6.8 JavaScript object basics

Learning outcomes:

- Understand that in JavaScript most things are objects, and you've probably used objects every time you've touched JavaScript.
- Basic syntax:
  - Object literals.
  - Properties and methods.
  - Nesting objects and arrays in objects.
- Using constructors to create a new object.
- Object scope, and `this`.
- Accessing properties and methods — bracket and dot syntax.
- Object destructuring.

Resources:

- JavaScript object basics
- Object destructuring assignment

6.9 DOM scripting

Learning outcomes:

- Understand what the DOM is — the browser's internal representation of the document's HTML structure as a hierarchy of objects, which can be manipulated using JavaScript.
- Understand the important parts of a web browser and how they are represented in JavaScript — `Navigator`, `Window`, and `Document`.
- Understand how DOM nodes exist relative to each other in the DOM tree — root, parent, child, sibling, and descendant.
- Getting references to DOM nodes, for example with `querySelector()` and `getElementById()`.
- Creating new nodes, for example with `innerHTML()` and `createElement()`.
- Adding and removing nodes to the DOM with `appendChild()` and `removeChild()`.
- Adding attributes with `setAttribute()`.
- Manipulating styles with `Element.style.*` and `Element.classList.*`.

Resources:

- Manipulating documents
- DOM Scripting, explainers.dev

6.10 Events

Learning outcomes:

- Understand what events are — a signal fired by the browser when something significant happens, which the developer can run some code in response to.
- Event handlers:
  - `addEventListener()` and `removeEventListener()`
  - Event handler properties.
  - Inline event handler attributes, and why you shouldn't use them.
- Event objects.
- Preventing default behavior with `preventDefault()`.
- Event delegation.

Resources:

Introduction to events

6.11 Async JavaScript basics

Learning outcomes:

- Understand the concept of asynchronous JavaScript — what it is and how it differs from synchronous JavaScript.
- Understand that callbacks and events have historically provided the means to do asynchronous programming in JavaScript.
- Modern asynchronous programming with async functions and `await`:
  - Basic usage.
  - Understanding async function return values.
  - Error handling with `try ... catch`.
- Promises:
  - Understand that `async` / `await` use promises under the hood; they provide a simpler abstraction.
  - Chaining promises.
  - Catching errors with `catch()`.

Resources:

Asynchronous JavaScript

6.12 Network requests with fetch()

Learning outcomes:

- Understand that `fetch()` is used for asynchronous network requests, which is by far the most common asynchronous JavaScript use case on the web.
- Common types of resources that are fetched from the network:
  - Text content, JSON, media assets, etc.

    - Data from RESTful APIs. Learn the basic concepts behind REST, including common patterns such as CRUD.
- Understand what single-page apps (SPAs) are, and the issues surrounding them:
    - Accessibility issues behind asynchronous updates, for example, content updates not being announced by screen readers by default.
    - Usability issues behind asynchronous updates, like loss of history and breaking the back button.
- Understand HTTP basics. You should look at common HTTP methods such as `GET`, `DELETE`, `POST`, and `PUT`, and how they are handled via `fetch()`.

Resources:

- Fetching data from the server

6.13 Working with JSON

Learning outcomes:

- Understand what JSON is — a very commonly used data format based on JavaScript object syntax.
- Understand that JSON can also contain arrays.
- Retrieve JSON as a JavaScript object using mechanisms available in Web APIs (e.g. `Response.json()` in the Fetch API).
- Access values inside JSON data using bracket and dot syntax.
- Converting between objects and text using `JSON.parse()` and `JSON.stringify()`.

Resources:

- Working with JSON
- JSON Review, Scrimba COURSE PARTNER

6.14 Libraries and frameworks

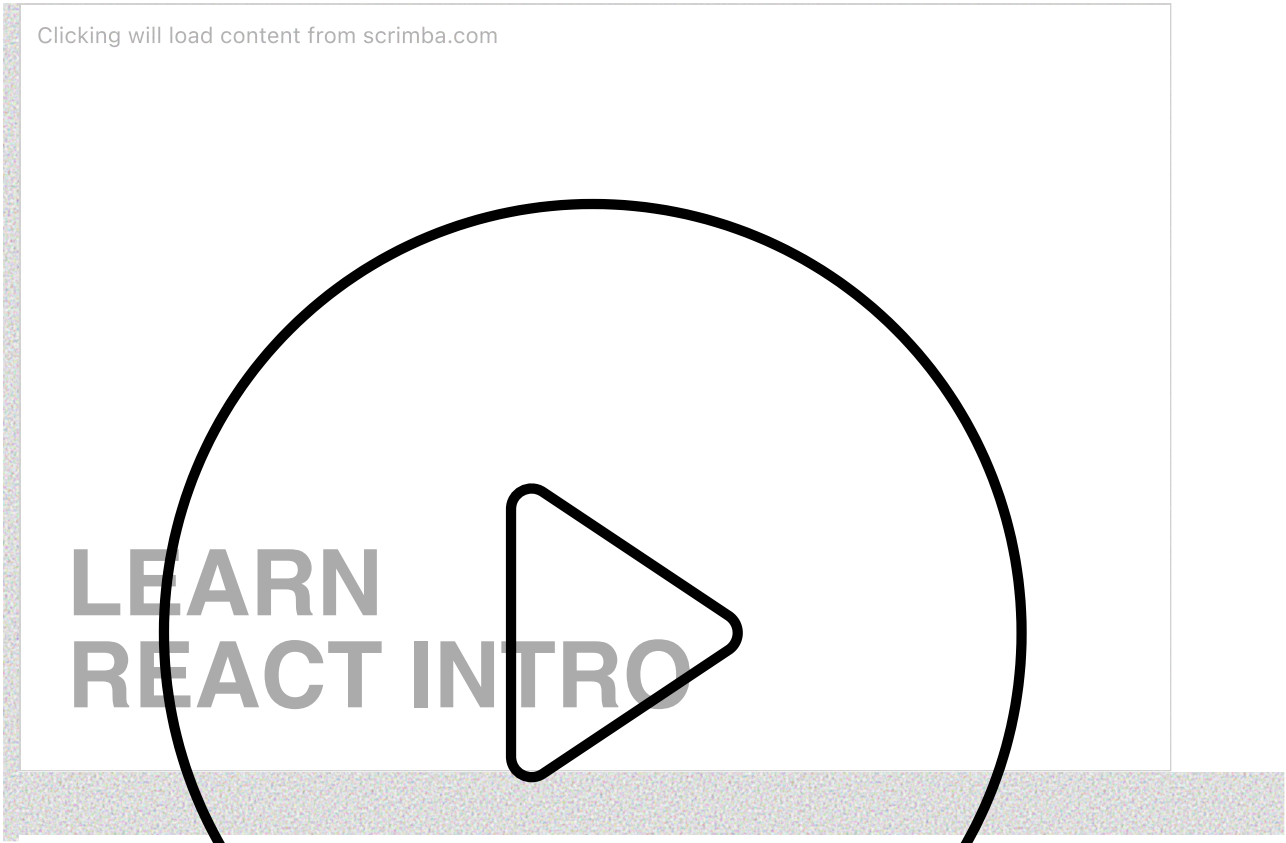
Learning outcomes:

- Understand what third-party code is — functionality written by someone else that you can use in your own project, so you don't have to write everything yourself.
- Why developers use third-party code:
  - Efficiency and productivity: A huge amount of complex functionality is already written for you to use, created in a way that enforces efficient, modular code organization.
  - Compatibility: Reputable framework code is already optimized to work across browsers/devices, for performance, etc. Many frameworks also have systems to output to specific platforms (e.g. Android or iOS) as build targets.
  - Support/ecosystem: Popular frameworks have vibrant communities and help resources to provide support, and rich systems of extensions/plugins to add functionality.
- The difference between libraries and frameworks:
  - A library tends to be a single code component that offers a solution to a specific problem, which you can integrate into your own app (for example, chart.js for creating `<canvas>` -based charts, or three.js for simplified 3D GPU-based graphics rendering), whereas a framework tends to be a more expansive architecture made up of multiple components for building complete applications.
  - A library tends to be unopinionated about how you work with it in your codebase, whereas a framework tends to enforce a specific coding style and control flow.
- Why should you use frameworks?
  - They can provide a lot of functionality and save you a lot of time.
  - A lot of companies use popular frameworks such as React or Angular to write their applications, therefore a lot of jobs list frameworks as requirements for applicants to have.
- Why is a framework not always the right choice? A framework:

    - Can easily be overkill for a small project — you might be better off writing a few lines of vanilla JavaScript to solve the problem or using a tailored library.
    - Usually adds a lot of JavaScript to the initial download of your application, leading to an initial performance hit and possible usability issues.
    - Usually comes with its own set of custom syntax and conventions, which can introduce a significant additional learning curve to the project.
    - May be incompatible with an existing codebase because of its architecture choice.
    - Will need to be updated regularly, possibly leading to extra maintenance overhead for your application.
    - May introduce significant accessibility issues for people using assistive technologies because of its architecture (for example, SPA-style client-side routing), which will need to be considered carefully.
- How to choose? A good library or framework must:
    - Solve your problems while offering advantages that significantly outweigh any negatives that it brings to the table.
    - Have good support and a friendly community.
    - Be actively maintained — don't choose a codebase that has not been updated for over a year, or has no users.

Resources:

- Introduction to client-side frameworks
- Introduction to React, Scrimba COURSE PARTNER



6.15 Debugging JavaScript

Learning outcomes:

- Understand the different types of JavaScript errors, for example, syntax errors and logic errors.
- Learn about the common types of JavaScript error messages and what they mean.
- Using browser developer tools to inspect the JavaScript running on your page and see what errors it is generating.
- Using `console.log()` and `console.error()` for simple debugging.
- Error handling:
  - Using conditionals to avoid errors.
  - `try ... catch`.
  - `throw`.
- Advanced JavaScript debugging with breakpoints, watchers, etc.

Resources:

- What went wrong? Troubleshooting JavaScript
- Control flow and error handling > Exception handling statements
- The Firefox JavaScript debugger, Firefox Source Docs
- Chrome > Console overview, developer.chrome.com (2019)
- Chrome > Debug JavaScript, developer.chrome.com (2017)