

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 = 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;  
    for (var i=fns.length; i--;) {  
      fn = fns[i];  
      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 un estado futuro
- El estado futuro puede ser:
 - La resolución de la promesa en un valor
 - El rechazo de la promesa con un error
- Mucho, mucho más fácil de manejar que los callbacks

Promesas

```
function onSuccess(data) {  
    /* ... */  
}
```

```
function onFailure(e) {  
    /* ... */  
}
```

```
var promesa = $.get('/mydata');  
promesa.then(onSuccess, onFailure);
```

Promesas

`promise.then(onSuccess [, onFailure])`

- En caso de éxito, se invoca a **onSuccess** con el valor
- En caso de error, se invoca a **onFailure**
- Devuelve, a su vez, una promesa

Promesas

¿Para qué sirven?

- Dar un aspecto más coherente al código
- Hacer más explícito el flow
- Gestionar los errores en cascada

Promesas

```
Parse.User.logIn("user", "pass", {
  success: function(user) {
    query.find({
      success: function(results) {
        results[0].save({ key: value }, {
          success: function(result) {
            // El objeto se guardó.
          },
          error: function(result, error) {
            // Error.
          }
        });
      },
      error: function(error) {
        // Error.
      }
    });
  },
  error: function(user, error) {
    // Error.
  }
});
```

Promesas

```
Parse.User.logIn("user", "pass").then(function(user) {  
    return query.find();  
}).then(function(results) {  
    return results[0].save({ key: value });  
}).then(function(result) {  
    // El objeto se guardó.  
}, function(error) {  
    // Error.  
});
```


Promesas

```
Parse.User.logIn("user", "pass").then(function(user) {  
    return query.find();  
}).then(function(results) {  
    return results[0].save({ key: value });  
}).then(function(result) {  
    // El objeto se guardó.  
}, function(error) {  
    // Error.  
});
```

Promesas

Casos: cuando **onSuccess** devuelve un valor

```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    return 42;  
}).then(function(valor) {  
    return "La respuesta es " + valor;  
}).then(function(mensaje) {  
    console.log(mensaje);  
});
```

Promesas

Casos: cuando **onSuccess** devuelve un valor

```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    return 42;  
}).then(function(valor) {  
    return "La respuesta es " + valor;  
}).then(function(mensaje) {  
    console.log(mensaje);  
});
```

Promesas

Casos: llamando varias a veces a `.then`

```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    console.log("primer onSuccess!");  
});  
  
promise.then(function() {  
    console.log("segundo onSuccess!");  
});
```

Promesas

Casos: llamando varias a veces a `.then`

```
/* siendo promise una promesa... */
```

```
promise.then(function() {  
    console.log("primer onSuccess!");  
}, function(e) {  
    console.log("primer onFailure...");  
});
```

```
promise.then(function() {  
    console.log("segundo onSuccess!");  
}, function(e) {  
    console.log("segundo onFailure...");  
});
```

Promesas

Casos: capturar errores

```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    throw new Error("Oops!");  
}).then(function() {  
    console.log("Nunca llegamos aquí...");  
}, function(e) {  
    console.log("Vaya por Dios!");  
    console.log(e);  
});
```

Promesas

Casos: capturar errores

```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    throw new Error("Oops!");  
}).then(function() {  
    console.log("Nunca llegamos aquí...");  
}, function(e) {  
    console.log("Vaya por Dios!");  
    console.log(e);  
});
```

Promesas

Casos: cascada de errores

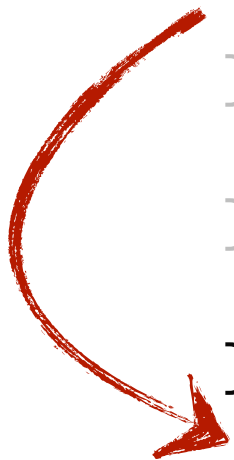
```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    throw new Error("Oh no!");  
}).then(function() {  
    console.log("Nunca se ejecuta.");  
}).then(function() {  
    console.log("Esto tampoco.");  
}, function(e) {  
    console.log("Vaya por Dios!");  
    console.log(e);  
});
```


Promesas

Casos: cascada de errores

```
/* siendo promise una promesa... */
```

```
promise.then(function() {  
    throw new Error("Oh no!");  
}).then(function() {  
    console.log("Nunca se ejecuta.");  
}).then(function() {  
    console.log("Esto tampoco.");  
}, function(e) {  
    console.log("Vaya por Dios!");  
    console.log(e);  
});
```



Promesas

Casos: errores localizados


```
/* siendo promise una promesa... */

promise.then(function() {
  throw new Error("Oh no!");
}).then(function() {
  console.log("Nunca se ejecuta.");
}, function(e) {
  console.log("Manejador del error");
}).then(function() {
  /* ... */
}, function(e) {
  /* este manejador no se ejecuta! */
});
```

Promesas

Casos: errores localizados

```
/* siendo promise una promesa... */  
  
promise.then(function() {  
    throw new Error("Oh no!");  
}).then(function() {  
    console.log("Nunca se ejecuta.");  
}, function(e) {  
    console.log("Manejador del error");  
}).then(function() {  
    /* ... */  
}, function(e) {  
    /* este manejador no se ejecuta! */  
});
```



Promesas

¿Cómo creo una promesa?

Promesas

Deferreds o diferidos

- Objetos que nos permiten crear y controlar promesas de valores futuros
- Dos operaciones:
 - **resolve**: resuelve la promesa como exitosa
 - **reject**: rechaza la promesa como fracasada

Promesas

Promesa	Diferido
Representa un valor futuro	Controla la generación del valor
<code>onSuccess</code>	<code>resolve(valor)</code>
<code>onFailure</code>	<code>reject(error)</code>

Promesas

```
function enDiezSegundos() {  
    var diferido = new R.Deferred();  
    setTimeout(function() {  
        diferido.resolve(new Date());  
    }, 10*1000);  
    return diferido.promise();  
}  
  
var promesa = enDiezSegundos();  
  
promesa.then(function(elFuturo) {  
    console.log("Ya han pasado diez segundos!");  
    console.log(elFuturo.getTime());  
});
```

Promesas

`Deferred#resolve([arg1, arg2, ...])`

- Resuelve la promesa (ejecuta el callback **onSuccess**)
- Los parámetros con los que se llame a `.resolve()` serán los que reciba el callback **onSuccess**
- Solo se debería llamar una vez

Promesas

`Deferred#reject([arg1, arg2, ...])`

- Rechaza la promesa (ejecuta el callback `onFailure`)
- Los parámetros con los que se llame a `.reject()` serán los que reciba el callback `onFailure`
- Solo se debería llamar una vez

Promesas

`Deferred#promise()`

- Devuelve la promesa asociada al diferido

Promesas

Deferred#then(onSuccess, onFailure)

- Exactamente igual que hacer:
`deferred.promise().then(...);`

Promesas

Vamos a crear una librería de promesas

- Una implementación sencilla
- Que satisfaga la especificación **Promises/A+**
 - <http://promises-aplus.github.com/promises-spec/>
- [tema2/r-promise/index.html](#)

Promesas

Por dónde empezar:

- Poder crear instancias de diferidos
- Poder poner un callback de éxito y uno de fracaso
- **.then()**
 - Por ahora, que no devuelva nada
 - Solo se puede llamar a una vez por diferido
- **.resolve([arg1, ...])** y **.reject([arg1, ...])**
 - Invocan el callback adecuado
 - Pasándole los parámetros adecuados

Promesas

Siguientes pasos:

- Poder invocar a `.then()` varias veces
 - Es decir, tener varios callbacks para cada caso en un mismo diferido
- Que funcione el primer ejemplo del ejercicio

Lo último a abordar:

- Que las llamadas a `.then()` se puedan encadenar
- Es decir, que `.then()` devuelva a su vez una promesa
- Que funcione el segundo ejemplo

Promesas

`when(pov1 [, pov2, ...])`

- Dos utilidades:
 - Homogeneizar promesas y valores en el código
 - Combinar varias promesas/valores
- Devuelve siempre una promesa
- La promesa devuelta:
 - Se resolverá si todas las promesas se resuelven.
 - Los parámetros del callback son los valores devueltos por cada una de las promesas.
 - Se rechazará en caso contrario

Promesas

```
R.Deferred.when(1, 2, 3).then(function(a, b, c) {  
    console.log(a, b, c); // 1 2 3  
});
```


Promesas

```
var p1 = new R.Deferred(),  
    p2 = new R.Deferred(),  
    p3 = new R.Deferred();
```

```
R.Deferred.when(p1, p2, p3).then(function(a, b, c) {  
    console.log(a, b, c); // 1 2 3  
});
```

```
p1.resolve(1);  
p2.resolve(2);  
p3.resolve(3);
```

Promesas

```
/* Homogeneizar */
```

```
var promesa0Valor = noSeQueDevuelve();
```

```
R.Deferred.when(promesa0Valor).then(function(valor) {  
    console.log(valor);  
});
```

Promesas

```
/* Homogeneizar */
```

```
var valor = 4,  
    promesa = new R.Deferred();
```

```
R.Deferred.when(valor, promesa).then(function(a, b) {  
    console.log(a, b); // 4, 5  
});
```

```
promesa.resolve(5);
```

Promesas

```
var valor = 4,  
    promesa = new R.Deferred();
```

```
R.Deferred.when(valor, promesa).then(function(a, b) {  
    console.log(a, b);  
}, function(e) {  
    alert("Oh, no!");  
});
```

```
promesa.reject("No funciona");
```

Promesas

Implementa `R.Deferred.when()`

- [tema2/when/index.html](#)