Async/Await Internals

Promises are the fundamental tool for integrating with async/await. Now that you've seen how promises work from the ground up, it's time to go from the micro to the macro and see what happens when you `await` on a promise. Even though async functions are flat like synchronous functions, they're as asynchronous as the most callback-laden banana code under the hood.

As you might have already guessed, `await` makes JavaScript call `then()` under the hood.

Example 3.1

```
const p = {
  then: onFulfilled => {
    // Prints "then(): function () { [native code] }"
    console.log('then():', onFulfilled.toString());
    // Only one entry in the stack:
    // Error
    //     at Object.then (/examples/chapter3.test.js:8:21)
    console.log(new Error().stack);
    onFulfilled('Hello, World!');
  }
};

console.log(await p); // Prints "Hello, World!"
```

The `await` keyword causes JavaScript to *pause* execution until the next iteration of the event loop. In the below code, the `console.log()` after the `await` runs **after** the `++currentId` code, even though the increment is in a callback. The `await` keyword causes the async function to pause and then resume later.

Example 3.2

```
const startId = 0;
let currentId = 0;
process.nextTick(() => ++currentId);
const p = {
  then: onFulfilled => {
    console.log('then():', currentId - startId); // "then(): 1"
    onFulfilled('Hello, World!');
  }
};

console.log('Before:', currentId - startId); // "Before: 0"
await p;
console.log('After:', currentId - startId); // "After: 1"
```

Notice that the `then()` function runs on the next tick, even though it is fully synchronous. This means that `await` always pauses execution until at least the next tick, even if the thenable is not async.The same thing happens when the awaited promise is rejected. If you call `onRejected(err)`, the `await` keyword throws `err` in your function body.

Example 3.3

```
const startId = 0;
let currentId = 0;
process.nextTick(() => ++currentId);
const p = {
  then: (onFulfilled, onRejected) => {
    console.log('then():', currentId - startId); // "then(): 1
    return onRejected(Error('Oops!'));
  }
};

try {
  console.log('Before:', currentId - startId); // "Before: 0"
  await p;
  console.log('This does not print');
} catch (error) {
  console.log('After:', currentId - startId); // "After: 1"
}
```

`await` vs `return`

Recall that `return` in an async function resolves the promise that the async function returns. This means you can `return` a promise. What's the difference between `await` and `return`? The obvious answer is that, when you `await` on a promise, JavaScript pauses execution of the async function and resumes later, but when you `return` a promise, JavaScript finishes executing the async function. JavaScript doesn't "resume" executing the function after you `return`.

The obvious answer is correct, but has some non-obvious implications that tease out how `await` works. If you wrap `await p` in a `try/catch` and `p` is rejected, you can catch the error. What happens if you instead `return p`?

Example 3.4

```
async function test() {
  try {
    return Promise.reject(new Error('Oops!'));
  } catch (error) { return 'ok'; }
}
// Prints "Oops!"
test().then(v => console.log(v), err => console.log(err.message));
```