Created	@May 15, 2023 10:18 PM
Class	
Type	
Materials	
Reviewed	☐

So until now we already have good context about promises. But still writing promise-based syntax where we use a lot of `.then` is not that readable. We can enhance readability by introducing `async-await`.

`async` and `await` are two keywords native to JS, which can help us to write cleaner promise-based codes.

async keyword

async keyword is used with a function to declare that the function might be doing some time-consuming async task, and will always return a promise.

```
async function task() {
    return 10;
}
```

Here we are returning a Number, but because we have declared the function async, it will wrap the number in a promise object and always returns a promise.


```
async function task() {
    return new Promise((res, rej) => {
        setTimeout(() => {
            res("done");
        }, 4000);
    });
}
```

In the above piece of code, we are returning a promise only, so when we call the task function, it immediately returns a pending promise, and that promise is resolved once the `res` function is called after the timer is completed. We could have achieved it with normal functions also but, marking something async gives us more power, one of them is that whether you return a non-promise value or a promise value, it will always return a promise.

One more capability it provides is the usage of `await` keyword.

await keyword

await keyword can only be used inside an async function(there is one exception to this). What awaits does it, the moment you right await and then put a value after it, it starts treating that value as a promise (even if it is not a promising value).

```
async function task() {
    await 10; // here await will assume 10 to be a promise
}
```

The moment your function hits any await keyword, you will be thrown outside the function just like how JS moves forward when it sees a promise object.

So technically everything is working how promises work, the moment you will be thrown outside the function, you will resume executing the remaining code. As I mentioned await treats the value like a promise (even if it is not a promise), so what will happen is when this promise gets resolved, the remaining code of your function waits inside the microtask queue

```
async function consume() {
    console.log("inside consume");
    let x = await fakeDownloader();
    console.log("first value downloaded is ",x);
    let y = await fakeDownloader();
    console.log("second downloaded value is", y);
    console.log("end");
}
console.log("start");
consume();
console.log("end");
```