Asincronía

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

```
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
  }
});
```

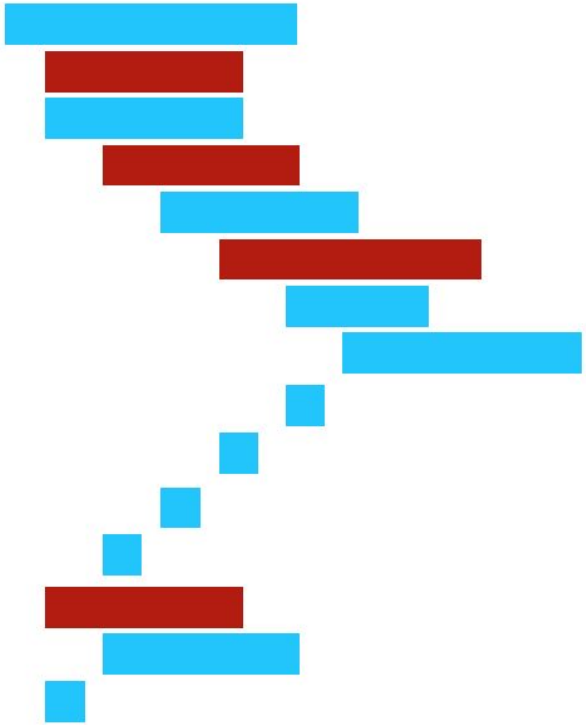
CALLBACK HELL

redradix

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 promesas



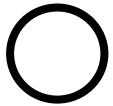
Promesas

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

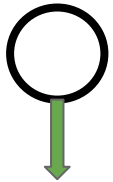


```
promesa.then(function() {
  // bloque
  return readFilePromise("./hola.txt");
})
.then(function(data) {
  // bloque
  return writeFilePromise("./copia.txt");
})
.then(function() {
  console.log("listo!");
})
.fail(function(err) {
});
```

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) {
});
```

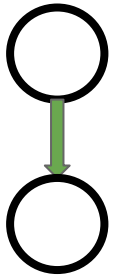


redradix

Promesas

Una promesa = un flujo de ejecución

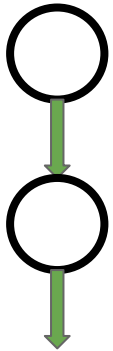
```
promesa.then(function() {
  // bloque
  return readFilePromise("./hola.txt");
})
.then(function(data) {
  // bloque
  return writeFilePromise("./copia.txt");
})
.then(function() {
  console.log("listo!");
})
.fail(function(err) {
});
```



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) {
});
```

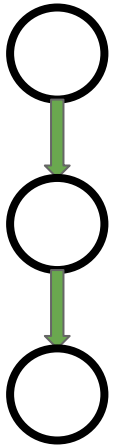


redradix

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) {
});
```



Promesas

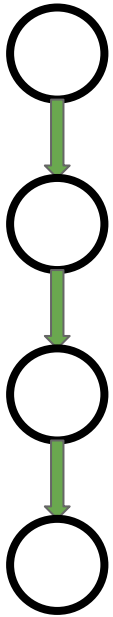
Una promesa = un flujo de ejecución

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

Promesas

Una promesa = un flujo de ejecución

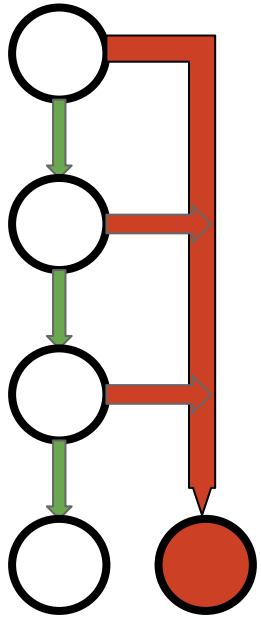
```
promesa.then(function() {
  // bloque
  return readFilePromise("./hola.txt");
})
.then(function(data) {
  // bloque
  return writeFilePromise("./copia.txt");
})
.then(function() {
  console.log("listo!");
})
.fail(function(err) {
});
```



Promesas

Una promesa = un flujo de ejecución

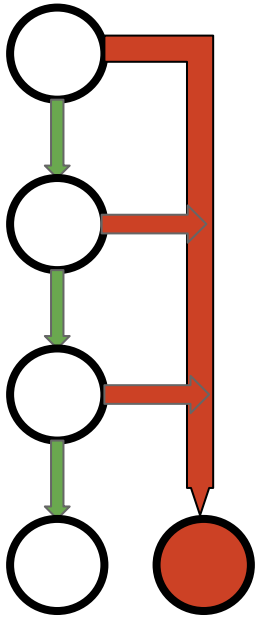
```
promesa.then(function() {
  // bloque
  return readFilePromise("./hola.txt");
})
.then(function(data) {
  // bloque
  return writeFilePromise("./copia.txt");
})
.then(function() {
  console.log("listo!");
})
.fail(function(err) {
});
```



redradix

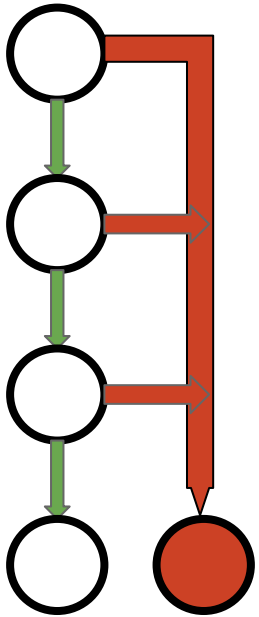
Promesas

Una promesa = un flujo de ejecución



Promesas

¡Pero aún no hemos ejecutado nada! Solamente hemos definido el flujo de ejecución.



Promesas

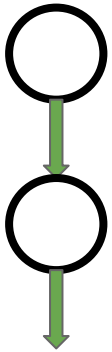
¿Ventajas?

- 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

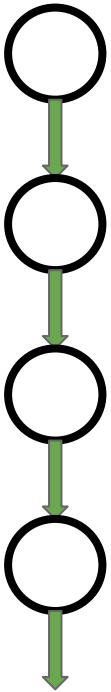
```
function copyFile(from, to) {
  return readFilePromise(from)
  .then(function(data) {
    // bloque
    return writeFilePromise(to);
  });
}
copyFile("./hola.txt", "./copia.txt")
.then(function() {
  return copyFile("./otraCosa.txt", "./copia2.txt");
})
.then(function() {
  console.log("listo!");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

Promesas



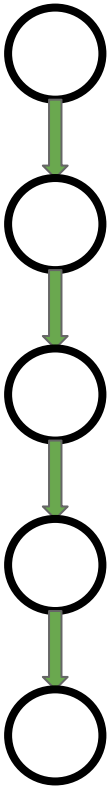
```
function copyFile(from, to) {
  return readFilePromise(from)
  .then(function(data) {
    // bloque
    return writeFilePromise(to);
  });
}
copyFile("./hola.txt", "./copia.txt")
.then(function() {
  return copyFile("./otraCosa.txt", "./copia2.txt");
})
.then(function() {
  console.log("listo!");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

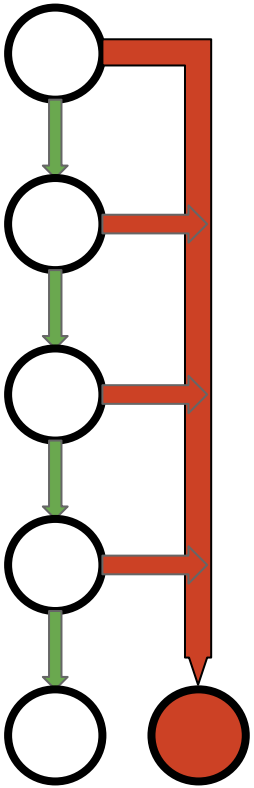
Promesas



```
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!");
})
```

redradix

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!");
})
```

redradix

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

Promesas

`.then(success, [error])`

- 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");

var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```

Promesas

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

```
var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```

Promesas

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

var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```

Promesas

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

var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```

Promesas

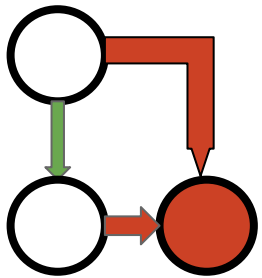
```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```

Promesas

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

var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```

Promesas

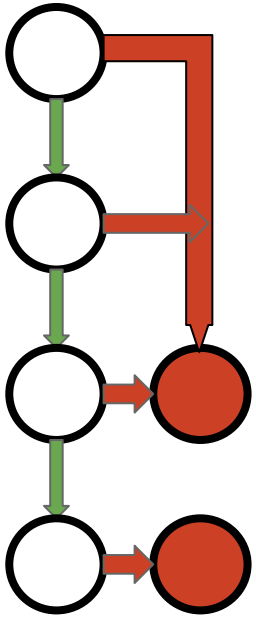
```
var promesa = readFilePromise("./hola.txt");

var promesa2 = promesa.then(function(data) {
  console.log("Contenido del fichero: ", data);
}, function(err) {
  console.log("Ooops!", err);
})
```



redradix

Promesas

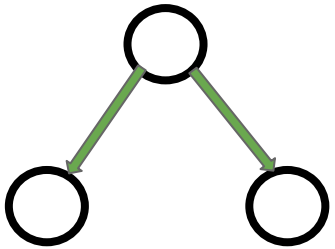
```
var promesa = readFilePromise("./hola.txt");
promesa.then(function(data) {
  return 1;
})
.then(function(uno) {
  return 2;
}, function(err) {
  console.log("Oh, oh...");
})
.then(function(dos) {
  return 3;
})
.fail(function(err) {
  console.log("Oops!");
});
```



redradix

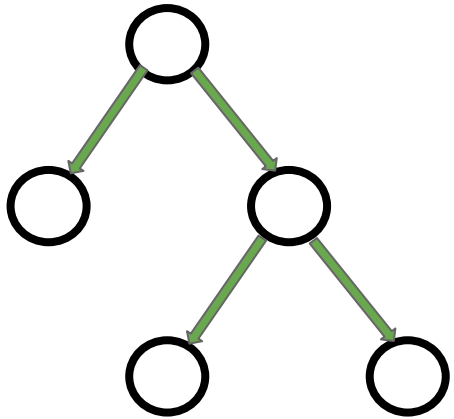
Promesas

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

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

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



Promesas

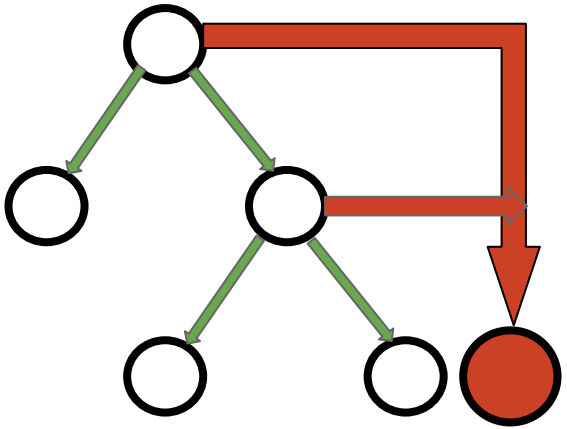
```
var promesa = readFilePromise("./hola.txt");

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

var promesa3 = promesa.then(function(data) {
  return 2;
});
promesa3.then(function(dos) {
  console.log("Ping!");
});
promesa3.then(function(dos) {
  console.log("Pong!");
});
```



redradix

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!");
});
promesa3.fail(function(err) {
  console.log("Fail!");
});
```



redradix

Promesas en ES6

```
const promise = new Promise((resolve, reject) => {
  const prob = Math.rand()
  if (prob < 0.5) {
    resolve(prob)
  } else {
    reject(prob)
  }
})

promise.then(console.log)
promise.catch(console.error)
```

A trastear

```
var promise = new Promise((resolve, reject) => {
})

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

Promesas

Una promesa tiene tres estados:

- Pendiente

- Resuelta - cuando llamamos a resolve(valor)

- Rechazada - cuando llamamos a reject(valor)

A trastear

```
var promise = new Promise((resolve, reject) => {
    resolve()
})
promise.then(function() {
  console.log("hola");
})
.then(function() {
  console.log("soy un flujo de ejecución");
})
.then(function() {
  console.log("expresado con promesas");
});
```

A trastear

```
var promise = new Promise((resolve, reject) => {
    reject()
})
promise.then(function() {
  console.log("hola");
})
.then(function() {
  console.log("soy un flujo de ejecución");
})
.then(function() {
  console.log("expresado con promesas");
});
```

A trastear

```
var promise = new Promise((resolve, reject) => {
    lalala
})
promise.then(function() {
  console.log("hola");
})
.then(function() {
  console.log("soy un flujo de ejecución");
})
.then(function() {
  console.log("expresado con promesas");
});
```

A trastear

```
var promise = new Promise((resolve, reject) => {
    lalala
})
promise.then(function() {
  console.log("hola");
})
.then(function() {
  console.log("soy un flujo de ejecución ");
})
.then(function() {
  console.log("expresado con promesas ");
})
.catch(function(err) {
  console.log('algo ha ido mal ', err);
})
```

A trastear++

¿Cómo podemos “promisificar” `fs.readFile` para que devuelva una promesa?

```
fs.readFile('/etc/passwd', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

A trastear++

El método `fs.stat` obtiene el estado de un fichero (última modificación, creación, tamaño....)

```
var p = new Promise((resolve, reject)=> {
        fs.stat('/tmp/world', (err, stats) => {
            if (err) reject(err);
            console.log(`stats: ${JSON.stringify(stats)}`);
            resolve(stats)
        });

})
p.then(....)
```

Promesas

La mayoría de funciones asíncronas de nodejs siguen el mismo convenio:

- Reciben un callback como último parámetro

- El callback recibe uno o más parámetros
  - El primero indica si ha habido un error
  - Si el primero es `falsy`, los siguientes son el resultado de la operación

Promesas

Casi todos los métodos de utilidades en node tienen la misma forma:

```
modulo.metodo(parametros, function(err, resultado) => {
  if (err) throw err;
  hacemos lo que sea con resultado;
});
```

A trastear++

Escribe una función `nodefcall` que:

- Reciba una función en “formato node”
- Devuelva una promesa
- Si la función devuelve error, la promesa falla
- Si la función ejecuta bien, resuelve la promesa con todos los valores que recibe el callback excepto el error

A trastear++

```
var fs = require('fs')

function nodefcall(fn) {
  // aquí tu código
}

nodefcall(fs.readFile, './files/uno.txt')
.then((data) => {
  console.log(data.toString())
})
```

redradix

A trastear++

Escribe una función nodeinvoke que:

- Reciba
  - Un objeto
  - Un nombre de método a invocar
  - Parámetros para pasarle al método
- Devuelva una promesa
- Si la función devuelve error, la promesa falla
- Si la función se ejecuta bien, resuelve la promesa con los parámetros que haya recibido la función de callback (sin el error)

A trastear++

```
var fs = require('fs')

function nodeinvoke(obj, method, params) {
  // aquí tu código
}

nodeinvoke(fs, 'readFile', './files/uno.txt')
.then((data) => {
  console.log(data.toString())
})
```

redradix

Promesas en ES6

`Promise.resolve(valor)` nos devuelve una promesa resuelta directamente con el valor que pasemos como parámetro.

`Promise.reject(valor)` nos devuelve una promesa rechazada directamente con el valor que pasamos como parámetro.

Promesas en ES6

El método `Promise.all(iterable)` nos devuelve una promesa que se resolverá cuando todas las promesas que le hemos pasado (en un array por ejemplo) se hayan resuelto. O nos devolverá una promesa rechazada cuando la primera de las promesas a resolver se haya rechazado.

Promesas en ES6

```
function promesasDeNumeros() {
  let result = []
  for (let i = 5; i > 0; i--)
    result.push(Promise.resolve(i))
  return result
}

Promise.all(promesasDeNumeros())
.then(function(data) {
  console.log(data)
})
```

Ejercicio

Intenta leer los ficheros del directorio `files` mediante promesas e imprime un “OK” más el contenido de los ficheros por pantalla cuando todo haya ido bien.

Ejercicio - Tu librería de promesas

Implementa `Promise.reduce(iterable, iteratee, initialValue)` que:

- Itera sobre los valores del `iterable`
- Reduce sus valores uno a uno y en orden usando la función `iteratee`
- Opcionalmente recibe un valor inicial como primer parámetro.

Ejercicio - Tu librería de promesas

```
reduce(["file1.txt", "file2.txt", "file3.txt"], function(total, fileName) {
  return readFilePromise(fileName, "utf8").then(function(contents) {
      return total + parseInt(contents, 10);
  });
}, 0).then(function(total) {
  //El total es 30, por ejemplo
});
```

Iteradores

Un iterador es un objeto que sabe cómo acceder a los elementos de una colección y llevar un registro de la posición actual dentro de ella.

- Arrays
- Mapas
- Sets
- Strings

Son aquellos que en su prototipo implementan una función de iteración en `Symbol.iterator`

Iteradores

```
function iterador(array){
  var indice = 0;

  return {
    next: function(){
      return indice < array.length ?
        {value: array[indice++], done: false} :
        {done: true};
       }
    }
}
```

redradix

Iteradores

```
var it = iterador(['hoy', 'programamos', 'fuerte']);
console.log(it.next().value); // 'hoy'
console.log(it.next().value); // 'programamos'
console.log(it.next().value); // 'fuerte'
console.log(it.next().done);  // true
```

Iteradores

El bucle `for...of` recorre todo un iterable y devuelve los valores

```
let iterable = 'boo';

for (let value of iterable) {
  console.log(value);
}
// "b"
// "o"
// "o"
```

Generadores

Una función se convierte en generador si contiene una o más expresiones `yield` y si usa la sintaxis `function*`

```
function* generadorDeIds(){
  var index = 0;
  while(true)
    yield 'id-' + index++;
}

var ids = generadorDeIds();

console.log(ids.next().value); // id-0
console.log(ids.next().value); // id-1
```

Generadores

Computan los valores dados por yield bajo demanda.

- Útiles para devolver valores de secuencias grandes o infinitas.

- Útil para representar secuencias de costosa computación.

Generadores

`next()`

Llama al siguiente valor dado por `yield` del generador y devuelve un objeto:

```
generador.next()

   {value: “cualquier valor dado por yield”, done: boolean}
```

Puede recibir un valor y reemplazarlo por el resultado del último yield dentro del generador.

Generadores

`next()`

Puede recibir un valor y reemplazarlo por el resultado del último yield dentro del generador.

```
function* migenerador(param) {
    let nuevoValor = yield param
    return nuevoValor
}
```

`return` termina con el generador.

Generadores

```
function* migenerador(param) {
    let nuevoValor = yield param
    return nuevoValor
}

var migen = migenerador(10)
migen.next(5)         // {value: 10, done: false}
migen.next("hola")    // {value: "hola", done: true}
```

Generadores

`throw()`

Reanuda la ejecución de un generador provocando un error dentro. Devuelve el mismo objeto que next.

Generadores

```
function* generador() {
  while(true) {
    try {
       yield 42;
    } catch(e) {
      console.log('Error capturado!');
    }
  }
}
var g = generador();
g.next();
// { value: 42, done: false }
g.throw(new Error('Algo ha ido mal'));
// "Error capturado!"
// { value: 42, done: false }
```

Generadores

Haz un generador de números desde 0 hasta un límite de `n` con la opción de poder resetear los valores devueltos a 0.

```
var g = resetableGenerator(3)
g.next()          // {value: 0, done: false}
g.next()          // {value: 1, done: false}
g.next(true)      // {value: 0, done: false}
g.next()          // {value: 1, done: false}
g.next() .        // {value: 2, done: false}
g.next()          // {value: 3, done: true}
g.next()          // {value: undefined, done: true}
```

Generadores

También podemos usar el bucle for...of para tomar valores de generadores

```
for(let val of resetedGenerator(3))
  console.log(val)
```

Generadores

¿Podríamos usar generadores para operaciones asíncronas?

Async / await

Funciones asíncronas de Javascript:

- Basadas en promesas
- Basadas en generadores
- Azúcar sintáctico
- Código asíncrono que parece síncrono

Async / await

```
async function sumaAsincrona() {
  var a = await resolverA()
  var b = await resolver B()
  return a + b
}
```

Async / await

Funciones asíncronas de Javascript:

- Devuelven promesas.
- Se pueden encadenar con `.then`
- El valor de resolución de la promesa de `await` se vuelca en el resultado.
- Solo se puede usar `await` dentro de una función declarada como `async`

Async / await

```
async function sumaAsincrona() {
    var a = await Promise.resolve(1)
    var b = await Promise.resolve(2)
    return a + b
}

sumaAsincrona().then(console.log)
```

Async / await

```
function getAsyncValuePromise(initVal) {
  return new Promise(function(success, err) {
    setTimeout(() => { success(initVal*2); }, 1000);
  });
};

function resolveAsyncValues(valuesArray) {
  var ps = valuesArray.map(v => getAsyncValuePromise(v, v*100));
  return Promise.all(ps);
};
```

Async / await

```
function* resolveAsyncValues(valuesArray) {
  var acc = [];
  for(var i = valuesArray.length; i--;) {
    var v = valuesArray[i];
    acc[i] = yield getAsyncValuePromise(v, v*100);
  };
  return acc;
};
```

Async / await

```
function*() {
  var valuesArray = [1, 2, 3, 4, 5, 6, 7];
  var asyncValues = yield getValuesFromGenerator(valuesArray);
  console.log(asyncValues);
}
```

redradix

Async / await - Ejercicio

Haz un adaptador de generadores para generar código asíncrono.

Se trata de una función que:

- Recibe un generador
- Devuelve una función que: Ejecuta `generador.next()` hasta que este ha acabado.
- Mientras no ha acabado, no devuelve ningún valor. Encadena promesas buscando el siguiente.
- Cuando ha acabado, devuelve el valor de retorno del generador.