

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);  
});
```

CPS

Los callbacks tienen muchas ventajas

- Muy fáciles de entender e implementar
- Familiares para el programador JavaScript
- Extremadamente flexibles (clausuras, funciones de primer orden, etc, ...)
- Un mecanismo universal de asincronía/continuaciones

Pero...

CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

Pyramid of Doom

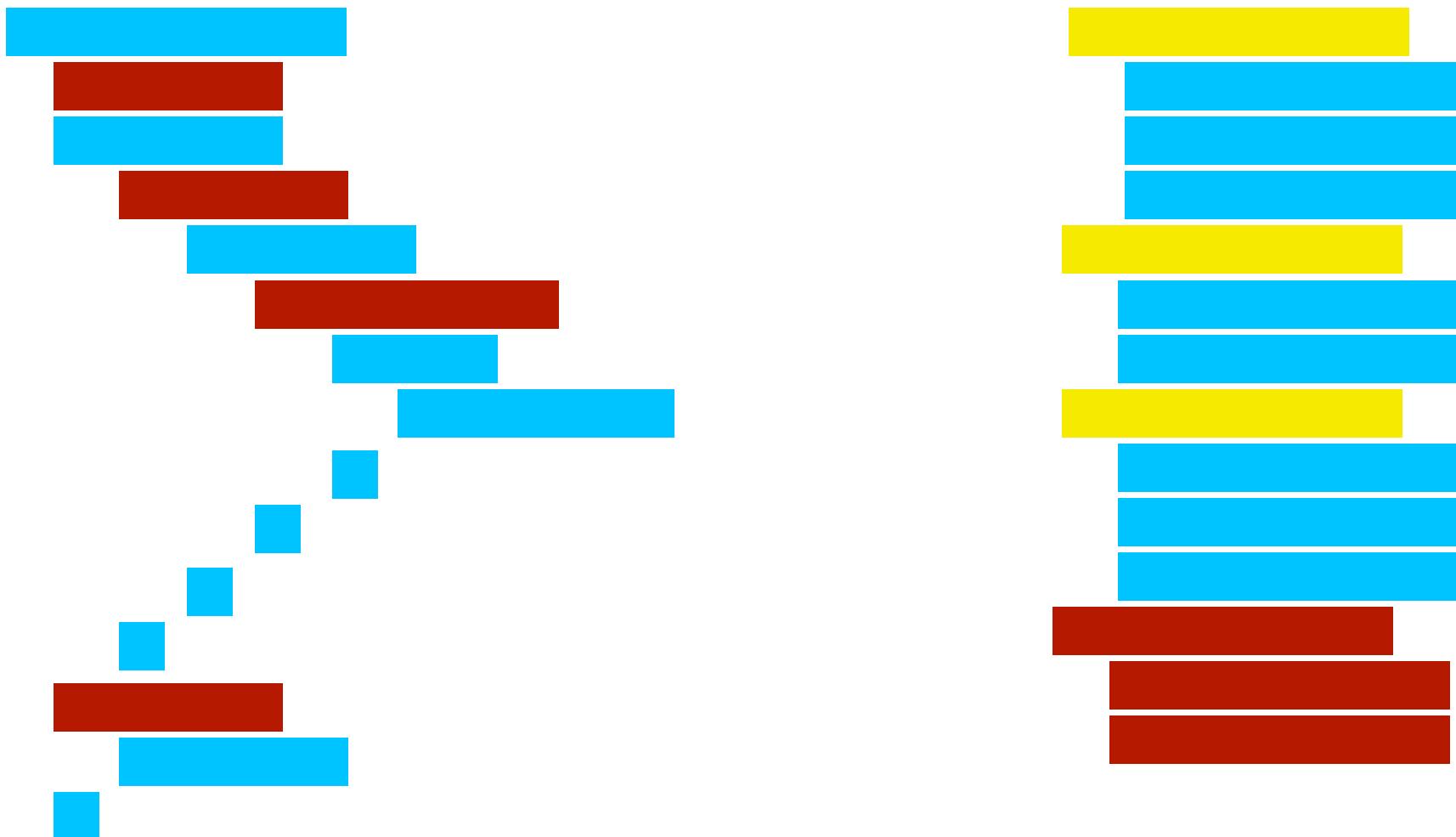
Callback Hell

CPS

```
var fs = require("fs");

fs.exists("./hola.txt", function(exists) {
  if (exists) {
    fs.readFile("./hola.txt", function(err, data) {
      if (err) {
        // MANEJO DE ERROR
      } else {
        fs.writeFile("./copia.txt", data, function(err) {
          if (err) {
            // MANEJO DE ERROR
          } else {
            console.log("OK!");
          }
        })
      }
    })
  } else {
    // MANEJO DE ERROR
  }
});
```

CPS vs. Promises



Promesas

Una manera alternativa de modelar asincronía

- Construcción explícita del flujo de ejecución
- Separación en bloques consecutivos
- Manejo de errores más controlado
- Combinación de diferentes flujos asíncronos

Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```

Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
  // bloque  
  return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
  // bloque  
  return writeFilePromise("./copia.txt");  
})  
.then(function() {  
  console.log("listo!");  
})  
.fail(function(err) {  
  // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

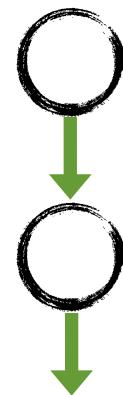
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
  // bloque  
  return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
  // bloque  
  return writeFilePromise("./copia.txt");  
})  
.then(function() {  
  console.log("listo!");  
})  
.fail(function(err) {  
  // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

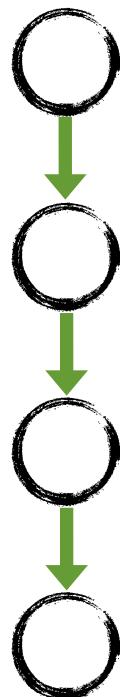
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

Una promesa = Un flujo de ejecución

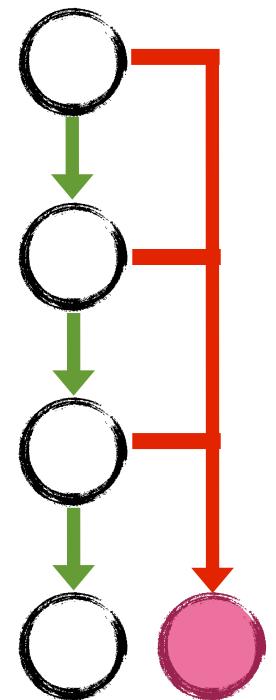
```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



Promesas

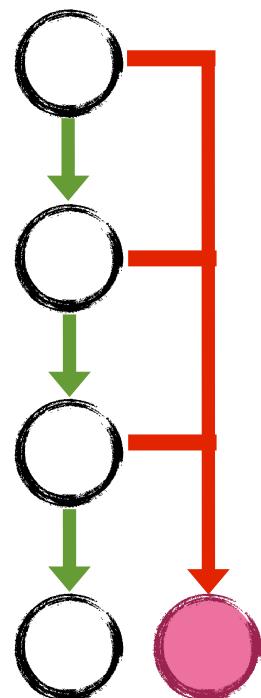
Una promesa = Un flujo de ejecución

```
promesa.then(function() {  
    // bloque  
    return readFilePromise("./hola.txt");  
})  
.then(function(data) {  
    // bloque  
    return writeFilePromise("./copia.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    // MANEJO DEL ERROR  
});
```



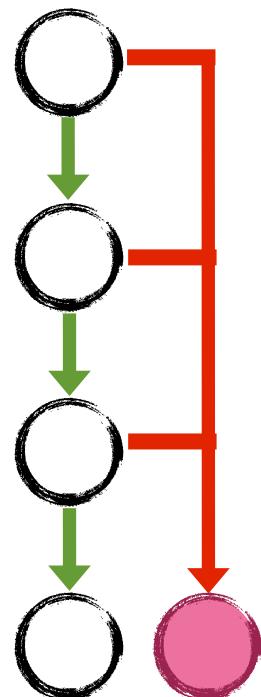
Promesas

Una promesa = Un flujo de ejecución



Promesas

¡Pero aún **no hemos ejecutado nada!** Solamente hemos construído el flujo



Promesas

¿Ventajas?

- Código mucho más ordenado y más legible
- Mejor control de errores
- **Podemos manipular el flujo**
 - Añadir nuevas etapas
 - Devolverlo en funciones
 - Pasarlo como parámetro
- **Podemos combinar varios flujos**

Promesas

```
function copyFile(from, to) {
  return readFilePromise(from);
  .then(function(data) {
    // bloque
    return writeFilePromise(to);
  });
}

copyFile("./hola.txt", "./copia.txt")
  .then(function() {
    return copyFile("./otraCosa.txt", "./copia2.txt");
  })
  .then(function() {
    console.log("listo!");
  })
  .fail(function(err) {
    console.log("Oops!");
  })
}
```

Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
}
```

```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```

Promesas

```
function copyFile(from, to) {  
    return readFilePromise(from);  
    .then(function(data) {  
        // bloque  
        return writeFilePromise(to);  
    });  
}  
}
```



```
copyFile("./hola.txt", "./copia.txt")  
.then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
})  
.then(function() {  
    console.log("listo!");  
})  
.fail(function(err) {  
    console.log("Oops!");  
})
```

Promesas

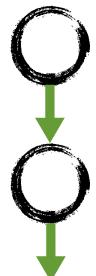
```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
}
```

```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```

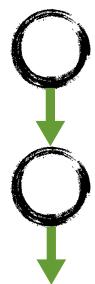


Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  });  
}  
}
```

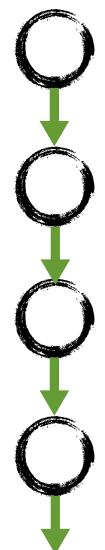


```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
  
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  } );  
}  
}
```

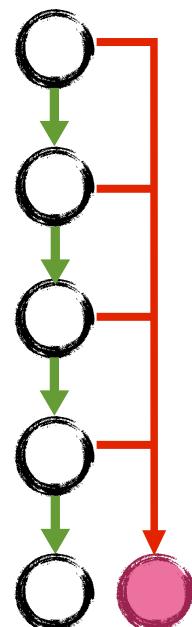
```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

```
function copyFile(from, to) {  
  return readFilePromise(from);  
  .then(function(data) {  
    // bloque  
    return writeFilePromise(to);  
  });  
}  
}
```

```
copyFile("./hola.txt", "./copia.txt")  
  .then(function() {  
    return copyFile("./otraCosa.txt", "./copia2.txt");  
  })  
  .then(function() {  
    console.log("listo!");  
  })  
  .fail(function(err) {  
    console.log("Oops!");  
  })
```



Promesas

.then(success, [error])

- Concatena bloques
- El nuevo bloque (**success**)...
 - Sólo se ejecuta si el anterior se ha ejecutado sin errores
 - Recibe como parámetro el resultado del bloque anterior
 - Devuelve el valor que se le pasará el siguiente bloque
 - ➔ Si es un **dato inmediato**, se pasa tal cual
 - ➔ Si es una **promesa**, se resuelve antes de llamar al siguiente bloque
- El segundo parámetro pone un manejador de error
 - Equivalente a llamar a .fail(error)
- **.then(...)** **siempre devuelve una nueva promesa**

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");
```

```
promesa = promesa.then(function(data) {  
    console.log("Contenido del fichero: ", data);  
}, function(err) {  
    console.log("Ooops!", err);  
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

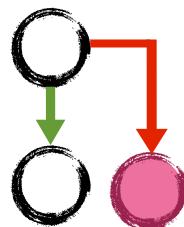
```
var promesa = readFilePromise("./hola.txt");

promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```

Promesas

```
var promesa = readFilePromise("./hola.txt");

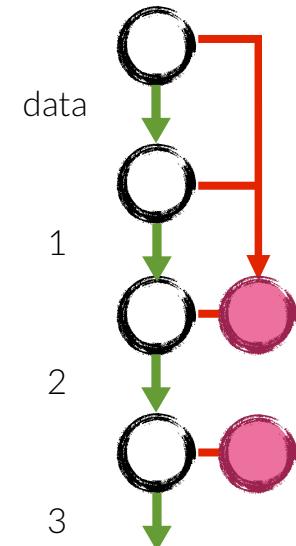
promesa = promesa.then(function(data) {
    console.log("Contenido del fichero: ", data);
}, function(err) {
    console.log("Ooops!", err);
})
```



Promesas

```
var promesa = readFilePromise("./hola.txt");

promesa.then(function(data) {
    return 1;
})
.then(function(uno) {
    return 2;
}, function(err) {
    console.log("Oh, oh...");
})
.then(function(dos) {
    return 3;
})
.fail(function(err) {
    console.log("Oops!");
});
```

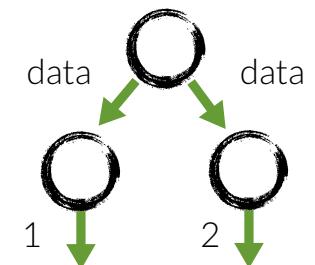


Promesas

```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});
```



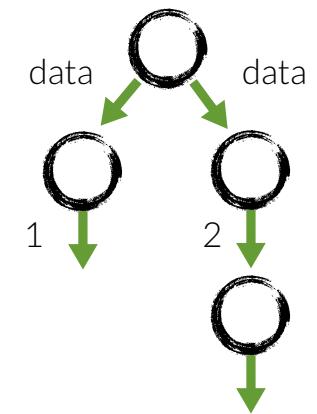
Promesas

```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});

promesa3.then(function(dos) {
    console.log("Ping!");
});
```



Promesas

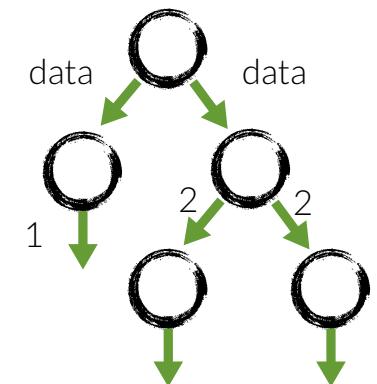
```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});

promesa3.then(function(dos) {
    console.log("Ping!");
});

promesa3.then(function(dos) {
    console.log("Pong!");
});
```



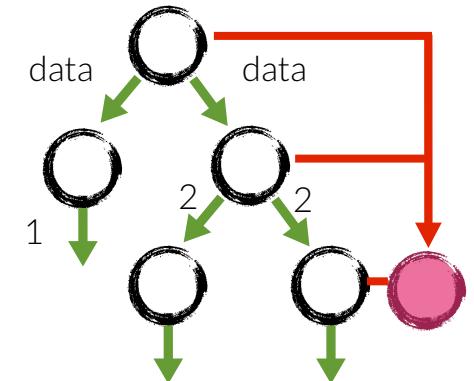
Promesas

```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
    return 1;
});

var promesa3 = promesa.then(function(data) {
    return 2;
});

promesa3.then(function(dos) {
    console.log("Ping!");
});
promesa3.then(function(dos) {
    console.log("Pong!");
}, function(err) {
    console.log("Oh, oh...");
});
```



Promesas: walled garden

Vamos a empezar a trastear con promesas...

- Pero, de momento, con una librería de mentira
- Para asentar conceptos
- (y perder el miedo)

Promesas: walled garden

¿Cómo creo una promesa?

Promesas: walled garden

¿Cómo creo una promesa?

```
var promise = fakePromise.create();
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    console.log("hola");
})
.then(function() {
    console.log("soy un flujo de ejecución");
})
.then(function() {
    console.log("expresado con promesas");
});
```

¿Qué se muestra por consola al ejecutar esto?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    console.log("hola");
})
.then(function() {
    console.log("soy un flujo de ejecución");
})
.then(function() {
    console.log("expresado con promesas");
});
```

¿Por qué?

Promesas: walled garden

Una promesa = un flujo de ejecución

- Configuramos un árbol de flujos e ejecución
- Añadimos bloques a promesas
- Pero... **¿Cuándo se empieza a ejecutar?**

Promesas: walled garden

Una promesa = un flujo de ejecución

- Configuramos un árbol de flujos e ejecución
- Añadimos bloques a promesas
- Pero... ¿Cuándo se empieza a ejecutar?
- **Cuando se resuelva la primera promesa del árbol**

Promesas: walled garden

Las promesas se **resuelven** o se **rechazan**

Si se resuelven:

- Se resuelven a un valor (si es un bloque, su valor de retorno)
- Representan el estado “OK, puede seguir el siguiente”

Si se rechazan:

- Representan un error
- La ejecución cae hasta el siguiente manejador de errores
- Se saltan todos los estados desde el error hasta el manejador

Promesas

¿Cómo **resuelvo** mi promesa?

```
var promise = fakePromise.create();
```

Promesas

¿Cómo **resuelvo** mi promesa?

```
var promise = fakePromise.create();  
promise.resolve("Un valor!");
```

Promesas

¿Cómo **rechazo** mi promesa?

```
var promise = fakePromise.create();
```

Promesas

¿Cómo **rechazo** mi promesa?

```
var promise = fakePromise.create();
promise.reject(new Error("Oh, oh..."));
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    console.log("hola");
})
.then(function() {
    console.log("soy un flujo de ejecución");
})
.then(function() {
    console.log("expresado con promesas");
});
```

¿Cómo puedo hacer que ejecute?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    console.log("hola");
})
.then(function() {
    console.log("soy un flujo de ejecución");
})
.then(function() {
    console.log("expresado con promesas");
});

promise.resolve();
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    console.log("hola");
})
.then(function() {
    console.log("soy un flujo de ejecución");
})
.then(function() {
    console.log("expresado con promesas");
});

promise.reject();
```

¿Qué pasa?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    console.log("hola");
})
.then(function() {
    console.log("soy un flujo de ejecución");
})
.then(function() {
    console.log("expresado con promesas");
});

promise.reject();
```

¿Qué pasa?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    console.log("La promesa se resolvió con:", valor);
});

promise.resolve(42);
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    console.log("La promesa se resolvió con:", valor);
});

promise.resolve(42)
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    console.log("La promesa se resolvió con:", valor);
})
.fail(function(e) {
    console.log("La promesa falló porque:", e.message);
});

promise.reject(new Error("fue rechazada"));
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    console.log("La promesa se resolvió con:", valor);
})
.fail(function(e) {
    console.log("La promesa falló porque:", e.message);
});

promise.reject(new Error("fue rechazada"));
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
  console.log("La promesa se resolvió con:", valor);
}, function(e) {
  console.log("La promesa falló porque:", e.message);
})
.then(function() {
  console.log("¿Me muestro?");
});

promise.reject(new Error("fue rechazada"));
```

¿Qué sucede aquí?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
  console.log("La promesa se resolvió con:", valor);
}, function(e) {
  console.log("La promesa falló porque:", e.message);
})
.then(function() {
  console.log("¿Me muestro?");
});

promise.reject(new Error("fue rechazada"));
```

¿Qué sucede aquí?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    console.log("La promesa se resolvió con:", valor);
}, function(e) {
    console.log("La promesa falló porque:", e.message);
})
.then(function() {
    console.log("¿Me muestro?");
});

promise.reject(new Error("fue rechazada"));
```

¿Qué sucede aquí?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    console.log("La promesa se resolvió con:", valor);
}, function(e) {
    console.log("La promesa falló porque:", e.message);
})
.then(function() {
    console.log("¿Me muestro?");
});

promise.reject(new Error("fue rechazada"));
```

Se considera que el error ha sido “controlado”

Promesas: walled garden

```
var promise = fakePromise.createJquery();  
  
promise.then(function(valor) {  
    console.log("La promesa se resolvió con:", valor);  
, function(e) {  
    console.log("La promesa falló porque:", e.message);  
})  
.then(function() {  
    console.log("¿Me muestro?");  
});  
  
promise.reject(new Error("fue rechazada"));
```

No todas las implementaciones lo respetan..

Promesas: walled garden

```
var promise = fakePromise.create();

promise.resolve(42);

promise.then(function(valor) {
  console.log("La promesa se resolvió con:", valor);
});
```

¿Qué pasa ahora?

Promesas: walled garden

Las promesas se resuelven o rechazan **una sola vez**

- Recuerdan su **valor** y su **estado**
- Cualquier llamada posterior a `.then` o `.fail` se ejecuta inmediatamente

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    return valor * 2;
})
.then(function(n) {
    console.log(n);
});

promise.resolve(42);
```

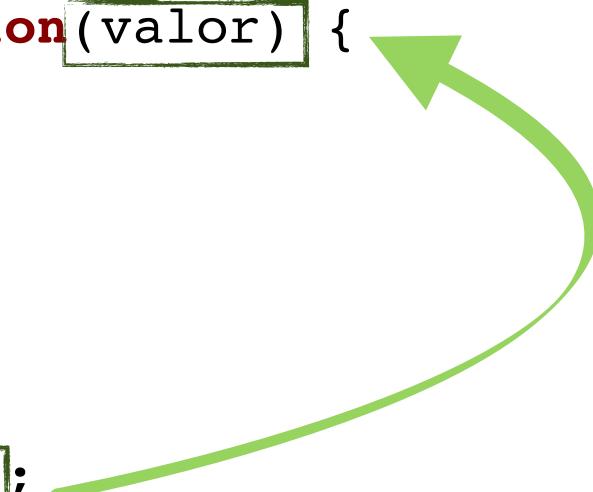
¿Qué se muestra por la consola?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    return valor * 2;
})
.then(function(n) {
    console.log(n);
});

promise.resolve(42);
```



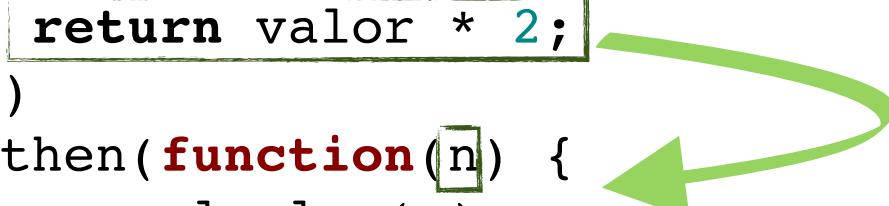
¿Qué se muestra por la consola?

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function(valor) {
    return valor * 2;
})
.then(function(n) {
    console.log(n);
});

promise.resolve(42);
```



¿Qué se muestra por la consola?

Promesas: walled garden

Otra manera de rechazar una promesa es lanzar una excepción desde el interior de un bloque

```
promise.then(function() {
  return "hola";
})
.then(function(msg) {
  console.log(msg);
  throw new Error("Oh, oh...");
  return "mundo";
})
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    return "hola";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("Oh, oh...");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

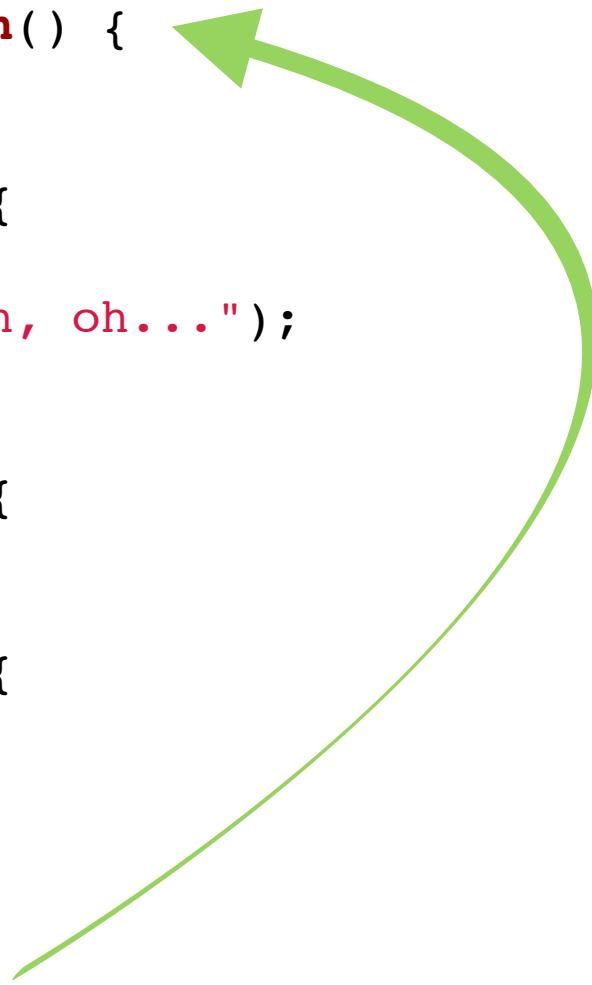
promise.resolve(42);
```

Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    return "hola";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("Oh, oh...");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

promise.resolve(42);
```



Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    return "hola";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("Oh, oh...");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

promise.resolve(42);
```



Promesas: walled garden

```
var promise = fakePromise.create();

promise.then(function() {
    return "hola";
})
.then(function(msg) {
    console.log(msg);
    throw new Error("Oh, oh...");
    return "mundo";
})
.then(function(msg) {
    console.log(msg);
})
.fail(function(err) {
    console.log(err);
    return "MAL!";
})

promise.resolve(42);
```



```
throw new Error("Oh, oh...");
```

¡A teclear!

Vamos a crear una librería de promesas

- Empezando por satisfacer los casos que hemos visto
- Completándola poco a poco
- Hasta satisfacer el estándar
 - <http://promises-aplus.github.com/promises-spec/>
- tema2/r-promise/index.html

r-promise.js

El primer paso:

```
var promise = new Prom();

promise.then(function() {
    console.log("Me resuelvo!");
});

promise.resolve();
```

r-promise.js

```
window.Prom = function() {
    var self = {},
        callback;

    self.then = function(onSuccess) {
        callback = onSuccess;
    };

    self.resolve = function() {
        callback();
    };

    return self;
};
```

r-promise.js

¿Cómo le podríamos pasar un valor de resolución?

```
var promise = new Prom();

promise.then(function(v) {
    console.log("Me resuelvo con:", v);
});

promise.resolve(42);
```

r-promise.js

A una promesa le podemos añadir varios callbacks llamando varias veces a `.then()`

```
var promise = new Prom();

promise.then(function(v) {
    console.log("Me resuelvo con:", v);
});

promise.then(function() {
    console.log("Soy otro callback");
});

promise.resolve(42);
```

r-promise.js

Una promesa ya resuelta, recuerda su estado y su valor (y no se puede cambiar!)

```
var promise = new Prom();

promise.resolve(12);

promise.then(function(v) {
    console.log("Resuelta con:", v);
});

promise.resolve(25);
```

r-promise.js

Se puede pasar un callback de error como segundo parámetro a `.then()`

```
var promise = new Prom( );

promise.then(function(v) {
    console.log("Resuelta con:", v);
}, function(e) {
    console.log("Oh, oh...", e.message);
});

promise.reject(new Error("Una catástrofe!"));
```

r-promise.js

Todos los callbacks de error deberían ejecutarse (en orden)

```
var promise = new Prom();

promise.then(function(v) {
    console.log("Resuelta con:", v);
}, function(e) {
    console.log("Oh, oh...", e.message);
});

promise.then(function(v) {
    console.log("segundo cb de éxito:", v);
}, function(e) {
    console.log("segundo cb de error:", e.message);
});

promise.reject(new Error("Una catástrofe!"));
```

r-promise.js

Nivel 2: .then() ha de devolver una promesa

```
var promise = new Prom(),
promise2;

promise2 = promise.then(function(v) {
    console.log("promesa 1 resulta con:", v);
});

promise2.then(function() {
    console.log("promesa 2 resuelta!");
});

promise.resolve(42);
```

r-promise.js

El valor de retorno de un bloque es el parámetro del siguiente

```
var promise = new Prom();

promise.then(function(v) {
    console.log("promesa 1 resulta con:", v);
    return v * 2;
})
.then(function(v2) {
    console.log("El doble es:", v2);
});

promise.resolve(42);
```

r-promise.js

Si un bloque captura un error, el siguiente se ejecuta normalmente

```
var promise = new Prom();

promise.then(function() {
    console.log("aquí no llego!");
}, function(e) {
    console.log("Nos recuperamos de:", e.message);
})
.then(function() {
    console.log("Todo ok!");
});

promise.reject(new Error("la frustración"));
```

r-promise.js

Si el bloque no captura el error, cae en cascada

```
var promise = new Prom();

promise.then(function() {
    console.log("aquí no llego!");
})
.then(function() {
    console.log("Aquí tampoco...");
})
.then(null, function(e) {
    console.log("El ERROR ha caído hasta aquí!", e.message);
});

promise.reject(new Error("Imparable"));
```

r-promise.js

Si un bloque levanta una excepción, la promesa que devolvió se rechaza

```
var promise = new Prom();

promise.then(function() {
    throw new Error("Oh my!");
})
.then(null, function(e) {
    console.log("Capturo el error:", e.message);
});

promise.resolve(42);
```

r-promise.js

De nuevo, el error cae en cascada hasta ser capturado

```
var promise = new Prom();

promise.then(function() {
    throw new Error("Oh my!");
})
.then(function() {
    console.log("soy saltado");
})
.then(null, function(e) {
    console.log("Capturo el error:", e.message);
});

promise.resolve(42);
```

checkpoint

```
var promise = new Prom();

promise.then(function(v) {
    console.log("Me llega:", v);
    return v*2;
})
.then(null, function(e) {
    console.log("Capturo el error:", e.message);
});

promise.then(function(v) {
    console.log(v, "entre 2 es...");
    return v/2;
})
.then(function(vp2) {
    console.log(vp2);
});

promise.resolve(42);
```

checkpoint

```
var promise = new Prom();

promise.then(function(v) {
    console.log("Me llega:", v);
    return v*2;
})
.then(null, function(e) {
    console.log("Capturo el error:", e.message);
});

promise.then(function(v) {
    console.log(v, "entre 2 es...");
    return v/2;
})
.then(function(vp2) {
    console.log(vp2);
});

promise.reject(new Error("tests"));
```

r-promise.js

Nivel 3: Si un bloque devuelve una promesa, la ejecución se para hasta que se resuelva

```
var promise = new Prom(),
    promise2 = new Prom();

promise.then(function() {
    return promise2;
})
.then(function() {
    console.log("Qué pasa aquí?");
});

promise.resolve();
```

r-promise.js

Nivel 3: Si un bloque devuelve una promesa, la ejecución se para hasta que se resuelva

```
var promise = new Prom(),
    promise2 = new Prom();

promise.then(function() {
    console.log("Promesa 1 OK!");
    return promise2;
})
.then(function() {
    console.log("Promesa 2 OK!");
});

promise.resolve();
setTimeout(promise2.resolve.bind(promise2), 1000);
```

r-promise.js

Nivel 3: Si un bloque devuelve una promesa, la ejecución se para hasta que se resuelva... **o se rechace!**

```
var promise = new Prom(),
    promise2 = new Prom();

promise.then(function() {
  console.log("Promesa 1 OK!");
  return promise2;
})
.then(null, function(e) {
  console.log("Error capturado:", e.message);
});

promise.resolve();
setTimeout(function() { promise2.reject(new Error("promesa rota")); }, 1000);
```

r-promise.js

El valor del siguiente bloque es el valor con el que se resuelva la promesa

```
var promise = new Prom(),
promise2 = new Prom();

promise.then(function() {
    console.log("Promesa 1 OK!");
    return promise2;
})
.then(function(v) {
    console.log("promise2 se resolvió con:", v);
});

promise.resolve();
promise2.resolve(56);
```

r-promise.js

El error del siguiente manejador de error es el error con el que se rechace la promesa

```
var promise = new Prom(),
    promise2 = new Prom();

promise.then(function() {
    console.log("Promesa 1 OK!");
    return promise2;
})
.then(null, function(e) {
    console.log("promise2 se falló con:", e.message);
});

promise.resolve();
promise2.reject(new Error("No funciono"));
```

r-promise.js

Nivel 4: Combinación de promesas

```
var promise = new Prom(),
    promise2 = new Prom();

Prom.all([promise, promise2])
.then(function(resultados) {
    console.log("promise devolvió:", resultados[0]);
    console.log("promise2 devolvió:", resultados[1]);
});
promise.resolve(42);
promise2.resolve(13);
```

r-promise.js

Epílogo: Diferidos

- Las librerías de promesas suelen presentar el interfaz fragmentado en dos:
- Diferidos: `.reject()` y `.resolve()`
- Promesas: `.then()` [y `.fail()`]

r-promise.js

Deferreds o diferidos

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

r-promise.js

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

r-promise.js

```
function elFuturo() {
  var defer = new Defer();
  setTimeout(defer.resolve.bind(defer), 1000);
  return defer.promise;
}

elFuturo().then(function() {
  console.log("El futuro está aquí!");
  return elFuturo();
})
.then(function() {
  console.log("Ya se pasó...");
});
```

r-promise.js

```
function elFuturo() {
  var defer = new Defer();
  setTimeout(defer.resolve.bind(defer), 1000);
  return defer.promise;
}

elFuturo().then(function() {
  console.log("El futuro está aquí!");
  return elFuturo();
})
.then(function() {
  console.log("Ya se pasó...");
});
```

r-promise.js

- **defer.promise**
 - La promesa asociada al diferido
- **promise.fail**
 - Poner solo callbacks de error
- **promise.done()**
 - Levatar excepciones no manejadas
- **promise.always(cb)**
 - Método que se ejecute tanto en éxito como en fracaso