: function( assert ) {
    assert.ok( true, "y otra más después de cada prueba" );
  }
});
QUnit.test("Prueba con setup y teardown", function(assert) {
  assert.ok( esPar(2), "dos es par");
});
QUnit.test("Prueba con setup y teardown", function(assert) {
  assert.ok( esPar(4), "cuatro es par");
});
```

Expectativas

- Permiten indicar el número de aserciones que esperamos que se ejecuten.
- *Best Practice*
- **`expect`**`(numAserciones)`
  - Si no se cumple, la prueba falla

```
QUnit.test('comparacionesPropiedades', function(assert) {
  expect(4);
  // aserciones
}
```

Pruebas Asíncronas

- `QUnit.asyncTest(nombre, funcionPrueba)`

- Cuando *QUnit* ejecuta un caso de prueba asíncrona, automáticamente detiene el *testrunner*
- Este permanecerá parado hasta que el caso de prueba que contiene las aserciones invoque a `QUnit.start()`
- Para detener una prueba usaremos `QUnit.stop()`.
- Al hacerlo se incrementa el número de llamadas necesarias a `QUnit.start()` para que el *testrunner* vuelva a funcionar

- Tanto `start()` como `stop()` aceptan un entero como argumento opcional para fusionar múltiples llamadas en una sola.

Ejemplo asíncrono

```
function mayor() {
  var max=-Infinity;
  for (var i=0, len=arguments.length; i<len; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
```

```
QUnit.asyncTest('max', function (assert) {
  expect(1);
  window.setTimeout(function() {
    assert.strictEqual(mayor(3, 1, 2), 3, 'Todo números positivos');
    QUnit.start();
  }, 0);
});
```

Ejemplo asíncrono múltiple - `stop()`

```
QUnit.asyncTest('max', function (assert) {
  expect(4);
  QUnit.stop(3);

  window.setTimeout(function() {
    assert.strictEqual(mayor(), -Infinity, 'Sin parámetros');
    QUnit.start();
  }, 0);
  window.setTimeout(function() {
    assert.strictEqual(mayor(3, 1, 2), 3, 'Todo números positivos');
    QUnit.start();
  }, 0);
  window.setTimeout(function() {
    assert.strictEqual(mayor(-10, 5, 3, 99), 99, 'Números positivos y negativos');
    QUnit.start();
  }, 0);
  window.setTimeout(function() {
    assert.strictEqual(mayor(-14, -22, -5), -5, 'Todo números negativos');
    QUnit.start();
  }, 0);
});
```

Librerías *Testing*

- **Jasmine** → BDD (Behaviour Driven Development) → requisitos de negocio

```
describe("Una suite", function() {
  it("contiene un requisito con una expectativa", function() {
    expect(true).toBe(true);
  });
});
```

- **Mocha** → BDD, permite añadir nuevas aserciones (Chai.js) y emplear promesas
- **Sinon.js** → objetos *mock*
- **Blanket.js** → informes de cobertura
- **Plato** → complejidad de código

¿Preguntas?