Chapter 82: Async Iterators

An `async` function is one that returns a promise. `await` yields to the caller until the promise resolves and then continues with the result.

An iterator allows the collection to be looped through with a `for-of` loop.

An async iterator is a collection where each iteration is a promise which can be awaited using a `for-await-of` loop.

Async iterators are a stage 3 proposal. They are in Chrome Canary 60 with `--harmony-async-iteration`

Section 82.1: Basics

A JavaScript `Iterator` is an object with a `.next()` method, which returns an `IteratorItem`, which is an object with `value : <any>` and `done : <boolean>`.

A JavaScript `AsyncIterator` is an object with a `.next()` method, which returns a `Promise<IteratorItem>`, a *promise* for the next value.

To create an AsyncIterator, we can use the *async generator* syntax:

```
/**
 * Returns a promise which resolves after time had passed.
 */
const delay = time => new Promise(resolve => setTimeout(resolve, time));

async function* delayedRange(max) {
  for (let i = 0; i < max; i++) {
    await delay(1000);
    yield i;
  }
}
```

The `delayedRange` function will take a maximum number, and returns an `AsyncIterator`, which yields numbers from 0 to that number, in 1 second intervals.

Usage:

```
for await (let number of delayedRange(10)) {
  console.log(number);
}
```

The `for await of` loop is another piece of new syntax, available only inside of async functions, as well as async generators. Inside the loop, the values yielded (which, remember, are Promises) are unwrapped, so the Promise is hidden away. Within the loop, you can deal with the direct values (the yielded numbers), the `for await of` loop will wait for the Promises on your behalf.

The above example will wait 1 second, log 0, wait another second, log 1, and so on, until it logs 9. At which point the `AsyncIterator` will be `done`, and the `for await of` loop will exit.