# I PROMISED MYSELF

Christian Ulbrich, Zalari UG

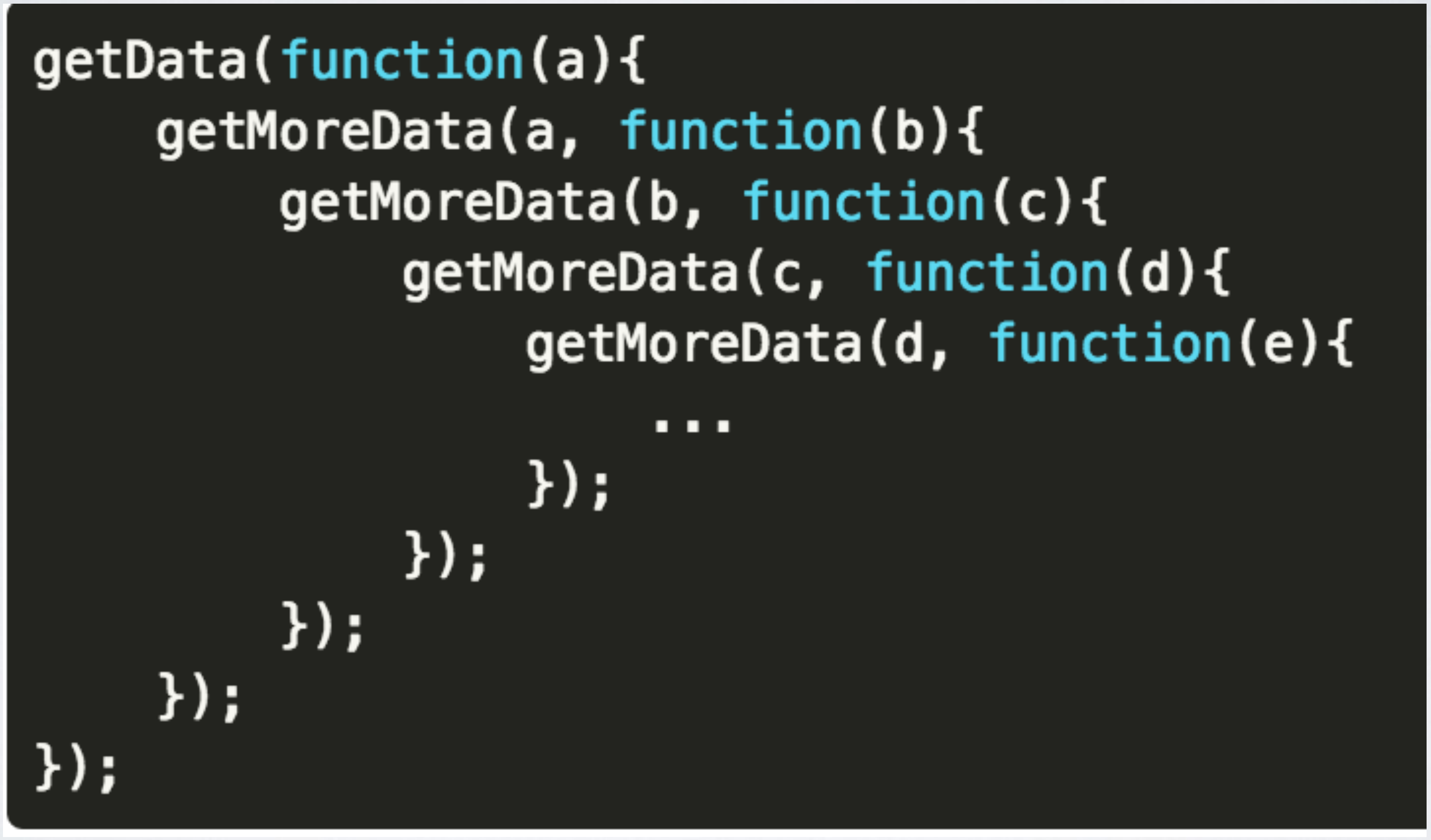
## AGENDA PROPAGANDA

- Recap: Promises
- Anti-Patterns
- Common use cases
- Advanced Example - lazyCache
- Links

## RECAP - PROMISES

„Promises are the monad of asynchronous programming"

# RECAP - PROMISES

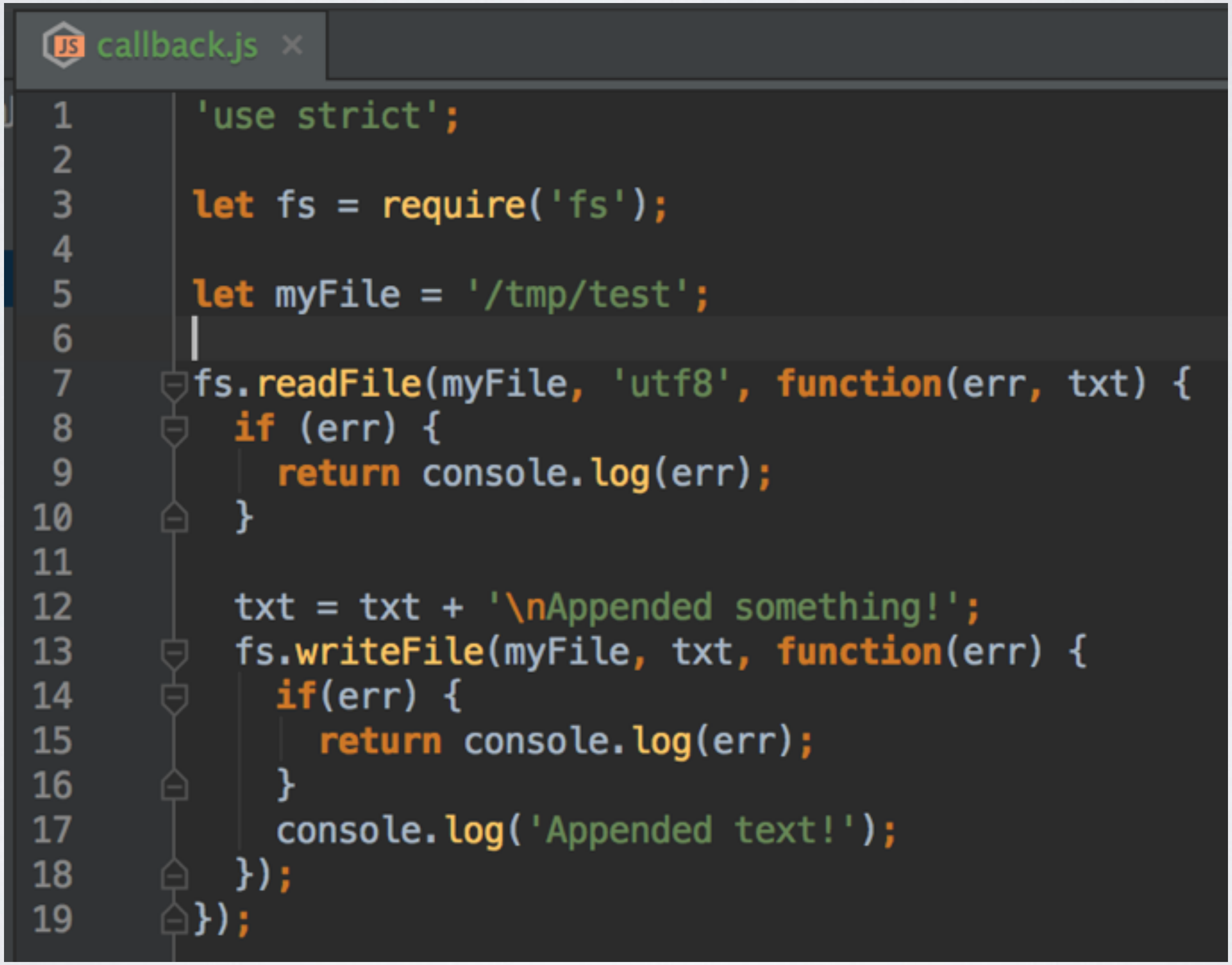


```
getData(function(a){
    getMoreData(a, function(b){
        getMoreData(b, function(c){
            getMoreData(c, function(d){
                getMoreData(d, function(e){
                    ...
                });
            });
        });
    });
});
```

## RECAP - PROMISES

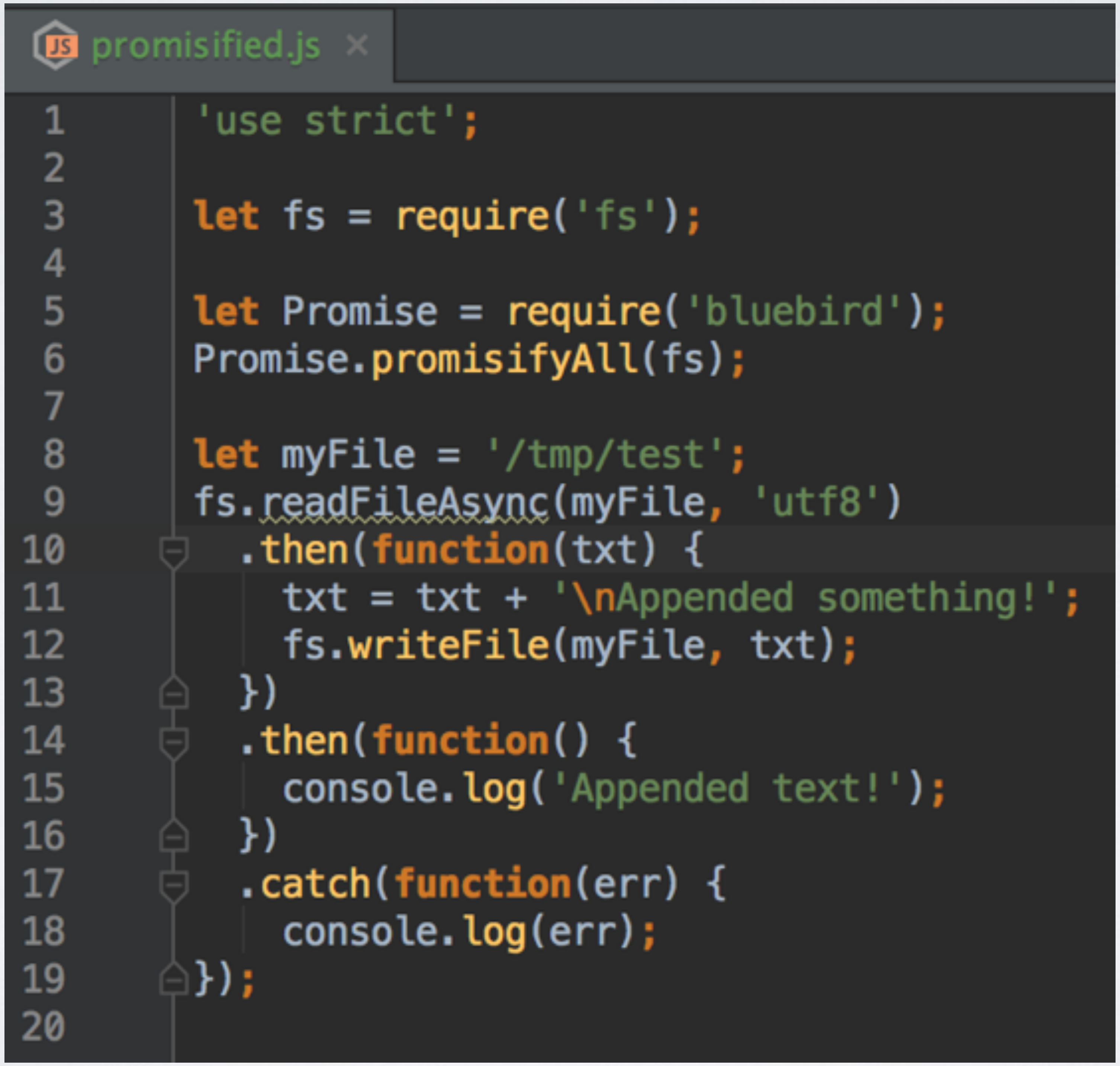
- Promises are an elegant concept of modeling asynchronous data flows
- Promises are not JS-specific, but can be found in other language as well
- Promises avoid the pyramid of doom aka callback hell

# RECAP - PROMISES


callback.js

```
'use strict';

let fs = require('fs');

let myFile = '/tmp/test';

fs.readFile(myFile, 'utf8', function(err, txt) {
  if (err) {
    return console.log(err);
  }

  txt = txt + '\nAppended something!';
  fs.writeFile(myFile, txt, function(err) {
    if(err) {
      return console.log(err);
    }
    console.log('Appended text!');
  });
});
```

## RECAP - PROMISES


promisified.js

```javascript
'use strict';

let fs = require('fs');

let Promise = require('bluebird');
Promise.promisifyAll(fs);

let myFile = '/tmp/test';
fs.readFileAsync(myFile, 'utf8')
  .then(function(txt) {
    txt = txt + '\nAppended something!';
    fs.writeFile(myFile, txt);
  })
  .then(function() {
    console.log('Appended text!');
  })
  .catch(function(err) {
    console.log(err);
});

```

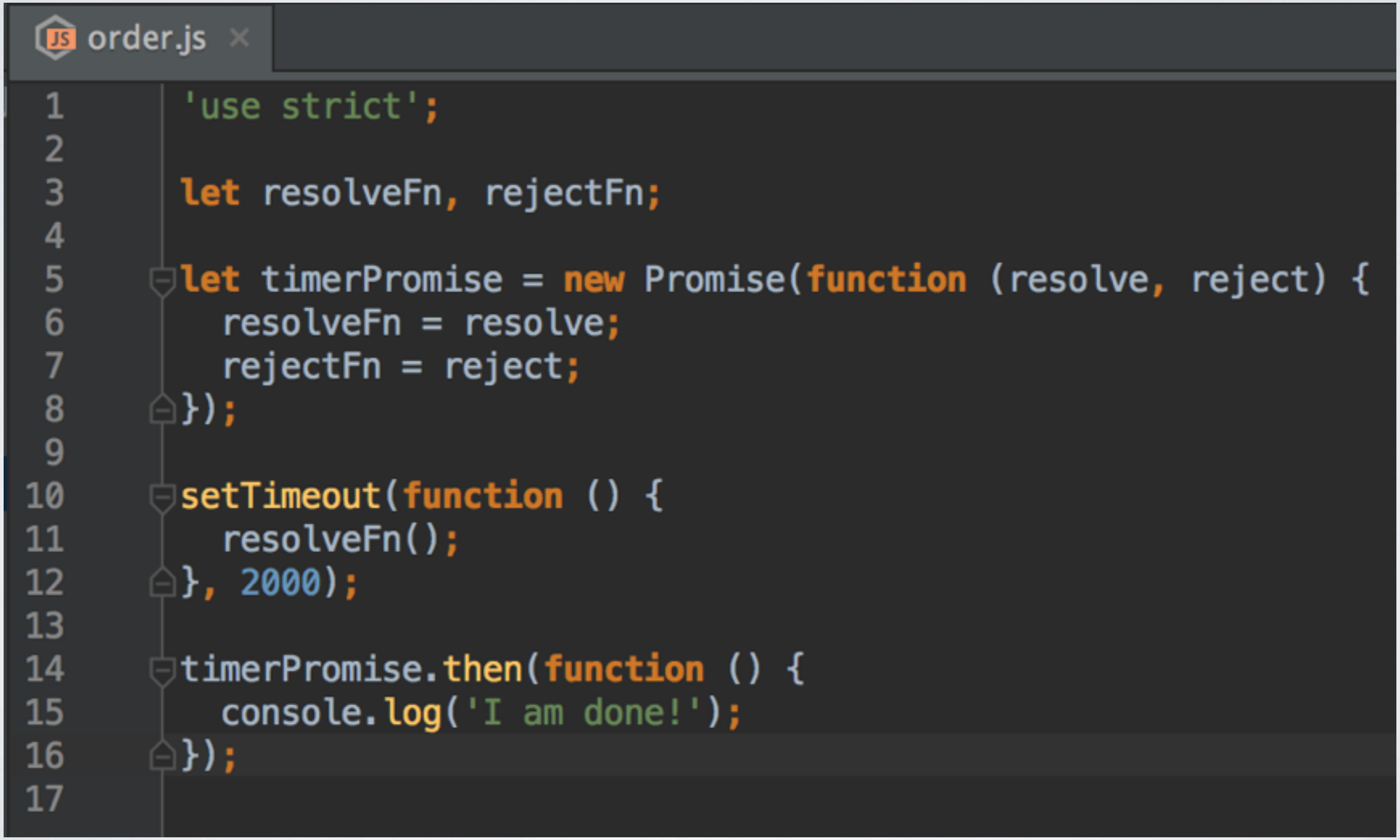
# RECAP - PROMISES

- Promises are something, that can either be *fulfilled* or *rejected*
- a Promise has two main methods
    - then(callback) , that gets called, when the promise is fulfilled
    - catch(callback), that gets called, when the promise is rejected

## WHY - PROMISES

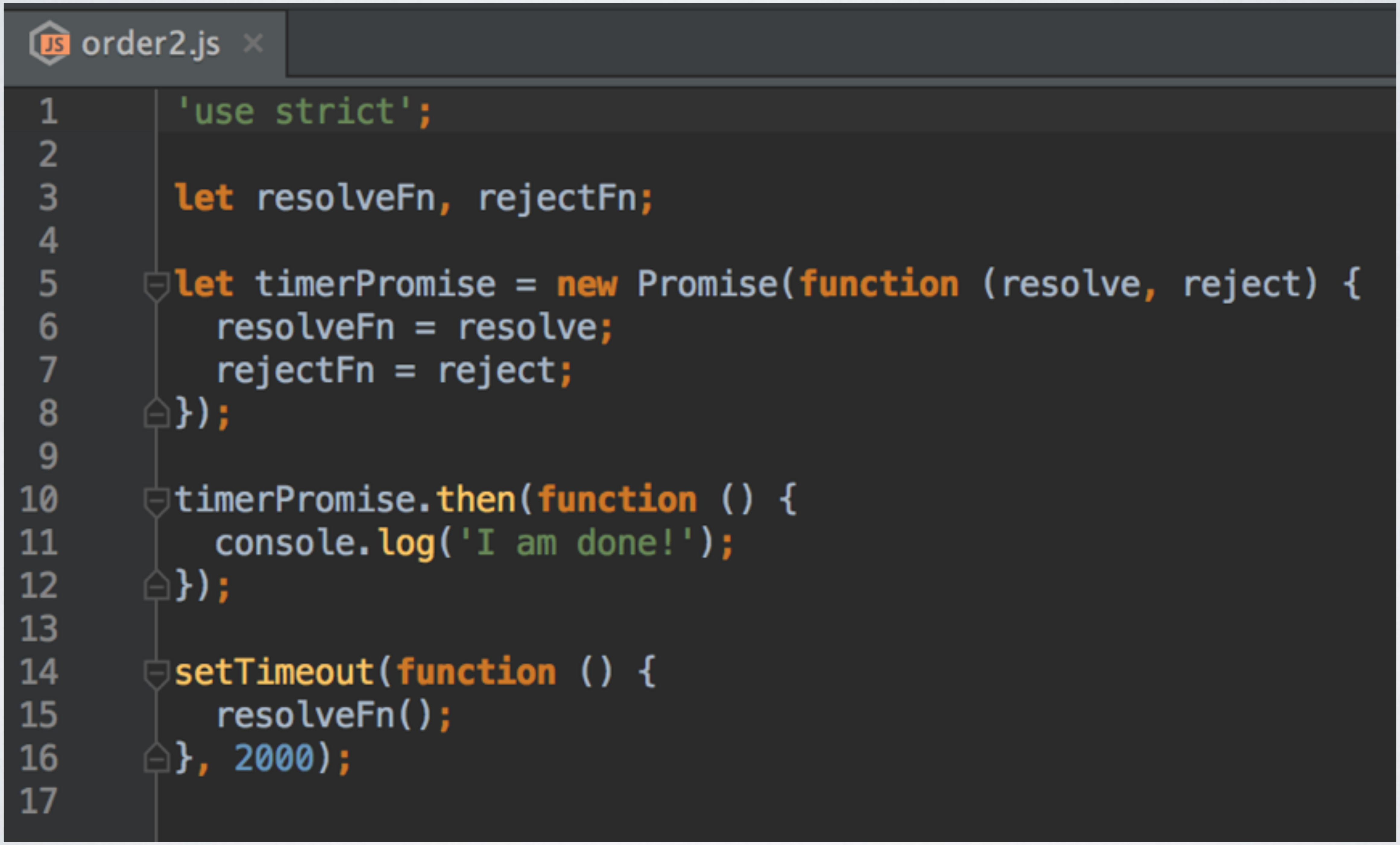
- Promises are native in ECMAScript 6 and current NodeJS
- a growing number of standard functions return Promises
- the order does not matter, i.e. attach then before or after a promise resolves / rejects

# PROMISES - ORDER



```javascript
'use strict';

let resolveFn, rejectFn;

let timerPromise = new Promise(function (resolve, reject) {
  resolveFn = resolve;
  rejectFn = reject;
});

setTimeout(function () {
  resolveFn();
}, 2000);

timerPromise.then(function () {
  console.log('I am done!');
});

```

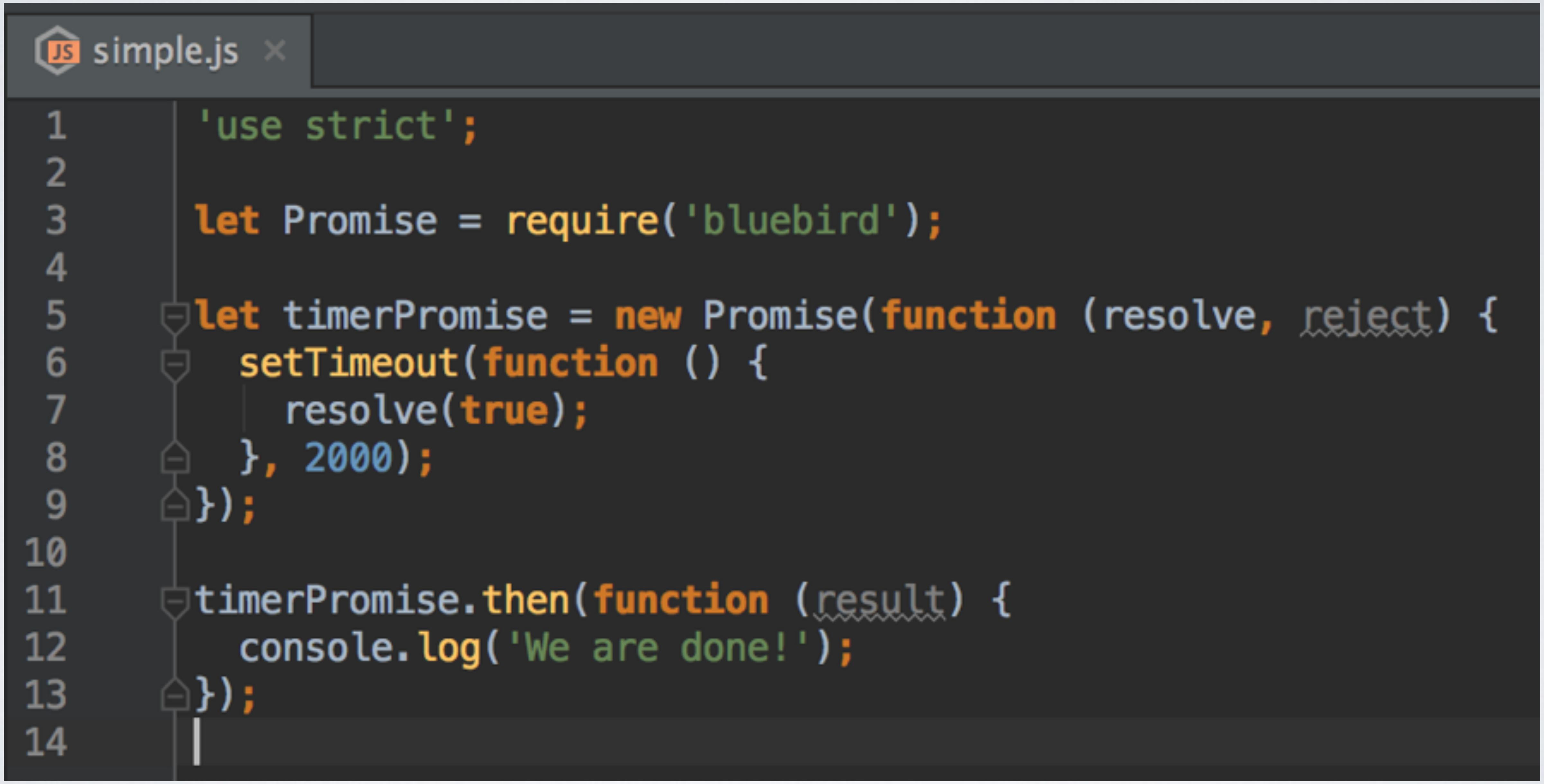
# PROMISES - ORDER



order2.js

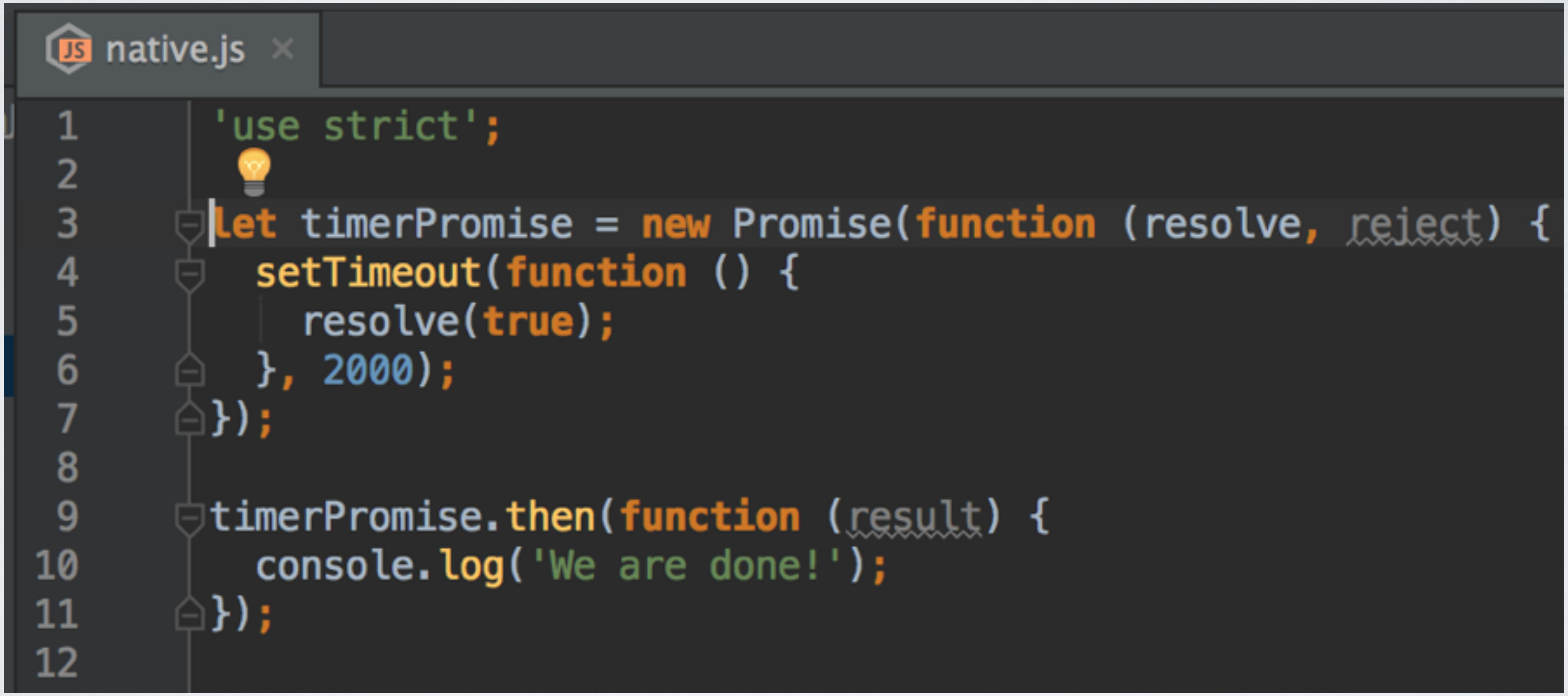
```
'use strict';

let resolveFn, rejectFn;

let timerPromise = new Promise(function (resolve, reject) {
  resolveFn = resolve;
  rejectFn = reject;
});

timerPromise.then(function () {
  console.log('I am done!');
});

setTimeout(function () {
  resolveFn();
}, 2000);

```

# PROMISES - NATIVE



```
'use strict';

let Promise = require('bluebird');

let timerPromise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve(true);
  }, 2000);
});

timerPromise.then(function (result) {
  console.log('We are done!');
});

```

# PROMISES - NATIVE



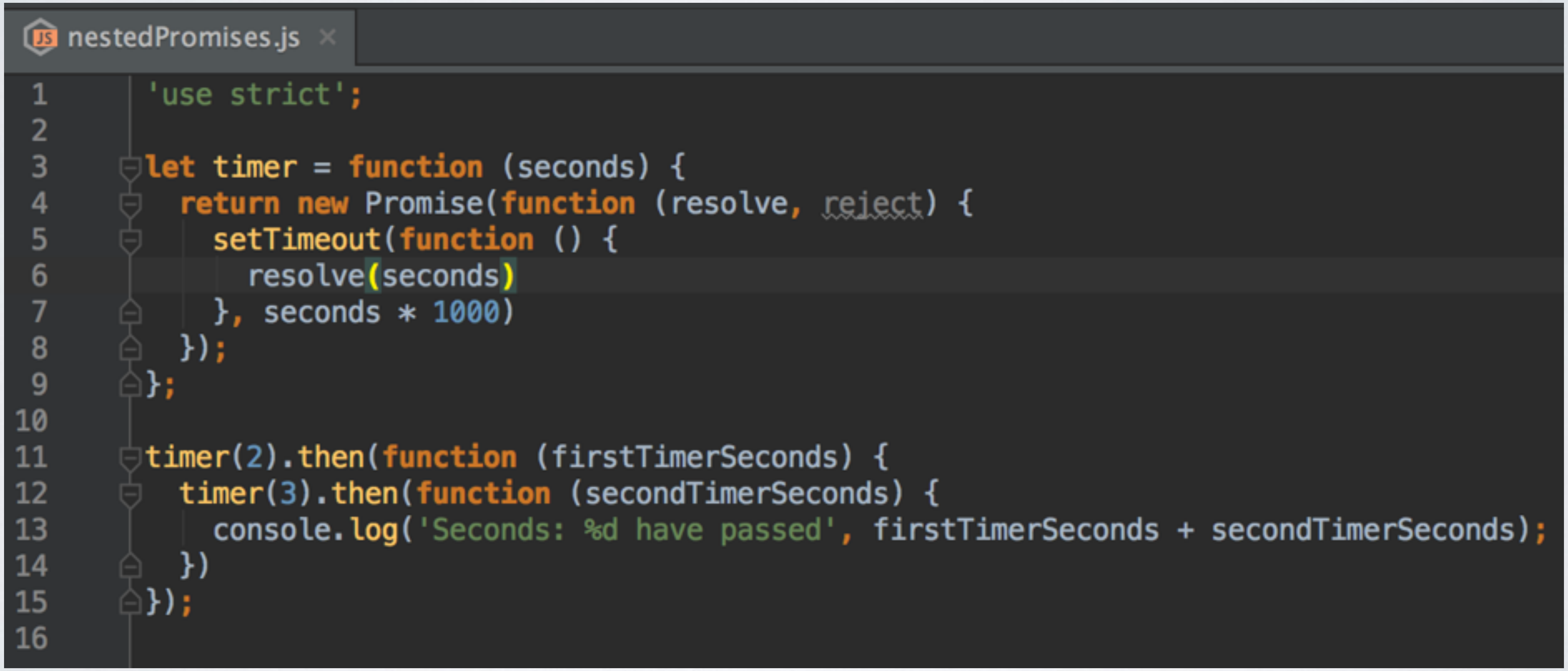
native.js

```
'use strict';

let timerPromise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    resolve(true);
  }, 2000);
});

timerPromise.then(function (result) {
  console.log('We are done!');
});
```

## ANTI-PATTERNS

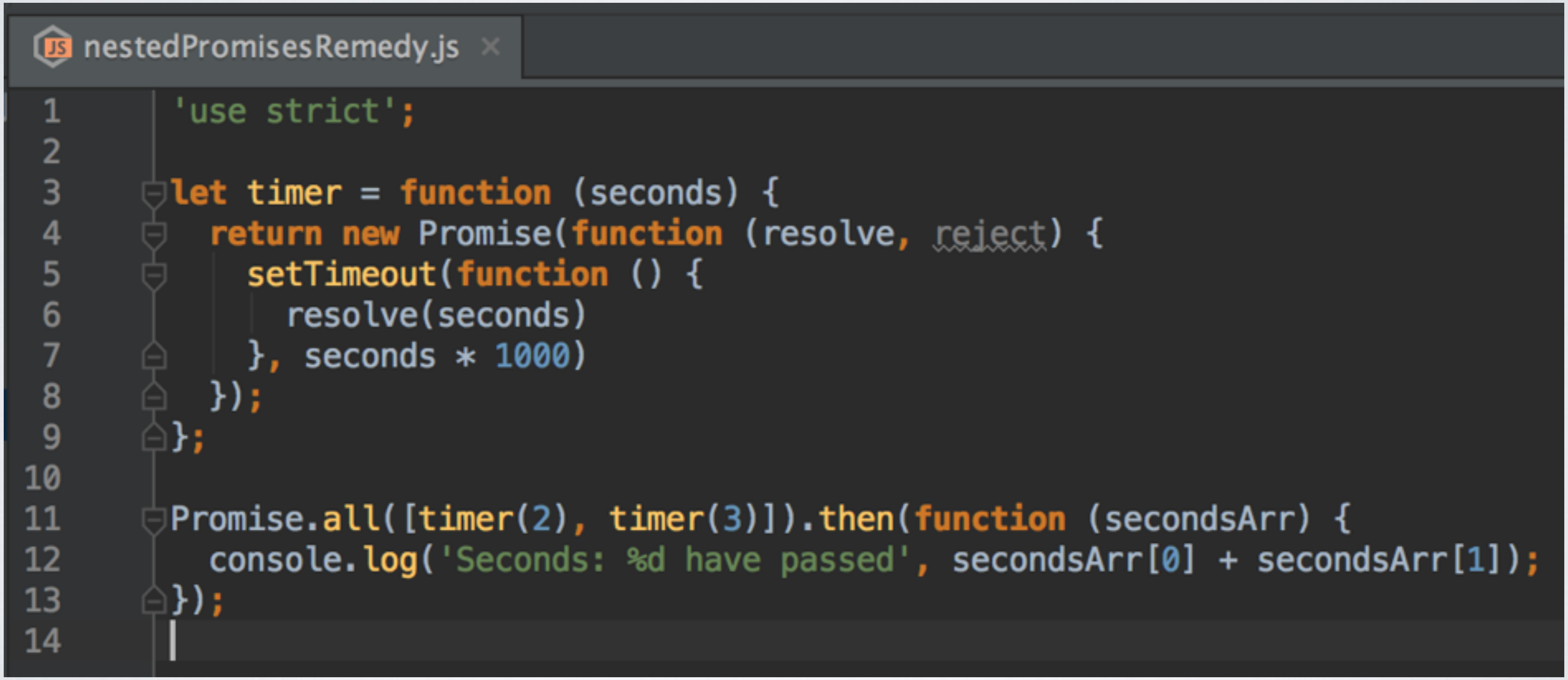
- nested promises
- superfluous deferred
- then-callback-style

# ANTI-PATTERNS
# NESTED PROMISES


nestedPromises.js

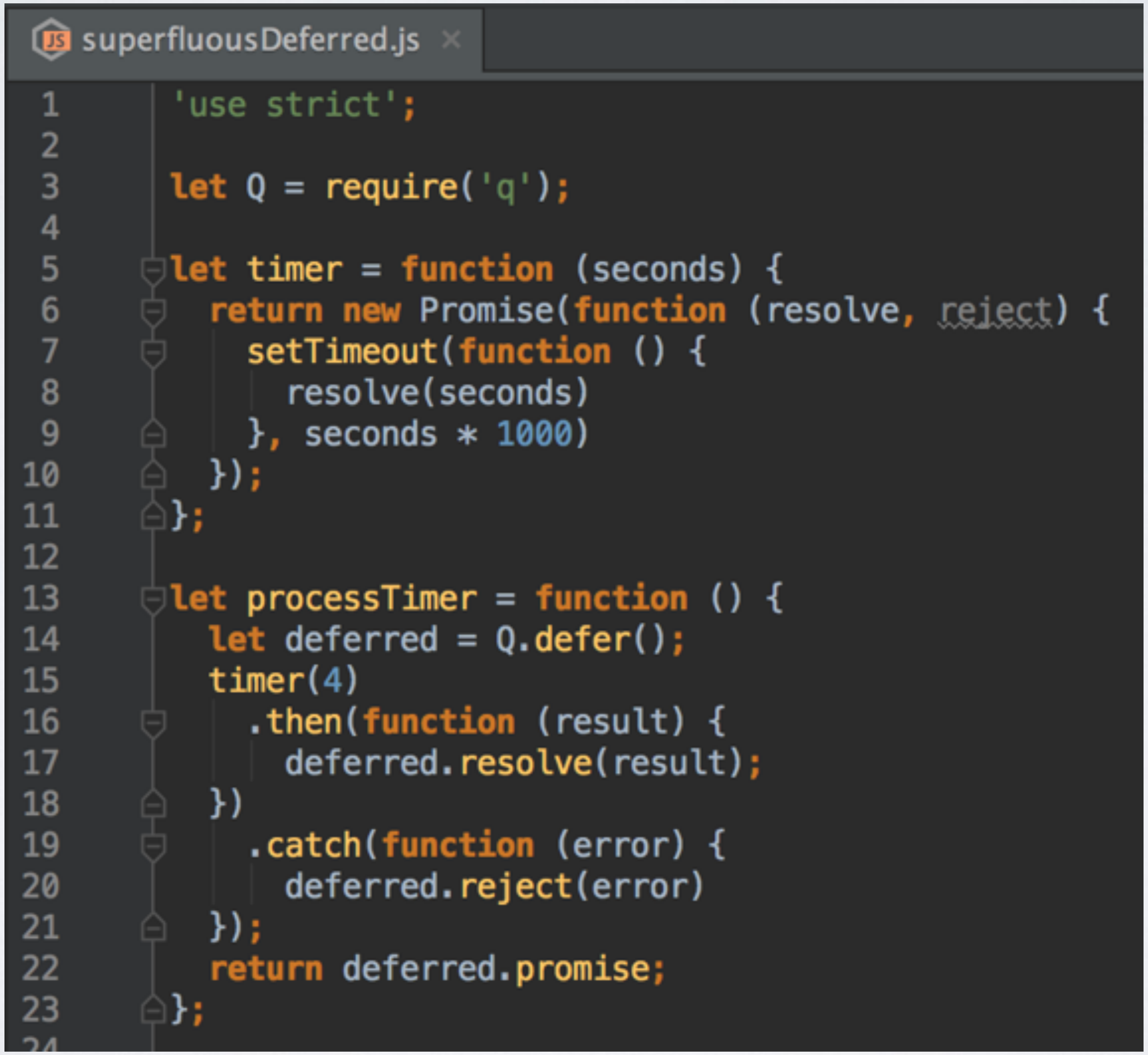
```javascript
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(seconds)
    }, seconds * 1000)
  });
};

timer(2).then(function (firstTimerSeconds) {
  timer(3).then(function (secondTimerSeconds) {
    console.log('Seconds: %d have passed', firstTimerSeconds + secondTimerSeconds);
  })
});

```

# ANTI-PATTERNS NESTED PROMISES



```javascript
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(seconds)
    }, seconds * 1000)
  });
};

Promise.all([timer(2), timer(3)]).then(function (secondsArr) {
  console.log('Seconds: %d have passed', secondsArr[0] + secondsArr[1]);
});
```

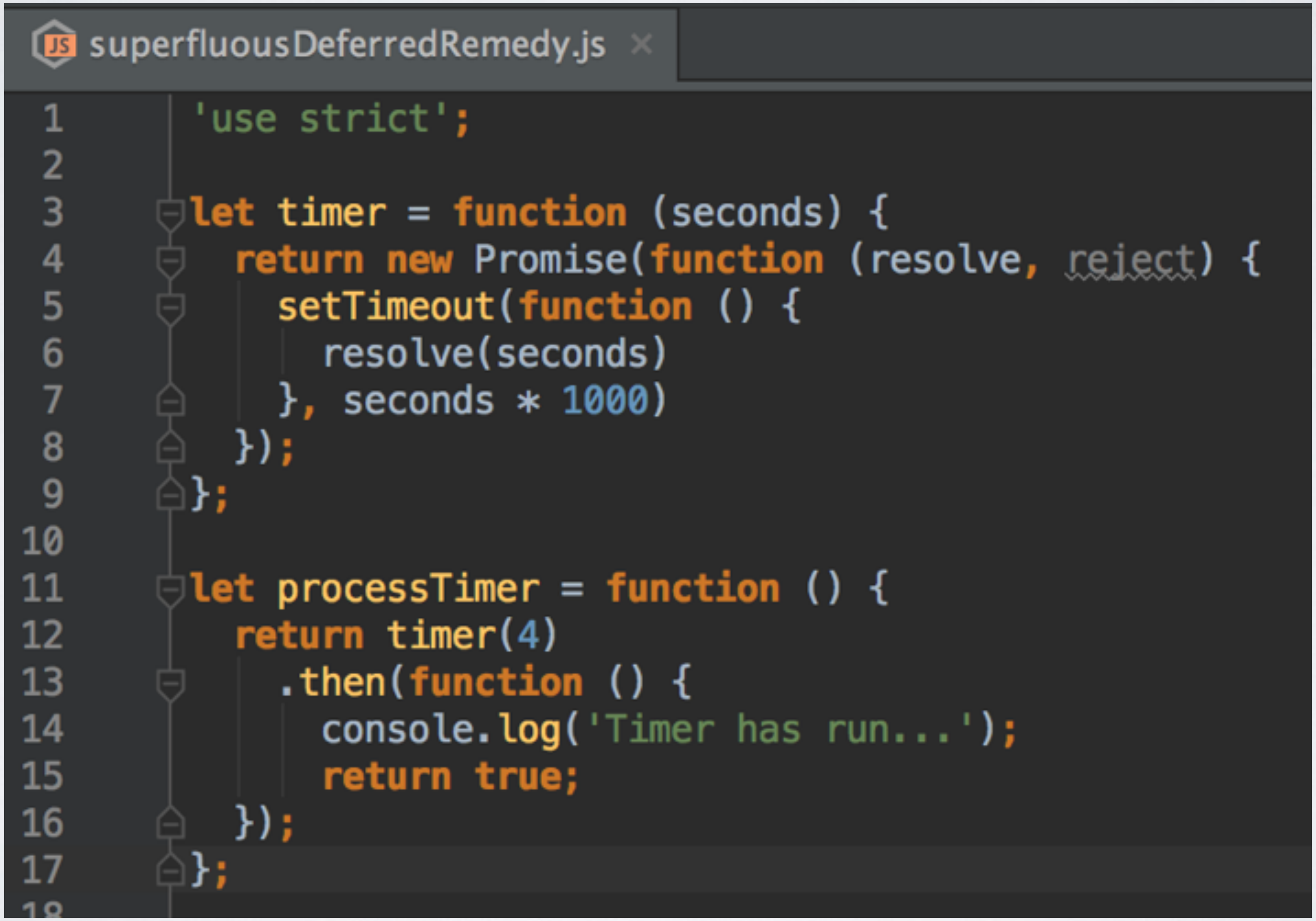
# ANTI-PATTERNS
# SUPERFLUOUS DEFERRED



superfluousDeferred.js

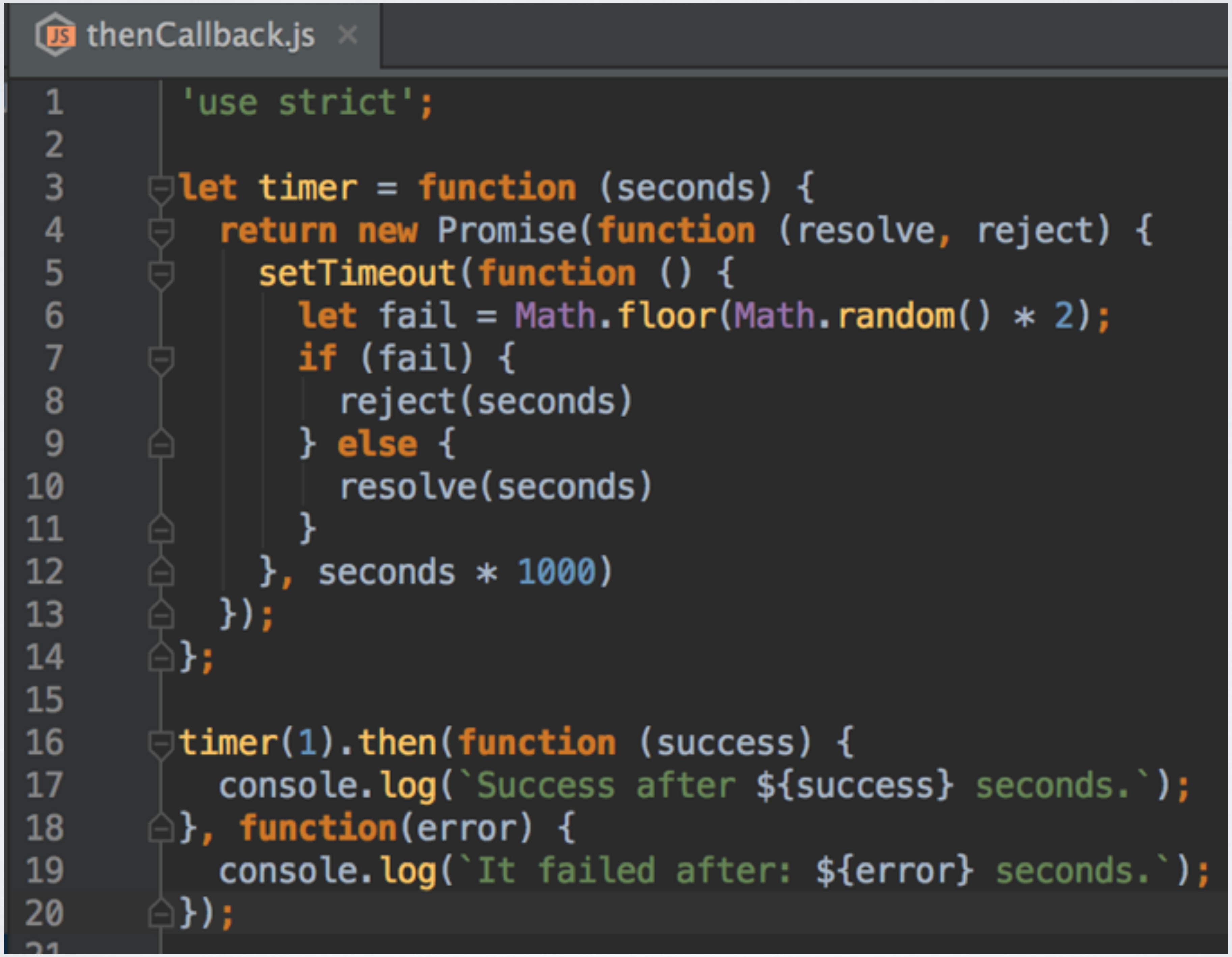
```javascript
'use strict';

let Q = require('q');

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(seconds)
    }, seconds * 1000)
  });
};

let processTimer = function () {
  let deferred = Q.defer();
  timer(4)
    .then(function (result) {
      deferred.resolve(result);
  })
    .catch(function (error) {
      deferred.reject(error)
  });
  return deferred.promise;
};
```

# ANTI-PATTERNS SUPERFLUOUS DEFERRED


superfluousDeferredRemedy.js

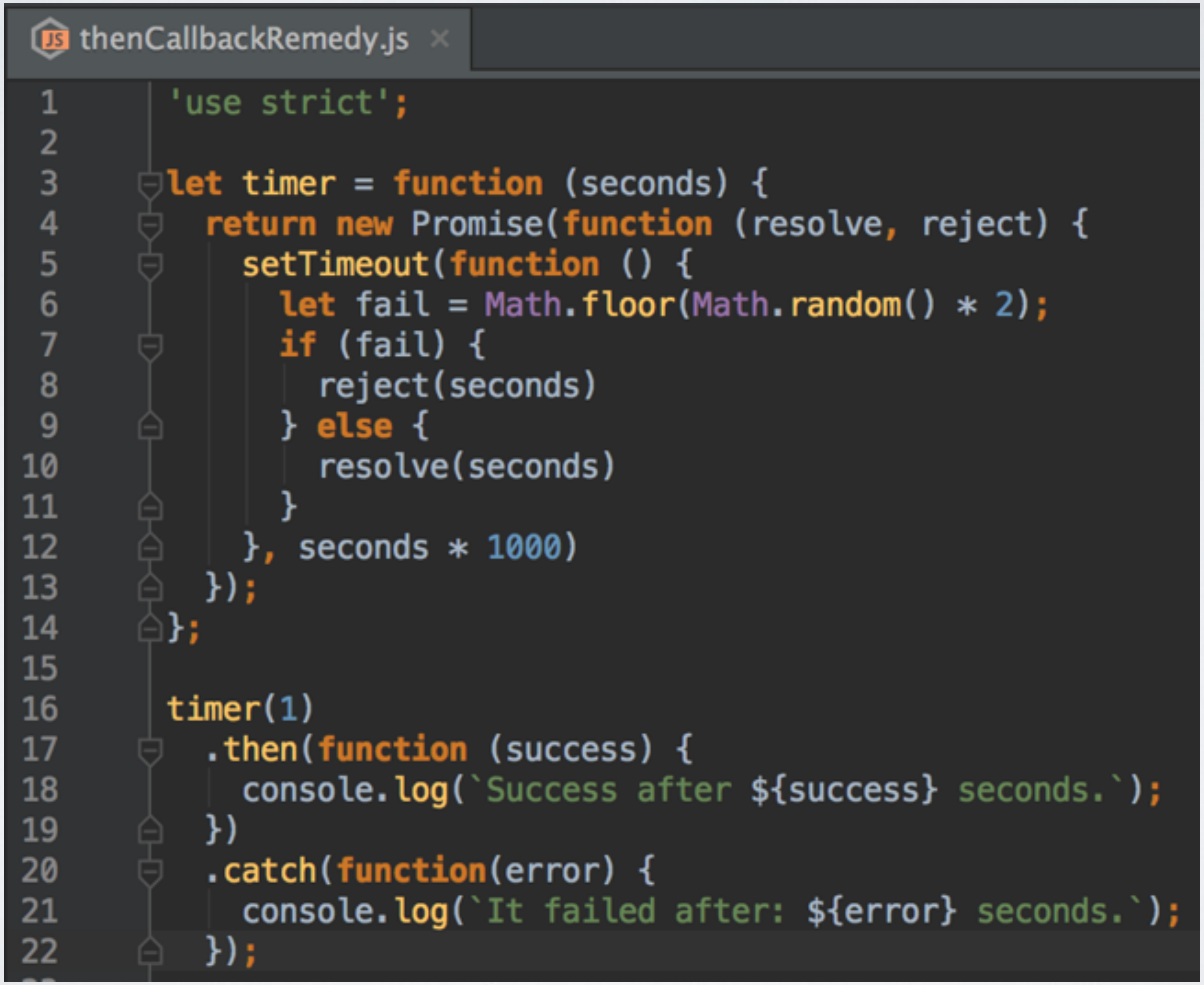
```javascript
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(seconds)
    }, seconds * 1000)
  });
};

let processTimer = function () {
  return timer(4)
    .then(function () {
      console.log('Timer has run...');
      return true;
  });
};
```

# ANTI-PATTERNS THEN-CALLBACK STYLE


thenCallback.js

```javascript
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let fail = Math.floor(Math.random() * 2);
      if (fail) {
        reject(seconds)
      } else {
        resolve(seconds)
      }
    }, seconds * 1000)
  });
};

timer(1).then(function (success) {
  console.log(`Success after ${success} seconds.`);
}, function(error) {
  console.log(`It failed after: ${error} seconds.`);
});
```

# ANTI-PATTERNS THEN-CALLBACK STYLE

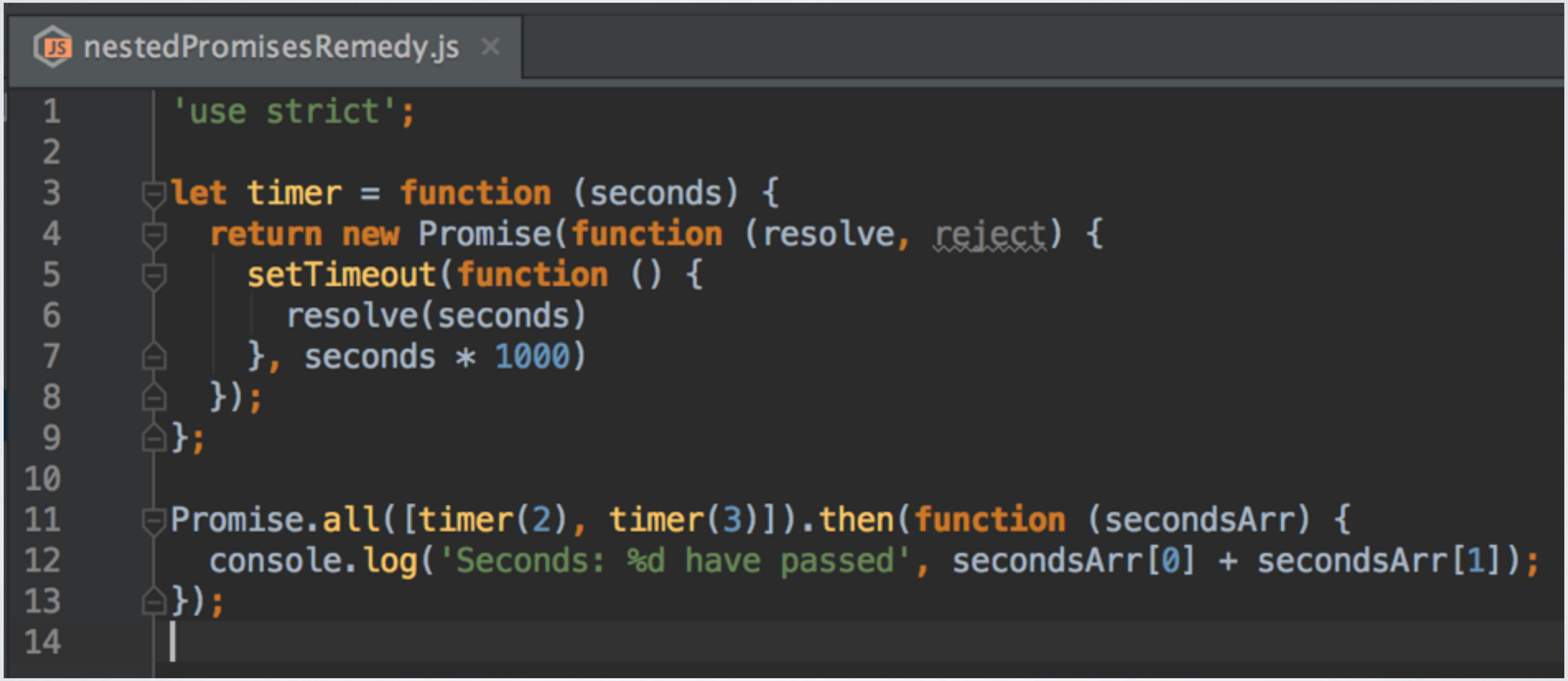

thenCallbackRemedy.js

```javascript
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      let fail = Math.floor(Math.random() * 2);
      if (fail) {
        reject(seconds)
      } else {
        resolve(seconds)
      }
    }, seconds * 1000)
  });
};

timer(1)
  .then(function (success) {
    console.log(`Success after ${success} seconds.`);
  })
  .catch(function(error) {
    console.log(`It failed after: ${error} seconds.`);
  });
```

## COMMON USE CASE

- doing things in parallel
- doing things in sequence
- same interfaces for sync / async code

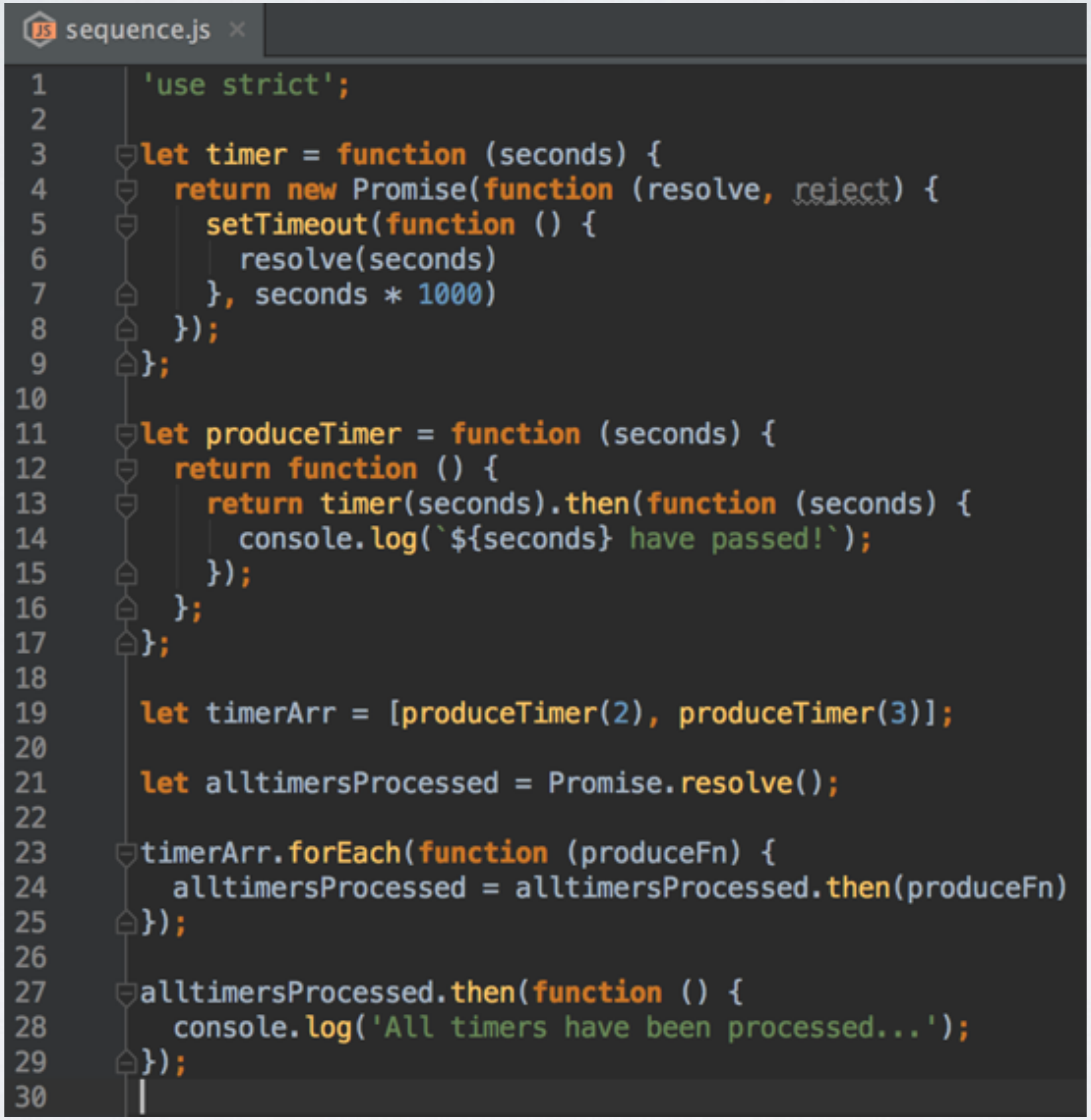
# COMMON USE CASE
# PARALLELISM



nestedPromisesRemedy.js

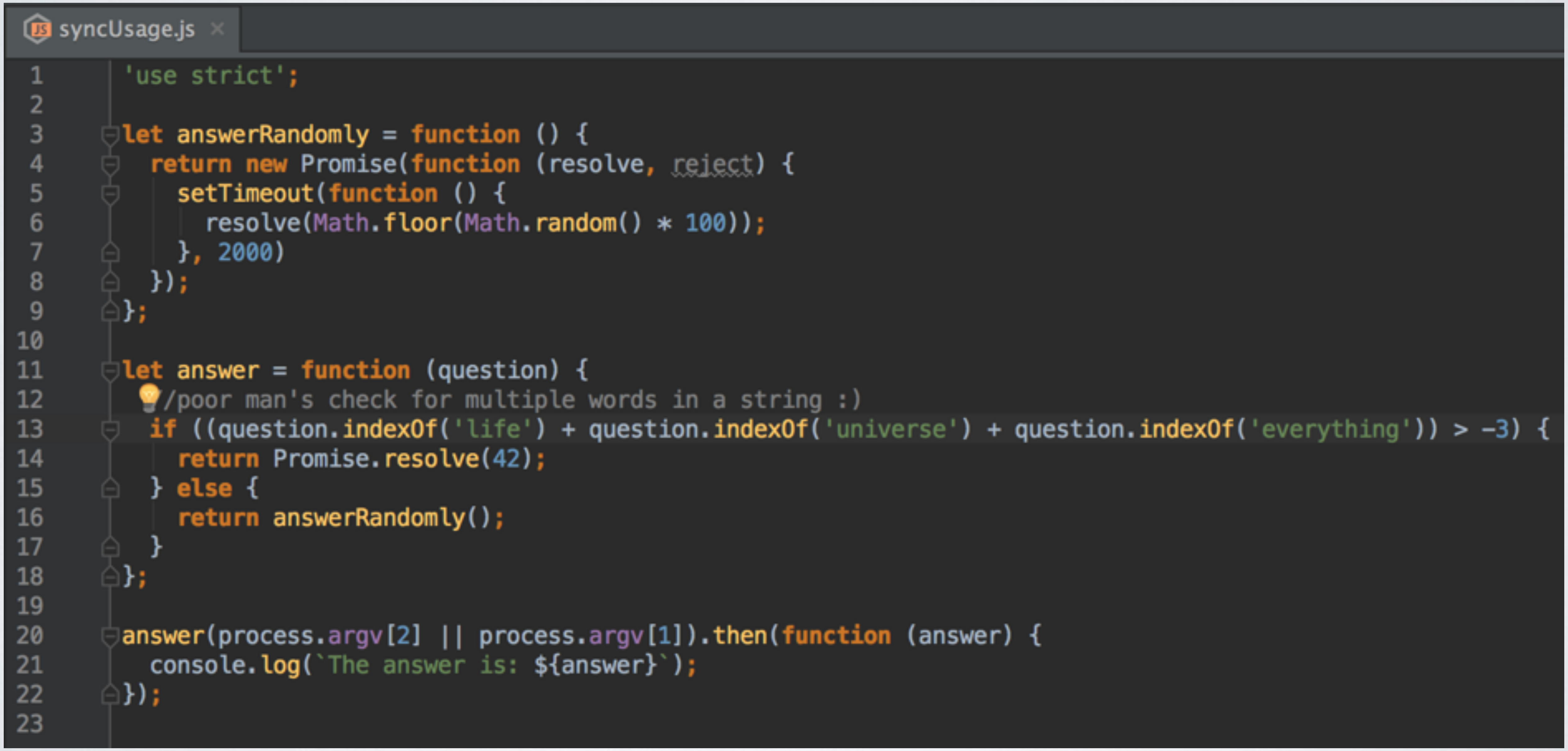
```javascript
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(seconds)
    }, seconds * 1000)
  });
};

Promise.all([timer(2), timer(3)]).then(function (secondsArr) {
  console.log('Seconds: %d have passed', secondsArr[0] + secondsArr[1]);
});
```

# SEQUENCE



```
'use strict';

let timer = function (seconds) {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(seconds)
    }, seconds * 1000)
  });
};

let produceTimer = function (seconds) {
  return function () {
    return timer(seconds).then(function (seconds) {
      console.log(`${seconds} have passed!`);
    });
  };
};

let timerArr = [produceTimer(2), produceTimer(3)];

let alltimersProcessed = Promise.resolve();

timerArr.forEach(function (produceFn) {
  alltimersProcessed = alltimersProcessed.then(produceFn)
});

alltimersProcessed.then(function () {
  console.log('All timers have been processed...');
});

```

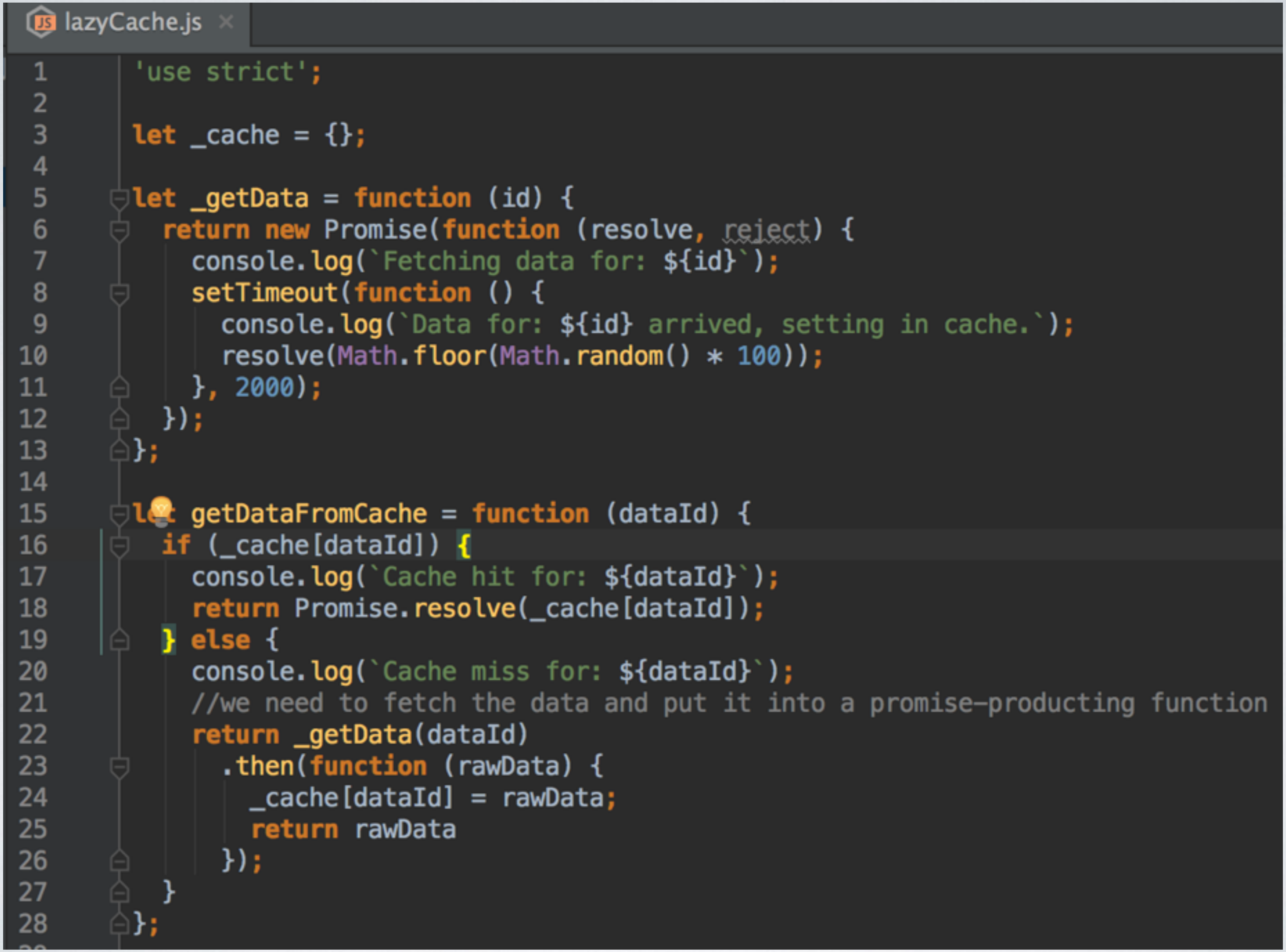
# COMMON USE CASE SAME INTERFACES


syncUsage.js

```javascript
'use strict';

let answerRandomly = function () {
  return new Promise(function (resolve, reject) {
    setTimeout(function () {
      resolve(Math.floor(Math.random() * 100));
    }, 2000)
  });
};

let answer = function (question) {
  //poor man's check for multiple words in a string :)
  if ((question.indexOf('life') + question.indexOf('universe') + question.indexOf('everything')) > -3) {
    return Promise.resolve(42);
  } else {
    return answerRandomly();
  }
};

answer(process.argv[2] || process.argv[1]).then(function (answer) {
  console.log(`The answer is: ${answer}`);
});
```

## ADVANCED EXAMPLE - LAZY CACHE

- we need to fetch data
- fetch is async
- we want to hide details of cache
- interface should always be the same



lazyCache.js

```javascript
'use strict';

let _cache = {};

let _getData = function (id) {
  return new Promise(function (resolve, reject) {
    console.log(`Fetching data for: ${id}`);
    setTimeout(function () {
      console.log(`Data for: ${id} arrived, setting in cache.`);
      resolve(Math.floor(Math.random() * 100));
    }, 2000);
  });
};

let getDataFromCache = function (dataId) {
  if (_cache[dataId]) {
    console.log(`Cache hit for: ${dataId}`);
    return Promise.resolve(_cache[dataId]);
  } else {
    console.log(`Cache miss for: ${dataId}`);
    //we need to fetch the data and put it into a promise-productinng function
    return _getData(dataId)
      .then(function (rawData) {
        _cache[dataId] = rawData;
        return rawData
      });
  }
};
```

## LINKS

- Promises are the monad of asynchronous programming http://bit.ly/1pH8qxJ
- Callbacks are Imperative, Promises are functional: http://bit.ly/1hwxDEf
- Promise Anti-Patterns http://bit.ly/1bKThTD
- Promise Tutorial http://bit.ly/1fgVvY9