Programación Funcional

¿Programación funcional?

La vamos a entender como

- Creación y manipulación de funciones
- Alteración de funciones
- Aplicación de funciones
- Asincronía

Funciones de orden superior

Funciones que devuelven funciones

- curry
- bind
- ¡Muchas otras!

Funciones de orden superior

Algunas de las más útiles:

- throttle
- debounce
- once
- after
- compose
- memoize

throttle

Controlar la frecuencia de invocación

- La función se invocará como máximo una vez
- Durante el periodo de tiempo especificado

throttle

```
var counter = 0,
    inc = function() { counter++; };

inc = throttle(inc, 10);

for (var i=100000; i--;) {
  inc();
}

alert(counter); // ~6
```

throttle

```
function throttle(fn, time) {
  var last = 0;
  return function() {
    var now = new Date();
    if ((now - last) > time) {
      last = now;
      return fn.apply(this, arguments);
    }
  }
}
```

debounce

Ejecutar la función cuando se deje de llamar

- La llamada se pospone hasta que pasen x ms
- Desde la última invocación

debounce

```
var counter = 0,
    inc = function() {
        counter++;
        alert(counter);
    };

inc = debounce(inc, 1000);

for (var i=100000; i--;) {
  inc();
}
```

debounce

```
function debounce(fn, time) {
  var timerId;
  return function() {
    var args = [].slice.call(arguments);
    if (timerId) clearTimeout(timerId);
    timerId = setTimeout(bind(this, function() {
      fn.apply(this, args);
    }), time);
  }
}
```

once

La función solo se puede invocar una vez

```
var counter = 0,
    inc = function() {
       counter++;
    };

inc = once(inc);

for (var i=100000; i--;) {
  inc();
}

alert(counter);
```

once

```
function once(fn) {
  var executed = false;
  return function() {
    if (!executed) {
      executed = true;
      return fn.apply(this, arguments);
    }
  }
}
```

after

La función se ejecuta solo tras haber sido invocada n veces

```
var counter = 0,
    inc = function() {
        counter++;
    };

inc = after(inc, 1000);

for (var i=100000; i--;) {
  inc();
}

alert(counter);
```

after

```
function after(fn, n) {
  var times = 0;
  return function() {
    times++;
    if (times % n == 0) {
      return fn.apply(this, arguments);
    }
  }
}
```

compose

Composición de funciones

```
function multiplier(x) {
  return function(y) { return x*y; }
}

var randCien = compose(Math.floor,
                       multiplier(100),
                       Math.random);
alert(randCien());
```

compose

```
function compose() {
  var fns = [].slice.call(arguments);
  return function(x) {
    var currentResult = x, fn;
    while (fn = fns.pop()) {
      currentResult = fn(currentResult);
    }
    return currentResult;
  }
}
```

memoize

Nunca calcules el mismo resultado 2 veces!

- La primera invocación calcula el resultado
- Las siguientes devuelven el resultado almacenado
- Solo vale para funciones puras

memoize

```
function fact(x) {
  if (x == 1) { return 1; }
  else { return x * fact(x-1); }
}

fact = memoize(fact);

var start = new Date();
fact(100);
console.log(new Date() - start);

start = new Date();
fact(100);
console.log(new Date() - start);
```

memoize

```
function memoize(fn) {
  var cache = {};
  return function(p) {
    var key = JSON.stringify(p);
    if (!(key in cache)) {
      cache[key] = fn.apply(this, arguments);
    }
    return cache[key];
  }
}
```

Asincronía

JS es, por naturaleza, asíncrono

- Eventos
- AJAX
- Carga de recursos

Asincronía

¿Qué significa asíncrono?

```
function asincrona() {
  var random = Math.floor(Math.random() * 100);
  setTimeout(function() {
    return random;
  }, random);
}
```

Asincronía

¿Cómo devuelvo el valor **random** desde dentro?

```
function asincrona() {
  var random = Math.floor(Math.random() * 100);
  setTimeout(function() {
    return random;
  }, random);
}
```

Asincronía

```
function asincrona(callback) {
  var random = Math.floor(Math.random() * 1000);
  setTimeout(function() {
    callback(random);
  }, random);
}

asincrona(function(valor) {
  alert(valor);
});
```

Asincronía

```
function asincrona(callback) {
  var random = Math.floor(Math.random() * 1000);
  setTimeout(function() {
    callback(random)
  }, random);
}

asincrona(function(valor) {
  alert(valor);
});
```

Asincronía

```
function asincrona(callback) {
  var random = Math.floor(Math.random() * 1000);
  setTimeout(function() {
    callback(random);
  }, random);
}

asincrona(function(valor) {
  alert(valor);
});
```

Asincronía

Promesas

- Otra forma de escribir código asíncrono
- Más fácil de manipular
- Más fácil de combinar

Asincronía

Promesas

- Una idea muy sencilla:
  - Un objeto que representa una tarea con dos posibles resultados: éxito o fracaso
- Podemos añadir callbacks a cualquiera de las dos resoluciones
- jQuery: deferreds

Promesas

```
var promesa = $.get('/mydata');
promesa.done(onSuccess);
promesa.fail(onFailure);
promesa.always(onAlways);
```

Promesas

Crear una promesa con jQuery:

```
var promesa = new $.Deferred();
```

Promesas

Intervienen dos actores:

- El creador de la promesa
  - Instancia la promesa
  - Controla el resultado
- El cliente
  - Recibe la promesa
  - Añade callbacks

Promesas

Empecemos por aquí:

- http://jsbin.com/ohuhiw/9/edit

Promesas

Quiero que:

- Cuando el usuario resuelva una fila
- Se le notifique con un alert
- Diciéndole “Ok!” o “Fallo...”

Promesas

Con eventos y callbacks:

- http://jsbin.com/ohuhiw/6/edit

```
list.subscribe('added', function(prompt) {
  prompt.subscribe('change:acabado', function() {
    if (prompt.get('resultado')) {
      alert("Ok!");
    } else {
      alert("Fallo...");
    }
  });
  var view = new PromptView({model: prompt}).render();
});
```

Promesas

Para implementar algo así con promesas:

- El modelo es el creador de la promesa
- Se encarga de resolverla o hacer que falle
- La devuelve para que el cliente la utilice
- http://jsbin.com/ohuhiw/11/edit

Promesas

Constructor

- Instancia la promesa
- `resolve()`: realiza la promesa con éxito
- `reject()`: realiza la promesa con fracaso

Cliente

- `done(callback)`: a llamar cuando se resuelva con éxito
- `fail(callback)`: a llamar cuando se resuelva con fracaso
- `always(callback)`: a llamar cuando se resuelva en cualquiera de los dos casos

Promesas

Por ahora ofrece pocas ventajas...

- Un interfaz fallo/éxito
- Pasar promesas y programar comportamientos desacoplados del origen de la promesa

Promesas

Ahora quiero que:

- Se muestre un alert
- Diciendo “Todo Ok!” si todas las filas son éxitos
- O diciendo “Fallo!” si alguna fila falla

Promesas

Con eventos y callbacks:

- Complejo
- Proclive a errores
- No muy flexible

Promesas

Con promesas:

- ¡Muy fácil!
- Solo hay que combinar promesas

Promesas

```
var prom3 = $.when(prom1, prom2);
```

- `prom3` es una nueva promesa
- Que combina `prom1` y `prom2`
- Con un `AND`

Promesas

Aquí tenemos un ejemplo:

- http://jsbin.com/ohuhiw/15/edit

Promesas

Ventajas sobre callbacks:

- Varios callbacks en cada posible resolución
- Las promesas se pueden combinar
- Más apropiadas para flujos complejos