ceviche.js

Capítulo 2: Funciones

> *Con lo aprendido en el capítulo anterior has iniciado con buen pie la tarea de desarrollar el sitio web de La Buena Espina, pero aún tienes camino por recorrer si deseas crear un sitio web fácilmente mantenible en el futuro.*
>
> *Para lograr una buena estructura en cualquier tipo de proyecto es necesario hacer uso de algunos patrones de diseño, que son soluciones probadas a situaciones comunes en todo tipo de sitio o aplicación web. Estos patrones utilizan funciones a un nivel más avanzado de lo visto en el capítulo anterior, que es justo de lo que trata esta parte.*

Funciones

Las funciones en JavaScript también son objetos, por lo que tienen propiedades y métodos. Además de ser objetos, son llamadas *ciudadanos de primera clase* (*first-class citizen*), el tipo de estructura más importante en un lenguaje, así que pueden ser pasadas como parámetros (*callbacks*), ser asignadas a una variable (constructores y funciones anónimas) o ser retornadas por otra función (*closures*).

En JavaScript se pueden crear funciones de 3 formas:

A. Declarando una función, con la sentencia `function`:

```
function sum(a, b) {
  return a + b;
};

sum(1, 2);
```

B. Expresando una función, con el operador `function`:

```
var sum = function sum(a, b) {
  return a + b;
};

sum(1, 2);
```

C. Creando una instancia del constructor `Function`:

```
var sum = new Function('a', 'b', 'return a + b');

sum(1, 2);
```

La forma A y B son similares en sintaxis. Sin embargo, la diferencia principal se da en cómo el navegador carga las funciones. En el primer caso, el navegador cargará todas las funciones declaradas y luego ejecutará el código en el orden en el que fue escrito, mientras que en el segundo caso, la función se cargará según la posición donde esté definida.

La forma C tiene la ventaja de permitir *evaluar sentencias en tiempo de ejecución*. Esto quiere decir que se pueden crear y ejecutar funciones a partir de datos que ingrese un usuario, como en el caso de una consola de JavaScript.

Scope y context

Una de las más grandes diferencias en JavaScript con respecto a los lenguajes de los cuales está influido es en el ámbito (*scope*) y en el contexto de función.

El ámbito de una variable es el lugar dentro de un programa en el cual dicha variable vive y por lo tanto, donde puede ser usada. El scope de una variable es a nivel de funciones, lo que significa que una variable definida dentro de una función (con la palabra reservada `var`) va a poder ser usada dentro de esa función, pero no fuera de la misma.

Por otro lado, no es recomendable declarar variables sin `var` ya que, si se omite esta palabra reservada, el programa buscará la variable en los ámbitos (o *scopes*) superiores hasta llegar al ámbito global. Si la variable existe, reemplaza su valor, y si no existe la crea en el ámbito global:

```
function globalFunction() {
  function innerFunction() {
    function deeperFunction() {
      globalVar = 'globalVar'; // sin `var`, `globalVar` se vuelve
global
    };

    deeperFunction();
    console.log('deeperFunction', globalVar);
  };

  innerFunction();
  console.log('innerFunction', globalVar);
};

globalFunction();
console.log('globalFunction', globalVar);

// deeperFunction globalVar
// innerFunction globalVar
// globalFunction globalVar
```

El contexto dentro de una función puede cambiar de valor, de acuerdo a la forma cómo está definida la función y cómo se la ejecuta. El contexto es el "dueño" del ámbito de la función que se está ejecutando y se puede acceder a él mediante la palabra reservada `this`.

El contexto cambia de valor según los siguientes casos:

A. Cuando se define una función como método de un objeto, el contexto de dicha función es el objeto:

```
var obj = {
  property: 'value'
};

obj.getValue = function() {
  console.log('context: ', this);
  return this.property;
};

obj.getValue();
// context:  Object {property: "value", getValue: function}
// "value"
```

B. Cuando la función es una función constructora, el contexto de dicha función es el objeto instanciado usando dicha función:

```
var Constructor = function Constructor(newValue) {
  this.property = newValue;

  console.log('context: ', this);
};

var obj = new Constructor('value');
// context:  Constructor {property: "value"}
obj;
// Constructor {property: "value"}
obj.property = 'value';
// "value"
```

C. Cuando la función solo es una función (creada de las 3 formas explicadas anteriormente), el contexto es el contexto global, el cual en navegadores es `window`.

```
function globalContext1() {
```

```
  console.log('context1: ', this);
};

var globalContext2 = function() {
  console.log('context2: ', this);
};

var globalContext3 = new Function("console.log('context3: ',
this);");

globalContext1();
// context1: Window {top: Window, window: Window, location:
Location, external: Object, chrome: Object…}
globalContext2();
// context2: Window {top: Window, window: Window, location:
Location, external: Object, chrome: Object…}
globalContext3();
// context3: Window {top: Window, window: Window, location:
Location, external: Object, chrome: Object…}
```

En JavaScript se puede ejecutar una función y cambiar el contexto utilizando los métodos `call` y `apply`. Ambos métodos, que también son funciones, son similares en propósito, pero difieren en el número y forma de sus parámetros.

```
function buildSiteTitle(part1, part2) {
  return part1 + ' – ' + part2;
};

buildSiteTitle('La Buena Espina', 'Carta');
// "La Buena Espina – Ceviches"
buildSiteTitle.call(null, 'La Buena Espina', 'Locales');
// "La Buena Espina – Locales"
buildSiteTitle.apply(null, ['La Buena Espina', 'Historia']);
// "La Buena Espina – Historia"
```

`call` y `apply` tienen como primer argumento el nuevo contexto de la función, el cual en este caso es `null` debido a que no es necesario tener un contexto definido para este ejemplo. Para el caso de `call` el resto de argumentos deben ser los mismos de la función al ser ejecutada, mientras que para el caso de `apply` solo toma un segundo argumento, un arreglo, el cual contiene todos los argumentos de la función a ejecutar.

`apply` tiene una ventaja con respecto a `call`, que es permitir pasar los argumentos de forma dinámica. En el caso de `call`, cada parámetro debe ser pasado dentro del método, como un parámetro más; en el caso de `apply`, solo basta agregar un elemento en el segundo parámetro, que es un arreglo.

```
var titleParts = [];

titleParts.push('La Buena Espina');
buildSiteTitle.apply(null, titleParts);
// "La Buena Espina – undefined"

titleParts.push('Historia');
buildSiteTitle.apply(null, titleParts);
// "La Buena Espina – Historia"
```

Esta flexibilidad al momento de pasar los argumentos en una función se ve limitada hasta este punto. Es aquí donde se empieza a utilizar la palabra reservada `arguments`, el cual es un objeto que representa a los argumentos de la función que se está ejecutando en ese instante.

```
function buildSiteTitle() {     // Ya no es necesario definir los
parámetros
  var separator = ' – ';
  var title = '';

  if (arguments.length > 2) {   // arguments tiene una propiedad
llamada length
```

```
    separator = ' › ';
  }

  for (var i = 0; i < arguments.length; i++) {
    if (i == 0) {
      title += arguments[i];    // La primera parte del título no
debe tener separador
    }
    else {
      title += separator + arguments[i];
    }
  }

  return title;
};

buildSiteTitle.apply(null, ['La Buena Espina', 'Historia']);
// "La Buena Espina — Historia"

buildSiteTitle.apply(null, ['La Buena Espina', 'Carta',
'Ceviches']);
// "La Buena Espina › Carta › Ceviches"
```

Cabe notar que aunque `arguments` tiene una propiedad llamada `length` y puede ser iterada mediante una estructura `for`, no es un arreglo, si no un objeto cuyas propiedades son los argumentos de la función, y donde el nombre de cada propiedad es un índice que empieza en 0 y termina en un número igual a `length` menos uno.

Funciones anónimas

Una función anónima es una función expresada con el operador `function` (forma B para crear funciones) y que no tiene nombre.

```
var nammedFunction = function funcionConNombre() {
  return 'función con nombre';
};

var anonymousFunction = function () {
  return 'función anónima';
};

nammedFunction();
// "función con nombre"
anonymousFunction();
// "función anónima"
```

Este tipo de funciones suelen ser utilizadas como funciones inmediatamente invocadas y callbacks, ya que al ser una función sin nombre, se espera que sea de un solo uso.

Funciones inmediatamente invocadas

Una función inmediatamente invocada (*Immediately-Invoked Function Expression - IIFE*) es una expresión que permite ejecutar una función anónima inmediatamente después de ser definida, lo cual hace que el valor devuelto por la expresión no sea la función en sí, si no el valor de su ejecución.

```
var sum = (function(a, b) {
  return a + b;
})(10, 15);

sum;
// 25
```

La función anónima de este ejemplo está encerrada por paréntesis, lo que permite tratarla como un objeto más, de igual forma a que si esa misma función anónima esté asignada a una variable:

```
var sumFn = function(a, b) {
  return a + b;
};

sumFn(10, 15);
// 25
```

En el caso de una función inmediatamente invocada, la función anónima es ejecutada una sola vez, por lo que no hay motivo para ser guardada en una variable.

Funciones constructoras

Las funciones constructoras permiten definir una especie de "clase" en JavaScript, con la cual luego se pueden instanciar objetos que tengan propiedades y métodos en común.

```
function Dish(options) {
  this.name = options.name;
  this.ingredients = options.ingredients;
  this.garnishes = options.garnishes;
  this.diners = options.diners;
};

var cevicheSimple = new Dish({
  name: 'Ceviche simple',
  ingredients: [
    '1 kilo de pescado',
    '2 cebollas',
    '1 taza de jugo de limón',
    '1 ají limo',
    'sal'
  ],
```

```
    garnishes: [
      'lechuga (2 hojas por plato)',
      'maíz cancha',
      '4 porciones de yuca',
      '4 choclos sancochados',
      'camote sancochado en rodajas (2 por plato)'
    ],
    diners: 4
  });

  cevicheSimple;
  // Dish {name: "Ceviche simple", ingredients: Array[5], garnishes:
  Array[5], diners: 4}

  cevicheSimple instanceof Dish;
  // true
```

Objeto prototype

La orientación a objetos en JavaScript no se maneja mediante clases, si no utilizando funciones constructoras y *prototypes*. Mientras las primeras fungen de clases, las segundas permiten aplicar herencia simple.

Todas los objetos que son instancias de una función constructora comparten las propiedades y métodos definidos en la propiedad `prototype` de dicha función. De igual forma, el *prototype* de un solo objeto puede definirse con el método `Object.create`, visto en el capítulo anterior.

Esta propiedad puede ser extendida (agregar o quitar elementos), así como ser reemplazada por otro objeto, que viene a ser el nuevo *prototype*, o incluso negarle la posibilidad de tener uno asignándole `null` a la propiedad `prototype`.

Al extender el *prototype* de una función, todos los objetos que comparten dicha propiedad actualizan automáticamente su valor:

```
function Dish(options) {
  this.name = options.name;
  this.ingredients = options.ingredients;
  this.garnishes = options.garnishes;
  this.diners = options.diners;
};

var cevicheSimple = new Dish({
  name: 'Ceviche simple',
  diners: 4
});

cevicheSimple.setIngredients([]);
// TypeError: Object #<Dish> has no method 'setIngredients'

Dish.prototype.setIngredients = function(ingredients) {
  return this.ingredients = ingredients;
};

cevicheSimple.setIngredients(['1 kilo de pescado']);
// ["1 kilo de pescado"]

var sudadoPescado = new Dish({
  name: 'Sudado de pescado',
  diners: 6
});

sudadoPescado.setIngredients(['6 filetes de 160 g. de pescado
blanco']);
// ["6 filetes de 160 g. de pescado blanco"]
```

Extendiendo objetos nativos

Extender el *prototype* de una función no está limitado a las funciones

constructoras propias, ya que también se pueden extender los *prototypes* de funciones nativas, como `String`, `Number`, `Date` o `Array`, entre otros.

Esta posibilidad permite *mejorar*, en cierto sentido, el lenguaje y dotar a los objetos de métodos utilitarios. Un ejemplo de esto se da en la biblioteca Sugar.js, la cual extiende los objetos nativos de JavaScript para simplificar y automatizar algunas operaciones comunes como son operaciones entre arreglos, manejar cadenas, números o fechas.

Sin embargo, también existe la posibilidad de extender el *prototype* de objetos que están definidos en el entorno en el cual el programa está ejecutándose, como los que se utilizan en el Document Object Model (la interfaz en JavaScript para manipular HTML). Estos objetos son denominados *host objects*, debido a que su implementación depende del entorno en el que se ejecuta JavaScript.

Mientras que los objetos nativos (`String`, `Number`, `Date` o `Array`) están explícitamente especificados por ECMA, con ECMAScript, los *host objects* difieren entre implementaciones ya que sus especificaciones no son tan explícitas y interpretadas a libertad por quien decida ejecutar JavaScript en su propio entorno.

Es precisamente por la falta de explicitud en la especificación de los *host objects* que extender sus *prototypes* no solamente es recomendable, si no que se evita a toda costa, ya que su comportamiento varía entre implementaciones (es decir, entre navegadores). Mayores detalles se pueden encontrar en "What's wrong with extending the DOM".

Otra de los situaciones que sucede al extender objetos nativos es el hecho que algun nuevo método a implementar pueda ser implementado nativamente en una siguiente versión del lenguaje. El caso más llamativo es el de los métodos para manipular arreglos, como `forEach` o `map`, los cuales fueron agregados en las últimas versiones de navegadores como Chrome o Firefox. Esto es fácilmente solucionable verificando que el método no exista antes de implementarlo en el *prototype* del constructor.

¿Por qué no se debería extender `Object`?

Como se detalló en el capítulo anterior, cada nueva propiedad de un objeto es

enumerable. Esto quiere decir que si se agrega un nuevo método al *prototype* de `Object`, esta aparecerá cuando se iteren las propiedades de un objeto con `for..in`. (un método en JavaScript no es más que una propiedad cuyo valor es una función):

```
Object.prototype.superMethod = function() {
  return 'instance of Object';
};

var obj = {};       // un objeto plano es una instancia de Object

for (var property in obj) {
  console.log(property);
}
// superMethod

var string = "";    // una cadena también es una instancia de
Object

for (var property in string) {
  console.log(property);
}
// superMethod
```

De igual forma, si luego de haber extendido el *prototype* de `Object` se agrega una propiedad a un objeto plano con el mismo nombre de la nueva propiedad o método, el valor que devolverá será el de la propiedad del objeto plano. A este comportamiento se le denomina *property shadowing*:

```
Object.prototype.superMethod = function() {
  return 'instance of Object';
};

var obj = {
```

```
  superMethod: 150
};

obj.superMethod();
// TypeError: Property 'superMethod' of object #<Object> is not a
function
```

Patrones de diseño

Debido al auge que ha tenido JavaScript en los últimos años se hizo necesario crear y aplicar técnicas probadas que permitan escribir mejor código y solucionar problemas comunes. Estas técnicas son llamadas patrones de diseño y representan uno de los pilares en cuanto al desarrollo tanto de JavaScript como lenguaje como del uso que se le da al momento de crear aplicaciones web del lado frontend y backend.

Closure

Un closure en JavaScript es una función definida dentro de otra función, teniendo esta función (la función interna) acceso al ámbito (*scope*) de la función que la contiene (la función externa). En JavaScript este comportamiento no sucede a la inversa; es decir, una función externa no tiene acceso al ámbito de la función interna.

```
function buildTitle(parts) {
  var baseTitle = 'La Buena Espina';

  function getSeparator() {
    if (parts.length == 1) {
      return ' — ';
    }
    else {
      return ' › ';
```

```
    }
  };

  var separator = getSeparator();
  parts.unshift(baseTitle);

  return parts.join(separator);
}

buildTitle(['Carta']);
// "La Buena Espina – Carta"

buildTitle(['Carta', 'Postres']);
// "La Buena Espina › Carta › Postres"
```

Este ejemplo tiene una función interna llamada `getSeparator`, la cual tiene acceso al ámbito de la función externa (`buildTitle`) y a sus argumentos (`parts`). En este caso el uso de `getSeparator` se limita a crear el título del sitio, por lo que no es necesario hacer que `buildTitle` la retorne (Las funciones, al ser *ciudadanos de primera clase*, pueden retornar otras funciones).

Sin embargo, de ser necesario, el closure puede ser devuelto por la función externa, y aún tener acceso al ámbito de esa función (incluso después de haber ejecutado dicha función).

```
function titleBuilder() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' – ';
    }
    else {
      return ' › ';
```

```
    }
  };

  function addPart(part) {
    parts.push(part);

    return parts.join(getSeparator());
  };

  return addPart;
};

var builder = titleBuilder();
builder('Carta');
// "La Buena Espina — Carta"

builder('Pescados');
// "La Buena Espina › Carta › Pescados"

builder('Ceviches');
// "La Buena Espina › Menú › Pescados › Ceviches"
```

En este caso, la función `titleBuilder` tiene definidos dos closures: `getSeparator` y `addPart`. A diferencia del ejemplo anterior, `titleBuilder` devuelve el closure `addPart`, por lo que, al guardar el valor devuelto en la variable `builder`, este se vuelve una referencia del closure `addPart`. Como los closures guardan acceso del ámbito de su función externa, aún después de haber sido ejecutadas, puede recrear el título del sitio utilizando la variable `parts`, que a su vez ha sido modificada por el closure.

Module

Un módulo utiliza las funciones inmediatamente invocadas y los closures para encapsular el comportamiento de una función y hacer públicos solo la funciones que se consideren necesarios, mientras que el resto de operaciones y variables

quedan inaccesibles.

```
var titleBuilder = (function() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' — ';
    }
    else {
      return ' › ';
    }
  };

  return {
    reset: function() {
      parts = [baseTitle];
    },
    addPart: function(part) {
      parts.push(part);
    },
    toString: function() {
      return parts.join(getSeparator());
    }
  };
})();

titleBuilder;
// Object {reset: function, addPart: function, toString: function}

titleBuilder.toString();
// "La Buena Espina"

titleBuilder.addPart('Carta');
```

```
titleBuilder.addPart('Pescados');
titleBuilder.addPart('Ceviches');

titleBuilder.toString();
// "La Buena Espina › Carta › Pescados › Ceviches"

titleBuilder.reset();

titleBuilder.toString();
// "La Buena Espina"
```

Un módulo es un función inmediatamente invocada, la cual devuelve un objeto. Este objeto, a su vez, contiene closures con acceso al ámbito de la función inmediatamente invocada y a sus variables internas. Sin embargo, quedan variables, como `parts`, que no pueden manipularse fuera del módulo (excepto al usar el closure `addPart`), así como funciones, como `getSeparator` que no pueden ser utilizadas fuera del módulo.

Callbacks

Un callback es una función pasada como parámetro en otra función, la cual ejecuta el callback luego de haber realizado sus propias operaciones. Usualmente los callbacks son funciones anónimas.

Este extiende el *prototype* de `Array` para crear el método `each`:

```
// como es un objeto nativo, se verifica que el método no exista
antes de crearlo
if (!Array.prototype.each) {
  Array.prototype.each = function (callback) {
    // luego, se verifica que el callback sea una función
    if (callback instanceof Function) {
      var i;
```

```
      for (i = 0; i < this.length; i++) {
        // el callback de Array.prototype.each puede recibir 2
argumentos: elemento e índice
        callback.call(this[i], this[i], i);
      }
    }
  };
}

[0, 1, 2, 3, 4, 5].each(function(item, index) {
  console.log(item.toString(2), index);
});

// "0"    0
// "1"    1
// "10"   2
// "11"   3
// "100"  4
// "101"  5
```

Publish / Subscribe

El patrón publish / subscribe permite la comunicación entre objetos de forma asíncrona y define dos tipos de objetos: aquel que se suscribe a un canal (*subscriber*) y aquel que envía el mensaje (*publisher*). Cada suscripción permite definir un callback que se ejecutará cuando el objeto *publisher* envíe un mensaje en el canal el objeto *subscriber* está suscrito.

```
var SiteNotifier = (function() {                        // Este
patrón utiliza el patrón módulo...
  var channels = {};

  return {
    subscribe: function(channelName, callback) {        // ...con 2
```

```
closures: subscribe y publish
      if (channels[channelName] === undefined) {        // en
subscribe verifica si el canal existe
        channels[channelName] = [];                     // y si no,
lo inicializa como un array vacío
      }

      if (callback instanceof Function) {
        channels[channelName].push(callback);           // al ser
un closure tiene acceso a la variable channels del ámbito externo
      }
    },
    publish: function(channelName, message) {
      if (channels[channelName] instanceof Array) {     // solo se
ejecutará si el valor del canal es un array
        var subscribers = channels[channelName];

        for (var i = 0; i < subscribers.length; i++) {  // itera a
través de todos los suscriptores (callbacks)...
          var callback = subscribers[i];

          callback.call(null, message);                 // ...y las
ejecuta sin contexto, pasándole el mensaje como parámetro
        }
      }
    }
  };
})();

SiteNotifier.subscribe('site_title:changed', function(message) {
  console.log(message.oldTitle, ' → ', message.newTitle);
});

// La ventaja de utilizar este patrón radica en que se pueden
asignar más de un callback a una acción (o canal)
```

```
SiteNotifier.subscribe('site_title:changed', function(message) {
  document.title = message.newTitle;
});

document.title;
""                // En el caso de una ventana o pestaña nueva,
esta no tiene título

var message = { oldTitle: 'La Buena Espina', newTitle: 'Bienvenidos
a La Buena Espina' };
SiteNotifier.publish('site_title:changed', message);
// "La Buena Espina  →  Bienvenidos a La Buena Espina"

document.title;
// "Bienvenidos a La Buena Espina"
```

Este patrón se utiliza en casos donde se requiera condicionar la ejecución de una o más de una acción a la ejecución de otra previamente. Por ejemplo, si quisiera automatizar el cambio de título de la ventana al pasar de una sección a otra, podría utilizar `SiteNotifier` dentro del módulo `titleBuilder`:

```
var titleBuilder = (function() {
  var baseTitle = 'La Buena Espina';
  var parts = [baseTitle];

  function getSeparator() {
    if (parts.length == 2) {
      return ' – ';
    }
    else {
      return ' › ';
    }
  };

  return {
```

```
    reset: function() {
      var message = {
        oldTitle: this.toString()
      };

      parts = [baseTitle];
      message.newTitle = this.toString();

      SiteNotifier.publish('site_title:changed', message);
    },
    addPart: function(part) {
      var message = {
        oldTitle: this.toString()
      };

      parts.push(part);
      message.newTitle = this.toString();

      SiteNotifier.publish('site_title:changed', message);
    },
    toString: function() {
      return parts.join(getSeparator());
    }
  };
})();

SiteNotifier.subscribe('site_title:changed', function(message) {
  // en vez de mostrar en la consola se puede cambiar el título de
la pestaña
  console.log(message.oldTitle, ' → ', message.newTitle);
});

titleBuilder.addPart('Carta');
// "La Buena Espina  →  La Buena Espina — Carta"
titleBuilder.addPart('Pescados');
// "La Buena Espina — Carta  →  La Buena Espina › Carta › Pescados"
```

```
titleBuilder.addPart('Ceviches');
// "La Buena Espina › Carta › Pescados  →  La Buena Espina › Carta
› Pescados › Ceviches"

titleBuilder.reset();
// "La Buena Espina › Carta › Pescados › Ceviches  →  La Buena
Espina"
```

De esta forma, cada vez que agregue una parte al título (con `addPart`), o la devuelva a su estado original (con `reset`), se ejecutarán los callbacks suscritos al canal `site_title:changed`.

Mixins

En la sección sobre *prototypes* creamos la función constructora `Dish`, la cual permite recrear los platillos que ofrece La Buena Espina:

```
function Dish(options) {
  this.name = options.name;
  this.ingredients = options.ingredients;
  this.garnishes = options.garnishes;
  this.diners = options.diners;
};
```

Así mismo, creamos una nueva función llamada `Beverage`, que servirá para modelar las distintas bebidas que ofrece el restaurante:

```
function Beverage(options) {
  this.name = options.name;
  this.quantity = options.quantity;
};
```

Pero el dueño de La Buena Espina quiere tener una calculadora en el sitio web,

que permita saber cuánto gastará un posible cliente según lo que vaya a pedir, y para esto necesitamos que todos los items de la carta (en este caso, `Dish` y `Beverage`) tengan un método que agregue el precio a una calculadora.

Podríamos tener un *prototype* en común para ambos pero suena un poco forzado. ¿Cómo es que `Dish` y `Beverage` podrían tener un objeto *padre* en común? Ambos necesitan el mismo comportamiento, pero son muy diferentes para compartir un *prototype*. Es aquí donde podemos usar un *mixin*.

Un *mixin* es una colección de métodos que pueden ser agregados a un objeto (generalmente al *prototype* de una función constructora) y así extender las funcionalidades que tiene dicho objeto. De esta forma podemos simular la herencia múltiple que el lenguaje no da por sí mismo (en JavaScript se maneja herencia simple al extender o reemplazar el *prototype* de una función):

```
var CalculatorItems = [];

var CalculatorMixin = {
  addToCalculator: function(price, quantity) {
    CalculatorItems.push({
      name: this.name,
      price: price,
      subtotal: price * quantity
    });
  }
};

for (var mixinMethodName in CalculatorMixin) {
  Dish.prototype[mixinMethodName] =
CalculatorMixin[mixinMethodName];
  Beverage.prototype[mixinMethodName] =
CalculatorMixin[mixinMethodName];
}
```

La forma de usar un *mixin* es iterando en él (para eso utilizamos un `for..in`) y añadiendo cada método del *mixin* en el *prototype* destino. De esta forma,

podemos agregar un platillo a la calculadora:

```
var cevicheSimple = new Dish({
  name: 'Ceviche simple',
  diners: 4
});

cevicheSimple.addToCalculator(20, 1);

var limonadaFrozen = new Beverage({
  name: 'Limonada frozen',
  quantity: '1 vaso'
});

limonadaFrozen.addToCalculator(7, 2);
```

Y calculando:

```
var total = 0;

for (var i = 0; i < CalculatorItems.length; i++) {
  total = total + CalculatorItems[i].subtotal;
}

// toFixed convierte un número a una cadena con determinado número
de decimales
console.log('Total:', 'S/.', total.toFixed(2));
// Total: S/. 34.00
```

← 1. Capítulo 1: Entendiendo JavaScript 3. Capítulo 3: DOM y CSSOM →

Acerca de | Disponible en Amazon.com