# PART 1: Structuring JavaScript

DWA\_03.1 Core Concepts Part 1 Another method of managing the complexity of modern software is diligent code documentation. In this module, we will explore how you can document your code to minimise the learning curve for new team members and act as a reference for yourself while working on the code itself.  
   
Core Concepts: Part 1  
  
   
00:00:00 Introduction  
00:00:50 Introducing "Behaviour Driven Development" (BDD)  
00:01:40 Documentation-related Tooling  
00:02:00 Overview of "user stories".  
00:02:50 How is work given to developers?  
00:03:40 User stories provide the problem, not the solution  
00:06:20 We don't always work in ideal situations  
00:07:30 Internal team process is also part of the complexity  
00:08:00 Example of a common "user story" template  
00:09:20 Understanding how to read user stories  
00:09:50 You will be the one solving user stories  
00:10:40 What is the ".md" file extension?  
00:11:10 Explaining "README.md" convention  
00:11:50 Examples of real-world README files  
00:13:30 Projects that have huge README files  
00:14:30 Repositories that only have a README file  
00:16:30 Overview of Markdown computer language  
00:17:40 How and why was Markdown created?  
00:18:30 Example of plain text documentation file  
00:19:20 Using AI in helping to create documentation  
00:21:00 Creating plain text documentation  
00:23:00 Converting plain text to Markdown  
00:23:30 Controlling heading levels with hashes (#)  
00:24:00 Including hyperlinks in markdown files  
00:24:30 Adding code snippets in markdown  
00:25:00 Installing the "Markdown All in One" plugin  
00:25:30 Adding auto-updating Table of Contents  
00:26:20 Preview of Markdown inside VS Code  
00:26:50 Introducing Unified Modeling Language (UML)  
00:27:30 Overview of Mermaid Diagrams  
00:28:40 Starting with super-rough sketches first  
00:29:20 Thinking visually through interactions  
00:30:40 Properly mapping software takes a long time to learn  
00:31:30 You will need a lot of support and input early on  
00:32:20 Completing your personal sketching process  
00:33:40 Sketches are just for you, and they are not documentation  
00:34:20 Create formal documentation only after sketches  
00:35:00 Thinking through remaining app interactions  
   
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_13.2 Utility Functions    
  
   
00:00:00 Introduction  
00:00:50 Add "Shoelace" library  
00:01:40 Add "sl-button" component  
00:02:10 Creating basic CSS styles  
00:02:50 Add "sl-dialog" component  
00:03:20 "lit-html" VS Code plugin  
00:04:50 Using "Shoelace" with HTML forms  
00:06:20 Embed buttons inside the dialog  
00:06:50 Add "sl-input" component  
00:07:40 Create custom "td-spacing"  
00:08:50 Component HTML auto-completion  
00:11:10 "lit-plugin" VS Code plugin  
00:12:10 Create "td-spacing" CSS styling  
00:14:00 What is ":host" pseudo-selector?  
00:13:50 Create CSS styles dynamically  
00:14:30 Define different values to combine  
00:15:15 ".map" higher-order function  
00:16:10 Compose ".map" inside ".map"  
00:16:50 Using the ".flat" method  
00:17:20 ".flatMap" higher-order function  
00:18:00 What is "currying" in programming?  
00:19:10 Self-documenting parameter names  
00:19:40 Moving into separate "helpers" file  
00:20:10 Creating a generic abstraction  
00:21:20 Custom "join" function parameter  
00:24:00 "Mapping" as a general concept  
00:24:40 Defining what CSS variables to use  
00:26:10 Using the native ".join" method  
00:26:40 Create actual string for "template"  
00:27:40 Connect template to Shadow DOM  
00:28:10 Example of "td-spacing" component  
00:29:00 Concept of "utilities" in programming  
00:30:40 Composing higher-order functions  
00:31:20 Calling "mergeArrays" inside itself00:00 - Introduction  
00:00:50 Add "Shoelace" library  
00:01:40 Add "sl-button" component  
00:02:10 Creating basic CSS styles  
00:02:50 Add "sl-dialog" component  
00:03:20 "lit-html" VS Code plugin  
00:04:50 Using "Shoelace" with HTML forms  
00:06:20 Embed buttons inside the dialog  
00:06:50 Add "sl-input" component  
00:07:40 Create custom "td-spacing"  
00:08:50 Component HTML auto-completion  
00:11:10 "lit-plugin" VS Code plugin  
00:12:10 Create "td-spacing" CSS styling  
00:14:00 What is ":host" pseudo-selector?  
00:13:50 Create CSS styles dynamically  
00:14:30 Define different values to combine  
00:15:15 ".map" higher-order function  
00:16:10 Compose ".map" inside ".map"  
00:16:50 Using the ".flat" method  
00:17:20 ".flatMap" higher-order function  
00:18:00 What is "currying" in programming?  
00:19:10 Self-documenting parameter names  
00:19:40 Moving into a separate "helpers" file  
00:20:10 Creating a generic abstraction  
00:21:20 Custom "join" function parameter  
00:24:00 "Mapping" as a general concept  
00:24:40 Defining what CSS variables to use  
00:26:10 Using the native ".join" method  
00:26:40 Create actual string for "template"  
00:27:40 Connect template to Shadow DOM  
00:28:10 Example of "td-spacing" component  
00:29:00 Concept of "utilities" in programming  
00:30:40 Composing higher-order functions  
00:31:20 Calling "mergeArrays" inside itself  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_08.3 Challenge 1 In this Module, you will continue with your “Book Connect” codebase, and further iterate on your abstractions. You must create an encapsulated abstraction of the book preview using a single factory function. If you are up for it, you can also encapsulate other aspects of the app into their own abstractions.  
   
If you are up for it, you can also convert other aspects of the app into web components.  
   
You will have a call with your coach where you highlight the following:  
  
What problems did you encounter converting the book preview to a component?  
What other elements make sense to convert into web components? Why?  
Why does keeping your HTML, CSS and JavaScript in a single file sometimes make sense?  
  
   
In your 1-on-1 session with your coach, you must demonstrate your understanding of all concepts covered in this module. It is at the coach's discretion to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_11.4 Project Submission DWA11 Project\_DWA11  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Global State Store and Redux Store.  
Worked through all the additional resources provided in the 'Further Reading' section at the end of the DWA\_11.2 Redux Store lesson.  
Completed DWA\_11: Challenge 1 and pushed code to the DWA11 GitHub folder.  
Finally, submit the link to your DWA\_11 Challenge Solutions via the [Projects] tab > DWA11 Submission  
  
  
To consider this module as “completed”, the above criteria need to have been met and demonstrated as done in your Code Review session.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_07.1 Creating Abstraction with SOLID As we learned in the previous module, abstraction is an important principle of software design that allows developers to create more flexible, maintainable, and reusable code. In this module, we will cover what exactly makes a good abstraction, specifically by introducing some common guidelines, and three different paradigms that have their own opinion on how abstraction should be done in software. In JavaScript, abstraction can be implemented using the SOLID principles, which are a set of guidelines for object-oriented programming.  
   
Let's do a quick overview of how to implement abstraction in JavaScript with SOLID, before diving into the first lecture, 'Creating Abstraction with SOLID'.  
  
  
Single Responsibility Principle (SRP): This principle states that a class or module should have only one reason to change. To apply SRP in JavaScript, you can create separate modules or classes that handle specific tasks or responsibilities. Each module or class should be responsible for a single functionality or feature, and should not be coupled with other functionalities.  
  
  
Open-Closed Principle (OCP): This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In JavaScript, you can apply this principle by creating abstract classes or interfaces that define a set of methods or properties. Concrete classes can then inherit from these abstract classes or interfaces and implement their methods. This way, you can add new functionalities without modifying existing code.  
  
  
Liskov Substitution Principle (LSP): This principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In JavaScript, you can implement LSP by ensuring that subclasses inherit all the properties and methods of their superclass, and do not alter their behaviour in unexpected ways.  
  
  
Interface Segregation Principle (ISP): This principle states that a client should not be forced to depend on methods it does not use. In JavaScript, you can apply ISP by creating small, focused interfaces that define only the methods that a client needs. This way, you can avoid creating large, bloated interfaces that are difficult to maintain and understand.  
  
  
Dependency Inversion Principle (DIP): This principle states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. In JavaScript, you can implement DIP by using dependency injection, which allows you to pass dependencies to a module or class rather than creating them inside. This way, you can decouple modules and make them more reusable and testable.  
  
  
   
Now, let's learn how you can implement abstraction in JavaScript and create more maintainable, flexible, and reusable code by following SOLID principles. The code used in this video is available in the following GitHub repository.  
   
  
   
00:00:00 Introduction  
00:01:00 Ignoring "Classes" for now  
00:02:00 Revisiting the Todo App  
00:02:30 Provided HTML and CSS  
00:03:00 What is "Procedural Programming"?  
00:03:40 Treating functions as instructions  
00:04:10 Commenting out dynamic HTML  
00:05:00 Creating task HTML with JavaScript  
00:05:30 Real example of error/type-checking  
00:07:40 Creating element and setting "innerHTML"  
00:08:10 Error/type-checking catches the hidden bug  
00:09:10 Creating the first abstraction with function  
00:10:20 Introducing the "SOLID" acronym as principles  
00:11:00 Important not to treat ideas as dogma  
00:12:10 "S" stands for "Single Responsibility"  
00:13:20 Splitting responsibilities between abstractions  
00:14:20 Configuring function to create a task  
00:17:00 Creating a function to update a task  
00:17:30 How to export type definitions  
00:18:00 "Pick" and "Omit" type helpers  
00:20:00 "Partial" and "Required" type helpers  
00:21:20 When to throw unhandled errors  
00:22:10 Knowing when to abstract behaviour  
00:22:40 Creating "getHtml" helper function  
00:24:40 Good abstractions prevent low-level bugs  
00:26:10 Duplication isn't always bad  
00:27:40 "D" stand for "Dependency Inversion"  
00:28:40 Thinking outside inwards vs inside outwards  
00:30:00 Remember that goal is a composition  
   
   
 Further reading To further unpack the concepts covered in this module, please work through the following resources:  
   
          - WATCH: Object Oriented vs Functional Programming with TypeScript by Fireship  
          - WATCH: Functional, Procedural & Object-oriented Programming - An Overview by Academind  
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_14.4 Challenge 1 In this challenge, you will return to the very first Tally App example and recreate it entirely using the Lit Framework. Apart from indicating competency with the Lit Framework, you will also be required to consider the following explicit state that the counter can be in:  
  
Normal  
Minimum Reached  
Maximum Reached  
  
   
Note that you do not need to call it these exact terms.  
   
You will have a call with your coach in which you will describe the HOW and WHY of decisions that you made as you rebuilt the Tally App with Lit.  
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.5 What is JavaScript even? Concluding the above, it is clear that what most people call JavaScript is technically titled ECMAScript. However, as quantum physicists don't preface all mentions of time with the relevant space-time in question, we will continue with the popular naming convention of "JavaScript".  
   
However, despite this, we need further qualifications for what we refer to when discussing JavaScript. Since what is known as JavaScript today is no longer confined to the browser. Today JavaScript is used in various applications, from NASA spacesuits to creating hybrid mobile applications (as covered in this course). However, for the sake of this course, we will consider JavaScript as a language similar to English. In short, a way to describe a set of instructions and what is done with these instructions is inconsequential.  
   
Following this metaphor, we can see the JavaScript spec as a dictionary indicating the syntax, words, and structure to be assumed when writing or reading the instructions. Similar to dictionaries, there is not a single "correct" dictionary, but several editions published at different times.  
   
Furthermore, this dictionary metaphor works exceptionally well for JavaScript since, while the language changes over time, there is no definitive point where specifications (dictionary rules) are implemented wholesale. Instead, the language itself fluxes and shifts on a smooth continuum. Yet, while the specification updates, it is adopted continually (for example, there is no clear point where English speakers switched from Shakespearian Old English to Victorian-era English or even Modern English).  
   
To further complicate this, there are also different ways the language is used in other circumstances. Yet all of them are covered in the specification and are still considered JavaScript (none are considered more "real" versions of JavaScript than the others). A fitting example would be the distinction between business jargon, everyday conversation and academic English. Yet none of these is a more "real" manifestation of English than the others.  
   
No wonder many developers colloquially refer to the mental impact that working with JavaScript has on developers as JavaScript Fatigue. Of course, it would be nice to have JavaScript be similar to stable runtime environments like PHP, Ruby or Python, where you know the exact syntax, functionality and limitation of how a compiler will execute your code. Yet, despite this back-to-front logic, JavaScript seems to be near-ubiquitous in the software world.  
   
The reason for the dramatic surge of JavaScript is a contentious topic, and it is clear that there are numerous factors. However, one possible factor is the development and specification process outlined above. To quote Dmitri Grabov in an article titled Why JavaScript is eating the World:  
   
The enterprise grade solutions which power all large companies were never designed to be easy to customise. Software was viewed much like a nuclear power plant. You build it and then it sits there doing its job. It will need maintenance and eventual decommissioning, but until then it functions as a unit. Those enterprise grade solutions were similarly designed as a unit. Sadly, this approach to software development is not even remotely viable today.  
— Dmitri Grabov, Why JavaScript is eating the World (2018)  
   
In short, in an ironic twist of fate, the very upside-down and make-shift process of standardising JavaScript has turned a simple scripting language with so many imperfections into one of the most widely-used languages. The fact that there will never be a new JavaScript 2 (but instead an unbroken continuing of "JavaScript" with incremental changes) makes this language so powerful.  
   
ES6 and further iterations of JavaScript are often incorrectly viewed as augmented versions of JavaScript, whereas ES5 is subsequently seen as the "real" version. There are several reasons for this, and they are understandable. However, as you'll probably guess from the content covered in the previous lesson, this view needs to be revised. Furthermore, by treating versions of JavaScript as equivalencies to, for example, PHP, Ruby or Python versions, we need to include more of what makes this little language so powerful.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_06.5 Challenge 1 This exercise presents you with a working version of the “Book Connect” website you previously audited as your final challenge. However, you must use objects and functions as abstractions to make the code more maintainable, extendable and easier to change.  
   
You might have already done this in your audit, but this is an opportunity to revisit your decisions and perhaps adjust, change, remove or add abstractions to improve the codebase. Also, please remember that you are encouraged to consider higher-level concepts discussed in the previous lessons, such as documentation, Styleguides and abstractions. Finally, you can start from scratch with the initially provided repository, merely using the script.js file below as a replacement or continue working on your version (if you created one) and updating the data.js file as listed below.  
   
As always, it is essential to note that you will be required to talk about the “how” and “why” you made specific decisions and not just present the “what” of the decisions that you made.  
   
Note that the code below (unlike the previous code you were provided with up until this point) does precisely what you need it to do; you need to rewrite (using abstraction and all the techniques you’ve learned up until now) to make it easier to maintain, change and understand.  
   
// Ensure that the data.js file exports all it's values. For example:  
  
export const BOOKS\_PER\_PAGE = 36;  
  
export const authors = {  
 "194e164b-9365-4358-b44a-f28a93cc528f": "Steven D. Levitt",  
 // remainder of object...  
}  
  
export const genres = {  
 "a4f80b3e-3e96-4266-b729-e09b71793182": "Economics",  
 // remainder of object...  
}  
  
export const books = [  
 {  
 id: "760b3450-9c86-42d0-8eff-e793bf823756",  
 // remainder of object...  
 }  
]  
   
   
// Fully working scripts.js file  
  
import { books, authors, genres, BOOKS\_PER\_PAGE } from './data.js'  
  
let page = 1;  
let matches = books  
  
const starting = document.createDocumentFragment()  
  
for (const { author, id, image, title } of matches.slice(0, BOOKS\_PER\_PAGE)) {  
 const element = document.createElement('button')  
 element.classList = 'preview'  
 element.setAttribute('data-preview', id)  
  
 element.innerHTML = `  
 <img  
 class="preview\_\_image"  
 src="${image}"  
 />  
   
 <div class="preview\_\_info">  
 <h3 class="preview\_\_title">${title}</h3>  
 <div class="preview\_\_author">${authors[author]}</div>  
 </div>  
 `  
  
 starting.appendChild(element)  
}  
  
document.querySelector('[data-list-items]').appendChild(starting)  
  
const genreHtml = document.createDocumentFragment()  
const firstGenreElement = document.createElement('option')  
firstGenreElement.value = 'any'  
firstGenreElement.innerText = 'All Genres'  
genreHtml.appendChild(firstGenreElement)  
  
for (const [id, name] of Object.entries(genres)) {  
 const element = document.createElement('option')  
 element.value = id  
 element.innerText = name  
 genreHtml.appendChild(element)  
}  
  
document.querySelector('[data-search-genres]').appendChild(genreHtml)  
  
const authorsHtml = document.createDocumentFragment()  
const firstAuthorElement = document.createElement('option')  
firstAuthorElement.value = 'any'  
firstAuthorElement.innerText = 'All Authors'  
authorsHtml.appendChild(firstAuthorElement)  
  
for (const [id, name] of Object.entries(authors)) {  
 const element = document.createElement('option')  
 element.value = id  
 element.innerText = name  
 authorsHtml.appendChild(element)  
}  
  
document.querySelector('[data-search-authors]').appendChild(authorsHtml)  
  
if (window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches) {  
 document.querySelector('[data-settings-theme]').value = 'night'  
 document.documentElement.style.setProperty('--color-dark', '255, 255, 255');  
 document.documentElement.style.setProperty('--color-light', '10, 10, 20');  
} else {  
 document.querySelector('[data-settings-theme]').value = 'day'  
 document.documentElement.style.setProperty('--color-dark', '10, 10, 20');  
 document.documentElement.style.setProperty('--color-light', '255, 255, 255');  
}  
  
document.querySelector('[data-list-button]').innerText = `Show more (${books.length - BOOKS\_PER\_PAGE})`  
document.querySelector('[data-list-button]').disabled = (matches.length - (page \* BOOKS\_PER\_PAGE)) > 0  
  
document.querySelector('[data-list-button]').innerHTML = `  
 <span>Show more</span>  
 <span class="list\_\_remaining"> (${(matches.length - (page \* BOOKS\_PER\_PAGE)) > 0 ? (matches.length - (page \* BOOKS\_PER\_PAGE)) : 0})</span>  
`  
  
document.querySelector('[data-search-cancel]').addEventListener('click', () => {  
 document.querySelector('[data-search-overlay]').open = false  
})  
  
document.querySelector('[data-settings-cancel]').addEventListener('click', () => {  
 document.querySelector('[data-settings-overlay]').open = false  
})  
  
document.querySelector('[data-header-search]').addEventListener('click', () => {  
 document.querySelector('[data-search-overlay]').open = true   
 document.querySelector('[data-search-title]').focus()  
})  
  
document.querySelector('[data-header-settings]').addEventListener('click', () => {  
 document.querySelector('[data-settings-overlay]').open = true   
})  
  
document.querySelector('[data-list-close]').addEventListener('click', () => {  
 document.querySelector('[data-list-active]').open = false  
})  
  
document.querySelector('[data-settings-form]').addEventListener('submit', (event) => {  
 event.preventDefault()  
 const formData = new FormData(event.target)  
 const { theme } = Object.fromEntries(formData)  
  
 if (theme === 'night') {  
 document.documentElement.style.setProperty('--color-dark', '255, 255, 255');  
 document.documentElement.style.setProperty('--color-light', '10, 10, 20');  
 } else {  
 document.documentElement.style.setProperty('--color-dark', '10, 10, 20');  
 document.documentElement.style.setProperty('--color-light', '255, 255, 255');  
 }  
   
 document.querySelector('[data-settings-overlay]').open = false  
})  
  
document.querySelector('[data-search-form]').addEventListener('submit', (event) => {  
 event.preventDefault()  
 const formData = new FormData(event.target)  
 const filters = Object.fromEntries(formData)  
 const result = []  
  
 for (const book of books) {  
 let genreMatch = filters.genre === 'any'  
  
 for (const singleGenre of book.genres) {  
 if (genreMatch) break;  
 if (singleGenre === filters.genre) { genreMatch = true }  
 }  
  
 if (  
 (filters.title.trim() === '' || book.title.toLowerCase().includes(filters.title.toLowerCase())) &&   
 (filters.author === 'any' || book.author === filters.author) &&   
 genreMatch  
 ) {  
 result.push(book)  
 }  
 }  
  
 page = 1;  
 matches = result  
  
 if (result.length < 1) {  
 document.querySelector('[data-list-message]').classList.add('list\_\_message\_show')  
 } else {  
 document.querySelector('[data-list-message]').classList.remove('list\_\_message\_show')  
 }  
  
 document.querySelector('[data-list-items]').innerHTML = ''  
 const newItems = document.createDocumentFragment()  
  
 for (const { author, id, image, title } of result.slice(0, BOOKS\_PER\_PAGE)) {  
 const element = document.createElement('button')  
 element.classList = 'preview'  
 element.setAttribute('data-preview', id)  
   
 element.innerHTML = `  
 <img  
 class="preview\_\_image"  
 src="${image}"  
 />  
   
 <div class="preview\_\_info">  
 <h3 class="preview\_\_title">${title}</h3>  
 <div class="preview\_\_author">${authors[author]}</div>  
 </div>  
 `  
  
 newItems.appendChild(element)  
 }  
  
 document.querySelector('[data-list-items]').appendChild(newItems)  
 document.querySelector('[data-list-button]').disabled = (matches.length - (page \* BOOKS\_PER\_PAGE)) < 1  
  
 document.querySelector('[data-list-button]').innerHTML = `  
 <span>Show more</span>  
 <span class="list\_\_remaining"> (${(matches.length - (page \* BOOKS\_PER\_PAGE)) > 0 ? (matches.length - (page \* BOOKS\_PER\_PAGE)) : 0})</span>  
 `  
  
 window.scrollTo({top: 0, behavior: 'smooth'});  
 document.querySelector('[data-search-overlay]').open = false  
})  
  
document.querySelector('[data-list-button]').addEventListener('click', () => {  
 const fragment = document.createDocumentFragment()  
  
 for (const { author, id, image, title } of matches.slice(page \* BOOKS\_PER\_PAGE, (page + 1) \* BOOKS\_PER\_PAGE)) {  
 const element = document.createElement('button')  
 element.classList = 'preview'  
 element.setAttribute('data-preview', id)  
   
 element.innerHTML = `  
 <img  
 class="preview\_\_image"  
 src="${image}"  
 />  
   
 <div class="preview\_\_info">  
 <h3 class="preview\_\_title">${title}</h3>  
 <div class="preview\_\_author">${authors[author]}</div>  
 </div>  
 `  
  
 fragment.appendChild(element)  
 }  
  
 document.querySelector('[data-list-items]').appendChild(fragment)  
 page += 1  
})  
  
document.querySelector('[data-list-items]').addEventListener('click', (event) => {  
 const pathArray = Array.from(event.path || event.composedPath())  
 let active = null  
  
 for (const node of pathArray) {  
 if (active) break  
  
 if (node?.dataset?.preview) {  
 let result = null  
   
 for (const singleBook of books) {  
 if (result) break;  
 if (singleBook.id === node?.dataset?.preview) result = singleBook  
 }   
   
 active = result  
 }  
 }  
   
 if (active) {  
 document.querySelector('[data-list-active]').open = true  
 document.querySelector('[data-list-blur]').src = active.image  
 document.querySelector('[data-list-image]').src = active.image  
 document.querySelector('[data-list-title]').innerText = active.title  
 document.querySelector('[data-list-subtitle]').innerText = `${authors[active.author]} (${new Date(active.published).getFullYear()})`  
 document.querySelector('[data-list-description]').innerText = active.description  
 }  
})  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_10.2 Component Slots    
  
00:00:00 Recapping "Polymorphism"  
00:01:50 Recapping "Over-abstraction"  
00:02:50 When to create abstractions?  
00:03:50 Example of Amazon.com buttons  
00:04:40 Multiple single-use components  
00:05:30 Attributes as "ad-hoc" polymorphism  
00:06:40 Aim for fewer modular components  
00:07:10 Only way to learn is to get wrong  
00:07:40 Introducing component "slots"  
00:08:10 What is "parametric polymorphism"?  
00:09:10 "Ad-hoc polymorphism" has limits  
00:10:30 Passing abstractions into abstractions  
00:11:10 What is "recursion" in programming?  
00:12:00 "Higher-order" is a complex concept to grasp  
00:13:30 Some frameworks rely on "higher-order"  
00:14:20 Different types of complexity  
00:15:00 Exchanging types of complexity  
00:15:20 Example of "slot" in component  
00:16:10 Somewhat similar to nesting in HTML  
00:16:40 "Parametric" based on mathematics  
00:19:00 Revisiting our "todo" example  
00:20:00 Creating "user-action" component  
00:22:10 Discussion on "Github Copilot"  
00:22:40 Finishing "user-action" component  
00:24:00 Treating components like regular HTML  
00:24:30 Adding "importance" dynamic styling  
00:25:40 Accessing CSS variables from components  
00:27:10 Adding "importance" getters and setters  
00:28:20 Directly targeting elements in components  
00:29:50 Narrowing "importance" type definition  
00:30:40 Check to confirm the "slot" value passed  
00:31:00 What is a "super" function call?  
00:32:20 Adding "user-action" to another codebase  
00:33:10 We Need to re-add CSS variables  
00:34:20 Passing custom CSS into components  
00:36:10 Sharing web components publicly  
00:36:40 Real-world web components in the Wild  
00:37:00 Brief overview of the "Stencil" tool  
00:39:00 "Material Design" components  
00:40:00 "Shoelace" components  
00:41:00 Adding custom "Shoelace" styling  
00:41:30 Using the "Shoelance" dialogue component  
00:43:20 Overview of "Shoelace" documentation  
00:44:30 "MUI" components for React framework  
00:45:10 Polymorphism allows you to extend components  
00:45:40 Components can be styled differently  
00:46:00 How much HTML and CSS do you write?  
00:47:40 What are "named slots" in components?  
00:49:00 Setting named slot attribute on an element  
00:49:40 Can pass any valid HTML into a slot  
00:50:10 Nesting components themselves  
00:50:50 Transforming slots inside a component  
00:51:10 What is CSS "::part" pseudo-element?  
00:53:00 Introduce "functional programming"  
   
   
 Further reading Please explore the following resources to understand better the concepts covered in this module.  
   
          - READ: The Wrong Abstraction by Sandi Metz  
          - READ: MDN: Using templates and slots  
          - READ: Shadow DOM slots, composition  
          - READ: Shoelace: A forward-thinking library of web components  
          - WATCH: AHA Programming - Kent C. Dodds  
          - WATCH: 10 Design Patterns Explained in 10 Minutes  
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.1 Introduction What is JavaScript?  
To effectively manage complexity in our JavaScript software, we must have a solid understanding of the nuances of the JavaScript language itself.  
   
You may be surprised to discover how distinct JavaScript is from other programming languages. Understanding these differences between JavaScript and more “traditional” programming languages is critical since JavaScript constitutes the low-level material we will use to create our software during this course. In the same way, an architect must understand the fundamental properties of steel, glass, concrete and wood; you must understand the nature of JavaScript inside-out.  
   
In a certain sense, this module is the inverse of the preceding “Managing Complexity” module since it contains no video content. However, you are required to work through the range of supplied reading material. Similar to the previous module, your understanding of the programming concepts and subject matter will be tested during your 1-on-1 session with your coach on the key concepts in this module.  
   
Introduction  
   
You might have heard someone refer to a specific version of JavaScript as ES5, ES6, ES7 or even by a suffixed date (for example, ES2015). You might even have used a particular version of JavaScript or even know the differences between different versions - yet you might not understand what the "ES" abbreviation refers to or why there are even different versions.  
   
The Web vs The Internet  
   
Before we get to JavaScript specifically, expanding our scope to include the Web is beneficial. A high-level understanding of how we came to where we are today will help demystify many concepts discussed later in this module.  
   
First and foremost, it is essential to distinguish between what is known as the "Web" and the "Internet". Both of these are often used interchangeably. However, the Internet refers to the underlying hardware and infrastructure on which the Web is built. Furthermore, the Web is only one of several ecosystems built on top of the Internet. The Internet is also used by various other platforms like IMAP/SMTP (for email) and a range of Internet of Things (IoT) devices. The Internet itself precedes the Web by several years. The Internet initially started as ARPANET, developed exclusively for military use by the United States government.  
   
Language of the Web  
   
Today, the Web, or more technically correct, the "World Wide Web" (hence the WWW abbreviation), is primarily accredited to a single individual: Tim Berners-Lee.  
While working as a physicist for the European Organization for Nuclear Research (CERN) in Switzerland, Berners-Lee developed the concept of an extensive decentralised collection of documents described by a single, unified markup language. This markup language eventually became known as HTML (Hypertext Markup Language), and the network itself, while initially called the "Information Mesh" and at a later point "Mine of Information", eventually became the "World Wide Web". The language and tools required to host a server on the Web were released for free and as open-source by Berner's Lee. By 1993 there were about 500 servers connected to the Web. These servers contained various documents styled in the HTML language as follows:  
   
<TABLE BGCOLOR="#CCCCFF" BORDER="10" ALIGN="CENTER">  
 <TR>  
 <TD>  
 <H1 ALIGN="CENTER">  
 <FONT SIZE="3" COLOR="red" FACE="arial">  
 Welcome to the World Wide Web!  
 </FONT>  
 </H1>  
 </TD>  
 </TR>  
</TABLE>  
   
Noted, this is quite a far cry from what we know today as HTML. Perhaps the most striking is the common usage of all-uppercase tags in the early days of the Web (as also seen in the HTML 4 specification). Only with the formulation of XHTML in 2000 did writing lowercase become the dominant convention (since XML, upon which it was built, requires tags to be lowercase). Similarly striking in the above example is the application of styling using HTML tags and attributes like <FONT>. While the notion of a dedicated styling language, eventually called Cascading Style Sheets (CSS), was several years away, a quest for a scripting language was burning hot. At this point, HTML was exclusively a static language. This means that it cannot be changed after loading the page.  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_11.1 Global State Store This module looks at a specific approach used in most modern JavaScript apps called a “Store”. It helps manage application state and side effects in a centralised, predictable manner. In the lecture material below, we will create our own “Redux”-inspired store using Functional Programming principles.  
   
   
  
   
   
00:00:00 Recapping programming paradigms  
00:01:00 What is Procedural Programming?  
00:02:00 What are side-effects in programming?  
00:02:30 Recapping OOP  
00:03:20 How does OOP do polymorphism?  
00:04:10 How does OOP do encapsulation?  
00:05:10 Example of carousel HTML component  
00:06:00 Summarising difference between paradigms  
00:06:30 Difference between Procedural and FP  
00:07:30 In FP you limit side-effects as far as possible  
00:08:00 Manage side-effects if unavoidable  
00:08:30 Basic Tally App with OOP  
00:09:15 Side-effects in basic Tally App  
00:09:50 Hard to untangle behaviour in OOP  
00:10:30 Moment.js vs Date-fns  
00:13:40 OOP is much harder to test than FP  
00:14:10 Example of managing side-effects in FP  
00:14:40 Creating "pure" (no side-effects) behaviour  
00:15:10 Complete new state object on change  
00:16:00 Example of how FP helps manage data  
00:16:40 Very basic example of a "lens" in FP  
00:17:20 Seperating data, behaviour and logic  
00:17:50 Example of common "Store" pattern  
00:18:40 Saving data with "middleware" in "Store"  
00:19:10 Still keeping components as OOP  
00:19:50 JavaScript uses FP to handle asynchrony  
00:20:20 Basic example of asynchrony in JavaScript  
00:21:50 JavaScript supports both OOP and FP  
00:22:40 Creating "Store" to handle our side-effects  
00:23:20 Create "update" method in "Store"  
00:24:20 Comparing performance of classes vs. factories  
00:24:50 Prototypes were added to solve a problem  
00:26:20 Avoid focusing on micro-optimisations  
00:26:50 Only important in super rare edge-cases  
00:27:30 Focus on more important performance issues  
00:29:10 What does "update" method do?  
00:29:40 Adding JSDoc interface to our "Store"  
00:30:20 In FP pass functions (not data) around  
00:31:50 Using JavaScript module functionality  
00:32:10 Breaking out behaviour into "actions.js"  
00:32:40 Continue adding JSDoc definitions  
00:33:10 Looking at "Observer Design Pattern"  
00:34:00 Discussing "Redux" and "Pinia"  
00:34:40 Adding "subscribe" method to "Store"  
00:36:10 Grabbing old state on "update" call  
00:36:40 Run "action" on old state to create new state  
00:37:10 Continue creating "subscribe" method  
00:38:50 "subscribe" returns its own "unsubscribe"  
00:39:30 Complete "subscribe" method skeleton  
00:40:30 Functions are primary unit of meaning in FP  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_13.3 Partial Application    
  
   
00:00:00 Introduction  
00:01:20 Defining Higher-order Function (HOF)  
00:03:00 Common uses for HOF  
00:06:30 Example without using HOF composition  
00:08:50 Recapping the "Single Responsibility" Principle  
00:09:40 Example using HOF composition  
00:10:50 "Partial Application" vs "Currying"  
00:12:10 Functions can create other functions  
00:12:40 ".reduce" higher-order function  
00:15:10 Dynamically create a new function  
00:15:40 Further composing created functions  
00:16:30 Introducting "Lodash" library.  
00:17:40 I Always prefer native behaviour  
00:18:20 Categories of utilities in "Lodash"  
00:19:00 Wrapping behaviour in function  
00:21:20 How deep do you want to explore FP?  
00:22:00 Careful to overdo FP in general code  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.4 The JavaScript Specification Two years before the founding of the Web Standards Project, Netscape caught wind of this growing frustration and, upon seeing its diminishing market share against Internet Explorer, approached the European Computer Manufacturers Association (ECMA) to create a standardised specification that could be used by Netscape's JavaScript and all its variants.  
   
If you're not familiar with the concept of a specification, and specifically a "requirement specification", it is described by Wikipedia as follows:  
   
A requirement specification is a documented requirement or set of documented requirements, to be satisfied by a given material, design, product, service, etc. It is a common early part of engineering design and product development processes, in many fields.  
— Wikipedia, Specification (technical standard)  
   
The specification is a collection of documents describing how JavaScript and its variants should work in the context of JavaScript and its variants. For example, here is an extract from the very first specification created by ECMA explaining how single-line commenting should work:  
   
Because a single-line comment can contain any character except a "LineTerminator" character, and because of the general rule that a token is always as long as possible, a single-line comment always consists of all characters from the "//" marker to the end of the line. However, the "LineTerminator" at the end of the line is not considered to be part of the single-line comment; it is recognized separately by the lexical grammar and becomes part of the stream of input elements for the syntactic grammar. This point is very important because it implies that the presence or absence of single-line comments does not affect the process of automatic semicolon insertion (see section 7.8.2).  
— ECMA 262: 1st Edition (1997)  
   
In short:  
  
The compiler should not execute single-line comments.  
Singe-line comments should always start with "//"  
Single-line comments accept any character except line breaks.  
Single-line comments end before the first line breaks  
  
   
This means that all JavaScript compilers should honour the following:  
   
// console.log('This code should be executed')  
  
console.log('This code should be executed')  
   
You can think of the original HTML specification created by Berners-Lee insofar that it was an open document that tells browser makers to make sure that their browser handles the HTML code in the manner specified in the document. So, for example, the HTML specification outlines that the <img> tag should be used to express an image and that browsers can't create their version called <netscape-img> or <super-image>. In the same way, the hope was that the specification created by ECMA could be used as a standard for how scripting should work in all browsers.  
   
Eventually, ECMA submitted their proposed specification, titled "ECMAScript: A general purpose, cross-platform programming language". After internal negotiations, it was adopted by both Netscape and Microsoft in 1997. Henceforth, what was previously known as JavaScript and JScript were to be unified into a single standardised language titled ECMAScript. At one point, it was considered even to change the file extension from .js to .es. Yet, the name never really caught on, and to this day, we still refer to it as JavaScript - even though JavaScript technically refers to the Netscape Navigator-specific scripting language. Interestingly enough, the original creator of JavaScript, Brendan Eich, attributes this to general blandness of the name "ECMAScript".  
   
"ECMAScript was always an unwanted trade name that sounds like a skin disease."  
— Brendan Eich, Will there be a suggested file suffix for ES4? (2006)  
   
Yet, the volunteer-driven group tasked with creating and maintaining new ECMAScript specifications was designated an even more ominous-sounding name, merely titled Technical Committee 39 (TC39). I often joke that this sounds like something from George Orwell's Nineteen Eighty-Four novel.  
   
Yet, despite this name, TC39 is still the primary caretaker of the ECMAScript specification used by all compilers. As a result, you can follow the development of the language in real-time on GitHub at https://github.com/tc39, and even propose unique additions or changes using issues or pull requests.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_05.4 Project Submission DWA5 Project DWA5  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Core Concepts Part 1 and Core Concepts Part 2.  
Worked through the 'Further Reading' at the end of the Core Concepts Part 2 lesson.  
Completed DWA\_05: Challenge 1 and pushed your code to the DWA5 GitHub folder.  
Finally, submit the link to your DWA\_05 Challenge Solutions via the [Projects] tab > DWA5 Submission.  
  
   
To prepare for your code review session with your coach, ensure you are able to talk through the code that provides solutions to the FOUR Unresolved User Stories before booking your 1-on-1 session.  
   
To consider this module as “completed”, the above criteria must have been met and demonstrated as done in your Code Review session.  
   
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_08.2 Working with Encapsulation    
  
   
00:00:00 Getting HTML elements  
00:00:40 Connecting state to HTML  
00:03:40 Moving types into abstraction  
00:04:00 Storing data inside closure itself  
00:05:00 Decoupling data state from DOM  
00:05:30 "id" getter and setter  
00:06:00 "created" getter and setter  
00:06:30 "completed" getter and setter  
00:07:10 "title" getter and setter  
00:08:20 "urgency" getter and setter  
00:09:00 "due" getter and setter  
00:09:30 Fixing problem with "urgency"  
00:10:00 Example of modifying a task  
00:10:30 Auto-update HTML in "setter"  
00:11:30 Example of behaviour in real-time  
00:12:40 Allowing adding of new tasks  
00:13:20 Creating HTML for "adding"  
00:14:10 Automatically enable "ts-check"  
00:14:40 VS Code uses TypeScript under hood  
00:15:30 Define level of browser support  
00:16:40 Adding HTML dialog code  
00:18:10 Seperating "adding" from tasks  
00:18:50 Creating higher-level of abstraction  
00:19:30 Adding event listeners to "adding"  
00:20:00 Return HTML references after creation  
00:21:30 Native "showModal" and "close" methods  
00:22:00 Adding HTML form to dialog  
00:22:40 Adding "submit" event listener  
00:23:10 Adding "due" and "urgency" fields  
00:24:20 Adding "callback" behaviour to "adding"  
00:26:00 Keeping "tasks" and "adding" decoupled  
00:27:00 Executing callback inside form submit  
00:28:00 Connecting "tasks" and "adding"  
00:28:30 When not to encapsulate with OOP  
00:29:00 Still not achieving full encapsulation  
00:29:20 CSS file can still change HTML  
00:29:50 Can still directly target via DOM  
00:30:20 We can encapsulate DOM and CSS  
00:31:00 Web Components help achieve this  
00:31:30 Web Components need "Class"  
   
   
 Further reading To extend your learning of concepts covered in this module, you are required to work through the following resources:  
  
           - JavaScript Factory Functions with ES6+ by Eric Elliot  
           - JAVASCRIPT.INFO: Property getters and setters  
           - WATCH: Factory Functions in JavaScript by Fun Fun Function  
           - WATCH: JavaScript getters and setters by Bro Code  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_11.2 Redux Store    
  
   
   
00:00:00 "subscribe" return own "unsubscribe"  
00:01:00 ".filters" higher-order function  
00:02:40 ".forEach" higher-order function  
00:03:40 Functional Programming is very expressive  
00:04:40 Using our created "Store"  
00:05:40 Passing the first "action"  
00:06:10 Creating a new state object  
00:06:40 Spreading into nested objects  
00:07:10 Debugging "Store" actions  
00:07:40 Example of "unsubscribe"  
00:08:10 Listen to a specific part of state  
00:09:40 Binding side-effects to state  
00:10:30 "Store" helps manage side-effects  
00:11:20 Mention of "prop-drilling" and "serialization"  
00:11:50 JS Frameworks used alongside "Store"  
00:12:30 "Redux" vs "Zustand"  
00:13:40 Three core "Redux" concepts  
00:14:20 Concept #1: Store  
00:14:50 Concept #2: Actions  
00:15:30 Concept #3: Reducers  
00:16:00 "Redux" requires a lot of code  
00:16:30 What problems does "Redux" solve?  
00:17:20 Traditional OOP struggles with asynchrony  
00:18:40 Mutations are complex with asynchrony  
00:19:40 "Store" is separate from the UI  
00:20:40 FP helps manage asynchrony  
00:21:10 Principle #1: Single data source  
00:22:20 Principle #2: Unidirectional data  
00:23:50 Principle #3: Pure data updates  
00:25:30 Redux "Store" methods  
00:26:00 Modeling data with "Redux"  
00:27:00 Creating documentation for "Store"  
00:30:20 Creating documentation for "Actions"  
00:32:30 "Actions" vs "Action Creators"  
00:35:30 "Redux" scales well as the app grows  
00:37:20 Create Redux "Reducer" skeleton  
00:39:20 Add "Store" methods as exports  
00:40:20 Creating "dispatch" method  
00:42:00 Creating the "subscribe" method  
00:42:30 Trading different types of complexity  
00:45:10 Finish remaining method logic  
00:47:40 Add Redux "Reducer" logic  
00:50:30 Example of working "Redux" store  
00:37:30 Remember to "dispatch" actions  
00:38:00 Remember to add the initial state  
00:39:30 "Redux" vs "React.js"  
   
   
 Further reading For a greater understanding of the concepts covered in this module, please work through the following resources:  
   
          - WATCH: Is Functional Programming a Good Idea? By Sammy Talks  
          - WATCH: Redux in 100 Seconds by Fireship  
          - READ: Redux. A Predictable State Container for JS Apps  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.6 JavaScript Compilers If you've seen the first The Matrix movie, you might be familiar with the scene where Neo (played by Keanu Reeves) visits The Oracle and encounters a boy bending a spoon by means of what appears to be psychokinesis.  
   
Neo, upon enquiring how it is possible, is met with the following response:  
   
"Do not try and bend the spoon. That's impossible. Instead, only try to realise the Truth... There is no spoon... Then you'll see that it is not the spoon that bends; it is only yourself."  
— The Matrix (1999)  
   
This is a highly apt analogy for the current JavaScript ecosystem. Making sense of JavaScript in this day and age is nigh impossible at face value. This sentiment has become so ubiquitous that (as mentioned in the previous session) it is often referred to as Javascript Fatigue. However, this is mainly due to misunderstanding what JavaScript (or technically ECMAScript as discussed in the previous lesson) is and how to write it.  
   
Central to this is a misconception is an expectation that (much like other languages) you know ahead of time which compiler will run your code, and not only that, but you know ahead of time the specific set-in-stone, pre-defined functionality and syntax supported by the compiler. This is similar to how developers think about other language compilers, for example, PHP 7, Python 3 or the upcoming Ruby 3. All these have specific versioned compilers you can often install and control on your server. By this logic, one would assume that we would have different compilers for ES5, ES6, ES7, etc. Instead, I've heard remarks like "I don't think ES6 comes installed on Linux" or "We're using regular JavaScript, not ES7".  
   
However, due to the decentralised nature of the web itself, there are around 39 different JavaScript compilers being used at the time of writing, each maintained by a different team. You might have even heard of them before; some most well-known ones are:  
  
v8  
SpiderMonkey  
Chakra  
Tamarin  
  
   
This conception of a controlled shift from ES5 to ES6 and beyond (similar to the shift from the older PHP 6 to modern PHP 7) is perhaps due to the period between 1999 and 2015, where JavaScript remained primarily unchanged. In 16 years, JavaScript only saw the addition of the following three features:  
  
The JSON native object  
"strict mode" for more strict error checking  
Minor changes to JavaScript objects.  
  
   
The reason for this is complicated, and a large part of it can be attributed to infighting amongst the TC39 committee itself. For this reason, the specification itself jumps straight from ES3 to ES5. Finally, the level of disagreement became so extreme that the ES4 specification itself had to be abandoned. Interestingly enough, to date, no ES4 specification has been published.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_05.3 Challenge 1 PLEASE NOTE:   
This exercise provides user stories in the Gherkin syntax, covered in Documentation - Reading Material. Please make sure you understand the use and meaning of this syntax before continuing.  
   
Below is the HTML and JavaScript code for a “Whole Number Divider”. At the moment it only meets two user stories (see below under “Resolved Stories”). Please ensure that it meets all user stories listed below:  
   
Resolved Stories  
Scenario: Starting program state  
  
GIVEN that the submit button has not been pressed yet  
AND the code has just loaded  
THEN “NO calculation performed” should be displayed at the bottom  
  
Scenario: Dividing numbers result in a whole number  
  
GIVEN that the submit button is pressed  
WHEN 20 is entered into the first input  
AND 10 is entered into the second input  
THEN 2 should be displayed at the bottom  
  
   
Unresolved Stories  
Scenario: Dividing numbers result in a decimal number  
  
GIVEN that the submit button is pressed  
WHEN 20 is entered into the first input  
AND 3 is entered into the second input  
THEN the number 6 with no decimal should be shown  
  
   
Scenario: Validation when values are missing  
  
GIVEN that the submit button is pressed  
WHEN either or both inputs are empty  
THEN the divisions should not be done  
AND the following should be displayed: “Division not performed. Both values are required in inputs. Try again”.  
  
   
Scenario: An invalid division should log an error in the console  
  
GIVEN that the submit button is pressed  
WHEN 20 is entered into the first input  
AND -3 is entered into the second input  
THEN the division should not be done  
AND the following should be displayed: “Division not performed. Invalid number provided. Try again”.  
AND an error should be logged in the browser console that shows the call stack  
BUT the program should not crash entirely  
  
   
Scenario: Providing anything that is not a number should crash the program  
  
GIVEN that the submit button is pressed  
WHEN ‘YOLO’ is entered into the first input  
AND ‘+++’ is entered into the second input  
THEN the entire screen should be replaced with “Something critical went wrong. Please reload the page  
AND an error should be logged in the browser console that shows the call stack.  
  
   
<!-- index.html -->  
  
<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <meta charset="UTF-8" />  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />  
 <title>Whole Number Division</title>  
 <script src="./scripts.js" defer type="module"></script>  
 </head>  
   
 <body>  
 <h1>Whole Number Division</h1>  
 <form data-form="data-form">  
 <label  
 ><span>Dividend</span>  
 <input name="dividend" />  
 </label>  
 <label>  
 <span>Divider</span>  
 <input name="divider" />  
 </label>  
 <button type="submit">Calculate</button>  
 </form>  
 <hr />  
 <div data-result="data-result">No calculation performed</div>  
 </body>  
</html>  
   
   
// scripts.js  
  
const form = document.querySelector("[data-form]");  
const result = document.querySelector("[data-result]");  
  
form.addEventListener("submit", (event) => {  
 event.preventDefault();  
 const entries = new FormData(event.target);  
 const { dividend, divider } = Object.fromEntries(entries);  
 result.innerText = dividend / divider;  
});  
   
   
In your 1-on-1 session with your coach, you will be required to demonstrate your understanding of all concepts covered in this module. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.3 The Browser Wars Due to the success of Netscape Navigator, Microsoft decided to develop its own competitor: Internet Explorer. Just to let you know, at this point, JavaScript was exclusively a proprietary scripting language and safe-guarded internally by the Netscape team. Consequently, Microsoft was forced to reverse engineer JavaScript and create their own called JScript, which they launched alongside CSS (an Internet Explorer exclusive). Despite the rapid development of the Web platform during this time, no overarching standardisation process existed. This meant that you effectively needed to write either JavaScript, JScript or both (depending on what browsers you want to support). This was further exacerbated by the additional JavaScript knock-off languages - the most well-known being ActionScript, created by Macromedia.  
   
The resulting period is often referred to as The Browser Wars. In short:  
   
"During these releases, it was common for web designers to display 'best viewed in Netscape' or 'best viewed in Internet Explorer logos. These images often identified a specific browser version and were commonly linked to a source from which the stated browser could be downloaded. These logos generally recognised the divergence between the standards supported by the browsers and signified which browser was used for testing the pages."  
— Wikipedia, Browser Wars  
   
Things were further complicated by a massive surge in usage of Internet Explorer - eventually peaking at 95% of the market share. As a result, JScript has overtaken JavaScript in usage. Yet, the difference between the various flavours of JavaScript used by different browsers grew larger.  
   
This could only continue until developers started demanding some over-arching standardisation amongst browsers. This culminated in 1998 with establishment of the Web Standards Project, headed by several prominent web developers. The organisation aimed to pressure browser makers to standardise their languages so that developers didn't need to maintain several versions of the same codebase.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.2 Birth of JavaScript In the early years, several competing pieces of software allowed users to view documents on the web; the most popular was Berners-Lee's browser, known as the WorldWideWeb browser. However, most browsers were highly technical and exclusively aimed at academics to be used to share documents on the Web. However, in 1993, the Nation Center for Supercomputing Applications (NCSA) created a new Mosaic browser. In contrast to the preceding generation of browsers, Mosaic was primarily designed for ease of use. In the words of Gary Wolfe, in a 1994 edition of Wired Magazine:  
   
"Mosaic is the celebrated graphical "browser" that allows users to travel through the world of electronic information using a point-and-click interface. Mosaic's charming appearance encourages users to load their own documents onto the Net, including color photos, sound bites, video clips, and hypertext "links" to other documents. By following the links - click, and the linked document appears - you can travel through the online world along paths of whim and intuition.  
   
Mosaic is not the most direct way to find online information. Nor is it the most powerful. It is merely the most pleasurable way, and in the 18 months since it was released, Mosaic has incited a rush of excitement and commercial energy unprecedented in the history of the Net."  
— Gary Wolfe, Wired Magazine (1994)  
   
Mosaic quickly became the web browser of choice. However, with the widespread adoption of user experience at the forefront came a burgeoning desire to add interactivity to web pages.  
After disputes around licensing, the original creators left NSCA and founded Netscape. Netscape was slated to release a follow-up to Mosaic called Netscape Navigator. From inception, this browser was envisaged to include a scripting language eventually. So in 1995, Netscape commissioned Brandon Eich to create an interactive scripting language for Netscape Navigator. This language was initially called "Mocha" and was meant to be modelled on a functional programming language called Scheme. It was later renamed to LiveScript and finally, at the behest of Netscape management, to JavaScript.  
   
This rebranding to "JavaScript" was made in response to the meteoric popularity of the Java programming language. This was meant to imply some connection between JavaScript and Java - the latter having no real relationship with the former. Java was owned by a completely different organisation called Oracle. However, this change to "JavaScript" entailed more than just rebranding; the direction of the language itself was slated to change from its functional origins to appear closer to the object-oriented approach popularised by Java.  
   
Unfortunately, to this day, no one outside of the original Netscape team knows precisely what resulted in this radical shift in direction mid-project. However, many contemporary developers speculate that this single reorientation is to blame for all the inconsistencies and design flaws in the JavaScript language. To date, JavaScript is infamous for being extraordinarily inconsistent and confusing in design. Pulling one's nose up at JavaScript has become a favourite pastime of many developers, including JavaScript developers. An open-source repository, WTF JS, catalogues these strange, unexpected JavaScript rules and behaviours.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_10.1 Introducing Polymorphism Critical to creating abstractions that scale well grows is the concept of “polymorphism”. In this lesson, we will reflect on how we have used polymorphism until now and introduce a new type of polymorphism, called “parametric polymorphism”, as an additional tool when creating abstractions.  
   
  
   
00:00:00 Introduction  
00:01:00 What are Polyfills?  
00:03:00 Example of "padStart" polyfill  
00:04:00 Brief mention of "polyfill.io"  
00:04:30 Dont confuse "Poyfills" and "Polymorphism"  
00:05:10 Polymorphism means "many forms"  
00:06:30 Polymorphism helps manage complexity  
00:07:00 Recapping concept of "encapsulation"  
00:07:30 Idea of "Big Ball of Mud"  
00:08:00 What is "Spaghetti Code"?  
00:09:50 Examples of "encapsulation"  
00:10:30 Value in having re-usable components  
00:11:20 Components will grow indefinitely  
00:12:10 Aim for a few reusable components  
00:13:00 You Need to learn to evaluate yourself  
00:13:50 Example #1: Subtype Polymorphism  
00:14:20 Example #2: Structural Polymorphism  
00:16:10 Example #3: Ad-hoc Polymorphism  
00:17:10 Example #4 Coercion Polymorphism  
00:19:10 What is "over-abstraction"?  
00:20:10 Examples of abstraction in design  
00:22:00 Abstraction in the World of Art  
00:25:50 Example of "over-abstraction" in art  
00:26:30 "Over-abstraction" in visual design  
00:27:20 Abstracting too much can be harmful  
00:28:40 "design patterns as Idea" vs "Design Patterns"  
00:29:20 Attempt to create abstract, general solutions  
00:29:50 "Design Patterns" conceived before JavaScript  
00:30:20 Programming changed considerably since  
00:31:10 "Design Patterns" possible "over-abstraction"  
00:32:10 Still valuable learning "Design Patterns"  
00:33:10 Some warnings about "Design Patterns"  
00:35:10 Encouraged to read about "Design Patterns  
00:36:00 Dont blindly follow "Design Patterns" solutions  
00:36:20 Example of using the "Singleton" pattern  
00:39:00 Why is "Singleton" bad in JavaScript?  
00:41:40 What are "idioms" and "idiomatic"  
00:42:10 Example of "idioms" in human language  
00:43:20 Some developers aware not aware of this  
00:44:40 Adding missing "static" to class  
00:45:20 Careful of universal, silver-bullet answers  
00:46:00 Important to start evaluating yourself  
   
   
   
 Further reading Please explore the following resources to understand better the concepts covered in this module.  
   
          - READ: The Wrong Abstraction by Sandi Metz  
          - READ: MDN: Using templates and slots  
          - READ: Shadow DOM slots, composition  
          - READ: Shoelace: A forward-thinking library of web components  
          - WATCH: AHA Programming - Kent C. Dodds  
          - WATCH: 10 Design Patterns Explained in 10 Minutes  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_07.5 Project Submission DWA7 Project\_DWA7  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Creating Abstraction with SOLID, Extending Abstraction with SOLID, and Programming Paradigms.  
Worked through all the additional resources provided in the 'Further Reading' section at the end of the DWA\_07.3 Programming Paradigms lesson.  
Completed DWA\_07: Challenge 1 and pushed code to the DWA7 GitHub folder.  
Save your 'Discussion Questions' answers to the DWA7 GitHub folder before your next code validation session with your coach.  
  
   
You are required to submit your answers to all the questions below, via the Knowledge Check template, before your next code validation session with your coach. Please follow these instructions:  
   
  
Open the DWA7 Discussion Questions template. Copy this file to your own Google Drive by clicking [File] > [Make a Copy].   
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your own details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Ensure your DWA7 Discussion Questions PDF is saved in the same repository as your code for this challenge.  
When you have pushed your code and discussion questions.pdf to GitHub please submit the link to your repository via the [Projects] tab on the LMS.  
  
   
To consider this module as “completed”, the above criteria need to have been met and demonstrated as done in your Code Review session.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_03.6 FreeCodeCamp Now that you have worked through the first 3 modules of this course, it's time to practice.  
   
To complete this module, please navigate to the freeCodeCamp ES6 course and complete all of the exercises up to and including “Import a Default Export”.  PLEASE NOTE: You are required to complete all the exercises in preparation for your next 1-on-1 code review session with your coach.  
   
ECMAScript, or ES, is a standardized version of JavaScript. Because all major browsers follow this specification, the terms ECMAScript and JavaScript are interchangeable.  
   
Most of the JavaScript you've learned up to this point was in ES5 (ECMAScript 5), which was finalized in 2009. While you can still write programs in ES5, JavaScript is constantly evolving, and new features are released every year.  
   
ES6, released in 2015, added many powerful new features to the language. In this course, you'll learn these new features, including arrow functions, destructuring, classes, promises, and modules.  
   
   
  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_06.1 Where are We? Abstraction simplifies complex systems by breaking them into manageable components. It streamlines code by hiding and elevating the essence of something while reducing the low-level noise. Becoming good at abstraction is the primary task of all software developers.  
   
   
  
   
00:00:00 Looking back at the course  
00:00:30 Where are we now?  
00:01:40 Combine low-level with high-level concepts  
00:02:10 Importance of mastering previous content  
00:02:40 Will focus on building projects  
00:03:40 Are you confident in your understanding?  
00:04:40 Talk to your coach about this new phase  
00:05:20 Intensity coming up  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_12.3 Project Submission DWA12 To complete this Knowledge Check, ensure you have worked through all the lessons in Module 12: Declarative Abstractions.  
   
Knowledge Check\_DWA12  
   
Please follow the instructions below to successfully submit the DWA12 Knowledge Check before your next code validation session with your coach.  
  
Open the Knowledge Check\_DWA12 template and copy the file to your own Google Drive by clicking [File] > [Make a Copy].  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Upload the PDF to the DWA12 Knowledge Check via the [Projects] tab  
  
   
During your 1-on-1 session with your coach, they will determine whether you understand the concepts covered in this module They may ask you questions about the concepts covered in this module or your responses submitted for this knowledge check. It is at the coach's discretion to determine what they will ask you and how deeply they require you to understand specific concepts.  
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_14.2 Rendering Component List    
  
   
   
00:00:00 Recap general framework concepts  
00:02:10 Benefit of using declarative UI  
00:02:50 One-way vs. Two-way binding  
00:04:20 What are "components"?  
00:04:50 What are component "props"?  
00:05:20 What does "lifting state" mean?  
00:06:10 Two versions of "open" state  
00:07:00 "Controlled" vs "Uncontrolled"  
00:08:20 Two ways to "lift" open state  
00:09:00 Using custom events in Lit  
00:10:15 Readding form submit logic  
00:10:40 Define custom event in component  
00:12:30 Create "save" custom event logic  
00:14:30 Dispatch event to parent  
00:15:50 Manually setting "open" state  
00:16:40 Passing function vs. calling function  
00:17:10 Prevent dispatch if invalid  
00:18:10 Create "td-task" component  
00:19:20 How to style LitElement?  
00:21:30 Passing attributes to "td-task"  
00:22:30 Add HTML inside Shadow DOM  
00:24:50 Adding fake task data  
00:25:40 Update "Store" actions  
00:26:40 ".map" to create tasks list  
00:27:30 Configure "completed" logic  
00:28:20 Add "sl-dialog" for task  
00:29:20 Update "reactive properties"  
00:30:30 Labels for "urgency" values  
00:31:00 "Shoelace" relative time utility  
00:31:30 Function to return Lit AST  
00:33:00 "Shoelace" date utility  
00:33:40 Remaining "sl-dialog" HTML  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_13.1 Using Lit-HTML This module will explore how Higher-Order functions are critical in modern JavaScript code. This is because they are the primary means by which we create and compose increasingly more powerful behaviour in our apps while still keeping the complexity, that comes with the scale of interaction in modern apps, under control.  
   
   
  
   
00:00:00 Recapping key concepts  
00:00:30 Concept #1: Global State Store  
00:01:40 Concept #2: Components  
00:02:20 Concept #3: Declarative UI  
00:04:30 Realities of modern software  
00:05:50 Higher-order functions are declarative  
00:06:50 Introducing "lit-HTML"  
00:07:20 "lit-HTML" vs "Lit Framework"  
00:08:20 Using "lit-HTML" via CDN  
00:10:30 Creating AST via "html" function  
00:12:20 Wrapping "html" inside a function  
00:12:50 "lit-html" uses string interpolation  
00:13:40 "render" converts AST into HTML  
00:14:10 Compares AST before update DOM  
00:15:40 "lit-HTML" only updates differences  
00:16:20 "HTML" and "hyperscript" are similar  
00:17:00 "libs" folder for third-party code  
00:17:45 Preventing "Prettier" from formatting  
00:18:20 Removing our custom abstraction  
00:19:20 Treating our app "view" as a function  
00:20:10 Our "view" will return a "lit-HTML" AST  
00:22:20 "lit-HTML" vs the previous approach  
00:22:50 Still need to register own event listeners  
00:24:30 Full JS Frameworks also handle events  
00:25:00 Add task to "Store" on submit  
00:25:40 Pass "Store" state directly to "View"  
00:26:50 Subscribing "View" to store changes  
00:27:20 Dynamic values in "View" output  
   
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_04.1 Core Concepts Part 1 This module will dive into code style and Styleguides to promote consistency, readability, and maintainability in our JavaScript code. Setting up and enforcing a specific Styleguide is critical to keeping our code complexity in check.  
   
Core Concepts: Part 1  
   
  
   
00:00:00 Introduction  
00:00:40 All functional things have a style  
00:01:20 Style determines how you experience something  
00:02:00 Similarities between human and computer languages  
00:02:00 Human language style guides go far back  
00:02:30 Can apply similar concepts to computer languages  
00:03:00 Code style should never be an accident  
00:04:00 Styleguides are found in various domains  
00:04:50 Stylistic predictability aids in reading and understanding  
00:05:20 Studies conducted into the influence of code style  
00:06:40 Forcing a style guide results in fewer bugs  
00:07:50 JavaScript highest ranking in benefits from Styleguides  
00:08:40 Code style is critical in JavaScript code  
00:09:20 Style decision #1: Code indentation  
00:10:00 JavaScript compiler does not care about indentation  
00:11:00 Two different ways to indent  
00:12:30 Controlling indentation in VS Code  
00:13:10 You should not manually indent spaces  
00:13:40 VS Code highlights and shows indentation  
00:14:20 Keyboard shortcut for increasing/decreasing indent  
00:15:00 "Tab" button is used when set to spaces too  
00:15:40 Very easy to change between indentation styles  
00:16:10 Tabs/Spaces is about the underlying character used  
00:17:30 Only difference is tab character is dynamic by default  
00:18:10 Most important point is to be consistent  
00:18:50 Stressing about which one is irrelevant  
00:20:00 Style decision #2: Line length  
00:20:30 Early on, we had to control line-breaks manually  
00:22:00 Influence of line-length on Readability  
00:23:30 Also, decisions around when to break lines  
00:24:20 Often break based on brackets and keywords  
00:25:30 It Also aids in reading methods and arguments  
00:26:40 Common line length differs by languages  
00:27:00 Style decision #3: Whitespace  
00:28:00 Using whitespace to communicate intent  
00:28:30 Example of arrays vs destructuring  
00:29:00 Style decision #4: Naming conventions  
00:30:00 Common character casing examples used in JS  
00:32:00 Style decision #5: Commenting  
00:32:00 Several different opinions on comments  
00:33:00 Agreement to avoid redundant comments  
00:33:30 Use common sense when writing comments  
00:34:00 Summary of code style  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_12.2 Declarative DOM Updates    
  
   
   
00:00:00 HTML as a result of a function  
00:01:10 Unidirection is important FP concept  
00:02:30 Separating state from HTML  
00:03:20 Currently relying on DOM mutations  
00:04:00 Updating DOM based on state  
00:04:50 Initial approach was manual changes  
00:05:40 Backbone.js focus on the MVC approach  
00:08:00 Angular.js focus on two-way binding  
00:09:30 React.js focus on FP unidirection  
00:10:30 React treats view as function output  
00:11:20 Example of "encapsulation" in React  
00:12:00 Example of "polymorphism" in React  
00:12:30 Popularity of React at the moment  
00:13:00 Example of declarative UI with "render"  
00:13:30 Basic declarative UI with ".innerHTML"  
00:14:15 Replace the entire view with each change  
00:15:10 Changing example to declarative  
00:16:30 Passing the entire store as a param  
00:17:10 ".map" higher-order function  
00:18:40 Replace all HTML in the example  
00:19:30 Basic declarative UI principles  
00:20:30 Up until now have done imperative UI  
00:21:10 Very basic declarative UI  
00:22:20 Declarative HTML bad performance  
00:23:00 Directly mutating the DOM is faster  
00:23:30 However, mutations are harder  to debug  
00:24:10 Problem with event listeners  
00:24:50 Problem with losing focus on render  
00:27:30 Frameworks abstract away mutations  
00:28:10 Declarative UI without drawbacks  
00:29:10 Express HTML as a result of a function  
00:29:40 Frameworks abstract away the DOM  
00:30:10 Understanding what gets abstracted  
00:30:40 Higher-order functions (HOF)  
00:31:10 HOF tend to be more declarative  
00:33:10 HOF keep data unidirectional  
00:34:10 I will focus on frameworks soon  
00:34:50 Frameworks' most important abstraction  
   
   
 Further reading Please work through the following resources to further explore the concepts covered in this module:  
   
          - WATCH: STOP LYING, The truth about Software Engineering by ThePrimeagen  
          - WATCH: ReactJS Basics - #9 How does ReactJS update the DOM? by Academind  
          - WATCH: Imperative vs Declarative Programming by uidotdev  
          - WATCH: Declarative APIs in an Imperative World by InfoQ  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_08.4 Project Submission DWA8 Project\_DWA8  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Creating Encapsulation and Working with Encapsulation.  
Worked through all the additional resources provided in the 'Further Reading' section at the end of DWA\_08.2 Working with Encapsulation lesson.  
Completed DWA\_08: Challenge 1 and pushed code to the DWA8 GitHub folder.  
Save your 'Discussion Questions' answers to the DWA8 GitHub folder before your next code validation session with your coach.  
  
   
Please follow these instructions:  
  
Open the DWA8 Discussion Questions template. Copy this file to your own Google Drive by clicking [File] > [Make a Copy].  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your own details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Ensure your DWA8 Discussion Questions PDF is saved in the same repository as your code for this challenge.  
When you have pushed your code and discussion questions.pdf to GitHub please submit the link to your repository via the [Projects] tab on the LMS.  
  
   
You will have a call with your coach, in which they will determine whether you understand the concepts covered in this module. They might ask you questions related to the concepts covered in this module or in your responses to the discussion questions. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_04.2 Core Concepts Part 2 Core Concepts: Part 2  
   
  
   
00:00:00 Introducing Styleguides  
00:40:00 First recorded coding Styleguide  
00:01:30 Type #1: Officially endorsed Styleguides  
00:02:00 Example: PEP8 in Python  
00:02:30 Type #2: Third-party Styleguide  
00:03:00 Example: Google third-party Styleguide  
00:03:30 Official style guide might not be the most popular  
00:04:00 Example: Ruby third-party Styleguides  
00:04:30 Type #3: community-driven Styleguides  
00:05:00 There is no official JavaScript Styleguide  
00:05:40 Mentioning resources that we won't cover  
00:07:30 Styleguides should be easily testable  
00:08:00 Styleguide focus on decisions beyond only bugs  
00:08:30 Three most popular JavaScript style guides  
00:09:00 Option #1: Google JS Styleguide  
00:09:40 Option #2: Standard JS Styleguide  
00:10:10 Option #3: Airbnb style guide  
00:10:40 Current lesson exercise instructions  
00:11:50 Airbnb style guide is extremely popular  
00:12:40 Overview of linting as a concept  
00:13:00 Very first software linter created  
00:13:40 Almost all modern languages have linters  
00:14:30 Overview of JSLint  
00:15:30 JSHint was created as an alternative to JSLint  
00:16:15 ESLint extended customizability further  
00:16:50 When running ESLint shows all problems in the code  
00:17:00 ESLint distinguishes between errors and warnings  
00:18:00 VSCode plugin for ESLint that highlights as you type  
00:18:30 Example of using ESlint inside VS Code  
00:19:00 ESLint links to documentation of specific rules  
00:19:40 Rule also shows what is expected  
00:20:45 Setting rules in the ESLint configuration file  
00:21:10 Showing basic ESLint built-in rules  
00:21:50 Installing NodeJS on your machine  
00:22:50 Explaining NodeJS LTS vs Current  
00:24:35 Checking NodeJS version is correct  
00:25:20 Checking NPM installed  
00:26:00 Using terminal inside VS Code  
00:26:00 Installing ESLint via NPM  
00:28:00 Remember running "npm init"  
00:29:00 Cycling through previous commands  
00:29:30 Basic ESLint configuration steps  
00:30:20 Selecting Airbnb Styleguide  
00:31:00 Automatically created ESLint files  
00:31:30 Creating ".gitignore" file  
00:32:00 Ignoring "node\_modules" folder  
00:32:40 Creating a basic project structure  
00:33:40 Confirming our code works  
00:41:10 Installing VS Code plugin for ESLint  
00:34:50 Following a link to a specific rule page  
00:35:30 How to disable specific ESLint rule  
00:37:40 How to override a specific area  
00:38:30 Overriding only a single line  
00:38:40 Overriding only a specific rule  
00:38:10 Overriding multiple lines  
00:40:20 View all ESLint errors in a single place  
00:41:10 Overview of Prettier  
00:42:40 Creating Prettier config file  
00:42:10 Example of how Prettier works  
00:43:00 Installing Prettier via NPM  
00:44:10 Integrate Prettier into ESLint  
00:44:40 Running "format document"  
00:45:10 Setting Prettier as default formatter  
00:46:00 ESLint now relies on Prettier to auto-fix  
00:46:20 Overriding config in Prettier  
00:46:40 Auto-run Prettier when saving a file  
00:47:00 Overview of VS Code Settings area  
00:47:30 Toggling "Editor: Format On Save"  
00:48:00 Recapping ESLint and Prettier  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_07.3 Programming Paradigms    
  
   
00:00:00 Basic Overview  
00:01:20 Discussing JavaScript "Class"  
00:02:30 "Clean Architecture" Book  
00:04:10 What are "Programming Paradigms"?  
00:05:40 Paradigms say how to create abstractions  
00:06:50 What is "Procedural Programming"?  
00:08:00 What is a "GOTO" statement?  
00:09:00 Procedures hide "GOTO" logic  
00:09:30 Encourages nesting functions as an abstraction  
00:10:30 What is "Object Oriented Programming" (OOP)?  
00:11:30 OOP used to be the primary paradigm for a long time  
00:12:00 However, OOP struggles with concurrency  
00:13:00 Functions as constructors for objects  
00:13:50 All paradigms remove some functionality  
00:14:40 OOP doesn't treat behaviour as instructions  
00:15:20 OOP abstracts code according to "things"  
00:15:50 All behaviour is handled through "interfaces"  
00:16:30 Can not directly access inside from outside  
00:17:10 What is "Functional Programming" (FP)?  
00:17:40 FP is based on mathematic principles  
00:18:10 OOP combines data and behaviour  
00:19:20 FP keeps data and behaviour separate  
00:19:50 OOP and FP remove different functionality  
00:20:30 FP says the problem is data changing  
00:21:30 The web is getting more decentralised  
00:22:20 FP mitigates race-conditions  
00:23:10 OOP makes it hard to split up behaviour  
00:23:50 Type of problem depends on what you are building  
00:25:10 JavaScript is becoming more FP-like  
00:26:00 You always need to do both OOP and FP  
00:26:50 FP says abstractions should avoid changing  
00:28:40 In FP, behaviour is always predictable  
00:29:00 Abstractions are immutable in FP  
00:29:40 Recapping paradigms  
00:30:00 Example of OOP approach  
00:30:50 Example of FP approach  
   
   
 Further reading To further unpack the concepts covered in this module, please work through the following resources:  
   
          - WATCH: Object Oriented vs Functional Programming with TypeScript by Fireship  
          - WATCH: Functional, Procedural & Object-oriented Programming - An Overview by Academind  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_06.3 Example with JavaScript    
  
   
00:00 Abstraction as process  
00:50 Abstraction is about elevating the essence  
01:20 Example of creating a UUID in JavaScript  
02:00 Abstraction is about removing low-level noise  
02:30 Example of function as an abstraction  
03:00 Making use of abstractions  
03:30 Creating objects as an abstraction  
04:40 Adding another level of abstraction  
05:20 Abstractions should hide the unimportant parts  
05:50 Good abstractions should be composable  
06:30 Further combining levels of abstractions into new abstractions  
07:30 When composed, abstractions should be expressive  
08:00 Using props also adds expressiveness  
08:30 Abstractions are only helpful if they are documented  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_09.4 Project Submission DWA9 Project\_DWA9  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Classes and Prototypes and Web Components.  
Worked through all the additional resources provided in the 'Further Reading' section at the end of the DWA\_09.2 Web Components lesson.  
Completed DWA\_09: Challenge 1 and pushed code to the DWA9 GitHub folder.  
Finally, submit the link to your DWA\_09 Challenge Solutions via the [Projects] tab > DWA9 Submission  
  
  
To consider this module as “completed”, the above criteria need to have been met and demonstrated as done in your Code Review session.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_06.4 Reading Material The process of abstraction helps us structure our code according to our built-in human ability to think, reason and interact with complex things without having to understand all the parts that make a thing.  
   
Please take a moment and look around the room you are in. You might notice a couple of different things:  
  
Computer  
Cellphone  
Desk  
Chair  
Door  
Window  
  
   
You most likely take for granted how good your brain is at identifying an item, even if this is the first time you see that specific instance of an object. This is because our brains are good at understanding the world around us by grouping related low-level complex parts into much simpler “things”.  
   
Take, for example, your computer; it consists of a range of physical parts from your keyboard, processor, motherboard, logic gates, circuit boards and even electron electrons that flow through all these. Furthermore, it also consists of several non-physical graphics shown on the screen from your cursor, buttons, taskbars and text inputs.  
   
All this complexity is grouped under the simple concept known as a “computer”, which can have several properties.  
  
Switched on or off  
Processing, idle or even frozen  
Fast or slow  
  
   
We haven’t even considered more permanent properties like, whether it is a cheap, expensive laptop, desktop, PC, Mac, gaming or work computer.  
   
There are also some things you can do with a computer:  
  
Browse the internet  
Write an essay  
Shop on Takealot  
Install a game  
Design and print a flyer  
Coding  
  
   
Not only do our brains automatically take a group of related low-level complex parts and group them into simpler “things”,. You might even be curious how your brain instinctively “knows” what something is without having to inspect every element it is made from in detail until you can conclude, “I think this is maybe a door”.  
   
A well-known American Psychologist, Oliver Sacks, wrote a book comically titled The Man Who Mistook His Wife for a Hat (1985) that tells a real-life story of a condition called visual agnosia in which a man lost his ability to reason about “objects” in a manner that doesn’t require considering every single part that makes up the whole. In the story, after leaving therapy, there is a brief moment when the man instinctively cannot tell the difference between his wife and his hat without carefully considering each in great detail.  
   
This ability to instinctively think and talk about “objects”, whether in programming or real life, is a process of what is called “abstraction”. Abstraction is our ability to simplify entities to a core set of properties and interactions without understanding the whole.  
   
Furthermore, you can identify general parts that make up these items.  
  
Screen  
Keyboard  
Mouse or Touchpad  
  
   
Each of these might further be broken down into sub-items; for example, a keyboard might consist of the following:  
  
Enter Key  
Escape Key  
Arrow Keys  
Spacebar  
  
   
Furthermore, each of these keys has parts that can be further reasoned about until we get down to the physical atoms that make up the physical material these are made of (and even beyond that into the realm of quantum physics).  
   
However, despite the reservations you might have about the “simplicity” of the above, it repeatedly comes up time and time again as the essential part of modern-day software development. Furthermore, often the ability to meaningfully abstract code is what separates bad software developers from good software developers.  
   
Unfortunately, most developers often only learn this later on in their careers. In his seminal book, titled “Composing Software (2018)”, Eric Elliot remarks that:  
   
One of my biggest regrets in life is that I failed to understand the significance of that lesson early on. I learned the essence of software design far too late in life. I have interviewed hundreds of developers. What I’ve learned from those sessions is that I’m not alone. […] If you’re a software developer, you compose functions and data structures every day, whether you know it or not. You can do it consciously (and better), or you can do it accidentally, with duct-tape and crazy glue. The process of software development is breaking down large problems into smaller problems, building components that solve those smaller problems, then composing those components together to form a complete application.  
   
   
 Further reading Please work through the following resources to further enhance your understanding of the concepts discussed in this module.  
   
          - Medium.com: Abstraction & Composition  
          - Watch: The Art of Abstraction - Computerphile by Computerphile  
          - Watch: What is abstraction in programming? by IAmDev Grant  
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_03.3 JSDoc Crash Course In order to understand the concepts covered in this module, please work your way through Documenting Your JavaScript | JSDoc Crash Course by Traversy Media.  
   
In the video below, Brad Traversy goes over JSDoc for documenting your JavaScript code as well as using it for type checking. You can find the code used in the video in this GitHub repository.   
   
  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_01.1 Core Concepts Part 1 Welcome to the first lesson on "Structuring JavaScript."  
   
In this lesson, we will take a high-level view of questions about how JavaScript should be structured when it forms part of a larger piece of software. So that you know, there is no reading material. Instead, you are required to watch two videos. So please pay attention and think through these ideas in detail since all lessons from this point onwards will constantly revisit these concepts.  
   
You don't need to do a technical challenge. Instead, you will have to talk with your coach about the concepts covered in the videos to confirm that you understand how to structure your JavaScript code.  
   
Core Concepts Part 1  
  
   
00:00:00 Introduction  
00:00:40 Reflecting on JavaScript Fundamentals  
00:02:00 Getting from fundamentals to full-on software  
00:02:30 Scale complexity incrementally by using composition  
00:04:00 Importance of mid-level code structures  
00:05:00 Comparing JavaScript to rocket science  
00:08:00 Thinking about software as workable material  
00:08:00 Software often fails catastrophically  
00:09:00 Programming requires patience regardless of skill  
00:10:30 First ever recorded computer bug in history  
00:11:30 It is impossible to create bug-free code  
00:12:30 Mid-level bugs are much harder to find and fix  
00:13:30 Getting code to work is the easy part  
00:14:30 Keep the focus on what is really hard in programming  
00:15:30 We can only equip you to walk this journey  
00:16:00 Talking about your career path  
00:16:30 Fundamentals is not enough  
00:17:20 Strive towards thinking about architecture  
00:18:00 Understand both fundamentals and architecture  
00:19:30 Building a programming career takes a long time  
00:20:10 Programming is complex, scary and frustrating  
00:21:40 Extreme focus on precision and accuracy  
00:23:20 Overwhelming amount of moving parts and people  
00:24:10 Should write that is readable by other people  
00:24:50 Software has an extremely long lifecycle  
00:26:45 Introducing four stories to illustrate concepts  
00:27:10 First example: Ariane 5 Rocket Incident  
00:28:20 Very rare for codebases to be rewritten entirely  
00:29:00 Problem was caused by automatic coercion of a value  
00:30:00 What Ariane 5 code might have looked like  
00:31:20 Hidden bugs more dangerous than obvious bugs  
00:32:20 Second example: Mars Climate Orbiter  
00:34:00 Problem was an incorrect assumption about the code  
00:34:40 What code might have looked like  
00:35:00 Real-world use of AI in day-to-day coding  
00:36:30 Third example: Knight Capital Incident  
00:38:40 Fourth example: Facebook DNS Incident  
00:40:20 Not obvious how complex maintaining software is  
00:41:00 It is impossible to build fail-proof software  
00:42:00 Even when less severe, there are still implications  
00:43:00 Mentioning the "Eloquent JavaScript" book  
00:44:00 Getting confused by your own code  
00:45:00 Complexity is the most critical problem in coding  
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_04.3 Knowledge Check\_DWA4 To complete this Knowledge Check, ensure you have worked through all the lessons in Module 4: Code Style and submit your responses to the following two items:  
  
1. Select three rules from the Airbnb Style Guide that you find useful and explain why.  
2. Select three rules from the Airbnb Style Guide that you find confusing and explain why.  
  
   
Knowledge Check\_DWA4  
   
Please follow the instructions below to successfully submit the DWA4 Knowledge Check before your next code validation session with your coach.  
   
  
Open the Knowledge Check\_DWA4 template. Copy this file to your own Google Drive by clicking [File] > [Make a Copy].  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your own details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Now save the [PDF Document] to the DWA4 GitHub folder.  
Submit the link for the DWA4 Submission via the [Projects] tab.  
  
   
You will have a call with your coach, in which they will determine whether you understand the concepts covered in both lectures, Core Concepts 1 and Core Concepts 2. They might ask you questions related to the concepts covered in this module or in your responses submitted for this knowledge test. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts. For this session, your coach will select a couple of Airbnb rules at random which you will have to discuss.  
  
  
To ensure you have a constructive code validation session, please ensure that you have covered all the content provided in the module and come prepared for your coaching session.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_01.2 Core Concepts Part 2 Core Concepts Part 2  
  
   
   
00:00:00 Revisiting Mars Climate Orbiter code  
00:00:30 Concept of code style and style guides  
00:01:00 JavaScript Standard Styleguide  
00:01:40 The Airbnb Styleguide  
00:02:30 Find the JS one, not the React/JSX one  
00:02:50 Example of function-related styles  
00:03:40 Even if disagree, style guides are invaluable  
00:04:00 Example of variable-related styles  
00:04:30 How do you name variables and functions  
00:05:00 Ensure variables visually look different at a glance  
00:06:20 Include extra information in variable names  
00:07:30 Avoid vague or ambiguous variable names  
00:08:00 Ensure all related code is grouped together  
00:08:30 Attempt to have code readable as plain English  
00:09:00 Use upper snake case for global constants  
00:09:30 Aim for expressiveness even if the code is longer  
00:10:00 Ensuring data is correct before running  
00:10:30 Group related data together in object literals  
00:11:10 Passing single object or multiple arguments  
00:12:00 Add checks to prevent code from running  
00:12:30 How strict checks depend on context  
00:14:00 Example of attempting run with wrong data  
00:14:30 With experience it is easier to gauge strictness  
00:15:00 Preventing incorrect code from running  
00:16:00 Comments to provide additional context  
00:17:00 Describing the props shape with JSDoc  
00:17:30 Use dot notation to indicate nested values  
00:18:40 Quotes in JSDoc to indicate literal values  
00:19:30 Including auto-conversion in our function  
00:20:00 Object properties are also more readable  
00:22:00 Checking if the value is one of the possible outcomes  
00:23:00 Scale strictness with the importance of operation  
00:23:50 Reviewing and recapping changes to code  
00:24:20 Using code styles to increase readability  
00:25:00 Describing shapes and behaviour with JSDoc  
00:25:40 Aiming for modularity and flexibility of code  
00:27:10 Introducing the idea of abstraction  
00:27:30 Abstraction at the core of managing complexity  
00:28:00 Abstraction and composition are used together  
00:28:40 Only expose what is needed from the abstraction  
00:30:00 Abstractions make use of interfaces  
00:30:30 Unable to avoid complexity in software  
00:31:00 Complexity due to evolving requirements  
00:31:30 Managing your technical debt  
00:32:20 Code changes as software scales  
00:33:00 Developers spend time maintaining software  
00:34:00 Introducing programming paradigms  
00:34:00 Introducing JavaScript frameworks  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_10.4 Project Submission DWA10 Project\_DWA10  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Introducing Polymorphism and Component Slots.  
Worked through all the additional resources provided in the 'Further Reading' section at the end of the DWA\_10.2 Component Slots lesson.  
Completed DWA\_10: Challenge 1 and pushed code to the DWA10 GitHub folder.  
Finally, submit the link to your DWA\_10 Challenge Solutions via the [Projects] tab > DWA10 Submission  
  
  
To consider this module as “completed”, the above criteria need to have been met and demonstrated as done in your Code Review session.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_09.2 Web Components In this lecture, we will explore using what we've learnt about inheritance to build custom web components. Then, we can use these custom components in our websites, just like the standard ones.  
   
   
  
   
00:00 Recapping previous content  
01:00 Rewriting "createTask" as a class  
02:00 Setting properties to private  
03:00 Adding JSDoc inside the class itself  
03:40 Overriding ESLint warning  
04:30 Confirming changes in the browser  
05:30 Example of extending HTMLElement  
05:40 "connectedCallback" method  
06:30 Introducing "custom elements"  
07:40 Example of the "shadow DOM"  
08:50 Separate "modules" and "components"  
09:30 Creating HTML "template" element  
10:40 Inline styles in shadow DOM  
11:10 CSS only applied to shadow DOM  
11:40 Remember to set "box-sizing"  
12:20 Why "custom elements" require dash "-"  
14:00 How to create a shadow DOM  
14:30 Assign shadow DOM to property  
15:30 Attaching template to shadow DOM  
16:30 Passing attributes into component  
17:50 Cloning template in shadow DOM  
18:50 Placing class inside the custom element  
20:00 Handling attributes inside a component  
21:10 Binding attributes to shadow DOM  
23:10 Extending "getHTML" for shadow DOM  
24:00 Adding JSDoc types inside a class  
25:10 Why does the listener not catch the error?  
25:50 Creating elements programmatically  
26:40 Devtools allows a peek into Shadow DOM  
27:10 Actual JavaScript code can't access  
27:40 Creating "task-adding" template  
28:50 Configuring "task-adding" class  
30:50 Adding open "getters" and "setters"  
34:00 Setting "public" value for side-effects  
34:40 Using event bubbling to replace callback  
35:40 Passing the "open" attribute to the component  
36:30 Ensure shadow DOM already exists  
37:30 HTML elements inside Shadow DOM  
38:30 Connecting custom elements in "script.js"  
39:40 Export class to do "instanceof" check  
41:00 Add event listener to a custom element  
41:30 Creating a new custom HTML event  
42:10 Dispatching the custom event to DOM  
43:40 Conclusion  
   
   
 Further reading Please explore the following resources to understand better the concepts covered in this module.  
   
          - READ: How do Classes Work in JavaScript?  
          - WATCH: Classes in JavaScript with ES6 - p5.js Tutorial by The Coding Train  
          - READ: Web Components Are Easier Than You Think  
          - WATCH: A Complete Introduction to Web Components by Kinsta  
   
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_06.6 Project Submission DWA6 Project DWA6  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Where are We?, Abstraction as a Concept?, and Example with JavaScript.  
Worked through DWA\_06.4 Reading Material and completed the 'Further Reading' at the end of this lesson.  
Completed DWA\_06: Challenge 1 and pushed code to the DWA6 GitHub folder.  
Finally, submit the link to your DWA\_06 Challenge Solutions via the [Projects] tab > DWA6 Submission.  
  
   
To prepare for your code review session with your coach, ensure you can talk through your code and demonstrate your understanding of abstraction, before booking your 1-on-1 session.  
   
To consider this module as “completed”, the above criteria need to have been met and demonstrated as done in your Code Review session.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_01.3 Knowledge Check\_DWA1 Now that you have worked through both pre-recorded lessons, (hopefully, replayed sections for better understanding, made notes, and come to grips with the concepts covered) take some time to work through the following questions:  
  
1. Why is it important to manage complexity in Software?  
2. What are the factors that create complexity in Software?  
3. What are ways in which complexity can be managed in JavaScript?  
4. Are there implications of not managing complexity on a small scale?  
5. List a couple of codified style guide rules, and explain them in detail.  
6. To date, what bug has taken you the longest to fix - why did it take so long?  
  
   
Knowledge Check\_DWA1  
   
Please submit your answers to all the questions, using the Knowledge Check template. Please follow the instructions below:  
  
Open the Knowledge Check\_DWA1 template. Copy this file to your own Google Drive by clicking [File] > [Make a Copy]. (If you do not have a Google account, you can sign up for one here.)  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Now save the [PDF Document] to the DWA1 GitHub folder.  
Finally, submit the GitHub link to your  DWA1 Knowledge Check via the [Projects] tab.  
  
   
To prepare and complete this submission, please ensure you have worked through the lectures, Core Concepts 1 and Core Concepts 2, and any additional resources. You will then be required to complete and submit the Knowledge Check, after which you will be given the opportunity to discuss your findings with some of your peers.   
  
This will prepare you for the group coaching session, where you will share your understanding of the programming concepts with the group, and complete your code validation.   
  
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_03.4 Knowledge Check 1 Now that you have completed "Documentation", it is time to review and verify your understanding of all the concepts covered in this module.  
   
For this Knowledge Check - DWA3.1 - you will be required to illustrate a good understanding of all the concepts covered in the two lectures, Core Concepts Part 1 and Core Concepts Part 2, as well as the provided reading material and additional JSDoc Crash Course.   
   
The DWA3.1 Submission requires you to respond to the following:  
  
1. Please show how you applied a Markdown File to a piece of your code.  
2. Please show how you applied JSDoc Comments to a piece of your code.  
3. Please show how you applied the @ts-check annotation to a piece of your code.  
4. As a BONUS, please show how you applied any other concept covered in the 'Documentation' module.  
  
   
To prepare for your next coaching code review session, please take a piece of code from any part of the IWA JavaScript Fundamentals (2023) or this course and apply the concepts learned to any piece of code that you have written thus far, as part of the JavaScript component of the Software Development Program.  
  
Open the Knowledge Check\_DWA3.1 template. Copy this file to your own Google Drive by clicking [File] > [Make a Copy].  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your own details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Now save the [PDF Document] to the DWA3.1 GitHub folder.  
Submit the link for the DWA3.1 Submission via the [Projects] tab  
  
   
Please Note: It is vital that you focus more on the WHY and HOW of what you did, rather than what you did.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_09.1 Classes and Prototypes This module will look at built-in prototypal chaining on which JavaScript classes are built. We will specifically look at how prototypes are used in the JavaScript DOM and utilise this functionality to extend the HTML. This gives us the means to create custom, fully-encapsulated HTML elements.  
   
In the first lecture, we will look at how to use classes and the concept of inheritance to create objects with defined structure and behaviour quickly.  
   
  
   
00:00:00 Introduction  
00:00:50 Example of "encapsulation" benefit  
00:02:00 Re-adding error event listener  
00:03:00 What are JavaScript "class" keywords?  
00:03:50 What is "syntactic sugar"?  
00:04:30 Creating an example of a JavaScript class  
00:06:00 Using "new" with JS Classes  
00:06:30 Classes require the usage of "this"  
00:07:00 Hiding class properties with "#"  
00:07:30 "constructor" method in JS Classes  
00:08:20 How to use JSDoc with JS Classes  
00:10:00 Historically, "classes" in OOP are the interface  
00:10:30 Can still wrap JS class in the factory function  
00:11:20 Setting "@param" on the constructor method  
00:11:50 Unless made private, it is public by default  
00:12:10 Examples of built-in JS Classes  
00:12:40 Some classes do not have constructors  
00:13:10 Example of "static" properties in classes  
00:14:00 "setters" and "getters" in JS Classes  
00:14:30 "setters" and "getters" need a unique name  
00:15:00 Classes are generally harder to split up  
00:15:30 Misconception that inheritance is the core of OOP  
00:16:00 "Favour composition over inheritance"  
00:16:30 Avoid inheritance as far as you can  
00:17:00 How does inheritance work?  
00:17:30 Using "extends" with JS classes  
00:18:00 Example of the inheritance hierarchy  
00:19:30 Example of how properties are inherited  
00:20:00 Inheritance creates extremely tight coupling  
00:20:30 Example accidentally overriding behaviour  
00:21:30 Inheritance hides too many things  
00:22:00 "Mixins" as an alternative to inheritance  
00:23:00 Focus on "has" or "can" instead of "is"  
00:24:00 Side note about lexical scope in methods  
00:24:30 Example of "mixin"-based composition  
00:25:20 Composition more flexible than inheritance  
00:25:50 Problem of "inheritance" in real-world  
00:26:40 Unable to always avoid inheritance  
00:27:10 Consistency is always the most important  
00:27:50 JavaScript DOM is frustrating to work with  
00:28:30 Classes and factory functions closer than think  
00:29:00 JS classes are not the same as other language classes  
00:29:30 What are JavaScript "prototypes"?  
00:30:20 Example of DOM "prototypal chain"  
00:31:20 Everything in JavaScript is based on "Object"  
00:31:40 Prototypal chain in MDN documentation  
00:32:20 JavaScript classes can only extend one prototype  
00:33:10 JavaScript does not have "implements"  
00:33:50 JavaScript does not have "abstract"  
00:34:30 What happens under the hood of JS class?  
00:35:30 JS class use function prototype  
00:36:00 We assigned manually before JS Classes  
00:36:40 "Prototypes" do not actually "inherit"  
00:37:20 "Prototypes" are merely fallbacks  
00:37:50 Can introduce bugs that are hard to find  
00:38:20 Example of prototypes in DOM  
00:39:20 JS prototypes are dangerous  
00:41:00 Fallbacks can update after the instance is created  
00:42:20 In the early days of JS, this was abused  
00:42:50 Today this is considered bad practice  
00:43:20 Example of why it is dangerous  
00:45:00 Has too much hidden behaviour  
00:45:30 Why might seem useful at first  
00:46:20 Early JS libraries extended prototypes  
00:47:00 Eventually realised extending prototypes is bad  
00:48:30 Prototypes still hidden under classes  
00:50:30 Classes syntax does not expose prototypes directly  
00:51:00 Yet JS Classes are not true classes  
00:51:30 JS Classes closer to factory functions  
00:51:00 Web Components require DOM prototypes  
   
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_06.2 Abstraction as a Concept?    
  
   
00:00:00 Contextualising discussion  
00:01:00 Understanding upcoming content  
00:01:30 Abstraction is hard to understand  
00:02:00 Connecting low-level and high-level concepts  
00:02:40 Abstraction in the World of Mathematics  
00:03:40 If we did not have a mathematic abstraction  
00:05:20 It is problematic precisely because you are used to it  
00:05:40 Algebra is built on the idea of abstractions  
00:06:10 Abstractions are useful when composed  
00:07:00 We have a Practical example of "Gestalt Psychology."  
00:09:00 Letters, words and sentences are abstractions  
00:10:00 Abstraction is an essential concept in programming  
00:10:30 A practical example of "Chunking."  
00:12:30 Abstraction explained through cooking  
00:14:10 Good abstractions provide means to learn concepts  
00:15:30 Great abstractions are seen as "common sense."  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_12.1 Creating a UI Framework When restricting ourselves to immutability, pure functions and separation of data from behaviour, we can write code more declaratively. This helps us focus on the "what" rather than the "how”, which significantly reduces complexity, increases readability and makes our code easier to debug.  
   
   
  
   
00:00:00 Introduction  
00:00:30 Need both FP and OOP  
00:01:30 Abstracting UI components  
00:02:40 Creating class inside closure  
00:03:10 Check if there is a hyphen in the element name  
00:03:40 Register element inside closure  
00:04:10 Shadow DOM in abstraction  
00:04:40 HTML template in abstraction  
00:05:10 Event listeners in abstraction  
00:05:50 Store subscription in abstraction  
00:06:40 Creating a JavaScript Framework  
00:07:30 Good exercise in re-creating tools  
00:08:30 Framework's most important abstraction  
00:09:30 Be careful when using AI tools  
00:10:40 Adding listeners in "connectedCallback"  
00:11:20 Removing "disconnectedCallback"  
00:12:20 Remove even if not needed  
00:13:30 "Containers" vs "Components" pattern  
00:14:40 Working example of our framework  
00:15:50 Adding interactivity to our example  
00:17:00 Add means to get HTML elements  
00:18:50 Passing "getHTML" into callbacks  
00:19:30 Wrapper around the event handler  
00:21:00 NodeList coercion helper function  
00:21:40 Array to remove event listeners  
00:22:20 Relax if you do not understand at all  
00:23:30 Handling form submission  
00:24:00 Overriding ts-check for a few lines  
00:24:50 Convert form response to an object  
00:25:20 Type-checking requires more work  
00:26:30 Add submission to the "store"  
00:27:10 Structured approach takes longer  
00:28:00 Dispatch action to "store"  
00:28:30 Update HTML imperatively  
00:29:20 Find tasks created in the change  
00:29:50 Check which is not in the previous state  
00:30:10 Check which is not in the next state  
00:32:30 Is there a better way?  
00:33:00 Understand the problem first  
00:34:30 Learn mechanics under abstractions  
00:34:00 Using abstractions without knowing  
00:34:30 "The Law of Leaky Abstractions"  
00:35:20 Low-level TCP mechanics with metaphor  
00:36:00 Explain the TCP abstraction itself  
00:37:30 Abstractions aim to hide details  
00:38:10 All abstractions sometimes "leak"  
00:39:00 Example of ASP.NET as a leaky abstraction  
00:39:30 Learn mechanisms under abstractions  
00:40:40 Reason #1: Required to debug  
00:43:50 Reason #2: Discuss your code  
00:45:00 Reason #3: Compare abstractions  
00:46:20 Reason #4: Learn from incorrect choices  
00:47:10 All abstractions have trade-offs  
00:49:00 Understand mechanics to use abstraction better  
00:49:30 Better abstractions make learning harder  
00:50:30 A lot of developers are stuck in junior positions  
00:52:00 Find similar opinions and keep this in mind  
00:53:00 Next video declarative DOM abstractions  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_13.6 Project Submission DWA13 Project DWA13  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Using Lit-HTML, Utility Functions, Partial Application and Built-in HOF  
Worked through the 'Further Reading' at the end of the Built-in HOF lesson.  
Completed DWA\_13: Challenge 1 and pushed your code to the DWA13 GitHub folder.  
Finally, submit the link to your DWA\_13 Challenge Solutions via the [Projects] tab > DWA13 Submission.  
  
   
To consider this module as “completed”, the above criteria must have been met and demonstrated as done in your Code Review session.  
   
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_14.5 Project Submission DWA14 Project DWA14  
   
To successfully meet the outcomes of this module, you are required to:  
  
Have worked through both pre-recorded lectures, Entire Lit Framework, Rendering Component List and Explicit States.  
Worked through the 'Further Reading' at the end of the Explicit States lesson.  
Completed DWA\_14: Challenge 1 and pushed your code to the DWA14 GitHub folder.  
Finally, submit the link to your DWA\_14 Challenge Solutions via the [Projects] tab > DWA14 Submission.  
  
   
To consider this module as “completed”, the above criteria must have been met and demonstrated as done in your Code Review session.  
   
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_13.5 Challenge 1 To meet the outcome requirements of this module, please complete the following exercises (each with corresponding data).  
   
const provinces = ['Western Cape', 'Gauteng', 'Northern Cape', 'Eastern Cape', 'KwaZulu-Natal', 'Free State']  
const names = ['Ashwin', 'Sibongile', 'Jan-Hendrik', 'Sifso', 'Shailen', 'Frikkie']  
   
  
Use forEach to console log each name to the console. You are allowed to call console.log seven times.  
Use forEach to console log each name with a matching province (for example Ashwin (Western Cape). Note that you are only allowed to call console.log seven times.  
Using map loop over all province names and turn the string to all uppercase. Log the new array to the console.  
Create a new array with map that has the amount of characters in each name. The result should be: [6, 9, 11, 5, 7, 7]  
Using toSorted to sort all provinces alphabetically.  
Use filter to remove all provinces that have the word Cape in them. After filtering the array, return the amount of provinces left. The final value should be 3  
Create a boolean array by using map and some to determine whether a name contains an S character. The result should be [true, true, false, true, true, false]  
Using only reduce, turn the above into an object that indicates the province of an individual. In other words:  
  
   
{  
 Ashwin: 'Western Cape',  
 Sibongile: 'Gauteng',  
 'Jan-Hendrik': 'Northern Cape',  
 Sifso: 'Eastern Cape',  
 Shailen: 'KwaZulu-Natal',  
 Frikkie: 'Free State',  
  
   
Below are additional exercises. However note that in all the following exercises all code should be written inside the brackets of a single console.log , with the final result being logged to the console. This means that your code will probably look something as follows:  
   
console.log(  
 // Your code here  
)  
   
See the data and exercises below. As per the above, you will be required to do the following and explain the why and how to your coach in your assessment.  
   
const products = [  
 { product: 'banana', price: "2" },  
 { product: 'mango', price: 6 },  
 { product: 'potato', price: ' ' },  
 { product: 'avocado', price: "8" },  
 { product: 'coffee', price: 10 },  
 { product: 'tea', price: '' },  
]  
   
  
Use forEach to console.log each product name to the console.  
Use filter to filter out products that have a name longer than 5 characters  
Using both filter and map. Convert all prices that are strings to numbers, and remove all products from the array that do not have prices. After this has been done then use reduce to calculate the combined price of all remaining products.  
Use reduce to concatenate all product names to create the following string: banana, mango, potato, avocado, coffee and tea.  
Use reduce to calculate both the highest and lowest-priced items. The names should be returned as the following string: Highest: coffee. Lowest: banana.  
Using only Object.entries and reduce recreate the object with the exact same values. However, the following object keys should be changed in the new array:  
   
product should be changed to name  
price should be changed to cost  
  
  
  
   
In your 1-on-1 session with your coach, you will be required to demonstrate your understanding of all concepts covered in this module. Please Note: It is your understanding of the HOW and WHY that your coach will be trying to establish.  It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_03.2 Core Concepts Part 2 Core Concepts: Part 2  
   
  
   
   
00:00 Introduction  
00:40 Process can not be 100% split into phases  
01:00 BDD is not about doing more planning  
01:50 Start with concrete example not abstract  
02:00 Adding code-blocks to Figjam files  
02:30 Storing tasks in object instead of array  
03:00 Ensuring task keys are always unique  
03:50 Example of unique identifier value  
04:20 Universal Unique Identifiers (UUID)  
05:30 Including key in object itself as well  
06:00 Adding remaining properties to tasks  
07:00 Adding "displaying" and "filters" to state  
08:30 Discussing JSDoc documentation website  
09:00 "@param" keyword  
09:40 "@returns" keyword  
11:50 Indicating JSDoc optional parameters  
12:50 Creating type unions with pipe operator "|"  
13:20 Writing types as Pascal Case  
13:50 "@typedef" keyword  
14:30 Creating custom types for objects  
15:00 Creating "Task" custom type  
15:30 "@prop" keyword  
16:00 Creating "Filters" custom type  
16:30 Extending custom types in union  
17:00 Adding descriptions to custom types  
17:50 VS Code plugin for markdown in JSDoc  
18:50 Creating "State" custom type  
19:20 Dynamic objects via "Record" keyword  
19:50 Defining arrays via "Array" keyword  
20:20 "@type" keyword  
21:00 Creating function documentation  
21:30 Be careful that keys are not numbers  
22:10 Quick mention of "nominal types"  
22:40 "@link" keyword  
22:40 Complete remaining JSDoc descriptions  
25:00 Example of how state might be modified  
26:00 What is "static type checking"?  
26:30 Enabling type errors with "// @ts-check"  
27:00 Why is "static type checking" useful?  
28:00 Ensuring documentation and code match  
29:00 "any" JSDoc type  
28:30 When to use the "any" type?  
29:30 Diagrams should express high-level structures  
30:00 Example of Entity Relationship (ER) diagram  
31:20 Creating ER diagram in Markdown  
32:40 Different types of relationship in ER diagrams  
33:30 Example of documentation on Github  
35:00 Basic skeleton for app is now created  
   
   
 Further reading To complete this module, please ensure you work through the following resources:  
   
          - Markdown Guide: Getting Started  
          - TypeScript: JSDoc Reference  
          - User Story Syntax: Writing User Stories with Gherkin  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_03.5 Knowledge Check 2 At the start of the IWA JavaScript Fundamentals (2023) course, we rebuilt The Tally Count App to introduce you to various concepts of the JavaScript language.  
   
Using what you’ve learned about Gherkin Syntax, write user stories to describe the behaviour of the “+” and “-” buttons from that app.  
  
1. User story(ies) in Gherkin syntax for the “+” button.   
2. User story(ies) in Gherkin syntax for the “-” button.  
  
   
You will be required to discuss these User Stories during your session with your coach.  
   
  
Open the Knowledge Check\_DWA3.2 template. Copy this file to your own Google Drive by clicking [File] > [Make a Copy].  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your own details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Now save the [PDF Document] to the DWA3.2 GitHub folder.  
Submit the link for the DWA3.2 Submission via the [Projects] tab.  
  
   
Please ensure you have submitted your User Stories before your session with your coach.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_Part 2 Expectations Part 2 of this course, JavaScript Frameworks, will primarily focus on React.js due to its popularity and employability, but the ever-changing JavaScript landscape necessitates adaptability and understanding of the broader ecosystem. While honing React skills, comparisons to other frameworks will help deepen understanding and encourage exploration.  
   
A strong foundation in programming concepts is crucial for a successful career in software development.  
   
PLEASE NOTE: To complete the DWA course successfully, you are required to:  
  
Watch all pre-recorded lectures  
Thoroughly work through all resources and  
Complete the SCRIMBA React Tutorial: Learn React JS  
  
   
To introduce you to this part of the course, please watch the next 3 pre-recorded lectures with our JavaScript Subject Matter Expert, Schalk Venter, as he takes our students through:  
  
Framework Landscape  
What is JSX  
First React App  
  
   
   
 Further reading Please view the following resources for further understanding of the content covered in this module:  
   
          - WATCH: React.js: The Documentary  
          - npm trends  
          - state of js: Front-End Frameworks  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_05.1 Core Concepts Part 1 Errors often get a bad reputation. However, in this module, we will explore how proper throwing and handling errors in software results in much more maintainable software. We will look at several examples where proper usage of errors could have prevented bad outcomes.  
   
Core Concepts: Part 1  
   
  
   
00:00:00 Introduction  
00:00:30 Understanding errors are critical  
00:01:00 Errors are tools to help manage complexity  
00:01:50 First actual bug recorded in computer history  
00:02:30 Computers were more mechanical than electrical  
00:03:20 Word "bug" is still in use today  
00:03:50 Bug vs Error  
00:04:20 Bugs are unplanned effects in software  
00:05:00 Similar to side-effects in medication  
00:05:30 Bugs can be bad or good  
00:08:00 Some bad bugs do not stop operation  
00:08:30 Introducing the concept of "unsafe state"  
00:09:20 Compile-time vs runtime bugs  
00:10:00 Failing early and loudly  
00:10:30 Software crashing might be the best option  
00:11:00 Example of a bug in "FaceTime" app  
00:11:30 Mention of "defensive programming"  
00:12:40 Introducing Therac-25 incident  
00:13:10 Overview of what Therac-25 itself  
00:13:40 Mention of open-closed principle  
00:14:10 Upgrading is risky if complexity not controlled  
00:14:30 Summarising context of Therac-25  
00:15:00 Different configurations in Therac-25  
00:15:30 What interfaces might have looked like  
00:16:00 Industry learned much from this incident  
00:16:30 Overview of "Malfunction 54"  
00:17:00 Popular underappreciation of complexity  
00:17:50 Dangerous if you remove an error without understanding  
00:19:10 Race-condition bugs are tough to detect  
00:19:40 Common case study in computer science  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_10.3 Challenge 1 In this module, you will be required to use the Shoelace component library in a brand-new JavaScript project and build the original Tally App example from the very first lesson using only Shoelace components. You will be provided with three user stories that should be met at a minimum, however, you are encouraged to add additional functionality. You are free to choose the components you think are best suited to meet these user stories. Be mindful to examine all components and read the documentation thoroughly before proceeding.  
   
You are required to resolve the following User Stories (expressed in Gherkin syntax).  
   
  
SCENARIO: Increment the counter by one  
  
GIVEN the tally counter app is open  
AND the counter is at 0  
WHEN I click the "Add" button  
THEN the counter should display 1  
  
  
  
   
  
SCENARIO: Decrement the counter by one  
  
GIVEN the tally counter app is open  
AND the counter is at 1  
WHEN I click the "Subtract" button  
THEN the counter should display 0  
  
  
  
   
  
SCENARIO: Resetting the Tally Counter  
  
GIVEN the tally counter app is open  
AND the counter value is 10  
WHEN I click on the "Reset" button  
THEN the counter value should change to 0  
AND a confirmation message should be displayed that the counter has been reset  
  
  
  
   
   
In your 1-on-1 session with your coach, you will be required to demonstrate your understanding of all concepts covered in this module. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_14.3 Explicit States    
  
   
   
00:00:00 Introduction  
00:30:00 Avoiding dogmatism  
00:02:00 Value found in core ideas  
00:02:40 JS needs different approaches  
00:03:40 Finite state machine (FSM) examples  
00:04:40 Expressing how state changes  
00:05:20 Turnstile as example  
00:06:40 Polymorphism and FSM  
00:07:30 Example of JavaScript FSM  
00:09:10 State Charts as documentation  
00:10:30 No strict definitions  
00:11:00 Mutual exclusive phases  
00:11:40 Where idea comes from?  
00:12:50 Way to manage complexity  
00:14:20 SM in various disciplines  
00:14:50 Game as example of FSM  
00:16:00 FSM and functional programming  
00:16:30 Revisiting discussion on errors  
00:17:00 "Implicit State" vs "Explict State"  
00:17:40 Revisting discussion on complexity  
00:19:00 Why manage complexity?  
00:19:50 Actions in wrong states  
00:21:00 Technical discussion  
00:22:20 What is "Implicit State?  
00:23:30 Create "Implicit State" by default  
00:24:30 FSM might seem like a limitation  
00:25:40 FSM are self-documenting  
00:26:40 FSM not only idea in programming  
00:27:10 Introducing "XState" library  
00:28:00 Considering usage of "XState"  
00:28:30 Tools have learning requirments  
00:29:00 Example of interactive state charts  
00:30:00 FSM are concerned with actions  
00:30:40 Sketching our "td-task" component  
00:31:30 Example implicit "td-task" state  
00:33:00 Adding implicit state behaviour  
00:35:00 Example explicit "td-task" state  
00:36:30 When to use polymorphism?  
00:37:10 Documenting FSM in JSDoc  
00:38:50 Creating explicit state behaviour  
00:39:30 Adding "transition" function  
00:40:00 Consider transitions as direction  
00:41:00 Copy code to VS Code  
00:41:50 Using "partial application"  
00:42:40 Defining behaviour for each phase  
00:44:00 Extending current FSM behaviour  
00:45:40 Document FSM with "Markdown"  
00:46:10 Indicating different states  
00:46:40 Document transitions in FSM  
00:47:50 Additional documentation tools  
00:48:50 Add notes to documentation  
00:49:10 Avoid too much clutter  
00:50:50 Conclusion  
   
   
   
 Further reading For further learning and information on the content covered in this module, please work through the following resources.  
   
          - WATCH: What Is Lit? - A Web Component Based Framework  
          - WATCH: Introduction to Lit - Lit University (Basics)  
          - WATCH: Understanding State Machines, Part 1: What Are They?  
          - WATCH: Understanding State Machines, Part 2: Why Use Them?  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_14.1 Entire Lit Framework A mutually exclusive component state has several benefits. It provides clear transitions between different phases and prevents unpredictable behaviour caused by overlapping logic. State machines provide a formalised way to explain the state as explicitly and mutually exclusive.  
   
   
  
   
00:00:00 Using "td-spacing"  
00:01:10 Add remaining form inputs  
00:01:50 Add "sl-select" component  
00:02:30 Built-in HTML date selector  
00:03:10 Controlling dialogue visible state  
00:03:40 Can only calculate attribute value  
00:04:20 Different ways to pass values in Lit  
00:05:20 Pass as DOM property syntax  
00:05:50 DOM "properties" vs HTML "attributes"  
00:06:10 Introducing full "Lit Framework"  
00:06:50 What does "lit-HTML" solve for us?  
00:08:00 What does a full "Lit Framework" solves?  
00:09:00 Using "Lit Framework" via CDN  
00:09:40 "LitElement" replaces "render" function  
00:10:20 Lit build on "HTMLElement" prototype  
00:11:00 Lit internally built with TypeScript  
00:11:40 Example JSDoc in Lit source code  
00:12:50 NPM always prefered over CDN  
00:13:20 Copy all functionality into the project  
00:14:00 Custom "td-app" component  
00:14:30 "render" method on "LitElement"  
00:15:10 Move existing HTML to "td-app"  
00:15:40 LitElement "reactive properties"  
00:16:30 Ignore JavaScript "decorators"  
00:17:20 Avoid assigning on class itself  
00:18:00 Lit is the smallest framework abstraction  
00:19:00 Configuring "reactive properties"  
00:20:00 Override default equality with "hasChanged"  
00:21:00 "state" prevents access from outside  
00:21:30 Convert string attributes to properties  
00:22:00 Define defaults in the constructor  
00:22:40 "Lit" framework controls rerendering  
00:24:00 Rerender if reactive properties change  
00:24:30 Compare AST influences the performance  
00:25:00 Understanding when AST compared  
00:25:40 Hard to debug if you do not understand the mechanism  
00:26:20 Lit framework handles event listeners  
00:26:50 Ensure Lit documentation is not "TS"  
00:27:20 Create "td-adding" component  
00:28:15 Declare static "properties" object  
00:28:50 Bind "open" value as property  
00:29:20 Create "toggleOpen" method  
00:30:10 "open" state tracked in two places  
00:30:40 Balancing "global" vs "local" state  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_07.2 Extending Abstraction with SOLID    
  
   
00:00:00 Splitting logic into modules  
00:01:00 Miscellaneous "helpers" module  
00:02:00 "scripts.js" file is the highest abstraction  
00:03:00 Creating a "tasks" JS module  
00:04:00 Revisiting SOLID principles  
00:04:30 Treat SOLID with a degree of flexibility  
00:05:30 Should aid in abstraction, not hinder  
00:06:00 Why so much mention of "Class"?  
00:07:20 What is "Inversion of Control"?  
00:09:00 What do we mean by "Interface"?  
00:10:00 Abstraction in JavaScript Frameworks  
00:12:00 Focus on principles and not exact examples  
00:13:00 Don't overthink terminology  
00:14:00 Revisit employee example  
00:15:00 Extending the "event" abstraction  
00:16:30 Setting values in the "response" property  
00:17:30 Extention by directly changing abstraction  
00:18:00 Listing two remaining principles  
00:18:30 Don't think about SOLID too rigidly  
00:19:40 "O" is for 'Open-Closed'  
00:21:00 Extending while keeping the internals  
00:21:10 Example of modifying core logic  
00:22:10 Adding two base employee objects  
00:23:00 "L" is for "Liskov Substitution"  
00:23:30 Creating a shared abstraction "interface"  
00:24:30 Adding shared "interface" to all abstractions  
00:25:30 Adding method on employee object itself  
00:26:00 "@callback" JSDoc type  
00:26:40 Single method for shared "interface"  
00:27:20 Adding base "invite" behaviour  
00:27:50 Overriding "invite" inside new abstractions  
00:28:10 Anchor ourselves to shared "interface"  
00:29:00 Extending with "Open-Closed" principle  
00:29:40 Learning abstraction takes a long time  
   
   
 Further reading To further unpack the concepts covered in this module, please work through the following resources:  
   
          - WATCH: Object Oriented vs Functional Programming with TypeScript by Fireship  
          - WATCH: Functional, Procedural & Object-oriented Programming - An Overview by Academind  
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_08.1 Creating Encapsulation We need to introduce encapsulation to talk about Object Orientated Programming (OOP). Encapsulation is at the heart of OOP since it allows grouping of related data and behaviour behind an abstracted interface. This enhances code structure, reusability, and maintainability.  
   
  
   
00:00:00 Paradigms focus on abstraction  
00:01:20 There are different flavours of OOP  
00:02:00 Creating practical example  
00:02:50 Co-locating data and behaviour  
00:03:20 Appreciate how far you have come  
00:05:00 Setting up ESLint and Prettier  
00:06:30 Procedural Programming example  
00:07:00 Misconceptions about OOP  
00:07:30 OOP example of Tally App  
00:08:20 OOP with object literals  
00:10:00 Object literals partial encapsulation  
00:10:30 Function closure full encapsulation  
00:11:00 What is a "factory function"?  
00:13:00 Different ways to encapsulate in JS  
00:14:00 "Factory function" return objects  
00:14:50 Cannot directly modify value  
00:15:50 Create JSDoc for methods  
00:18:00 Different objects from the same factory  
00:18:30 Configuration when creating  
00:20:30 Can't update primitives from the interface  
00:22:00 "getters" and "setters"  
00:23:00 Throwing an error on assignment  
00:23:30 Working assignment behaviour  
00:23:50 "setters" can transform on assignment  
00:24:10 Completely override standard logic  
00:24:40 Returning something dynamic  
00:25:00 Recapping "destructuring"  
00:25:50 What is a "spread" operator?  
00:26:30 "destructuring" props object  
00:27:40 Careful of "truthy" or "falsey" bugs  
00:28:00 Discussing "Firacode" font  
00:29:00 Conditions in variables for readability  
00:29:40 Adding the "data attributes"  
00:30:20 Regular comments vs JSDoc  
00:31:15 Extending "getHtml" abstraction  
00:31:50 Realise whether abstraction was good/bad  
00:32:40 Be mindful of getting better at abstraction  
00:33:10 Whether the same or new responsibility  
00:33:40 Example of props vs params  
00:35:40 Be careful of over-abstracting  
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.7 JavaScript at War The critical implication is that this period was the exception, not the rule. Alternatively, it can be framed as a bug and not a feature of the JavaScript specification process.  
   
From its inception, the language was intended to be in constant development and flux as it tries to catch up to published specifications. As a result, the first three versions of the specification were released with about one year between them. This tradition is continued today, so you might have seen an understanding of the specification being referred to by the year it was published (for example, ES6 as ES2015, ES7 as ES2016, and so forth). However, the distinction between the specification and the actual JavaScript you can use at any time is essential.  
   
One of my favourite American Beat-era poets, William S. Burroughs, remarked that:  
   
This is a war universe. War all the time. That is its nature. There may be other universes based on all sorts of other principles, but ours seems to be based on war and games.  
— William S. Burroughs, Grand Street 37 (1991)  
   
In the same way, JavaScript itself is a language caught in endless war and games between the platonic, idealistic specification created by the TC39 and the actual piecemeal implementation by these different compilers. Furthermore, in some cases, teams either flatly decline to implement aspects of the specification. For example, Mozilla, responsible for the Spider Monkey compiler, wrote a public letter in 2015 on why they refuse to implement some aspects of the latest specification - specifically citing grievances with the team behind the V8 compiler.  
   
This tension between the writers of specifications and the individuals behind compilers (called vendors) is most clearly illustrated by a warning in the W3C Web App Manifest specification that reads as follows:  
   
Implementors who are not taking part in the discussions will find the specification changing out from under them in incompatible ways. Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation phase should subscribe to the repository on GitHub and take part in the discussions.  
— W3C, Web App Manifest (2020)  
   
This means there is a never-ending tug-of-war between the specification and what is supported in specific environments and between the specification and particular implementations.  
   
To quote one of the god-fathers of the modern web, Eric Meyer:  
   
“The Web is the most hostile environment to developer assumptions imaginable.”  
   
Therefore, we must have a good understanding of the landscape, to not get lost along the way.  
   
   
 Further reading PLEASE WORK THROUGH THE FOLLOWING REQUIRED RESOURCES FOR THIS MODULE.  
   
          - Caniuse and MDN compatibility data collaboration  
          - What’s the difference between JavaScript and ECMAScript?  
          - TC39 and its contributions to ECMAScript  
          - A Brief History of JavaScript  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_07.4 Challenge 1 In this challenge, you will revisit the previous abstractions you created in the “Book Connect” project and consider them through the lens of SOLID.  
   
Take a moment to rate all your abstractions in terms of the best three and the worst three. Attempt to apply SOLID principles to the worst three as means to improve them.  
   
You will have a call with your coach where you highlight the following:  
  
Which were the three best abstractions, and why?  
Which were the three worst abstractions, and why?  
How can the three worst abstractions be improved via SOLID principles?  
  
   
In your 1-on-1 session with your coach, you will be required to demonstrate your understanding of all concepts covered in this module. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts  
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_05.2 Core Concepts Part 2 Core Concepts: Part 2  
   
  
   
00:00 What is a "race-condition"?  
00:50 Explain race condition in Therac-25  
02:50 Desyncing of internal state and UI  
03:30 JavaScript frameworks mitigate the problem  
04:00 Made worse due to ways in which used  
04:30 When bugs are not reproducible  
05:00 Locking state to prevent desynced UI  
06:00 Throw errors when unlikely but not impossible  
06:40 Abstracting lock behaviour into function  
07:15 Don't build fallback states for uncommon problems, rather throw an error  
07:50 "setTimeout" as a means to delay execution  
09:00 Example of created error in the browser console  
09:30 Thrown errors stop the entire app from working  
10:30 Overview of JavaScript "callstack"  
10:30 Errors bubble up through the callstack in JS  
11:00 Preventing errors from stopping the entire app from working  
11:30 Creating basic JavaScript example  
13:30 Creating function that guarantees HTML element is available  
14:20 Fail loud and early when the HTML element is not found  
15:00 JSDoc static typing prevents runtime errors at compile time  
15:30 Better to catch errors in compile town than runtime  
16:00 Not failing loud and early results in bad outcomes  
16:30 Outcome might not be severe, but there is still a bad outcome  
17:00 Errors give us more peace of mind instead of more stress  
17:30 Creating basic logic and interactions  
19:00 Example of where static typing prevents hard-to-spot bug  
19:40 Sync UI with app state  
21:20 Starting in the "locked" state  
22:45 Example of using errors to prevent a bug  
24:00 Catch error at the top-most scope  
25:40 Using validation instead of stopping the entire process  
26:20 Decide where to intercept errors  
   
   
 Further reading To extend your understanding of the content covered in this module, please work through the following resources:  
   
          - 10 Common JavaScript Errors (And What They Mean) by Kinsta  
          - try, catch, finally, throw - error handling in JavaScript by freeCodeCacmp  
          - How to Handle Errors - Basics of Error Handling in JavaScript - Tutorial by dcode         
          - JavaScript error handling by Bro Code  
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_09.3 Challenge 1 In this challenge, you will continue with your “Book Connect” codebase and further iterate on your abstractions.  
   
Previously, you worked on adding abstraction around the book preview functionality of the project. Next, you must turn the book preview abstraction into a fully-working web component. Then, apply the techniques you’ve learned about this module to the book preview.  
   
If you are up for it, you can also convert other aspects of the app into web components.  
   
You will have a call with your coach where you highlight the following:  
  
What problems did you encounter converting the book preview to a component?  
What other elements make sense to convert into web components? Why?  
Why does keeping your HTML, CSS and JavaScript in a single file sometimes make sense?  
  
   
In your 1-on-1 session with your coach, you will be required to demonstrate your understanding of all concepts covered in this module. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_11.3 Challenge 1 For this challenge, you will be required to use the two supplied video lessons as a reference to create your own implementation of a Redux-inspired store to manage the state of a basic counting Tally App. Note that you are not required to render any HTML to the screen, but instead should add subscriptions that merely log the new state to the console if it changes.  
   
Please look at the video below, for a quick overview of the challenge.  
   
  
   
See below user stories (in Gherkin syntax):  
   
  
  
SCENARIO: Increment the counter by one  
  
GIVEN no interactions have been performed yet  
WHEN the “getState” method is run  
AND the result is logged to the console  
AND the browser console is open  
THEN the state should show a count of 0  
  
  
  
SCENARIO: Increment the counter by one  
  
GIVEN no interactions have been performed yet  
WHEN an “ADD” action is dispatched  
AND another “ADD” action is dispatched  
AND the browser console is open  
THEN the state should show a count of 2  
  
  
  
SCENARIO: Increment the counter by one  
  
GIVEN the current count in the state is 2  
WHEN a “SUBTRACT” action is dispatched  
AND the browser console is open  
THEN the state should display a count of 1  
  
  
  
SCENARIO: Resetting the Tally Counter  
  
GIVEN the current count in the state is 1  
WHEN a “RESET” action is dispatched  
AND the browser console is open  
THEN the state should display a count of 0  
  
  
  
   
In your 1-on-1 session with your coach, you will be required to demonstrate your understanding of all concepts covered in this module. It is at the discretion of the coach to determine what they will ask you, and how deeply they require you to understand specific concepts.  
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_02.8 Knowledge Check\_DWA2 To complete this Knowledge Check, ensure you have worked through all the lessons in Module 2: What is JavaScript? and complete the following questions to successfully complete the module:  
  
1. What do ES5, ES6 and ES2015 mean - and what are the differences between them?  
2. What are JScript, ActionScript and ECMAScript - and how do they relate to JavaScript?  
3. What is an example of a JavaScript specification - and where can you find it?  
4. What are v8, SpiderMonkey, Chakra and Tamarin? Do they run JavaScript differently?  
5. Show a practical example using caniuse.com and the MDN compatibility table.  
  
   
Knowledge Check\_DWA2  
   
Please submit your answers to all the questions, using the Knowledge Check template. Please follow the instructions below:  
  
Open the Knowledge Check\_DWA2 template and copy the file to your own Google Drive by clicking [File] > [Make a Copy].  
Rename the file (click on the filename, top left of the browser window) and replace StudentNumber, ClassCode, Group and FirstAndLastName with your details.  
Complete the template with your answers.  
Download the file as a PDF to your computer by clicking [File] > [Download] > [PDF Document] (.pdf).  
Now save the [PDF Document] to the DWA2 GitHub folder.  
Submit the link for the DWA2 Submission via the [Projects] tab.  
  
   
Please complete and submit your DWA2 Submission prior to your next coaching session, where you will share your understanding of the programming concepts with the group, and complete your code validation.   
   
   
   
   
   
 Previous Lesson   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA: Structuring JavaScript In Part 1 of this course, Structuring JavaScript, we teach students how to write clean, maintainable, and scalable code in JavaScript. The content covers various areas of JavaScript development, including code style, documentation, error handling, abstraction, object-oriented programming, functional programming, and state machines, teaching students the skills and knowledge they need to structure their code in a way that makes it easy to manage and maintain.  
  
Structuring JavaScript Learning objectives:  
  
Understand strategies for managing complexity and structuring code to make it more maintainable.  
Learn the fundamentals of JavaScript, including its syntax, data types, and control structures.  
Understand the importance of documentation and how to write effective comments that enhance code readability.  
Develop an appreciation of the importance of code style and consistency and how to implement conventions that make your code easy to read and maintain.  
Learn how to handle errors in JavaScript and improve debugging capabilities.  
Develop skills in abstraction and how to create modular code that is easier to reuse.  
Understand the principles of object-oriented programming (OOP) and how to create objects, classes, and methods.  
Learn the principles of encapsulation and how to create classes that prevent data and methods from being accessed externally.  
Understand the principles of polymorphism and inheritance and how to use them to extend classes and reuse code.  
Develop an understanding of functional programming concepts, including purity and immutability, and how to use higher-order functions to transform data.  
Learn how to create state machines to model complex systems.  
Practice writing clean code through hands-on coding exercises and examples.  
  
   
By the end of this section, you will be able to write JavaScript code that is modular, maintainable, and scalable, and you will have the skills to tackle complex coding challenges with confidence.   
   
   
   
   
 Next Lesson

# PART 1: Structuring JavaScript

DWA\_13.4 Built-in HOF    
  
   
00:00:00 Recapping array concepts  
00:01:30 Creating example data  
00:02:20 What is the ".find" method?  
00:03:50 What is ".forEach" method?  
00:04:50 What is ".map" method?  
00:06:30 Example of ".map" with HTML  
00:07:30 Using "index" inside method  
00:08:20 ".some" vs ".includes"  
00:09:10 What is the ".some" method?  
00:10:10 Example of "Partial Application"  
00:11:30 OOP composition vs FP composition  
00:13:00 We Need a mix of OOP and FP  
00:13:30 What is the ".every" method?  
00:14:00 What is the ".filter" method?  
00:14:50 ".sort" vs ".toSorted"  
00:15:20 ".sort" MDN documentation  
00:16:40 Default "alphabetical" sorting  
00:17:30 Setting custom sorting  
00:18:50 How to sort numbers?  
00:20:40 How to sort other types?  
00:22:00 How to sort dates?  
00:23:00 Setting custom alphabetic sorting  
00:24:30 Defining custom matching order  
00:25:30 ".reduce" MDN documentation  
00:26:20 Four arguments passed to ".reduce"  
00:27:20 Basic ".reduce" example  
00:29:40 Advanced ".reduce" example  
00:30:50 Composing ".filter" inside ".reduce"  
00:31:40 Adding "if" inside ".reduce"  
00:33:00 ".map" and ".reduce" is useful  
00:33:40 Conclusion  
   
   
   
   
 Previous Lesson   
   
 Next Lesson