JavaScript async and await in loops

Basic `async` and `await` is simple. Things get a bit more complicated when you try to use `await` in loops.

In this article, I want to share some gotchas to watch out for if you intend to use `await` in loops.

Before you begin

I'm going to assume you know how to use `async` and `await`. If you don't, read the previous article to familiarize yourself before continuing.

Preparing an example

For this article, let's say you want to get the number of fruits from a fruit basket.

```
const fruitBasket = {
  apple: 27,
  grape: 0,
  pear: 14
}
```

You want to get the number of each fruit from the fruitBasket. To get the number of a fruit, you can use a `getNumFruit` function.

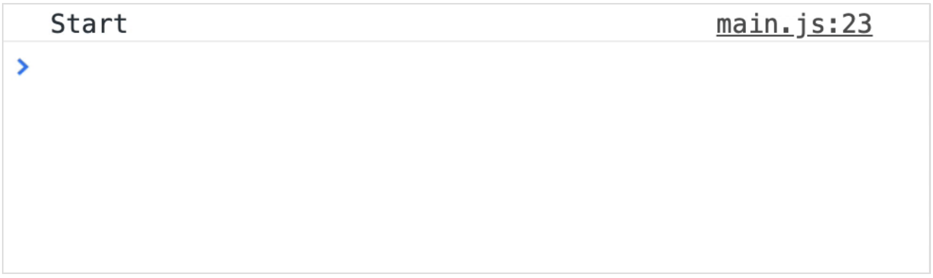
```
const getNumFruit = fruit => {
  return fruitBasket[fruit]
}

const numApples = getNumFruit('apple')
console.log(numApples) // 27
```

Now, let's say `fruitBasket` lives on a remote server. Accessing it takes one second. We can mock this one-second delay with a timeout. (Please refer to the previous article if you have problems understanding the timeout code).

```
const sleep = ms => {
  return new Promise(resolve => setTimeout(resolve, ms))
}

const getNumFruit = fruit => {
  return sleep(1000).then(v => fruitBasket[fruit])
}

getNumFruit('apple')
  .then(num => console.log(num)) // 27
```

Finally, let's say you want to use `await` and `getNumFruit` to get the number of each fruit in asynchronous function.

```
const control = async _ => {
  console.log('Start')

  const numApples = await getNumFruit('apple')
  console.log(numApples)

  const numGrapes = await getNumFruit('grape')
  console.log(numGrapes)

  const numPears = await getNumFruit('pear')
  console.log(numPears)

  console.log('End')
}
```



```
Start                                              main.js:23
>
```

With this, we can begin looking at `await` in loops.

Await in a for loop

Let's say we have an array of fruits we want to get from the fruit basket.

```
const fruitsToGet = ['apple', 'grape', 'pear']
```

We are going to loop through this array.

```
const forLoop = async _ => {
  console.log('Start')

  for (let index = 0; index < fruitsToGet.length; index++) {
    // Get num of each fruit
  }

  console.log('End')
}
```

In the for-loop, we will use `getNumFruit` to get the number of each fruit. We'll also log the number into the console.

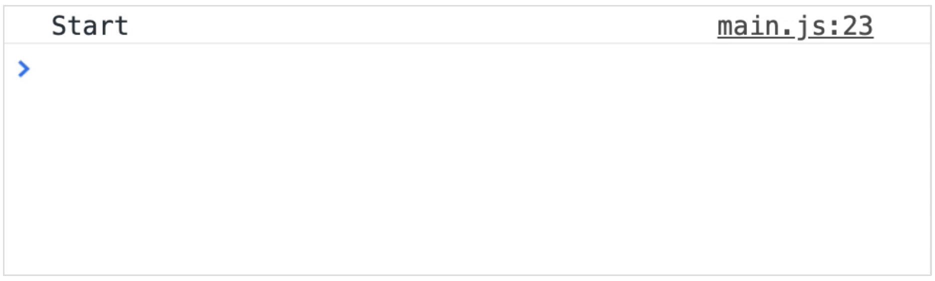
Since `getNumFruit` returns a promise, we can `await` the resolved value before logging it.

```
const forLoop = async _ => {
  console.log('Start')

  for (let index = 0; index < fruitsToGet.length; index++) {
    const fruit = fruitsToGet[index]
    const numFruit = await getNumFruit(fruit)
    console.log(numFruit)
  }

  console.log('End')
}
```

When you use `await`, you expect JavaScript to pause execution until the awaited promise gets resolved. This means `await`s in a for-loop should get executed in series.

The result is what you'd expect.

```
'Start'
'Apple: 27'
'Grape: 0'
'Pear: 14'
'End'
```



```
Start                                              main.js:23
```

This behaviour works with most loops (like `while` and `for-of` loops)…

But it won't work with loops that require a callback. Examples of such loops that require a fallback include `forEach`, `map`, `filter`, and `reduce`. We'll look at how `await` affects `forEach`, `map`, and `filter` in the next few sections.

Await in a forEach loop

We'll do the same thing as we did in the for-loop example. First, let's loop through the array of fruits.

```
const forEachLoop = _ => {
  console.log('Start')

  fruitsToGet.forEach(fruit => {
    // Send a promise for each fruit
  })

  console.log('End')
}
```

Next, we'll try to get the number of fruits with `getNumFruit`. (Notice the `async` keyword in the callback function. We need this `async` keyword because `await` is in the callback function).

```
const forEachLoop = _ => {
  console.log('Start')

  fruitsToGet.forEach(async fruit => {
    const numFruit = await getNumFruit(fruit)
    console.log(numFruit)
  })

  console.log('End')
}
```

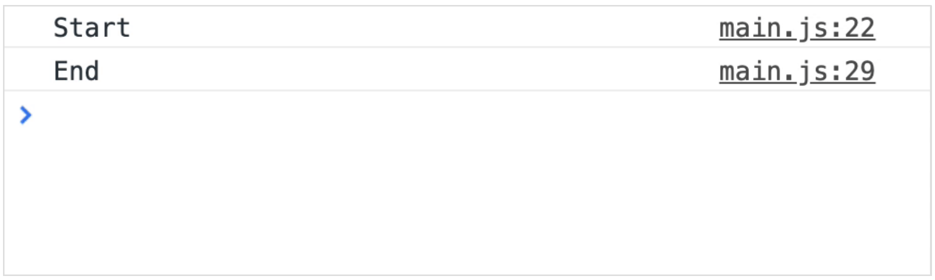
You might expect the console to look like this:

```
'Start'
'27'
'0'
'14'
'End'
```

But the actual result is different. JavaScript proceeds to call `console.log('End')` before the promises in the forEach loop gets resolved.

The console logs in this order:

```
'Start'
'End'
'27'
'0'
'14'
```



```
Start                                              main.js:22
End                                                main.js:29
>
```

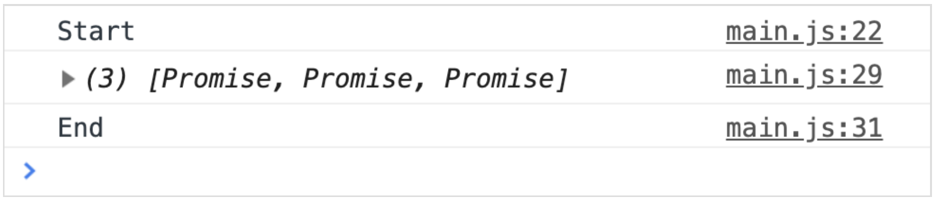
JavaScript does this because `forEach` is not promise-aware. It cannot support `async` and `await`. You *cannot* use `await` in `forEach`.

Await with map

If you use `await` in a `map`, `map` will always return an array of promise. This is because asynchronous functions always return promises.

```
const mapLoop = async _ => {
  console.log('Start')

  const numFruits = await fruitsToGet.map(async fruit => {
    const numFruit = await getNumFruit(fruit)
    return numFruit
  })

  console.log(numFruits)
  console.log('End')
}
```

```
'Start'
'[Promise, Promise, Promise]'
'End'
```



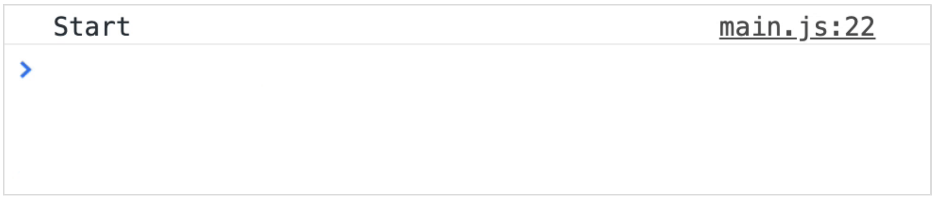
```
Start                                         main.js:22
▶ (3) [Promise, Promise, Promise]             main.js:29
End                                           main.js:31
>
```

Since `map` always return promises (if you use `await`), you have to wait for the array of promises to get resolved. You can do this with `await Promise.all(arrayOfPromises)`.

```
const mapLoop = async _ => {
  console.log('Start')

  const promises = fruitsToGet.map(async fruit => {
    const numFruit = await getNumFruit(fruit)
    return numFruit
  })

  const numFruits = await Promise.all(promises)
  console.log(numFruits)

  console.log('End')
}
```

Here's what you get:

```
'Start'
'[27, 0, 14]'
'End'
```



```
Start                                         main.js:22
>
```

You can manipulate the value you return in your promises if you wish to. The resolved values will be the values you return.

```
const mapLoop = async _ => {
  // ...
  const promises = fruitsToGet.map(async fruit => {
    const numFruit = await getNumFruit(fruit)
    // Adds onn fruits before returning
    return numFruit + 100
  })
  // ...
}
```

```
'Start'
'[127, 100, 114]'
'End'
```

Await with filter

When you use `filter`, you want to filter an array with a specific result. Let's say you want to create an array with more than 20 fruits.

If you use `filter` normally (without await), you'll use it like this:

```
// Filter if there's no await
const filterLoop = _ => {
  console.log('Start')

  const moreThan20 = await fruitsToGet.filter(fruit => {
    const numFruit = fruitBasket[fruit]
    return numFruit > 20
  })

  console.log(moreThan20)
  console.log('End')
}
```

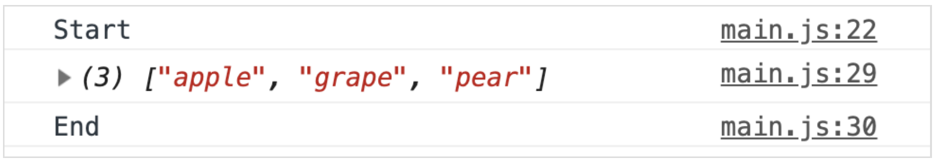
You would expect `moreThan20` to contain only apples because there are 27 apples, but there are 0 grapes and 14 pears.

```
'Start'
['apple']
'End'
```

`await` in `filter` doesn't work the same way. In fact, it doesn't work at all. You get the unfiltered array back...

```
const filterLoop = _ => {
  console.log('Start')

  const moreThan20 = await fruitsToGet.filter(async fruit => {
    const numFruit = getNumFruit(fruit)
    return numFruit > 20
  })

  console.log(moreThan20)
  console.log('End')
}
```

```
'Start'
['apple', 'grape', 'pear']
'End'
```



```
Start                                          main.js:22
▶ (3) ["apple", "grape", "pear"]               main.js:29
End                                            main.js:30
```

Here's why it happens.

When you use `await` in a `filter` callback, the callback always a promise. Since promises are always truthy, everything item in the array passes the filter. Writing `await` in a `filter` is like writing this code:

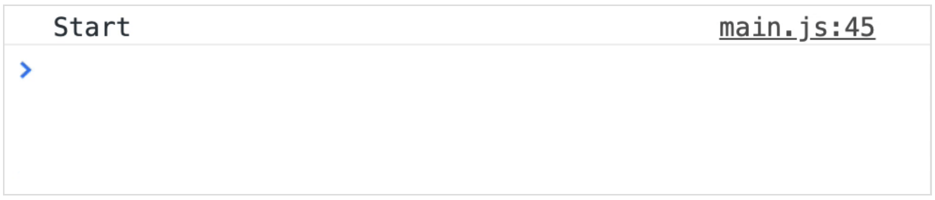
```
// Everything passes the filter...
const filtered = array.filter(true)
```

There are three steps to use `await` and `filter` properly:

1. Use `map` to return an array promises
2. `await` the array of promises
3. `filter` the resolved values

```
const filterLoop = async _ => {
  console.log('Start')

  const promises = await fruitsToGet.map(fruit =>
getNumFruit(fruit))
  const numFruits = await Promise.all(promises)

  const moreThan20 = fruitsToGet.filter((fruit, index) => {
    const numFruit = numFruits[index]
    return numFruit > 20
  })

  console.log(moreThan20)
  console.log('End')
}
```

```
Start
[ 'apple' ]
End
```



```
Start                                                  main.js:45
>
```

Await with reduce

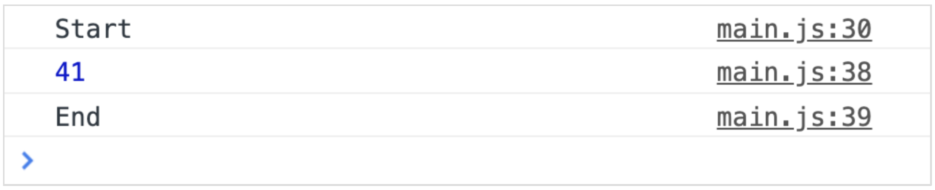
For this case, let's say you want to find out the total number of fruits in the fruitBastet. Normally, you can use `reduce` to loop through an array and sum the number up.

```
// Reduce if there's no await
const reduceLoop = _ => {
  console.log('Start')

  const sum = fruitsToGet.reduce((sum, fruit) => {
    const numFruit = fruitBasket[fruit]
    return sum + numFruit
  }, 0)

  console.log(sum)
  console.log('End')
}
```

You'll get a total of 41 fruits. (27 + 0 + 14 = 41).

```
'Start'
'41'
```

```
'End'
```

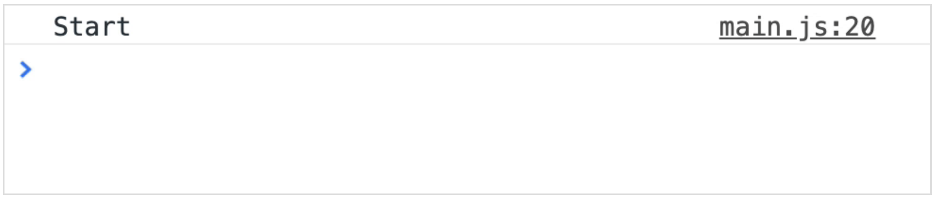


```
Start                                              main.js:30
41                                                 main.js:38
End                                                main.js:39
>
```

When you use `await` with reduce, the results get extremely messy.

```
// Reduce if we await getNumFruit
const reduceLoop = async _ => {
  console.log('Start')

  const sum = await fruitsToGet.reduce(async (sum, fruit) => {
    const numFruit = await getNumFruit(fruit)
    return sum + numFruit
  }, 0)

  console.log(sum)
  console.log('End')
}
```

```
'Start'
'[object Promise]14'
'End'
```



```
Start                                              main.js:20
>
```

What?! `[object Promise]14` ?!

Dissecting this is interesting.

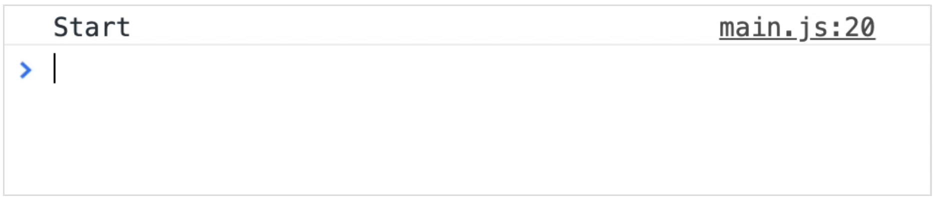
- In the first iteration, `sum` is `0`. `numFruit` is 27 (the resolved value from `getNumFruit('apple')`). `0 + 27` is 27.
- In the second iteration, `sum` is a promise. (Why? Because asynchronous functions always return promises!) `numFruit` is 0. A promise cannot be added to an object normally, so the JavaScript converts it to `[object Promise]` string. `[object Promise] + 0` is `[object Promise]0`
- In the third iteration, `sum` is also a promise. `numFruit` is `14`. `[object Promise] + 14` is `[object Promise]14`.

Mystery solved!

This means, you can use `await` in a `reduce` callback, but you have to remember to `await` the accumulator first!

```
const reduceLoop = async _ => {
  console.log('Start')

  const sum = await fruitsToGet.reduce(async (promisedSum,
fruit) => {
    const sum = await promisedSum
    const numFruit = await getNumFruit(fruit)
    return sum + numFruit
  }, 0)

  console.log(sum)
  console.log('End')
}
```

```
'Start'
'41'
'End'
```

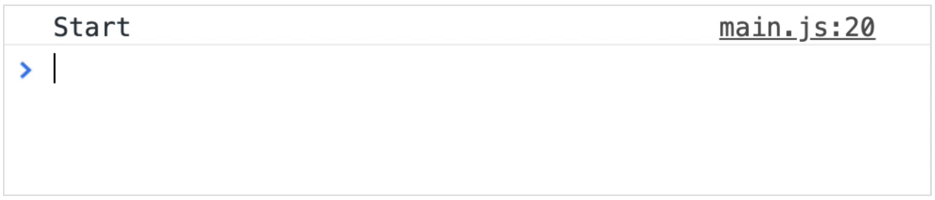


```
Start                                                    main.js:20
> |
```

But... as you can see from the gif, it takes pretty long to `await` everything. This happens because `reduceLoop` needs to wait for the `promisedSum` to be completed for each iteration.

There's a way to speed up the reduce loop. (I found out about this thanks to [Tim Oxley](#)). If you `await getNumFruits()` first before `await promisedSum`, the `reduceLoop` takes only one second to complete:

```
const reduceLoop = async _ => {
  console.log('Start')

  const sum = await fruitsToGet.reduce(async (promisedSum,
fruit) => {
    // Heavy-lifting comes first.
    // This triggers all three `getNumFruit` promises before
waiting for the next interation of the loop.
    const numFruit = await getNumFruit(fruit)
    const sum = await promisedSum
    return sum + numFruit
  }, 0)

  console.log(sum)
  console.log('End')
}
```



```
Start                                                    main.js:20
> |
```

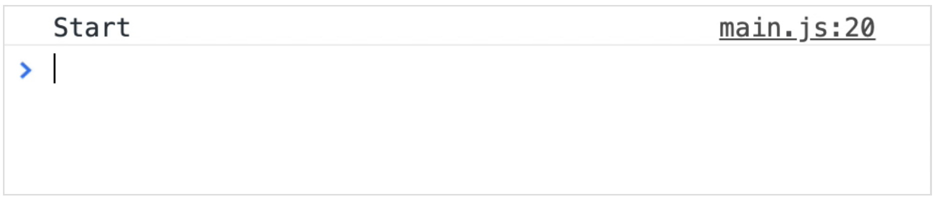
This works because `reduce` can fire all three `getNumFruit` promises before waiting for the next iteration of the loop. However, this method is slightly confusing since you have to be careful of the order you `await` things.

The simplest (and most efficient way) to use `await` in reduce is to:

1. Use `map` to return an array promises
2. `await` the array of promises
3. `reduce` the resolved values

```
const reduceLoop = async _ => {
  console.log('Start')

  const promises = fruitsToGet.map(getNumFruit)
  const numFruits = await Promise.all(promises)
  const sum = numFruits.reduce((sum, fruit) => sum + fruit)

  console.log(sum)
  console.log('End')
}
```

This version is simple to read and understand, and takes one second to calculate the total number of fruits.



```
Start                                    main.js:20
>
```

Key Takeaways

1. If you want to execute `await` calls in series, use a for-loop (or any loop without a callback).
2. Don't ever use `await` with `forEach`. Use a for-loop (or any loop without a callback) instead.
3. Don't `await` inside `filter` and `reduce`. Always `await` an array of promises with `map`, then `filter` or `reduce` accordingly.