Front-End Web Development

Unit 12: JavaScript – Advanced

Course Outline

1. Getting Started
2. HTML - Structuring the Web
3. CSS - Styling the Web
4. JavaScript - Dynamic client-side scripting
5. CSS - Making Layouts
6. Introduction to Websites/Web Applications
7. CSS – Advanced
8. Reviewing Progress
9. JavaScript - Modifying the Document Object Model (DOM)
10. Dynamic HTML
11. Web Forms - Working with user data
12. **JavaScript – Advanced**
13. Building a Web Application with JavaScript
14. Introduction to CSS Frameworks – Bootstrap
15. Other Frameworks, SEO, Web security, Performance
16. Walkthrough project

Course Learning Outcomes

- Competently write HTML and CSS code
- Create web page layouts according to requirements using styles
- Add interactivity to a web page with JavaScript
- Access and display third-party data on the web page
- Leverage Bootstrap and Static Site Generator

Assessment

- Final Project - 100% of the grade

  - Design and Build functioning Website using HTML5, CSS (including Bootstrap), JavaScript (browser only)

  ✓ Code will be managed in GitHub
  ✓ Website will be deployed to GitHub Pages
  ✓ All code to follow best practice and be documented

  - Details and How-To-Guide are available on the course page under the section called Assessments

12. JavaScript - Advanced

In This Unit

Title
Objects, Prototypes and Classes
Asynchronous Programming, Promises, Async and Await
JSON, Ajax and Fetch

Objects, Prototypes and Classes

Object Basics

- An object is a collection of related data and/or functionality
- Consists of several variables (properties) and functions (methods)
- Properties only

```
const person = {
  name: ["Bob", "Smith"],
  age: 32,
}
```

- Properties and methods

```
const person = {
  name: ["Bob", "Smith"], // an object property value can be another object
  age: 32,
  bio: function () {
    console.log(`${this.name[0]} ${this.name[1]} is ${this.age} years old.`);
  },
  introduceSelf() { // preferred syntax
    console.log(`Hi! I'm ${this.name[0]}.`);
  },
};
```

Object Basics

- Accessing the object using dot notation

```
// access a property
person.name;
person.name[0];
person.age;
// invoke a method
person.bio(); // "Bob Smith is 32 years old."
person.introduceSelf(); // "Hi! I'm Bob."
```

- An alternative way is using square bracket notation

```
// access a property
person['name'];
person['name'][0];
person['age'];
// invoke a method
person.bio(); // "Bob Smith is 32 years old."
person.introduceSelf(); // "Hi! I'm Bob."
```

- dot notation is preferred but there are situations where square bracket notation is used, e.g. if the property name is held in a variable

Object Basics

- Setting the value of a property

```
// access a property
person.name = "John";
person.age = 21;
```

- Extending an object - if the property doesn't already exist it's added to the object

```
// add a new property
person.address = "no 1, O'Connell Street, Dublin 1";
```

- Deleting a property - the property and it's value is deleted

```
// delete a property
delete person.name;
```

Object basics - "this"

- The `this` keyword refers to the current object the code is being written inside — so in this case `this` is equivalent to person

```
introduceSelf() { // preferred syntax
  console.log(`Hi! I'm ${this.name[0]}.`);
},
```

- Iterate thought an objects properties

```
for (const prop in person) {
  console.log(`${prop}: ${person[prop]}`);
}
```

```
// name: John
// age: 21
```

JavaScript Objects in the browser

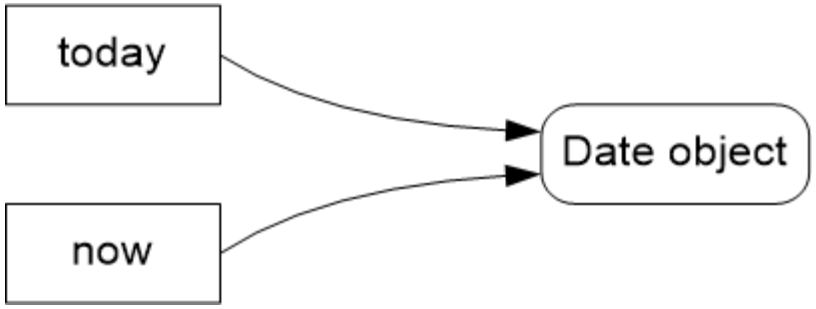
- JavaScript Objects/ Global Objects
  - Number, Boolean, String
  - Array
  - Date - manipulate dates and time
  - Math - various mathematics contants and functions, e.g. phi, random()
  - JSON - convert JSON formatted strings to and from JS Objects for exchange/storage purposes
  - Regular Expression - find patterns in strings
  - Promise, Fetch, localStorage
- Browser Objects
  - Document - access to the Document Object Model(DOM) properties and methods
  - Window - access to key browser properties and methods
  - Console - display messages in the console
- User-defined Objects

Object Creation Syntax

```
let variableName = new <ObjectType>(arguments);

let invoice = new Object();   // with new keyword
let invoice = {}; // with braces

let today = new Date(); // new Date object
let lastName = "John"; // = new String("John");
let taxRate = 0.0875 // = new Number(0.0875);
let validFlag = true; // = new Boolean(true);
```

Object References

```
// two variables that refer to the same object
let today = new Date()
let now = today;
```




```
let today = new Date()
let now = today;
delete today;
console.log(today); // displays undefined
console.log(now.getFullYear); // displays the year
delete now;
console.log(now.getFullYear); // displays undefined
```

Constructors

Previous examples don't scale if you want to create more than one object of the same type as you need to duplicate the code, etc...

Two approaches available:

- Factory Design Pattern
- Constructor

Factory Design Pattern

- use a function to create a new version of the object

```
function createPerson(name) {
  const obj = {};
  obj.name = name;
  obj.introduceSelf = function () {
    console.log(`Hi! I'm ${this.name}.`);
  };
  return obj;
}
```

```
const salva = createPerson("Salva");
salva.introduceSelf();
// "Hi! I'm Salva."

const frankie = createPerson("Frankie");
frankie.introduceSelf();
// "Hi! I'm Frankie."
```

- https://en.wikipedia.org/wiki/Software_design_pattern

Constructors

A constructor is just a function called using the new keyword. When you call a constructor, it will:

- create a new object
- bind this to the new object, so you can refer to this in your constructor code
- run the code in the constructor
- return the new object

Constructors, by convention, start with a capital letter and are named for the type of object they create. So we could rewrite our example like this:

```
function Person(name) {
  this.name = name;
  this.introduceSelf = function () {
    console.log(`Hi! I'm ${this.name}.`);
  };
}
```

Constructors

To call Person() as a constructor, we use `new`:

```
const salva = new Person("Salva");
salva.introduceSelf();
// "Hi! I'm Salva."

const frankie = new Person("Frankie");
frankie.introduceSelf();
// "Hi! I'm Frankie."
```

Other examples of objects you've used on this course so far include:

```
myString.split(','); // split is a method of the String object
document.getElementById('myID'); // getElementById is a method of the document object
```

Object Prototypes

- Prototypes are the mechanism by which JavaScript objects inherit features from one another
- Every object in JavaScript has a built-in property, which is called its prototype, typically called `__proto__`
- The prototype is itself an object, so the prototype will have its own prototype, making what's called a prototype chain
- The chain ends when we reach a prototype that has null for its own prototype
- The standard way to access an object's prototype is the `Object.getPrototypeOf()` method

```
const myObject = {
  city: "Madrid",
  greet() {
    console.log(`Greetings from ${this.city}`);
  },
};

myObject.greet(); // Greetings from Madrid

// toString() method is not implemented in myObject but is "inherited" from Object
myObject.toString(); // "[object Object]"
```

Prototype and Inheritance

- Prototypes are a powerful and very flexible feature of JavaScript, making it possible to reuse code and combine objects
- They support a version of inheritance. Inheritance is a feature of object-oriented programming languages that lets programmers express the idea that some objects in a system are more specialized versions of other objects
- For example, if we're modelling a school, we might have professors and students: they are both people, so have some features in common (for example, they both have names), but each might add extra features (for example, professors have a subject that they teach), or might implement the same feature in different ways
- In an OOP system we might say that professors and students both inherit from people
- You can see how in JavaScript, if Professor and Student objects can have Person prototypes, then they can inherit the common properties, while adding and redefining those properties which need to differ

Object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm fundamental to many programming languages, including Java, Python and C++
- Main concepts:
  - classes and instances
  - inheritance, polymorphism
  - encapsulation
  - abstraction

Classes and instances

- When we model a problem in terms of objects in OOP, we create abstract definitions or templates representing the types of objects we want to have in our system
- For example if we are modelling a university, we might have Professors
- So `Professor` could be a class in our system. The definition of the class lists the data and methods that every professor has
- In pseudocode, a `Professor` class could be written like this:

```
class Professor
    properties
        name
        teaches
    constructor
        Professor(name, teaches)
    methods
        grade(paper)
        introduceSelf()
```

Classes and instances

- Now we can create some professors using the `new` and invoke their methods

```
const walsh = new Professor("Walsh", "Psychology");
const lillian = new Professor("Lillian", "Poetry");

walsh.teaches; // 'Psychology'
walsh.introduceSelf(); // 'My name is Professor Walsh and I will be your Psychology professor.'

lillian.teaches; // 'Poetry'
lillian.introduceSelf(); // 'My name is Professor Lillian and I will be your Poetry professor.'
```

Inheritance

- Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support

```
class Person
    properties
        name
    constructor
        Person(name)
    methods
        introduceSelf()

class Professor : extends Person
    properties
         teaches
     constructor
        Professor(name, teaches)
    methods
        grade(paper)
        introduceSelf()

class Student : extends Person
    properties
        year
    constructor
        Student(name, year)
    methods
        introduceSelf()
```

Inheritance, Polymorphism

- `Person` is the **superclass** or parent class of both `Professor` and `Student`
- Conversely, `Professor` and `Student` are **subclasses** or child classes of `Person`
- also `introduceSelf()` is defined in all three classes. The reason for this is that while all people want to introduce themselves, the way they do so is different
- `Polymorphism` means one name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality and arguments
- When a method in a **subclass** replaces the superclass's implementation, we say that the subclass overrides the version in the **superclass**

Encapsulation

- Encapsulation means hiding the internal representation of an object from view outside of the object's definition
- Only the object's own methods can directly inspect or manipulate its properties/fields
- Encapsulation is the hiding of data implementation by restricting access to accessors and mutators
- A benefit of encapsulation is that it can reduce system complexity
- In many OOP languages, we can prevent other code from accessing an object's internal state by marking some properties as private. This will generate an error if code outside the object tries to access them

```
class Student : extends Person
    properties
      private year
    constructor
        Student(name, year)
    methods
      introduceSelf()
      canStudyArchery() { return this.year > 1 }

student = new Student('Weber', 1)
student.year // error: 'year' is a private property of Student
```

Abstraction

- Data abstraction and encapuslation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details
- Abstraction denotes a model, a view, or some other focused representation for an actual item

> “An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.” — G. Booch

- In short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing

JavaScript - Classes and constructors

- You can declare a class using the class keyword:

```
class Person {
  name; // properties can also be initialised, e.g. name = '';

  constructor(name) {
    this.name = name;
  }

  introduceSelf() {
    console.log(`Hi! I'm ${this.name}`);
  }
}

const giles = new Person("Giles"); // create a new Person object

giles.introduceSelf(); // Hi! I'm Giles
```

- If you don't need to do any special initialization, you can omit the constructor, and a default constructor will be generated for you

JavaScript - Inheritance

- Given our Person class above, let's define the Professor subclass

```
class Professor extends Person {
  teaches;

  constructor(name, teaches) {
    super(name); // super() calls the superclass (Person) contructor. Is alsways called first
    this.teaches = teaches;
  }

  introduceSelf() {
    console.log(
      `My name is ${this.name}, and I will be your ${this.teaches} professor.`,
    );
  }

  grade(paper) {
    const grade = Math.floor(Math.random() * (5 - 1) + 1);
    console.log(grade);
  }
}
```

JavaScript - Inheritance

- With this declaration we can now create and use professors:

```
const walsh = new Professor("Walsh", "Psychology");
walsh.introduceSelf(); // 'My name is Walsh, and I will be your Psychology professor'

walsh.grade("my paper"); // some random grade
```

JavaScript - Encapsulation

- **Private** properties and methods

```
class Student extends Person {
  #year; // private property

  constructor(name, year) {
    super(name);
    this.#year = year;
  }

  introduceSelf() {
    console.log(`Hi! I'm ${this.name}, and I'm in year ${this.#year}.`);
  }

  #canStudyArchery() { // private method
    return this.#year > 1;
  }
}

const summers = new Student("Summers", 2);

summers.introduceSelf(); // Hi! I'm Summers, and I'm in year 2.
summers.canStudyArchery(); // true
summers.#year; // SyntaxError. Code run in the Chrome console can access private properties outside the class
```

Static methods and static properties

The `static` keyword is used to indicate that the method or property belongs to the Class itself and not any instance of the Class, e.g. `Data.now()`

```
class Colour {
  static defaultColour = 'Red';
  static isValid(r, g, b) {
    return r >= 0 && r <= 255 && g >= 0 && g <= 255 && b >= 0 && b <= 255;
  }
}

Colour.isValid(255, 0, 0); // true
Colour.isValid(1000, 0, 0); // false

console.log(Colour.defaultColour);  //  Red
console.log(new Colour().defaultColour); // undefined (not accessible from an instance)
```

Static initialization block

A static block is a block of code that runs when the class is first loaded

```
class MyClass {
  static {
    MyClass.myStaticProperty = "foo";
  }
}

console.log(MyClass.myStaticProperty); // 'foo'
```

Static initialization blocks are almost equivalent to immediately executing some code after a class has been declared

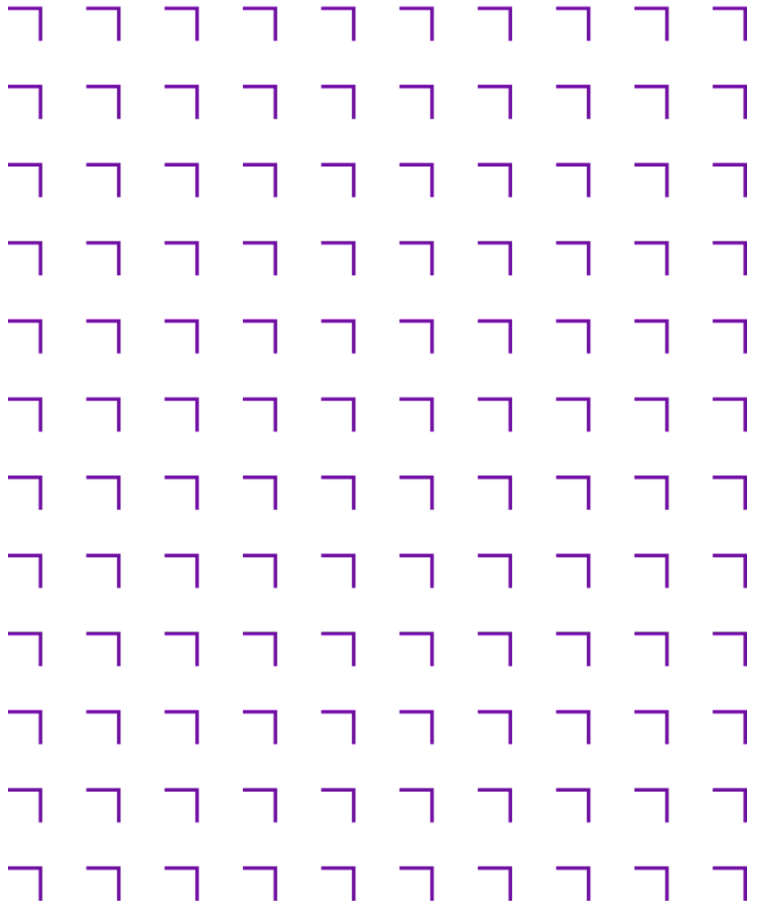
The only difference is that they have access to static private properties

Activity

Breakout

- Join a breakout room
- Download the unit 12 exercises
- Follow the instructions and complete the exercises
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

Asynchronous Programming, Promises, Async and Await



Asynchronous Programming

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result

The reason for this is that JavaScript is single-threaded. A thread is a sequence of instructions that a program follows

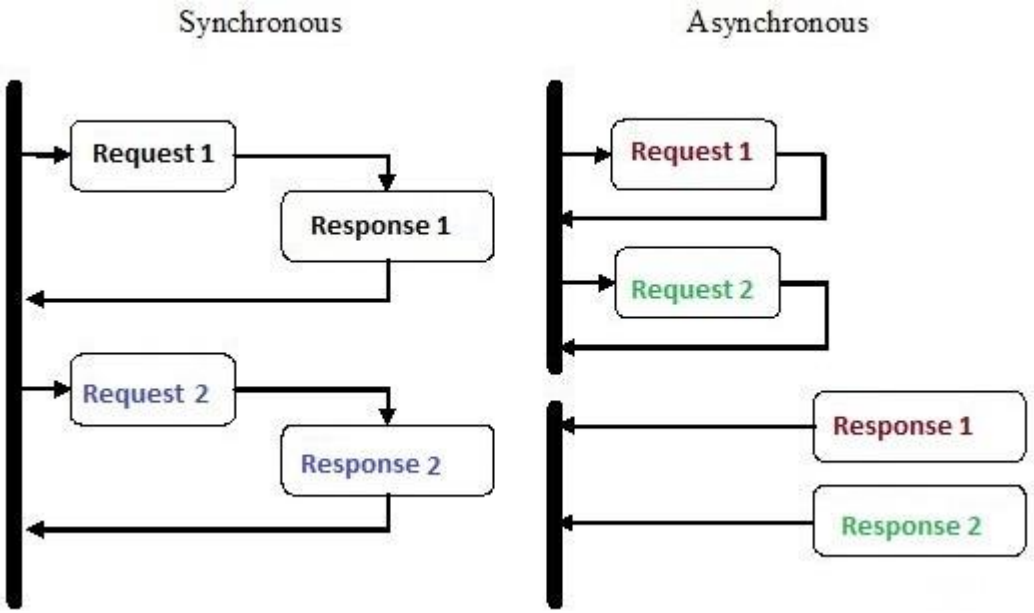
Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else

Many functions provided by browsers, especially the most interesting ones, can potentially take a long time, and therefore, are asynchronous. For example:

- Making HTTP requests using `fetch()`
- Accessing a user's camera or microphone using `getUserMedia()`
- Asking a user to select files using `showOpenFilePicker()`

Note: Web Workers enable JavaScript to utilize multithreading, allowing for concurrent execution of tasks without blocking the main thread

Event Handlers



- Synchronous: first request starts and then waits for the response/completion. Only then does the second request start and again it waits for the response/completion, i.e. the second request is blocked until the first request completes
- Asynchronous: first request starts followed by the second request. The responses are then received as they complete, i.e. the requests are non-blocking but you may be waiting for the response

Asynchronous Programming

There are different ways to implement Asynchronous in JavaScript:

- Callback function
- Promise
- Async & Await
- Generators (outside the scope of this course)

Callback function

A Callback function is a function that returns or uses another function as a parameter

JavaScript

```
console.log("first line");
setTimeout(function(){
  console.log("second line");
},5000);

console.log("third line");
setTimeout(function(){
  console.log("fourth line");
},4000);
console.log("fifth line");
```

Console

```
"first line"
"third line"
"fifth line"
"fourth line"
"second line"
```

Notice that the console messages are out of sysnc

While callbacks work fine for simple cases, they have major issues when callbacks are nested, i.e. a callback calls a callback which calls another callback, etc...

`Promises` was introduced in ES6(2015) to address these shortcomings

Promises and Fetch

`Promises` are the foundation of asynchronous programming in modern JavaScript. A promise is an object returned by an asynchronous function, which represents the current state of the operation

At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation

With a promise-based API, the asynchronous function starts the operation and returns a promise object

You can then attach handlers to this promise object, and these handlers will be executed when the operation has succeeded or failed

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
console.log(fetchPromise);

fetchPromise.then((response)  =>  {
  console.log(`Received response: ${response.status}`);
});
console.log("Started request…");
```

Promises

Explanation of the code:

1. calling the `fetch()` API, and assigning the return value to the `fetchPromise` variable
2. immediately after, logging the `fetchPromise` variable. This should output something like: `Promise { <state>: "pending" }`, telling us that we have a Promise object, and it has a state whose value is "pending". The "pending" state means that the fetch operation is still going on
3. passing a handler function into the Promise's `then()` method. When (and if) the fetch operation succeeds, the promise will call our handler, passing in a `Response` object, which contains the server's response
4. logging a message that we have started the request

```
// output to console
Promise {<pending>}
ajax-fetch.html:28 Started request…
ajax-fetch.html:25 baked beans
```

Chaining Promises

With the `fetch()` API, once you get a Response object, you need to call another function to get the response data

In this case, we want to get the response data as JSON, so we would call the `json()` method of the Response object. It turns out that `json()` is also asynchronous. This is called **promise chaining**

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise.then((response) => { // 1
  const jsonPromise = response.json();
  jsonPromise.then((data) => {
    console.log(data[0].name);
  });
});

fetchPromise // preferred way to chain promises. This replaces code above at 1
  .then((response) => response.json())
  .then((data) => {
    console.log(data[0].name);
});
```

Checking the response status

We need to check that the server accepted and was able to handle the request, before we try to read it. We'll do this by checking the status code in the response and throwing an error if it wasn't "OK":

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((data) => {
    console.log(data[0].name);
  });
```

Catching Errors

To support error handling, Promise objects provide a `catch()` method. This is a lot like `then()`: you call it and pass in a handler function

However, while the handler passed to `then()` is called when the asynchronous operation succeeds, the handler passed to `catch()` is called when the asynchronous operation fails

If you add `catch()` to the end of a promise chain, then it will be called when any of the asynchronous function calls fail. So you can implement an operation as several consecutive asynchronous function calls, and have a single place to handle all errors

```
const fetchPromise = fetch(
  "bad-scheme://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) { throw new Error(`HTTP error: ${response.status}`); }
    return response.json();
  })
  .then((data) => { console.log(data[0].name); })
  .catch((error) => { console.error(`Could not get products: ${error}`);});
```

Promise terminology

A promise can be in one of three states:

- **pending**: the promise has been created, and the asynchronous function it's associated with has not succeeded or failed yet. This is the state your promise is in when it's returned from a call to fetch(), and the request is still being made
- **fulfilled**: the asynchronous function has succeeded. When a promise is fulfilled, its then() handler is called
- **rejected**: the asynchronous function has failed. When a promise is rejected, its catch() handler is called

Note that what "succeeded" or "failed" means here is up to the API in question. For example, fetch() rejects the returned promise if (among other reasons) a network error prevented the request being sent, but fulfills the promise if the server sent a response, even if the response was an error like 404 Not Found

Sometimes, the term **settled** is used to cover both **fulfilled** and **rejected**

A promise is **resolved** if it is settled, or if it has been "locked in" to follow the state of another promise

See Let's talk about how to talk about promises and Promise API on MDN for more details

async and await

The `async` keyword gives you a simpler way to work with asynchronous promise-based code

Adding `async` at the start of a function makes it an async function:

```
async function myFunction() {
  // This is an async function
}
```

Inside an async function, you can use the `await` keyword before a call to a function that returns a promise

This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown

Using aysnc and await with fetch()

Rewriting the `fetch` code to use asynchronous functions but which looks like synchronous code:

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()` call to be settled
    // the `fetch()` call will either return a Response or throw an error
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // after this line, our function will wait for the `response.json()` call to be settled
    // the `response.json()` call will either return the parsed JSON object or throw an error
    const data = await response.json();
    console.log(data[0].name);
  } catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

Using aysnc and await with fetch()

If you want to return the data object from the fetchProducts() function, remember that it's a `Promise` and needs to be resolved:

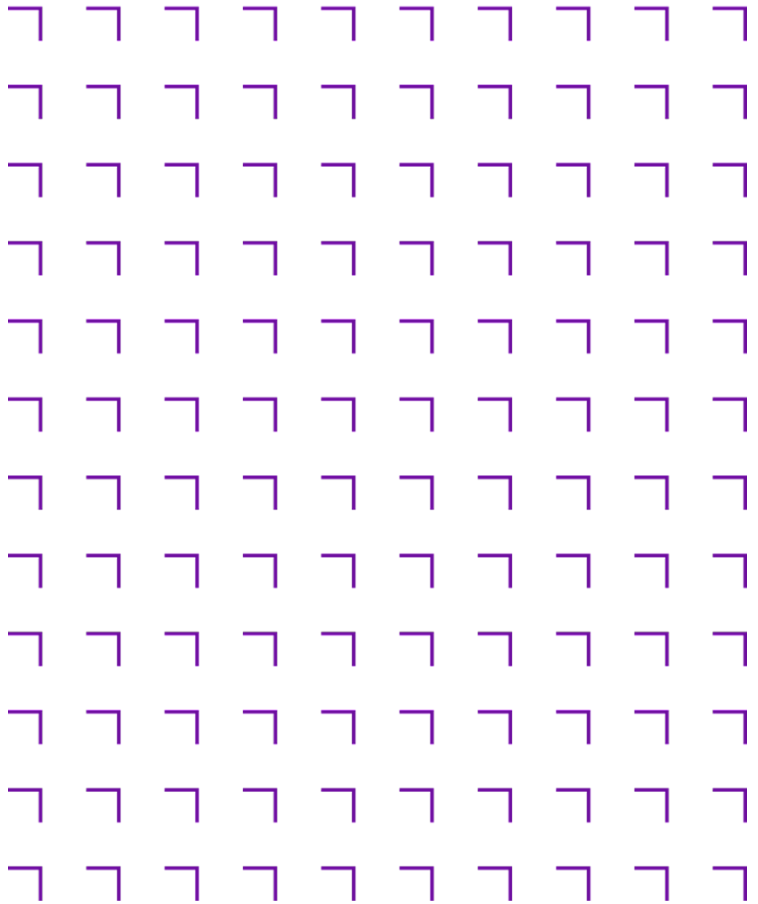
```
async function fetchProducts() {
  const response = await fetch(
    "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
  );
  if (!response.ok) {
    throw new Error(`HTTP error: ${response.status}`);
  }
  const data = await response.json();
  return data;
}

const promise = fetchProducts(); // returns a promise
promise
  .then((data) => {
    console.log(data[0].name);
  })
  .catch(() => {
    console.error(`Could not get products: ${error}`);
  });
```

Activity

Breakout

- Join a breakout room
- Continue working on the unit 12 exercises
- You have 35 minutes
- Lecturer will visit each room in turn, etc...
- Will start next topic on the hour

JSON, AJAX and Fetch



JSON

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax

It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa)

Converting a string to a native object is called deserialization, while converting a native object to a string so it can be transmitted across the network is called serialization

A JSON string can be stored in its own file, which is basically just a text file with an extension of .json, and a MIME type of application/json

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

JSON

- JSON is purely a string with a specified data format — it contains only properties, no methods
- JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string
- Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as long as the generator program is working correctly). You can validate JSON using an application like JSONLint
- JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON
- Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties

JSON Structure

You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals

This allows you to construct a data hierarchy, e.g:

```
const superHeroes = `{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
    },
    ....
  ]
}`;
```

Demo

```
// convert from JSON string to JS object
const superHeroesObj = JSON.parse(superHeroes);
console.log(superHeroesObj.squadName); // Super hero squad

// convert JS object to JSON string
const jsObject = {
  name: "Eternal Flame",
  age: 10000,
  secretIdentity: "Unknown",
  powers: ["Immortality", "Heat Immunity", "Inferno", "Teleportation", "Interdimensional travel"]
}
console.log(jsObject);
const jsObjectJsonString = JSON.stringify(jsObject);
console.log(jsObjectJsonString);
```

Fetching data from the server – AJAX and Fetch

A common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new pag

This seemingly small detail has had a huge impact on the performance and behavior of sites as they have become more data-driven

Background

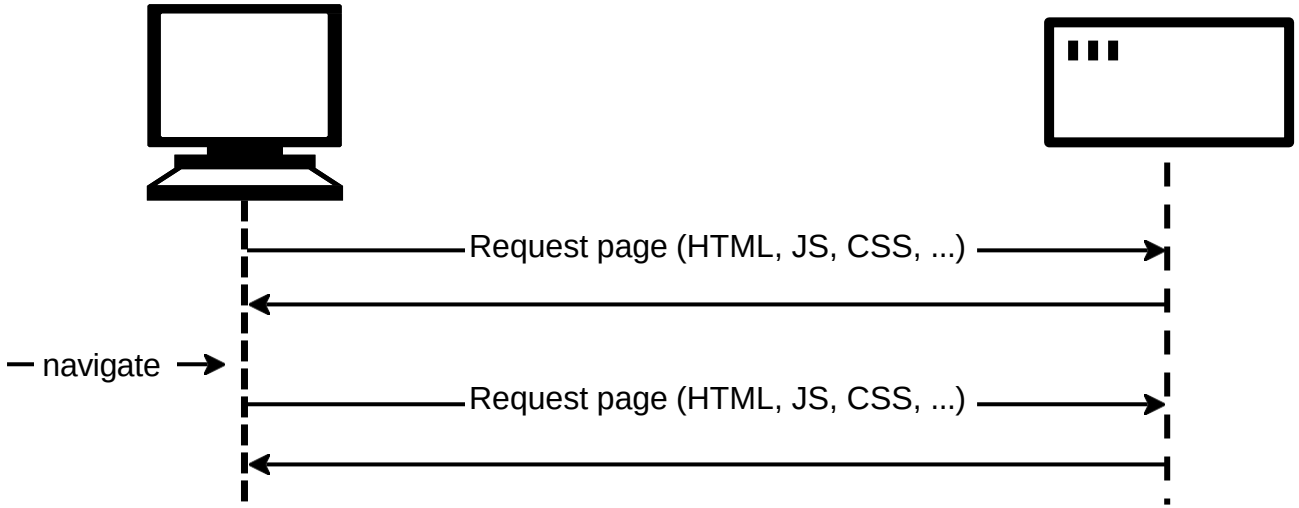
In 1999, Microsoft introduced the `XMLHttpRequest` object, an ActiveX control, in IE5

The term **AJAX** was coined on February 18, 2005, by Jesse James Garret in a short essay published a few days after Google released its Maps application

AJAX is a methodology using several web technologies together (HTML, XMLHttpRequest, DOM), in an effort to close the gap between the usability and interactivity of a desktop application and the ever demanding web application

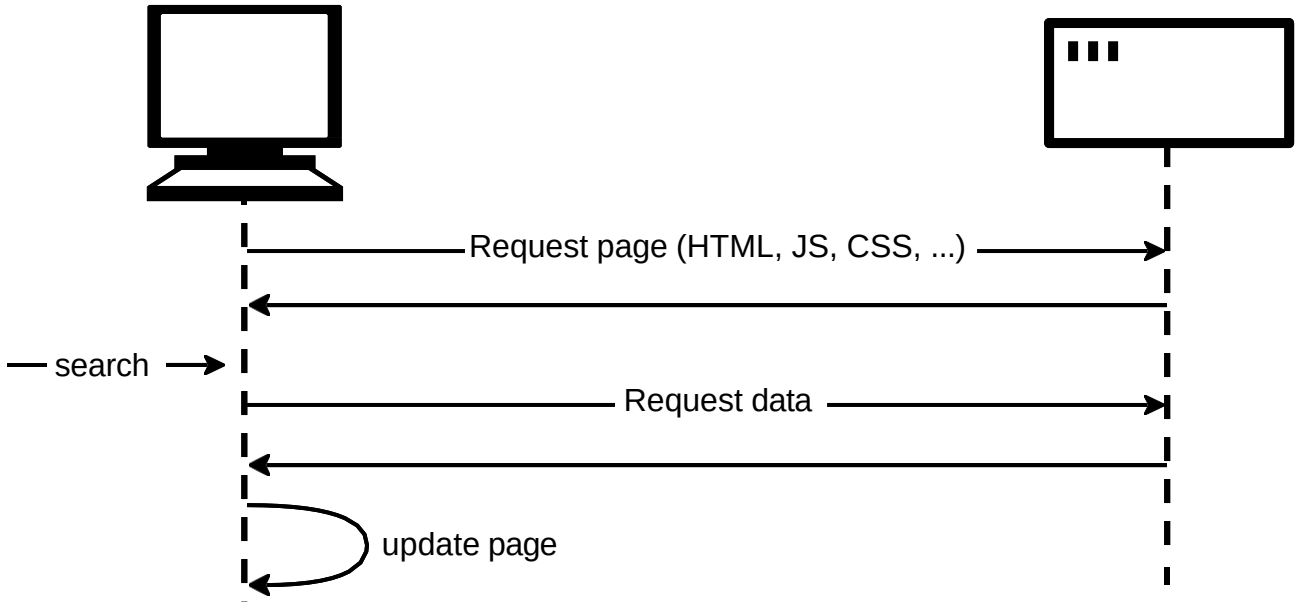
Web Page Loading - Basic Model

- A web page consists of a HTML page and (usually) various other files, such as stylesheets, scripts, and images
- The basic model of page loading on the Web is that your browser makes one or more HTTP requests to the server for the files needed to display the page, and the server responds with the requested files
- If you visit another page, the browser requests the new files, and the server responds resulting in the new page loading
- Reloading the entire page, even when we only need to update one part of it, is inefficient and can result in a poor user experience



Web Page Loading - Data-Driven Model

- So instead of the traditional model, many websites use JavaScript APIs (`XMLHttpRequest` , `Fetch`) to request data from the server and update the page content without a page load
- So when the user searches for a new product, the browser only requests the data which is needed to update the page — the set of new books to display, for instance



The Fetch API

- The `Fetch` API. This enables JavaScript running in a page to make an HTTP request to a server to retrieve specific resources
- When the server provides them, the JavaScript can use the data to update the page, typically by using DOM manipulation APIs
- The data requested is often `JSON`, which is a good format for transferring structured data, but can also be HTML or just text
- This is a common pattern for data-driven sites such as Amazon, YouTube, eBay, and so on. With this model:
  - Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive
  - Less data is downloaded on each update, meaning less wasted bandwidth. This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in countries that don't have ubiquitous fast internet service

Fetch Demo

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
console.log(fetchPromise);

fetchPromise.then((response) => {
  console.log(`Received response: ${response.status}`);
});
console.log("Started request…");
```

Activity

Breakout

- Join a breakout room
- Continue working on the exercises
- You have 35 minutes
- Lecturer will visit each room in turn, etc...

Summary

Completed this Week

- Objects, Prototypes and Classes
- Asynchronous Programming, Promises, Async and Await
- JSON, AJAX and Fetch

For Next Week

- Complete the remaining exercises for unit 12 before next class
- Review the slides and examples for unit 13