# Closures

Los closures son funciones que manejan variables independientes. En otras palabras, la función definida en el closure "recuerda" el entorno en el que se ha creado.

Consideremos lo siguiente:

```
function inicia() {
  var nombre = "Mozilla"; // 'nombre' es una variable local creada por l
  function muestraNombre() { // 'muestraNombre' es una función interna (
    alert(nombre); // dentro de esta función usamos una variable declara
  }
  muestraNombre();
}
inicia();
```

La función `inicia()` crea una variable local llamada `nombre`, a continuación, define una función denominada `muestraNombre()`. `muestraNombre()` es una función interna (un closure) definida dentro de `inicia ()`, y sólo está disponible en el cuerpo de esa función. `muestraNombre ()` no tiene ninguna variable propia, lo que hace es reutilizar la variable `nombre` declarada en la función externa.

Esto funciona bien, intenta correr el código para ver qué pasa. Este es un ejemplo de ámbito funcional: en JavaScript, el ámbito de una variable se define por su ubicación dentro del código fuente y las funciones anidadas tienen acceso a las variables declaradas en su ámbito externo.

Ahora consideremos el siguiente ejemplo:

```
function creaFunc() {
  var nombre = "Mozilla";
  function muestraNombre() {
    alert(nombre);
  }
```

```
  return muestraNombre;
}

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

Si se ejecuta este código tendrá exactamente el mismo efecto que el ejemplo anterior: se mostrará el texto "Mozilla" en un cuadro de alerta de Javascript. Lo que lo hace diferente (e interesante) es que la función externa nos ha devuelto la función interna `muestraNombre()` antes de ejecutarla.

Puede parecer poco intuitivo que este código funcione. Normalmente, las variables locales dentro de una función sólo existen mientras dura la ejecución de dicha función. Una vez que `creaFunc()` haya terminado de ejecutarse, es razonable suponer que no se pueda ya acceder a la variable `nombre`. Dado que el código funciona como se esperaba, esto obviamente no es el caso.

La solución a este rompecabezas es que `miFunc` se ha convertido en un *closure*. Un closure 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. En este caso, `miFunc` es un closure que incorpora tanto la función `muestraNombre` como el string "Mozilla" que existían cuando se creó el closure.

Este es un ejemplo un poco más interesante: una función `creaSumador`:

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

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

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

En este ejemplo, hemos definido una función `creaSumador (x)` que toma un argumento único `x` y devuelve una nueva función. Esa nueva función toma un único argumento `y`, devolviendo la suma de `x` + `y`.

En esencia, `creaSumador` es una fábrica de función: crea funciones que pueden sumar un valor específico a su argumento. En el ejemplo anterior utilizamos nuestra fábrica de función para crear dos nuevas funciones: una que agrega 5 a su argumento y otra que agrega 10.

`suma5` y `suma10` son ambos closures. Comparten la misma definición de cuerpo de función, pero almacenan diferentes entornos. En el entorno `suma5`, `x` es 5. En lo que respecta a `suma10`, `x` es 10.

## Closures prácticos

Hasta aquí hemos visto teoría, pero ¿son los closures realmente útiles? Vamos a considerar sus implicaciones prácticas. Un closure permite asociar algunos datos (el entorno) con una función que opera sobre esos datos. Esto tiene evidentes paralelismos con la programación orientada a objetos, en la que los objetos nos permiten asociar algunos datos (las propiedades del objeto) con uno o más métodos.

En consecuencia, puede utilizar un closure en cualquier lugar en el que normalmente pondría un objeto con un solo método.

En la web hay situaciones habituales en las que aplicarlos. Gran parte del código JavaScript para web está basado en eventos: definimos un comportamiento y lo conectamos a un evento que es activado por el usuario (como un click o pulsación de una tecla). Nuestro código generalmente se adjunta como una devolución de llamada (callback): que es una función que se ejecuta en respuesta al evento.

Aquí está un ejemplo práctico: Supongamos que queremos añadir algunos botones a una página para ajustar el tamaño del texto. Una manera de hacer esto es especificar el tamaño de fuente del elemento `body` en píxeles y, a continuación, ajustar el tamaño de los demás elementos de la página (como los encabezados) utilizando la unidad relativa `em`:

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

```

```
h1 {
  font-size: 1.5em;
}
h2 {
  font-size: 1.2em;
}
```

Nuestros botones interactivos de tamaño de texto pueden cambiar la propiedad `font-size` del elemento `body`, y los ajustes serán aplicados por los otros elementos de la página gracias a las unidades relativas.

Aquí está el código JavaScript:

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);
```

`size12`, `size14` y `size16` ahora son funciones que cambian el tamaño del texto de `body` a 12, 14 y 16 pixels, respectivamente. Podemos conectarlos a botones (en este caso enlaces) de la siguiente forma:

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
```

```
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

## Emulando métodos privados con closures

Lenguajes como Java ofrecen la posibilidad de declarar métodos privados, es decir, que sólo pueden ser llamados por otros métodos en la misma clase.

JavaScript no proporciona una forma nativa de hacer esto, pero es posible emular métodos privados utilizando closures. Los métodos privados no son sólo útiles para restringir el acceso al código: también proporcionan una poderosa manera de administrar tu espacio de nombres global, evitando que los métodos no esenciales embrollen la interfaz pública de tu código.

Aquí vemos cómo definir algunas funciones públicas que pueden acceder a variables y funciones privadas utilizando closures. A esto se le conoce también como el patrón módulo:

```
var Counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
})();

alert(Counter.value()); /* Muestra 0 */
Counter.increment();
Counter.increment();
alert(Counter.value()); /* Muestra 2 */
Counter.decrement();
alert(Counter.value()); /* Muestra 1 */
```

Hay mucho aquí. En los ejemplos anteriores cada closure ha tenido su propio entorno; aquí creamos un único entorno compartido por tres funciones: `Counter.increment`, `Counter.decrement` y `Counter.value`.

El entorno compartido se crea en el cuerpo de una función anónima, que se ejecuta en el momento que se define. El entorno contiene dos elementos privados: una variable llamada `privateCounter` y una función llamada `changeBy`. No se puede acceder a ninguno de estos elementos privados directamente desde fuera de la función anónima. Se accede a ellos por las tres funciones públicas que se devuelven desde el contenedor anónimo.

Esas tres funciones públicas son closures que comparten el mismo entorno. Gracias al ámbito léxico de Javascript, cada uno de ellas tienen acceso a la variable `privateCounter` y a la función `changeBy.`

En este caso hemos definido una función anónima que crea un contador, y luego la llamamos inmediatamente y asignamos el resultado a la variable `Counter`. Pero podríamos almacenar esta función en una variable independiente y utilizarlo para crear varios contadores:

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
};

var Counter1 = makeCounter();
var Counter2 = makeCounter();
alert(Counter1.value()); /* Muestra 0 */
```

```
Counter1.increment();
Counter1.increment();
alert(Counter1.value()); /* Muestra 2 */
Counter1.decrement();
alert(Counter1.value()); /* Muestra 1 */
alert(Counter2.value()); /* Muestra 0 */
```

Ten en cuenta que cada uno de los dos contadores mantiene su independencia del otro. Su entorno durante la llamada de la función `makeCounter()` es diferente cada vez. La variable del closure llamada `privateCounter` contiene una instancia diferente cada vez.

Utilizar closures de este modo proporciona una serie de beneficios que se asocian normalmente con la programación orientada a objectos, en particular la encapsulación y la ocultación de datos.

## Creando closures en loops: Un error común

Antes de la introducción de la palabra clave `let` en JavaScript 1.7, un problema común con closures ocurría cuando se creaban dentro de un bucle 'loop'. Veamos el siguiente ejemplo:

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>
<p>Age: <input type="text" id="age" name="age"></p>
```

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Dirección de correo electrónico'},
      {'id': 'name', 'help': 'Nombre completo'},
      {'id': 'age', 'help': 'Edad (debes tener más de 16 años)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
```

```
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();

```

Ver en el JSFiddle

El array `helpText` define tres avisos de ayuda, cada uno asociado con el ID de un campo de entrada en el documento. El bucle recorre estas definiciones, enlazando un evento onfocus a cada uno que muestra el método de ayuda asociada.

Si pruebas este código, verás que no funciona como esperabas. Independientemente del campo en el que se haga foco, siempre se mostrará el mensaje de ayuda relativo a la edad.

La razón de esto es que las funciones asignadas a onfocus son closures; que constan de la definición de la función y del entorno abarcado desde el ámbito de la función `setupHelp`. Se han creado tres closures, pero todos comparten el mismo entorno. En el momento en que se ejecutan las funciones callback de onfocus, el bucle ya ha finalizado y la variable `item` (compartida por los tres closures) ha quedado apuntando a la última entrada en la lista de `helpText.`

En este caso, una solución es utilizar más closures: concretamente añadiendo una fábrica de función como se ha descrito anteriormente:

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function makeHelpCallback(help) {
  return function() {
    showHelp(help);
  };
}

```

```
10
11  function setupHelp() {
12    var helpText = [
13        {'id': 'email', 'help': 'Dirección de correo electrónico'},
14        {'id': 'name', 'help': 'Nombre completo'},
15        {'id': 'age', 'help': 'Edad (debes tener más de 16 años)'}
16      ];
17
18    for (var i = 0; i < helpText.length; i++) {
19      var item = helpText[i];
20      document.getElementById(item.id).onfocus = makeHelpCallback(item.hel
21    }
22  }
23
24  setupHelp();
```

Ver en el JSFiddle

Esto funciona como se esperaba. En lugar de los tres callbacks compartiendo el mismo entorno, la función `makeHelpCallback` crea un nuevo entorno para cada uno en el que `help` se refiere a la cadena correspondiente del array `helpText`.

## Consideraciones de rendimiento

No es aconsejable crear innecesariamente funciones dentro de otras funciones si no se necesitan los closures para una tarea particular ya que afectará negativamente el rendimiento del script tanto en consumo de memoria como en velocidad de procesamiento.

Por ejemplo, cuando se crea un nuevo objeto/clase, los métodos normalmente deberían asociarse al prototipo del objeto en vez de definirse en el constructor del objeto. La razón es que con este último sistema, cada vez que se llama al constructor (cada vez que se crea un objeto) se tienen que reasignar los métodos.

Veamos el siguente caso, que no es práctico pero sí demostrativo:

```
1  function MyObject(name, message) {
2    this.name = name.toString();
     this.message = message.toString();
```

```
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}

```

El código anterior no aprovecha los beneficios de los closures. Podríamos modificarlo de la siguiente manera:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName: function() {
    return this.name;
  },
  getMessage: function() {
    return this.message;
  }
};
```

Sin embargo, no se recomienda redefinir el prototipo, así que el siguiente ejemplo es aún mejor que el anterior, porque lo que hace es añadir funcionalidad al prototipo existente:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

En los dos ejemplos anteriores, todos los objetos comparten el prototipo heredado y no se van a definir los métodos cada vez que se  crean de objetos. Ver Detalles del Modelo de Objetos para más información.