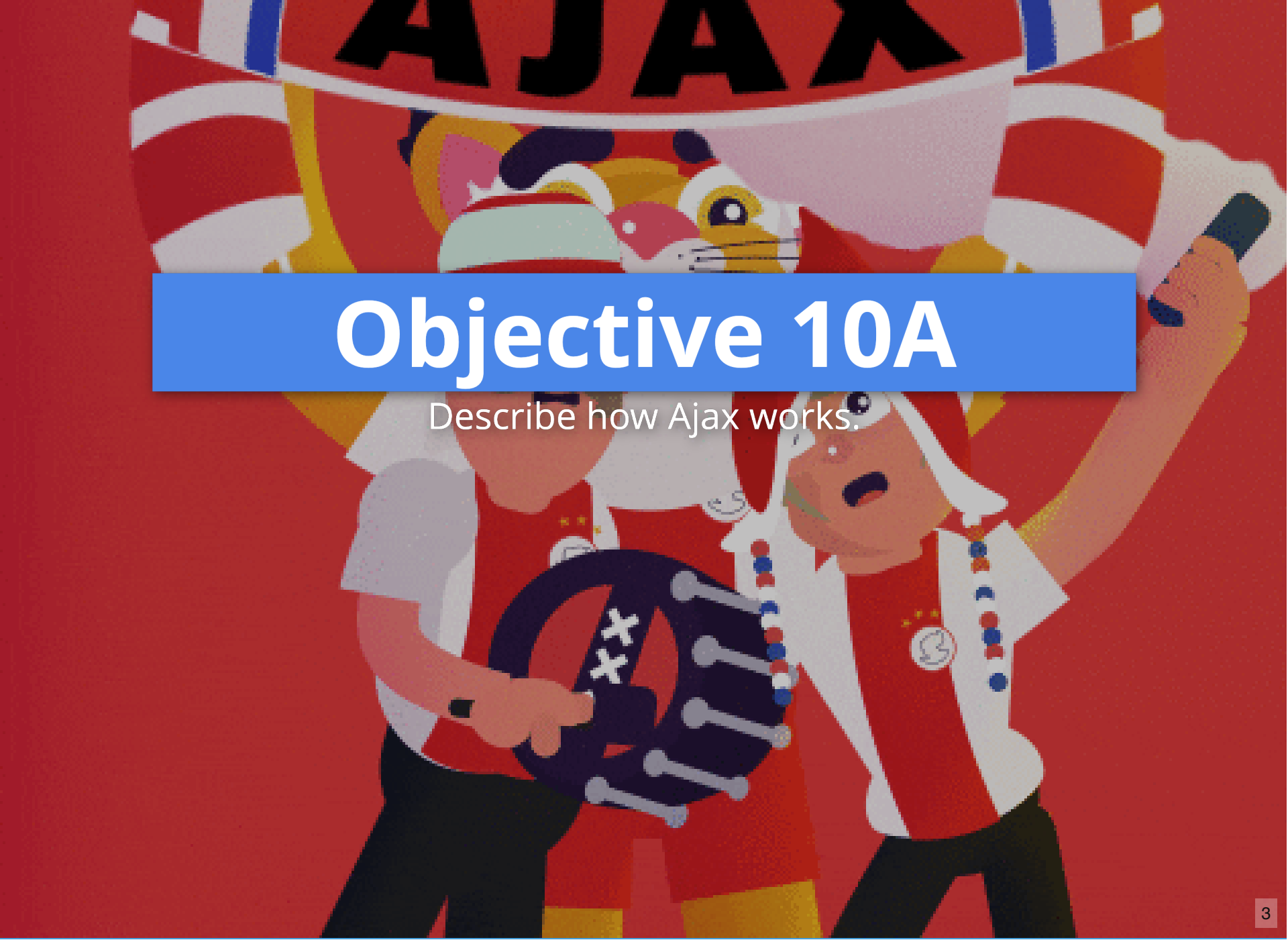
INFO 2124

JavaScript

```
const createSlideShow = () => {
        const title = "Module 10";
        const subTitle = "Asynchronous JavaScript and XML (AJAX)"
        return {
             getTitle () {
                return `<h1>${title}</h1>
                        <h2>${subTitle}<h2>`;
        }
      }
}

$(document).ready(()=> {
        const slideShow = createSlideShow();
        $('#container').html(slideShow.getTitle());
});
```

Module Objectives

A. Describe how Ajax works.
B. Distinguish between XML and JSON.
C. Name and describe the three states of a Promise object.
D. Describe how you can use the aync and await keywords to work with asynchronous functions.
E. Use your browser to review the response that's returned from a request to a web service.
F. Use the Fetch API to make Ajax requests that update a web page without reloading it.



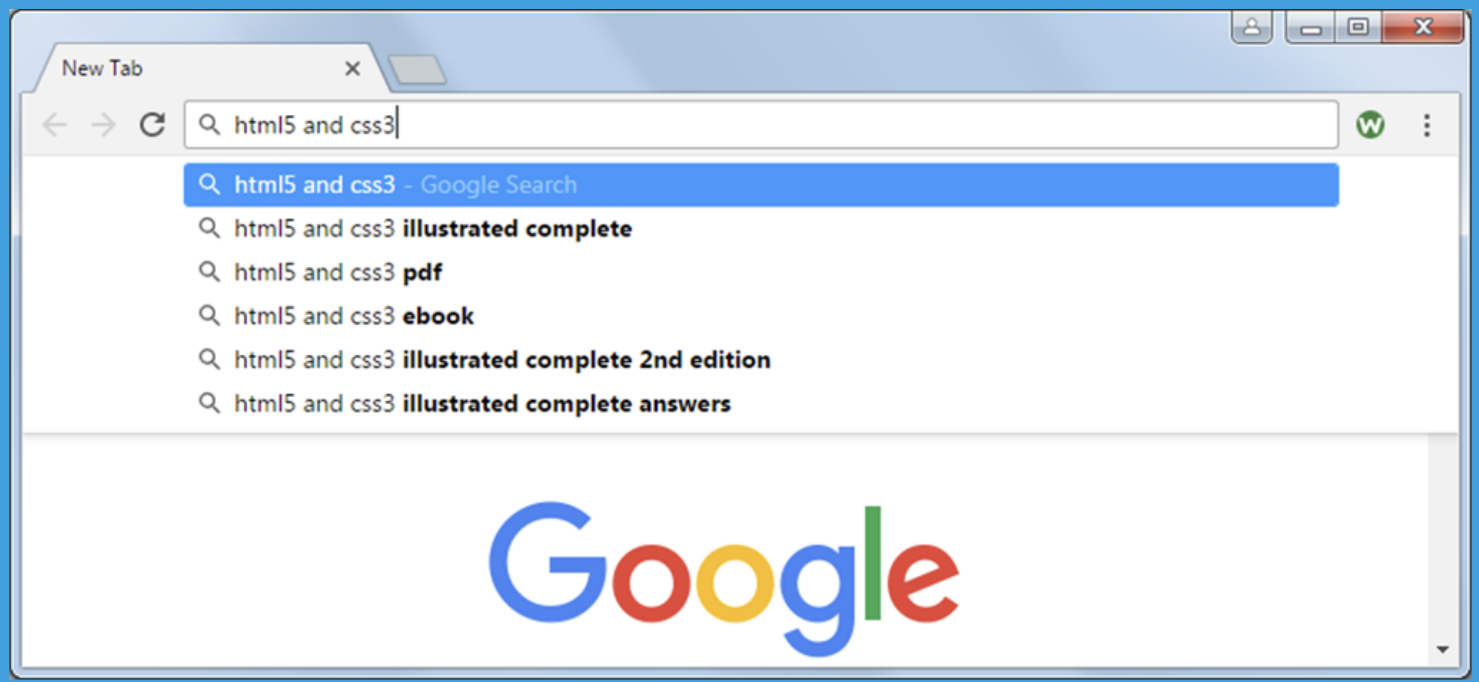
Objective 10A

Describe how Ajax works.

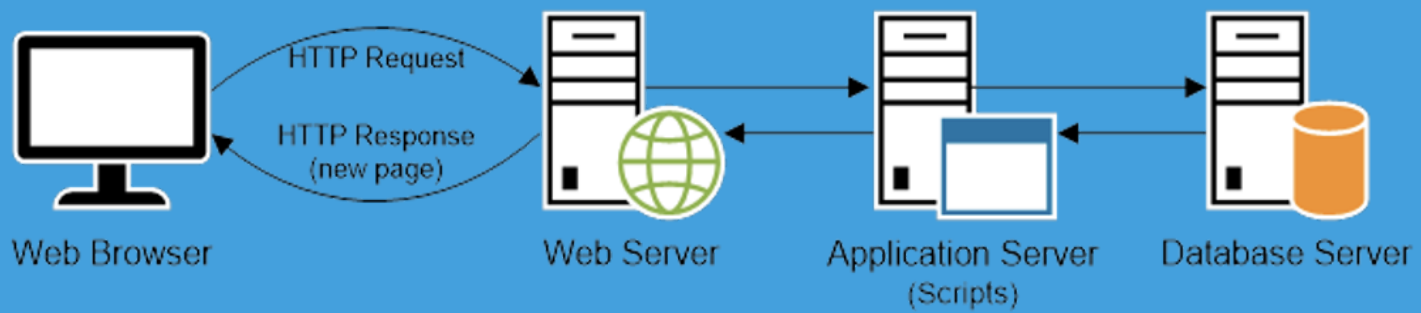
Ajax

- *Ajax* (*Asynchronous JavaScript and XML*) allows a web browser to update a web page with data from the web server without reloading the entire page. This is sometimes known as a "*partial refresh*."
  - It's what makes viewing this slideshow through the browser possible!
- When working with Ajax, JavaScript sends the request, processes the response, and updates the DOM with the new data. As a result, the browser doesn't need to refresh the whole page.
- To send an Ajax request, JavaScript can use a browser object known as the *XMLHttpRequest* (*XHR*) *object*, or it can use the *Fetch API*.
- Ajax requests are often made to *web services* that provide *Application Programming Interfaces* (*APIs*) that developers can use to interact with a website.
- Many popular websites provide APIs that let you use Ajax to get data from their sites.
- Many popular websites use Ajax to improve the way they function.

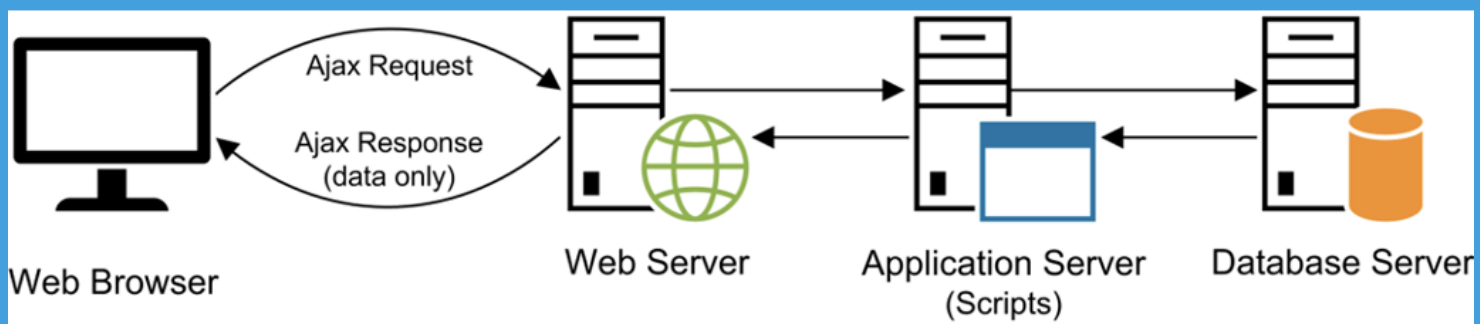
Google's Auto Suggest is an Ajax Application



HTTP vs. Ajax Requests



A normal HTTP Request



How an Ajax Request is Processed

Objective 10B

Distinguish between XML and JSON.

XML and JSON

- The two most common data formats for working with Ajax are XML and JSON
  - These data formats are used for data *serialization*.
  - Serialization is the, "the process of translating a data structure or object state into a format that can be stored...or transmitted (for example, over a computer network) and reconstructed later...." [1]
- Both *XML* (*eXtensoble Markup Language*) and *JSON* (*JavaScript Object Notation*) are formats that use text to store and transmit data.
- Most server-side languages provide methods for parsing JSON returned from a web service into a JavaScript object.
- JSON is less verbose than XML, so it uses less bandwidth when being sent from the server to the client.

Speaker notes

[1]: https://en.wikipedia.org/wiki/Serialization

XML and JSON

- XML
  - Stands for *eXtensible Markup Language*
  - File extension is .xml
- JSON
  - Stands for *JavaScript Object Notation*
  - File extension is .json

Sample XML Data

```xml
<?xml version="1.0" encoding="utf-8"?>
<management>
    <teammember>
        <name>Agnes</name>
        <title>Vice President of Accounting</title>
        <bio>With over 14 years of public accounting ... </bio>
    </teammember>
    <teammember>
        <name>Wilbur</name>
        <title>Founder and CEO</title>
        <bio>While Wilbur is the founder and CEO ... </bio>
    </teammember>
</management>
```

XML looks like HTML with matching opening and closing tags. XML documents can be *validated* by means of a *schema.*

Sample JSON Data

```
{"teammembers":[
    {
        "name":"Agnes",
        "title":"Vice President of Accounting",
        "bio":"With over 14 years of public accounting... "
    },
    {
        "name":"Wilbur",
        "title":"Founder and CEO",
        "bio":"While Wilbur is the founder and CEO ... "
    }
]}
```

JSON looks like a JavaScript object literal. Arrays are enclosed in square brackets just like they are in JavaScript. Unlike XML, JSON documents have no validation mechanism.

A little history

In the early days of Ajax, requests were sent uphill both ways, in the snow . . .

The XMLHttpRequest Object

- "*XMLHttpRequest (XHR) is an API in the form of an object whose methods transfer data between a web browser and a web server. The object is provided by the browser's JavaScript environment.*" [2]
  - The XMLHttpRequest object first appeared in the late 1990s in Internet Explorer 5.0.
  - Google was the first entity to push Ajax around 2004 with the release of Gmail.
  - A draft spec of the XMLHttpRequest object was published by the W3C in 2006.
- The XMLHttpRequest object set the basis for Ajax, allowing requests to be made independently of the browser (in the background).
  - Early web pages were static; if you wanted/needed new content, you would have to reload a whole page, or request a new page.
  - The XMLHttpRequest allowed for fetching and loading of content *asynchornously*.
    - *asynchronous* = not happening at the same time (as the page loads).
      - A one-way conversation.
    - *synchronous* = happening at the same time ("simultaneous")
      - A two-way conversation.

Speaker notes

[2]: https://en.wikipedia.org/wiki/XMLHttpRequest

The XMLHttpRequest Object

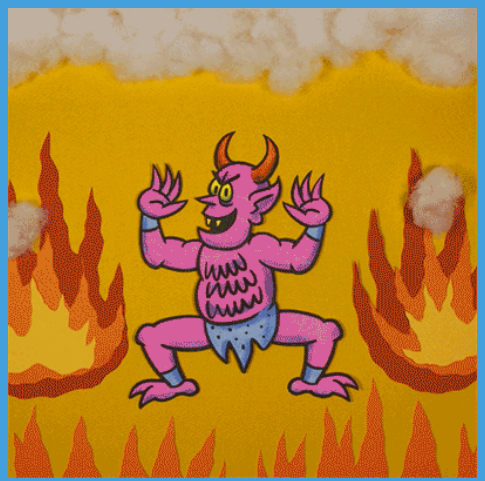
```
1  const xhr = new XMLHttpRequest();
2  xhr.responseType = "json";
3
4  xhr.onreadystatechange = () => {
5      if (xhr.readyState == 4 && xhr.status == 200) {
6          console.log(xhr.response);
7      }
8  };
9  xhr.onerror = e => console.log(e.message);
10
11 xhr.open("GET",
12     "https://jsonplaceholder.typicode.com/users");
13 xhr.send();
```

The code above illustrates a basic XMLHttpRequest. The *onreadystatechange* method is the method that fires when the XMLHttpRequest receives a response from the remote server. To handle events raised by the XMLHttpRequest object, you can assign a *callback function* to the event. The callback function runs when the event occurs; callback functions are essential to asynchronous JavaScript.

Callback Hell

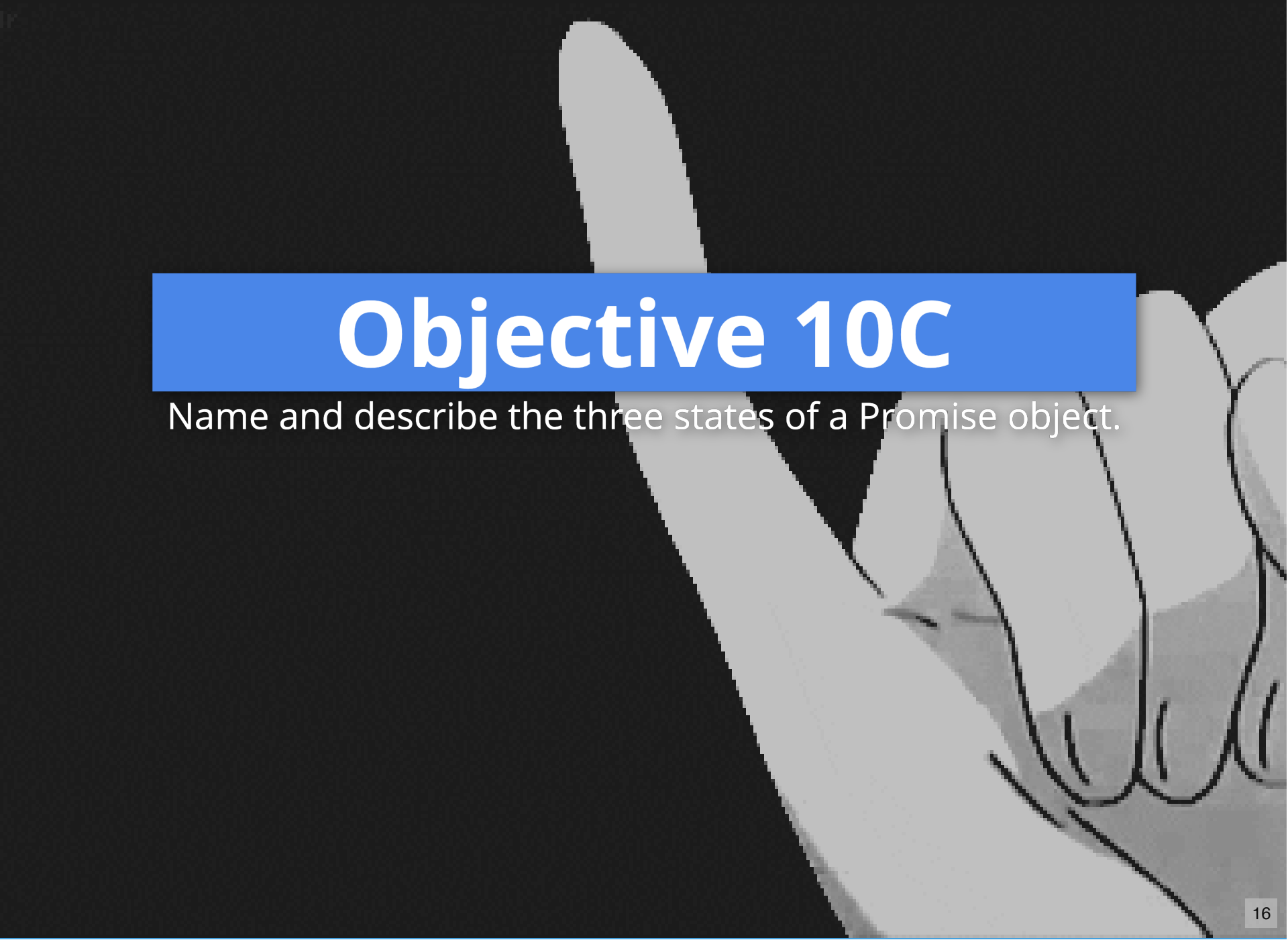
```
const xhr1 = new XMLHttpRequest();
xhr1.responseType = "json";

const domain = "https://jsonplaceholder.typicode.com";
let url = `${domain}/photos/1`;

xhr1.onreadystatechange = () => {
    if (xhr1.readyState == 4 && xhr1.status == 200) {
        const photo = xhr1.response;

        const xhr2 = new XMLHttpRequest();
        xhr2.responseType = "json";
        url = `${domain}/albums/${photo.albumId}`;

        xhr2.onreadystatechange = () => {
            if (xhr2.readyState == 4 && xhr2.status == 200) {
                const album = xhr2.response;

                const xhr3 = new XMLHttpRequest();
                xhr3.responseType = "json";
                url = `${domain}/users/${album.userId}`;
                xhr3.onreadystatechange = () => {
                    if (xhr3.readyState == 4 &&
                        xhr3.status == 200) {
                        const user = xhr3.response;
                        let html = `<img src="${photo.url}"
                            alt="${photo.title}">`;
                        html +=
                            `<h4>In album ${album.title}</h4>`;
                        html += `Posted by ${user.username}`;
                        $("#photo").html(html);
                    }
                };
                xhr3.open("GET", url);
                xhr3.send();
            }
        };
        xhr2.open("GET", url);
        xhr2.send();
    }
};
xhr1.open("GET", url);
xhr1.send();
```

- If you need to use the data from one asynchronous call to make another asynchronous call (as illustrated to the left), and you're using the *XMLHttpRequestObject*, you can nest one callback within another callback function --- this can create "callback hell" since nested callbacks can be difficulty to read and maintain.
  - We can solve this using the JavaScript *Fetch* API, instead of the classic *XMLHttpRequestObject*!



Objective 10C

Name and describe the three states of a Promise object.



The *Fetch* API

- Using the *XMLHttpResquest* object was the original method of create Ajax requests.
- For newer development, it's generally considered a best practice to use the *Fetch* API instead of the older *XMLHttpRequest* object.
  - The *Fetch* API returns *Promise* objects, which can be chained. This solves the "callback hell" problem that's common when you use the XMLHttpResponse object!

The *Fetch* API

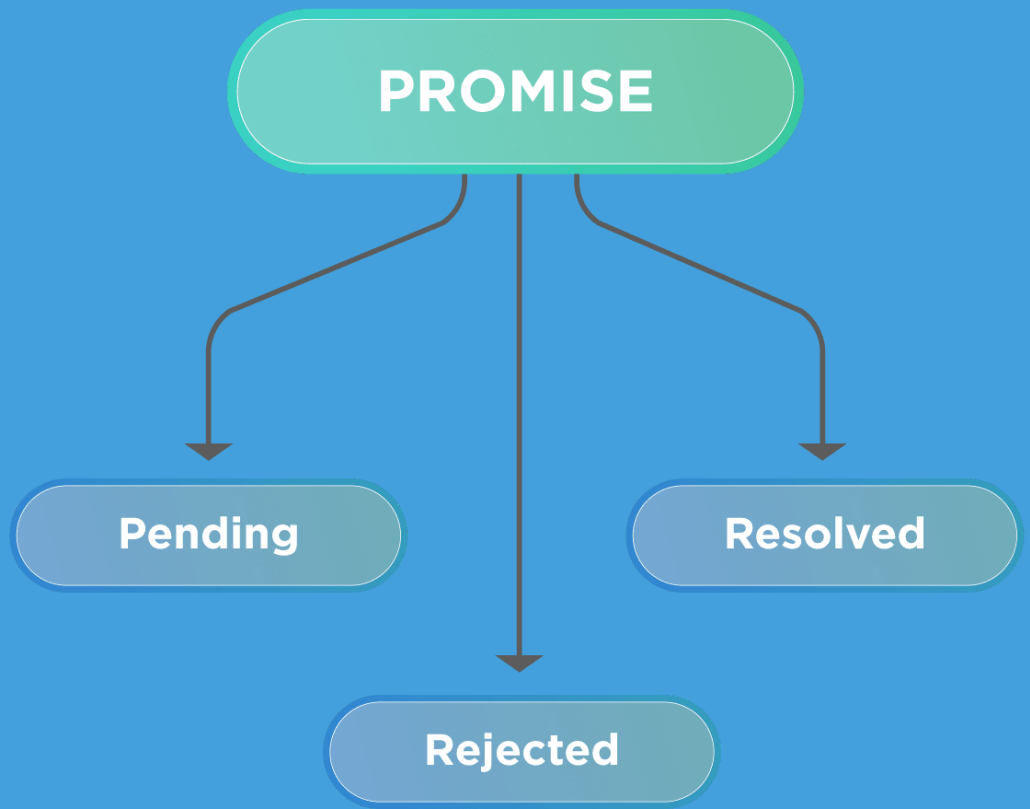
- The *fetch()* method of the *Fetch* API accepts the URL for a get request.
  - The *fetch()* method returns a *Promise* object, which represents the eventual return of an asynchronous request.

```
fetch("https://jsonplaceholder.typicode.com/users")
    .then( response => response.json() )
    .then( json => console.log(json) )
    .catch( e => console.log(e.message) );
```

The *Promise* Object

- The *fetch()* method of the *Fetch* API returns a *Promise* object.
  - A *Promise* represents the eventual return value of an asynchronous request.
  - The value eventually returned is a *Response* object, which represents an HTTP response.
- A *Promise* object has three states:
  1. *pending* - when the *promise* is first created, the *promise* in "pending"
  2. *fulfilled* - when the promise returns its value, the promise is "fulfilled"
  3. *rejected* - if there is an error, the promise is *rejected*
- A promise that is no longer *pending* is considered *settled*.
  - This is true whether the promise is *fulfilled* or *rejected*.
- A promise can be *resolved* **WITHOUT** being *fulfilled*.
  - For example, if a promise returns another promise, the first promise is considered *resolved* even though the requested data isn't returned yet and the promise isn't fulfilled.

The *Promise* Object



The Promise Object

- Here are two methods of the Promise object:
  - **then(*callback*)**
    - Registers the *callback* function to execute when the promise is resolved. the callback function receives a single parameter, which is the eventual return value of the asynchronous request. Returns a *Promise* object.
    - You can chain multiple *then()* statements together to build a callback chain.
  - **catch(*callback*)**
    - Registers the callback function to execute when the promise is rejected (an error occurs). The callback function receives a single parameter, which is usually an *Error* object. Returns a *Promise* object.

The Promise Object

"fetch" this

```
1 fetch("https://jsonplaceholder.typicode.com/users")
2     .then( response => response.json() )
3     .then( json => console.log(json) )
4     .catch( e => console.log(e.message) );
```

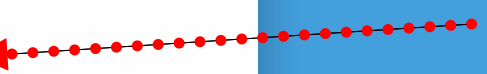
The Promise Object

```
1 fetch("https://jsonplaceholder.typicode.com/users")
2     .then( response => response.json() )
3     .then( json => console.log(json) )
4     .catch( e => console.log(e.message) );
```

"then" on the response, take the response and execute the .json() method on it...

The Promise Object

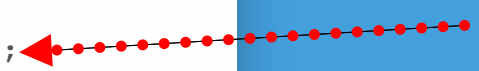
```
1 fetch("https://jsonplaceholder.typicode.com/users")
2     .then( response => response.json() )
3     .then( json => console.log(json) )
4     .catch( e => console.log(e.message) );
```



"then" take the resulting JSON object and write it to the Console using the .log() method...

The Promise Object

```
1 fetch("https://jsonplaceholder.typicode.com/users")
2     .then( response => response.json() )
3     .then( json => console.log(json) )
4     .catch( e => console.log(e.message) );
```



"catch" any errors that may occur; name the error "e" and then use the .log() method to output e.message to the Console.

The Promise Object

```
1 fetch("https://jsonplaceholder.typicode.com/users")
2     .then( response => response.json() )
3     .then( json => console.log(json) )
4     .catch( e => console.log(e.message) );
```


```
▼ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶ 0: {id: 1, name: "Leanne Graham", username: "Bret", email: "Sincere@april.biz", address: {…}, …}
  ▶ 1: {id: 2, name: "Ervin Howell", username: "Antonette", email: "Shanna@melissa.tv", address: {…}, …}
  ▶ 2: {id: 3, name: "Clementine Bauch", username: "Samantha", email: "Nathan@yesenia.net", address: {…}, …}
  ▶ 3: {id: 4, name: "Patricia Lebsack", username: "Karianne", email: "Julianne.OConner@kory.org", address: {…}…
```

Handling Errors

- It's good practice to end every promise chain with a *catch()* method for general errors.
  - You can also include a *catch()* method earlier in a promise chain for specific errors that can be recovered from.
- You can use the *finally()* method to perform any cleanup activities, if necessary.
  - **finally(*callback*)**
    - Registers the callback function to execute when a promise is settled. The callback function doesn't receive a parameter. Returns a Promise object.

```
readFile()
    .then(displayContents)
    .catch(logError)
    .finally(closeFile);
```

Working With Promises

Creating a Promise

- You can create your own *Promise* object using the *Promise* constructor.

```
1 const myPromise = new Promise(callback);
```

- The two parameters of the *callback* passed to the *Promise* constructor:
  1. **resolve**
     - The callback function that is passed to the then() method to execute when the promise is resolved.
  2. **reject**
     - The callback function that is passed to the catch() method to execute when the promise is rejected.

Creating a Promise

```
const wait = milliseconds => {
    return new Promise( (resolve, reject) => {
        if (milliseconds > 0) {
            setTimeout(resolve, milliseconds);
        } else {
            reject( new RangeError(
                "Milliseconds must be positive.") );
        }
    });
}
```

```
actionThatFailsIntermittently()
    .catch( e => wait(200).then(actionThatFailsIntermittently) ) // retry
    .then(doSomethingWithDataReturnedByAction)
    .catch(logError);
```

Creating a Promise

```
1  const wait = milliseconds => {
2      return new Promise( (resolve, reject) => {
3          if (milliseconds >= 0) {
4              setTimeout(resolve, milliseconds);
5          } else {
6              reject( new RangeError(
7                  "Milliseconds must be positive.") );
8          }
9      });
10 }
```

```
1 actionThatFailsIntermittently()
2     .catch( e => wait(200).then(actionThatFailsIntermittently) ) // retry
3     .then(doSomethingWithDataReturnedByAction)
4     .catch(logError);
```

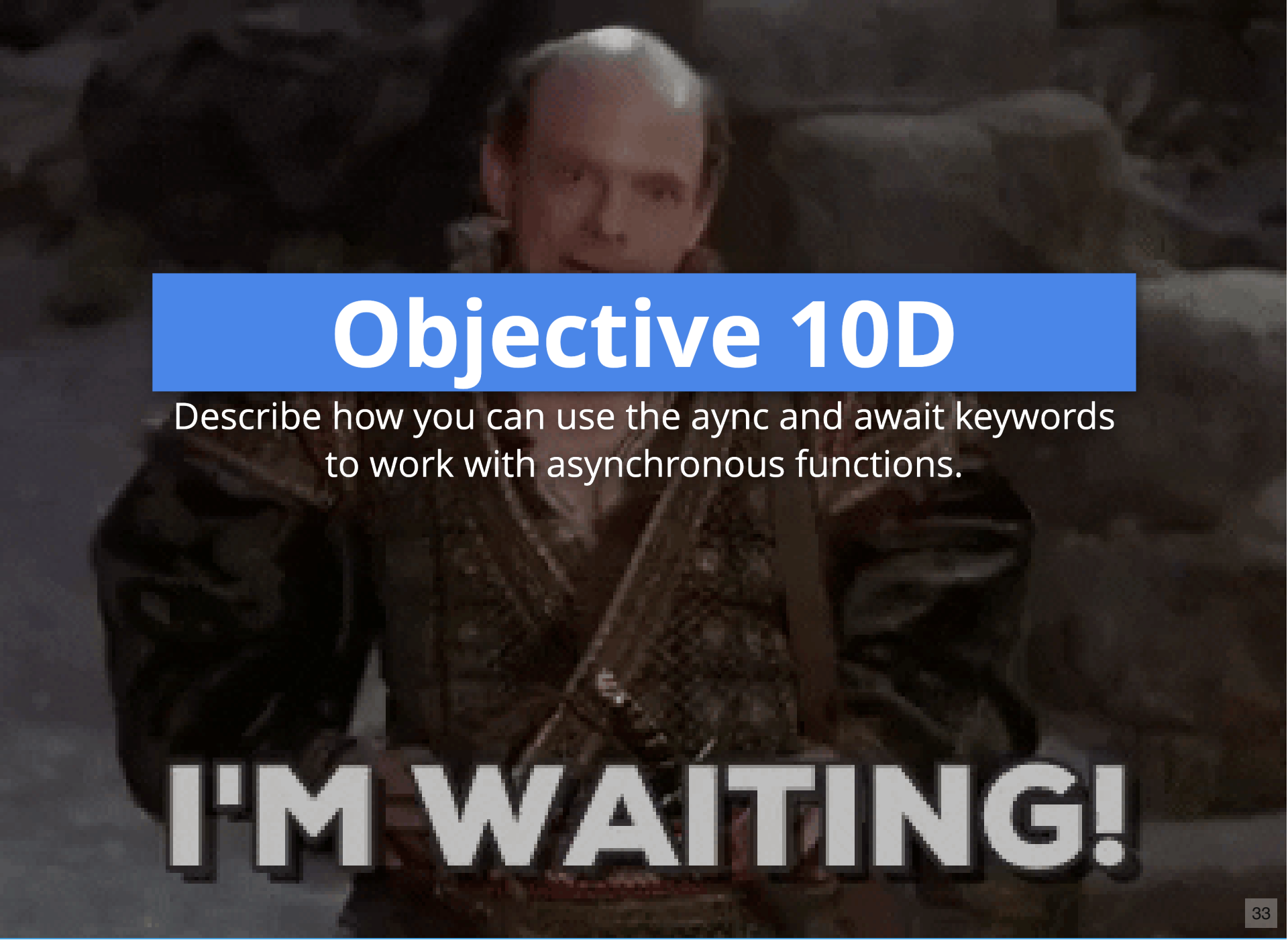
Creating a Promise

```
1  const wait = milliseconds => {
2      return new Promise( (resolve, reject) => {
3          if (milliseconds > 0) {
4              setTimeout(resolve, milliseconds);
5          } else {
6              reject( new RangeError(
7                  "Milliseconds must be positive.") );
8          }
9      });
10 }
```

```
1 actionThatFailsIntermittently()
2     .catch( e => wait(200).then(actionThatFailsIntermittently) ) // retry
3     .then(doSomethingWithDataReturnedByAction)
4     .catch(logError);
```

Objective 10D

Describe how you can use the aync and await keywords to work with asynchronous functions.



await for the Kitchen sync

- Most developers are used to working with synchronous code, not asynchronous code.
  - Code that works with promises *looks* different, so it can make it harder for some developers to understand.
  - We can use the *async* and *await* keywords to make asynchronous code look more synchronous, thus increasing the readibility.
- The *async* keyword declares an *asynchronous function* that wraps its return value in a Promise object.
- The *await* keyword tells JavaScript to wait until a promise is settled then return its result. It can only be used in asynchronous functions.

Comparisons . . .

```
"use strict";

const domain = "https://jsonplaceholder.typicode.com";

const getPhoto = id => {
    if (id < 1 || id > 5000) {
        return Promise.reject(
            new Error ("Photo ID must be between 1 and 5000."));
    } else {
        return fetch(`${domain}/photos/${id}`)
            .then( response => response.json() );
    }
};

const getPhotoAlbum = photo => {
    return fetch(`${domain}/albums/${photo.albumId}`)
        .then( response => response.json() )
        .then( album => {
            photo.album = album;
            return photo;
        });
};
const getPhotoAlbumUser = photo => {
    return fetch(`${domain}/users/${photo.album.userId}`)
        .then( response => response.json() )
        .then( user => {
            photo.album.user = user;
            return photo;
        });
};

const displayPhotoData = photo => {
    let html =
        `<img src="${photo.thumbnailUrl}" alt="${photo.title}">`;
    html    += `<h4>Title: ${photo.title}</h4>`;
    html    += `<p>Album: ${photo.album.title}</p>`;
    html    += `Posted by: ${photo.album.user.username}`;
    $("#photo").html(html);
};

const displayError = e => {
    let html = `<span>${e}</span>`;
    $("#photo").html(html);
};
$(document).ready( () => {
    $("#view_button").click( () => {
        const photo_id = $("#photo_id").val();
        getPhoto(photo_id)
            .then(getPhotoAlbum)
            .then(getPhotoAlbumUser)
            .then(displayPhotoData)
            .catch(displayError);
    });
});
```

```
"use strict";

const domain = "https://jsonplaceholder.typicode.com";

const getPhoto = async id => {
    if (id < 1 || id > 5000) {
        return Promise.reject(
            new Error ("Photo ID must be between 1 and 5000."));
    } else {
        const r1 = await fetch(`${domain}/photos/${id}`);
        const photo = await r1.json();

        const r2 = await fetch(`${domain}/albums/${photo.albumId}`);
        const album = await r2.json();
        photo.album = album;

        const r3 =  await fetch(`{$domain}/users/${photo.album.userId}`);
        const user = await r3.json();
        photo.album.user = user;

        return photo;
    }
};

const displayPhotoData = photo => {
        let html = `<img src="${photo.thumbnailUrl}" alt=${photo.title}">`;
        html += `<h4>Title: ${photo.title}</h4>`;
            html += `<p>Album:L ${photo.album.title}</p>`;
        html += `Posted by: ${photo.album.user.username}`;
    $('#photo').html(html);
}

const displayError = e => {
  let html = `<span>${e}</span>`;
  $('#photo').html(html);
}









$(document).ready( () => {
    $("#view_button").click( async () => {
        const photo_id = $("#photo_id").val();
        try {
            const photo = await getPhoto(photo_id);
            displayPhotoData(photo);
        }
        catch(e) {
            displayError(e);
        }
    });
});
```

aync/await

- async/await builds on top of Promises.
  - Remember, an *async* function always returns a *Promise*.

```
async function asyncFunc() {
  return "Hey!";
}

const returnedValue = asyncFunc();
console.log(returnedValue)                    //this is a Promise
```

aync/await

- The *await* keyword is used ***within*** an *async* block, otherwise it will throw an error.

```
1  "use strict"
2
3  const domain = "https://www.mccinfo.net/epsample/employees";
4
5  const asyncFunction = async ()=> {
6    const returnData = await fetch(domain);
7    const jsonData = await returnData.json();                    //convert returl to JSON
8    return jsonData;
9    /*
10   Using a promise:
11
12   return fetch(domain)
13         .then( response => response.json() );
14
15   */
16 }
17
18 $(document).ready( async ()=> {
19         //an await has to be in an async block
20         const asyncData = await asyncFunction();
21         console.log(asyncData);
22 });
```