Course Overview

Course Overview

Hello. My name is David Tucker. Welcome to JavaScript: The Big Picture. According to multiple reports, JavaScript is the world's most popular programming language. Because of this, there is a large amount of interest from developers to learn more about it. Few languages are as universal as JavaScript, and pretty much anyone would benefit from knowing a bit more about it. In this course, JavaScript: The Big Picture, you'll learn what JavaScript is and how it is used in the world today. First, you'll explore a definition of JavaScript and the language's core characteristics. Through this, you'll learn what makes JavaScript unique. Next, you'll discover the most common use cases for JavaScript. Whether you realize it or not, your life intersects with JavaScript on a daily basis. Finally, you'll learn about how JavaScript is governed, as well as the release process. This is certainly different from many other programming languages, so you will gain an understanding of what to consider when it comes to JavaScript versions. When you're finished with this course, you'll have the skills and knowledge of JavaScript needed to begin your journey into JavaScript development

Why Javascript?

What Is JavaScript?

My name is David Tucker, and we're getting ready to start a journey where we examine the big picture of JavaScript. Now whether this is your first or your tenth programming language, I want you to know that you are starting in the right place. Now instead of covering a bunch of background material, I want to cut to the chase and answer the question that many of you have, what is JavaScript? This information will be foundational to everything that I'll cover in this course, so let's dive in, okay? Here we go. JavaScript is a programming language that can be used to create web applications, backend services, games, mobile applications, and just about anything else you can imagine. Now I realize for many of you that might leave more questions than answers, and that's okay. We'll build on this definition with some additional technical details in the next clip. Next, I realize that many of you might not have ever even seen any JavaScript. So next, I want to answer the question, what does JavaScript look like? Now if you're new to programming, I realize that this may look like a foreign language. But if you work through this learning path, you will be able to understand the meaning of this code sample on the screen. Before we get too deep into the language, I want to cover a few things you will need to know when it comes to JavaScript. First, and I don't want you to miss this, JavaScript and Java are not the same thing. Yes, they are both programming languages. And yes, there are some aspects of the languages that have similarities. But these two are very different. The naming of JavaScript has caused a good deal of confusion over the years. Hopefully, I have already helped you avoid a common developer faux pas. Second, I want you to know that JavaScript is a language you can learn. Yes, you. In many ways, it is more approachable than traditional languages like C++ or even modern languages like Rust. If this is your first jump into programming, I want to encourage you that you are starting in a good place. Okay, since we've covered a basic definition, let's dive into the next question you may be asking, why should I care? Even though JavaScript was created initially for the browser when it launched in 1995, it has grown to become one of the most widely used programming languages in the world. According to an article by ZDNet, JavaScript is now used by more than 16.4 million developers globally, making it the world's most popular programming language by a wide margin. Now if you think about it, it's telling that the language once created for the web is now the most popular programming language in the world. In addition to this, HackerRank's Developer Skills Report states that globally, JavaScript is the most popular language hiring managers look for in a candidate. To help make this point, right before I recorded this episode, I checked 10 of the biggest tech companies in the world. Do you want to guess how many of them were currently hiring for JavaScript developers? 100%. I hope these data points are compelling for you. But let me just explain why I'm excited about you learning more about JavaScript in this course. Ultimately, I believe that knowing about JavaScript will open doors for you professionally. There are few areas that you will work in that JavaScript doesn't touch in some way. I see it as a value-add, even if it isn't your primary language or platform. Now in some ways, I've tackled the two biggest questions, what is JavaScript and why should you care? But in some ways, I've only scratched the surface on these questions. Up next, I want to review a few of the characteristics that make JavaScript what it is. We'll get to expand on our current definition with some technical details.

Characteristics of JavaScript

Let's quickly revisit the working definition of JavaScript that I covered in the last clip. Now we determined that JavaScript is a programming language that can be used to create web applications, backend services, games, mobile applications, and just about anything else you can imagine. Now while this definition worked well to get us into JavaScript, it doesn't really cover the technical details that provide JavaScript with its identity. I want to introduce you to a few technical terms that will help us build on our current definition of JavaScript, and this should help us understand the core characteristics of the language. First, JavaScript is a dynamic programming language. Let me explain what that means. First, to demonstrate this, in every programming language, we have the concept of a variable. A variable gives us the ability to store some bit of information that we can access at a later point. Let's say that I was building a contact list for my company. As a part of this, I would need to enter my first name, last name, and age. Now if I wanted to access this information, I could store it inside of variables. In JavaScript, we could create a variable to hold my first name like this. Don't worry too much about the syntax for this just yet. We'll cover that extensively in this learning path. Next, if I wanted to store my age in a variable, I could do it in the same way. Now, these two variables hold data of different types. The first variable, my first name, is a word. We generally refer to this as a string. The second variable, my age, is a number and, more specifically, an integer. We generally won't use a decimal here. Most people don't say they are 41.34 years old, for example. Now in JavaScript, we don't have to tell the language which type we are using. This is what makes it dynamic. The opposite of a dynamic language is a strongly typed language. Here is an example from C#, which is a strongly typed language. And don't worry. You don't need to know anything about C# to understand what I'm getting at here. With this example, you can see that we are actually telling the language what type of data my first name is. In addition, we are doing the same thing for the age. In this case, int is short for integer. Not having to define types for every variable is one of the benefits and also one of the biggest challenges when it comes to JavaScript. It can make it easier to learn the language, and it means that you usually have less code that you have to write. It can make it harder in that some errors in your code might not be as evident when you're writing it, and you might not find out about those problems until you actually run your code. JavaScript isn't alone in this. Other languages like Python and Ruby also share this characteristic. Now when JavaScript was created, it was an interpreted language. This is our next characteristic. In many programming languages like C++, after you write your code, you need to compile it before you can run it. The compiler takes your code and translates it into something that your computer can understand. Your code cannot run without taking this step first. Think of it this way. I speak English as my primary language. Some of you may have noticed that. Now, with this being said, imagine that I have a set of instructions for assembling a piece of furniture in German. I don't know any German. If we were following the compiled language paradigm, I would need to have someone come in and translate the complete set of instructions before I ever get started. Once I have the instructions, I don't need this person anymore. I can now assemble the furniture on my own. In this case, the person assisting us is the compiler. Now in most cases, JavaScript doesn't work this way. JavaScript runs inside some type of engine. The engine knows how to take in JavaScript code and translate it into the actions that it needs to take. In this case, the engine is serving as the interpreter of the JavaScript code. This is why we refer to JavaScript as an interpreted language. Let's shift back to our furniture assembly problem with the German instructions. Now in this situation, we don't need a translator to come in and fully translate the instructions. Instead, we can bring in someone who can instruct us on what we need to do for each instruction individually. That person can simply translate the current instruction, let me know what I need to do for that step, and then wait until I'm ready for the next step. In this case, the person assisting us is the engine or interpreter. Now you might be curious, where would you find a JavaScript engine? Now if you were watching this clip on your computer, you are using a JavaScript engine right now. Every browser has a JavaScript engine built in. While this was what JavaScript was created for, there are now other JavaScript engines that we'll be discussing throughout this course. Now I do want to mention that for those of you who are technical, you might be calling out that JavaScript is not as pure of an interpreted language as it was when it was created. Yeah, we'll get to all the specifics later in the learning path, so just be patient. Finally, I want to highlight something about the name, JavaScript. The second half of the name, script, is there for a reason. JavaScript was created as a scripting language. Some languages like C, for example, are not bound to a specific context. As I mentioned previously, JavaScript was created for the browser. If you wanted to add interactivity to a website or customize how the browser worked for a specific site, you would use JavaScript. In this way, the language was scripting the behavior of the browser. If you wanted to do something outside the browser, you wouldn't use JavaScript for it. Now the language has changed a great deal since its creation, and it can be used in many different contexts beyond just the browser. That being said, many people still refer to JavaScript as a scripting language, and I wanted you to know why. I also wanted you to know why script was stuck at the end of the name. So with all of that, let's update our definition of

JavaScript by adding more of the technical details. JavaScript is a dynamic interpreted scripting language that can be used to create web applications, backend services, games, mobile applications, and just about anything else you can imagine. In this clip, I've alluded to something many times, that JavaScript can run both in the browser, but also in other contexts. This leads to a question that I'll be covering in the next module, where is JavaScript used?

Where Is JavaScript Used?

Where to Find JavaScript

Whether you realize it or not, I would bet that your life has intersected with JavaScript at some point today. Here in this module, I want you to understand where to look for the JavaScript in your life. For each of the use cases we talk about here, I'll be diving into them more deeply in upcoming clips. Within these upcoming clips, I'll be diving into the core technologies that are leveraged for each use case. So when someone says they're building a full-stack JavaScript experience with React, Node, and Express, you at least have a high-level understanding about what they mean. It's probably no surprise, but the first place I want to start is the browser. If you opened your browser today, whether on your laptop, tablet, or phone, you've likely used JavaScript. Most all interactivity within web applications is facilitated by JavaScript. Think of it this way. Every time you open up a menu with a button press on a web page, JavaScript made that happen. Every time you get typeahead when searching in your favorite search engine, JavaScript made that happen. Every time you see your timeline updating with new posts, you guessed it, JavaScript did that too. Even though JavaScript was born in the browser, it certainly didn't stop there. When we're talking about web applications, there are usually two sides to the coin, the front end and the back end. We've just been talking about the front end that runs in the browser. But think of it this way. Where do your social media posts go when you close the browser? The answer is into a type of database. Something needs to take that post, validate which user posts in it, and then insert it into the database. In addition something needs to be able to fetch all the recent posts when you log in the next time. All of this happens on a server, and JavaScript can work there too. This doesn't run in that browser engine though. So, in a future clip, I'll explain how JavaScript works in that context. JavaScript does not end here though. One of the most popular desktop code editor apps is Visual Studio Code from Microsoft. This open source desktop app was built with JavaScript and TypeScript, and many of you may regularly use Slack. This desktop app was also written with JavaScript. Maybe you don't use Slack because your company leverages Microsoft Teams. Well, that is also an app built with JavaScript. These desktop apps utilize an engine that brings together aspects from the browser and the server. But we'll dive into those specifics a bit later. It doesn't even end there. Many games have been built fully or partially with JavaScript. People use JavaScript to help manage their smart home devices. Developers can even use JavaScript to launch servers and other infrastructure in the cloud with public cloud providers like Amazon Web Services, Microsoft Azure, and the Google Cloud Platform. Now I want to guide you through these different use cases so that you can understand the full reach of JavaScript within our current digital landscape. Also, I want to call out an additional resource I have included with this course. I've compiled a collection of links to various JavaScript resources on my personal blog. You can access the page with this link. And if I mention any platforms, frameworks, or tools during this course, I'll have the corresponding links on that page. I'll be referring back to this page at several different points throughout this course. Now, with all of that out of the way, let's jump into where JavaScript got its start, the browser.

JavaScript in the Browser

JavaScript has a longer history in the browser than in any of the other use cases that we're going to take a look at in this course. When the web was invented, it was a static and stationary experience. Between December 1995 and 96, we had JavaScript and CSS launched onto the world. And through steady evolution, we now have the modern web applications that we all use as a part of our lives. I mentioned previously that when JavaScript in the browser runs, it runs within an engine. Now there are three primary JavaScript engines that you will come across. V8 is the most popular JavaScript engine, and it is used by Google and Google Chrome, as well as Microsoft with Microsoft Edge. For those of you saying no, Microsoft Edge uses a different JavaScript engine instead of V8, you can go back and check. And since 2020, Microsoft Edge does indeed use the V8 engine. JavaScript Core is the engine that runs on Safari, although it sometimes goes by the name SquirrelFish. Finally, we have SpiderMonkey, which is a descendant of the original JavaScript engine, and it is used in the Firefox browser. By no means are these the only JavaScript engines, but they are the most common ones. I'll also give a hint for a future clip. Some of these engines even have uses outside of the browser, but we'll get to that a bit later. Now before we go any further, let's discuss some of the things that you can do with JavaScript and why it is such a crucial part of the web today. First, it is hard to talk about JavaScript without also talking about two other web technologies, HTML and CSS. Let's dive into these before we explore JavaScript's role. First up, we have HTML. HTML stands for HyperText Markup Language, and it is where we define the structure of any web page. If you want to put a paragraph of text on a web page, you do that with HTML. HTML is pretty distinctive because it uses a set of opening and closing tags to define the structure of the page. The p tag you see here represents a paragraph that will appear on the page. The a tag lets you define a link that will take you to a different web page. When the web first started, this was all we had. If you learned to master tags like the p and the a tag I mentioned, you could create your own web page. Now maybe it would look a bit like this. Do you think this page looks boring? This was the very first web page created by Tim Berners-Lee. So, while it might look boring at first glance, it has affected each of us in more ways than we could ever quantify. Now shortly after, we gained JavaScript as a tool, we also gained another tool, CSS. And CSS stands for Cascading Style Sheets, and it enables us to define and how our HTML should look when it is rendered in the browser. Now this really enabled us to bring elements of design into the pages that we create. While some of these things were actually possible before CSS, it was difficult to create a style across a collection of pages. So if you take a look at my blog, you can see that it is much different from the plain website we looked at previously. A big reason for this is the introduction of CSS. CSS works by defining rules for how specific elements in HTML should look. We often call this the presentation of the site. With CSS, we define what we want to style. In this case, we'll take the p tag on our site, and we can choose to change the default values that the browser uses. For example, we can change the color of the text to red and the size to be 40 px. A CSS file can contain any number of these style rules. And finally, we get to JavaScript, the one I'm here to talk about. And while we have the ability to define the structure of the page and we have the ability to style a collection of pages with a specific and consistent aesthetic, how do we configure how our users interact with the pages we create? Sure, some of the things work by default. Users click on a link and then navigate to another page. But what if we want to enable users to click on an image and have it open up a menu? This is where JavaScript comes in. With JavaScript, you can assign and code to run when specific things happen. In this case, there is an HTML tag that we enable to trigger a specific JavaScript function. Now don't worry if you don't know about functions yet. We'll cover that fully in this learning path. This function will take care of showing the menu, as well as hiding it. This is a very basic version of what is possible when we leverage JavaScript alongside HTML and CSS. Now more and more often today, we don't just have web pages. We also have web applications. Now, think of the difference between the first ever web page I showed you earlier from Tim Berners-Lee and then the experience you have visiting Gmail in the browser. Gmail is full of ways that you can interact with the interface to perform different tasks. This is an example of a web application. Now when it comes to web applications, there are tools and frameworks that all still build on the core elements of HTML, CSS, and JavaScript, but they make creating holistic web applications a bit easier for developers. Now I wanted to bring these to your attention as you may have run across these terms already. Now some of the most popular frameworks include React, which is maintained by Meta, Angular, which is maintained by Google, and then Vue.js and Svelte, which are both maintained by different core teams of open source contributors. Now while these frameworks enable you to create holistic web applications, they are still based on HTML, CSS, and JavaScript. In some cases, they do start to blur the lines a bit between these different technologies. Now first, it's important to know that JavaScript is at the center of most all modern web applications. Because of this, JavaScript developers wanted the ability to manipulate both HTML and CSS using JavaScript. One example of this is JSX. JSX is an extension for JavaScript most commonly used in the React framework that enables you to write HTML-like markup directly within JavaScript. Now this can enable you to do some very powerful things with the JavaScript you create. But the important thing to note is that even JSX will eventually boil down to HTML, CSS, and JavaScript. In addition, there are even tools and frameworks that even enable you to write CSS within JavaScript. But even that will still eventually boil down to, you guessed it, HTML, CSS, and JavaScript. These are the core building blocks of the web, and JavaScript plays an essential role in defining how our users interact with the pages that we create. While this is the original context for JavaScript, it certainly didn't stop there. Next up, we'll take a look at how JavaScript works on the server.

JavaScript on the Server

Up to this point, we have seen JavaScript work as a scripting language for the browser. Now there have been many attempts to expand JavaScript beyond just this environment. But to be honest, nothing really was widely adopted until a new project launched in May of 2009. This project, Node.js, dramatically altered the role JavaScript plays in development outside of the browser. Now, you might remember that I mentioned that our discussion of browser engines wouldn't end in the browser, right? Well this is why.

Node.js was built on the open source V8 JavaScript engine that is used in both Google Chrome, as well as Microsoft Edge. The most interesting piece is that the creator, Ryan Dahl, added APIs for things JavaScript couldn't do in the browser. Now this included reading and writing files, making and receiving network requests, as well as encrypting and decrypting data. So, if we piece all of this information together, we can define Node.js as a JavaScript environment which executes JavaScript code inside of the V8 JavaScript engine and provides an I/O library to do things that we can't do in the browser environment. Now to understand why this matters, let's chat a bit about how websites and web applications get delivered to end users. In the previous clip, I mentioned that HTML, CSS, and JavaScript are the building blocks of the web, and this is indeed true. But have you ever stopped to think about how that HTML, CSS, and JavaScript actually make its way to your computer or your phone? Now we won't cover all of this because that would take quite some time. But when you enter a URL, that request gets mapped to a specific server. Now this server is probably running in a data center somewhere. Now when your request makes it to that server, there is software that handles what happens next. So, let's say, for example, that you visit amazon.com on a brand new computer. But once you were there, you click on the link to view your orders. What happens next? Well, since it's a brand new machine, you're going to need to log into your amazon.com account before you can see a list of orders. Now, more than likely, this isn't a surprise to you. But did you ever stop and think about why it works that way? Well, the reason is that there is software, what we would call the web server, that defines what happens for each request. Now, one of the use cases that Node.js is commonly used for is to create this web server software for websites and applications. These web servers can deliver an e-commerce application, a social network, or even a video subscription service. Now while Node.js was very useful in the beginning, there was an addition added less than a year from its launch that made creating these types of applications even easier. First, let me give you a definition. Now in terms of software development, a package manager is a tool that enables you to install and manage software written by other developers into your own software projects. Now, these aren't just a JavaScript thing. Most languages have some sort of a package manager associated with them. You might have heard of tools like NuGet, which is the package manager for the Microsoft development stack, or pip, which is a package manager for Python. Now there are two different parts to a package manager. First, you have the tool that you use to download the software into your project, and we'll call this one the client. The next part is a listing of all of the available packages you can install, and we'll call this the registry. These two pieces work together to enable the package manager to do its thing. Well in January of 2010, npm was introduced as the official package manager for Node.js. Npm, short for Node Package Manager, was both the client and the registry. Now if we were to fast-forward to current-day JavaScript development, there are multiple ways that you can handle package management for Node.js projects. Now for the client, you can still leverage npm, and it is still the most popular client. But you can also leverage other solutions like yarn and pnpm. Now all of these are different clients that you can use to install and manage packages in your JavaScript project. However, some things haven't changed. The npm registry is still the primary registry used by all of these clients. And since 2020, this registry has been owned by GitHub, which is itself owned by Microsoft. Now let me explain a bit about why this is important. Let's say you want to build a web server application like the one I mentioned earlier. Now you could do this completely in Node.js by not using any code from other organizations or developers. It would just take a ton more time and a ton more code that you have to write and maintain. Now there are some common frameworks that you can install to make this easier. One of the most popular tools for creating web server applications is Express. Now if we had a Node.js project set up, we could install Express by simply typing npm install express. Now this would add it to our project, and then with just a little bit of code, we could configure our own web server and test it out on our browser. Now, I realize that there are a lot of things that might not fully make sense for you with that demo, and that's totally fine. I just wanted you to see both Node.js and npm in action and have you understand them just at a conceptual level. Now Express isn't the only popular Node.js package. There are many amazing tools that you can leverage, such as Axios to help with making web requests, Socket.IO for creating real-time apps like chat applications, and then Mocha and Jest for testing your JavaScript code. Npm provides a website where you can go search through all the packages that are included within the registry. Now I must admit that I've told you something that might be a bit misleading in the title for this clip. Now I mentioned that this clip is about JavaScript on the server, but it's actually much more than that. Node.js isn't just used in that server that delivers websites and applications. I have Node.js running on my machine here as well. Many of the modern tools that you will actually use to create JavaScript applications actually run on Node.js as well. Want to create a new React web application? Well the create-react-app tool runs on Node.js. Want to leverage Visual Studio Code to edit your JavaScript code? That runs on an application framework called Electron that enables you to build desktop apps with JavaScript and Node.js. The same framework powers the desktop apps for Slack and Microsoft Teams. Now I know it seems like everything on the server is very centered around Node.js, and that's true for the most part. But I did want to call your attention to one other runtime for running JavaScript on the server, and that's Deno. Now it was actually created by the same guy who created Node.js. It has some really cool features, but since it has a fraction of the adoption that Node.js has, I'll just be referring to Node.js for the remainder of this course. Now, think about something for a minute. When you do learn JavaScript, what could happen if you knew how to build all of the amazing things in both of these contexts? Well, that's what we're going to be diving into next.

Full Stack JavaScript

So we've explored how JavaScript works both in the browser and on the server. Now to quickly sum it up, let's revisit a few of the terms we've covered so far related to a typical web application. First, there will be some type of website or web application that will be built with HTML, CSS, and JavaScript, and this is what our users will interact with through their browser. And we generally call this part of the application the front end. Next, we will have a web server that delivers the web application to the end users, and it will handle things like loading in data from a database, as well as handling things like ensuring a user is logged in. This can be built in Node.js with frameworks like Express, and we generally call this the back end. Now what do these two have in common? You guessed it, JavaScript. The technologies we use across all of the tiers of our application, we generally call our stack. Many developers get jobs working on either the front end or the back end. If you really like creating user interfaces, you may prefer a job on the front end. If you like working with infrastructure and tools like databases, you may prefer the back end. But what would you do if you liked both? One of the things that's unique about JavaScript is that you can work across both of these different tiers using the same language, and this has led to the rise of what we would call the full stack JavaScript developer. And these types of developers are currently in demand for many different types of companies. So if you are a developer, having this skill set offers you many benefits, even if you don't have a job as a full stack engineer. First, you better understand the challenges that exist on other tiers of the technology stack. You get exposure to different tools and frameworks. But, and this is the one that has always excited me, if you want to build something yourself, you can do it just with your own skill set. Now, don't get me wrong. This doesn't mean that you are an expert in the whole stack. But it does mean that you can work within both contexts. Now since the popularity of this type of development has grown, some web application frameworks have truly embraced it. Next.js is a web application framework that enables developers to build both the front end and a portion of the back end in a single project. This framework builds on one of the frameworks that I've already mentioned, React. The concept of a full stack developer highlights one of the best assets of JavaScript, its prevalence. Up until this point, I've talked a great deal about how it is used to create experiences on the web. But in the next clip, I want to show you some use cases in completely different areas that help drive home the point that JavaScript is indeed everywhere.

Additional JavaScript Use Cases

JavaScript can be found in many different places besides the web. Now, I will tell you a majority of JavaScript developers do work in an area that is connected to the web, but there continue to be new use cases that JavaScript is expanding into, and I wanted to review some of these with you. And to be honest, I've already mentioned a few of them to you. So, if we rewind and you remember, I called out on a previous clip that there is a way you can build desktop applications using JavaScript. Now while there are a few different frameworks for this, Electron is the most popular way to currently utilize your JavaScript skills to create a experiences for Mac, Windows, and Linux. Now because of this, it probably isn't a surprise that JavaScript can also be used to create mobile applications. So let's say you want to build a mobile application on both iOS and Android for an e-commerce site that you've launched. If you went the traditional way, you would use the actual development kits provided by Apple and Google, respectively. You would have to build the front end for these applications with completely different languages, Swift for iOS and Kotlin or Java for Android. Now, some existing frameworks, and we could include React Native, for example, enable you to create one application using only JavaScript that just requires some slight customizations for each of the mobile operating systems you want to deploy to. Now these apps can end up in the App Store alongside apps that were built using the primary developer kits. Now I should mention here that these type of apps aren't always as seamless as the ones built in the original development kits, but that's a discussion for another day. Next, in addition to desktop and mobile apps, we also have IoT, or the Internet of Things. All around us, we have connected devices. In my home, we have smart thermostats. And to be honest, I even had to install a firmware update on my oven the other day. Through a combination of different frameworks and cloud services, JavaScript can play a role in building out these solutions too. Many devices, like a Raspberry Pi, have the ability to install and run Node.js, and packages are available in npm to interact with different sensors and input devices. Now, next up, I'm going to take you into a world we haven't really discussed much, the public cloud. Now there are three primary public cloud providers, Amazon Web Services, commonly known as AWS, Microsoft Azure, and the Google Cloud Platform, known as GCP. At a very high level, these platforms give you access to servers and services that you can leverage to build your own applications. You no longer have to create your own data center to deploy an application and make it available to the world. Instead, you can pay them to utilize a portion of their data center, and you don't even have to have your own server to send them. You can utilize a service like Amazon EC2, which runs on AWS, and they can spin up a virtual server that you can use. Now I brought up the public cloud because you you can even use JavaScript to configure and deploy your public cloud infrastructure. If I wanted to deploy servers in Singapore, Ireland, and the United States, I can do it with JavaScript in a matter of mere minutes. Solutions like Pulumi, as well as the Cloud Development Kit from AWS, make this possible. So I hope you can see that JavaScript has a wide reach in the digital landscape, and it has grown considerably since being born as a scripting language for the browser. This might create some questions for you. For example, what version of JavaScript are we even on, and who even gets to decide that? So in the next module, I'll be diving into those questions and oh, so many more.

How Is JavaScript Versioned?

JavaScript's History

Now if we were to rewind to 1993, the first web browser was released, and it was named Mosaic. Now it quickly was outpaced by Netscape Navigator the following year. There's actually a lot more to that story, but we're not going to dive into it. Now in the following year, 1995, Netscape wanted to integrate a dynamic and interactive element into the web, so they tasked Brendan Eich to create a new language for this purpose. Now this language first called LiveScript was soon renamed to JavaScript. And according to Wikipedia, the dot-com boom had begun and Java was the hot new language, so Eich considered the JavaScript name a marketing ploy by Netscape. So there you go, all that confusion caused by a marketing ploy. Now over the upcoming years, new browsers would come onto the scene. Hello, Internet Explorer. Now this dynamic created some problems. For example, when Microsoft reverse-engineered their own JavaScript interpreter, it wasn't based on any defined standards, so it didn't work exactly the same way. In 1996, it became obvious that the makers of web browsers needed to follow the same standard. Ecma International became the company that would manage the standards for the web. Ecma, which originally stood for the European Computer Manufacturers Association, had already been working to standardize all sorts of things for the tech industry. Now most all of the big tech companies that you can think of, they have some type of a membership with Ecma International, and these organizations can help shape the future of a wide variety of different technology standards. Now if we go back to the early days, standardizing web technologies was not easy. Even after a standard would be agreed upon, it would take a long time for it to be consistently available across browsers. Even now, versions in JavaScript work very differently from what we see in other languages or platforms, and this can lead to some confusion. So, before we continue our journey into understanding JavaScript versions, we actually have to take a step back and understand a bit more about its governance.

Who Governs JavaScript?

Now that you have an understanding of the history of JavaScript, let's dive more into its governance. In the last clip, I introduced you to Ecma International, which was formerly known as the European Computer Manufacturers Association. Since 1994, it has just been known as Ecma International, and it is a not-for-profit standards organization. This means that it is supported by donations from tech and manufacturing companies related to the computing realm. They work with a great deal more than just JavaScript. For example, they manage the standards for different technologies, including the CD-ROM file structure, the OpenOffice XML file format, and both the C# and Dart programming languages just to name a few. The specification that focused on JavaScript was not known as JavaScript initially. The specification was called ECMAScript. Now there were multiple reasons for this, but one of the reasons was that Oracle controlled the name JavaScript. In addition, there actually were other languages that were developed from the specification, but trust me. We don't have enough time to get into that drama. Just in the past year, the group managing the specification has started to phase out the use of ECMAScript in favor of JavaScript, but you'll still see ECMAScript in most places, so I'm just going to use ECMAScript for now. So let's start to piece this together. In 1997, a standard was defined called ECMA-262, which would become the standard that browser creators would be able to follow so that there was a level of consistency with JavaScript. This standard, as I mentioned, earlier was called ECMAScript. Now the specific technical committee at Ecma International that would work on the specification was called TC-39 with TC standing for Technical Committee. Now in summary, Ecma International manages the specification for JavaScript called the ECMAScript specification through the TC-39 committee. Now you might think that this committee works in secret, but that couldn't be further from the truth. These technical committees work out in the open. The TC-39 committee has a website where they post minutes of meetings, current proposals for new JavaScript capabilities, as well as information around how ideas can be submitted to the committee. In addition, they also house the ECMAScript specification in a GitHub repository. You might think, wow, this is great. I get it now. However, there's a bit more that we need to discuss. This specification applies to the core JavaScript language that you use in the browser. And if you remember, I mentioned that Node.js adds many different capabilities to JavaScript for things like reading and writing files, making and receiving network requests, as well as encrypting and decrypting data. This is not covered in the ECMAScript specification. This isn't even covered under Ecma International at all. Instead, this is all governed under the Node.js project. Given that Node.js is used in many of the different use cases that we've discussed, I believe it is also important to understand how governance works for it as well. The Node.js project has a technical steering committee, which is responsible for defining the project's capabilities. But in addition, they also have a community committee, which ensures that the community has a voice in the direction of the project. And together, these two constitute the governance model for Node.js. Just as with TC-39, you can see much of the work of these committees online where you can review members, meeting minutes, and even proposals. So, now that you understand how governance works within the JavaScript world, we can now dive in and talk about what will affect your life directly if you're working in the language, versions of JavaScript.

JavaScript Releases

Okay, now that you have the knowledge about how JavaScript is governed, we can now start the process of discussing JavaScript versions. Some of you right now want to know how this applies to JavaScript development today, and I promise I'm getting into that in this clip. But before I get there, I need to explain how we got our JavaScript versions. I alluded to this before, but I want to mention again that JavaScript is very different from other programming languages in how versions are handled. So for example, if you're building a Python application, you will target a specific version number. Occasionally, there would be a shift from one major version, so like 2.7 up to 3.10. And at this point, you would gain the new features from Python 3, but some of the things that you wrote in Python 2 wouldn't work in the same way in version 3. For many reasons that we'll cover, JavaScript doesn't work this way. Let's look at the history of JavaScript versions to understand why. First, if we look back at the original version of JavaScript that was defined with the first ECMAScript specification, we'll call this ES1. I'm guessing you've already picked up that the ES here refers to ECMAScript. This release was followed by ES2 in 1998, but this release was really just about neatening things up in the specification. It didn't have any substantial changes. Now, you might be looking at this and going okay. When there was a new version of the specification, the browsers just used that, and everything stayed in sync. But I'm guessing that you might already know that it didn't work that way. There was always a delay in the specification being released and it being available in a browser. Each browser operated on its own timeline. So there was no hard and fast date for when new functionality would be available. Over time, we would have many additional releases that would add new functionality. When ES3 was finalized in 1999, we got try/catch blocks and regular expressions that were added to the specification. When ES4 was finalized, who am I kidding? ES4 was never finalized. It had more drama than a Spielberg film. Now at this point in history, the web was still being formed into what it would become, and companies were fighting for their own interest in the TC-39 committee. Thankfully, those differences would be resolved. Now that might not be the right word. Some companies actually left the TC-39 committee over this debacle. Now, with all that said, things at least started to move forward again. In 2009, we got ES5, which was a huge step forward towards our modern concept of JavaScript. So, when do you think that all browsers fully supported this version of specification? Maybe 2010? Nope. 2011? No. It would take until 2013 for Internet Explorer, Chrome, Firefox, Safari, and Opera to be compliant with this version of the specification. Now I should mention here that many of the features that you will use for modern JavaScript weren't yet in place at this point. Things like promises, async await, ES modules, class declarations, none of those existed in ES5. But the vision for those were set by the next release, which was initially referred to as ES6. And something happened at this point. The naming began to change from these large major releases like ES3 and ES5 to a yearly release. In 2015, we finally got ES6, which is also known as ES2015 since it was released in 2015. This was also the start of that yearly cadence. Now I can't overstate how significant this release was. While it would take a while for its impact to touch all areas of JavaScript development, this release was the beginning of modern JavaScript development. Now many of the complaints that developers had about JavaScript as a whole were tied to how things were done before ES6. Problems like hoisting and the lack of block scope and dealing with a variable amount of arguments and dealing with classes, all of these were addressed in some way within ES6. Now I'm not going to even explain those issues to you right now. And to be honest, many of you will never have to write JavaScript that deals with those pre-ES6 concerns. Now there were still some good new features that were introduced after this. In ES2017, which is sometimes also called ES7, the addition of async and await made it much easier to write asynchronous code. Smaller, incremental updates would be added to the specification, and this cadence continues until today. Now I should tell you about another version, ES Next. This is the code name for the next release before it has been defined. At the time of this recording, ES 2022 has been released, ES 2023 has some functionality already planned to be with it, and everything else beyond that falls into the bucket of ES Next. So how exactly do JavaScript developers know which version they're using when they're building a web application, for example? Or how would they know what version they're using for a server-based application? Well, the answers to those questions are different. Let's first dive into the browser. Since it took a while for ECMAScript specification versions to end up in the browser, there were some sites that were created to track what functionality was available across the different browser versions. One of the most popular was a site called caniuse.com. Now here, you can see the browser support for the ES6 specification. You can see, for example, that Internet Explorer 6 through 10 never supported ES6, while version 11 had partial support. Sites like these were useful to track JavaScript capabilities based on the specific browser version. If we were to look here at the await operation, which was included in ES 2017, we can see it wasn't supported in Chrome until version 55. Also, it wasn't ever supported in Internet Explorer. Now, let's chat about the server. If you remember, Node.js used the V8 JavaScript engine. This is the same engine that is included in Google Chrome. Now there is another site, node.green, which shows supported functionality from different specification versions based on the specific version of Node.js. Now while you can see a ton of different versions of Node.js here, I want to explain something about Node.js versions that you need to be aware of. Node.js has LTS, or long-term support releases. On October 25th, 2022, Node 18 became the LTS version where it took over from Node 16. Now, these releases will have maintenance support for a few years unlike other releases. Finally, I need to cover an aspect of JavaScript that is also unique. JavaScript remains backwards-compatible. This means that valid JavaScript based on ES1 is still valid JavaScript. When JavaScript introduces a new capability, it's stuck with it forever. This is very different from other languages. But most languages don't have the burden of being the language for the entire web. So what JavaScript version are you on? Well, in every situation, the answer is it depends. If you're building the browser, you'll need to look at the target browsers and what they support. Thankfully, this is much easier than it used to be, and browser vendors can much more quickly implement those specifications. And if you're on the server, you can look at your Node.js version to see what is supported. So every JavaScript developer has to wait to use functionality from a release of the specification until it is fully supported on all browsers and Node.js, right? Not exactly. In the next clip, I'll explain how we have worked around that with some developer magic.

Extending JavaScript

Now I had to explain to you how JavaScript versions work for you to understand why this clip and any of the tools I mention here even exist. Now I'm going to explain for you a handful of tools that can be used to overcome the limitation of a specific bit of JavaScript functionality that isn't fully supported across all JavaScript engines that your code may run on. So first up, we have a polyfill. And according to Mozilla, a polyfill is a piece of code used to provide modern functionality on older browsers that do not natively support it. The reason why polyfills are not used exclusively is for better functionality and better performance. Native implementations of APIs can do more and be faster than polyfills. So polyfills were a huge part of navigating the ES5 and ES6 years of JavaScript development. Now they're less critical now, but there are still some very popular projects like core-js. So as of the time of recording, if you want to use some of the features from ES2023, even though it isn't released yet, you can core-js to make that happen. Now it's certainly not the only polyfill library, but it is one of the most common. And in addition to polyfills, we also have another tool, which is even more popular, transpilers. Now it might sound a bit familiar because earlier in this course I mentioned the term compiler. So when we're using that term, I mentioned that a compiler can actually take your code and convert it into something that your computer can understand. Well in this case, a transpiler works in a similar way, but with a different output. So according to Wikipedia, a transpiler is a translator that takes the source code of a program written in a programming language as its input and produces an equivalent source code in either the same or a different programming language. So think of the benefits here. You could write code that uses the absolutely latest and greatest JavaScript capabilities and somehow have this transpiler spit out code that can run on an engine that only supports ES5, for example. And it can inject the needed polyfills and transforms to make the code work in a completely different context than what it was originally written in. Let's use another example. Let's say that you were a Java developer. Note here that I said Java and not JavaScript. And you really like the static typing of Java instead of the dynamic nature of JavaScript. What if you could write in another language and then just have that translate into JavaScript that you could use in the browser? Well, in many ways, that's the kind of thinking that led to TypeScript. And TypeScript is an open source project from Microsoft that is based on JavaScript. It only adds features to JavaScript. So it can completely understand the JavaScript code that you write, it adds in static typing, which enables it to perform static type checking. And this means that many types of defects that might happen related to types within JavaScript could be found earlier in the development process with TypeScript. So their project includes a transpiler. So you can translate your TypeScript code into JavaScript that can run on the browser or in Node.js. The most popular transpiler is Babel, and it can even be used to convert TypeScript to JavaScript, just like the transpiler that comes with TypeScript. And Babel also utilizes core-js to in insert polyfills to provide functionality that isn't yet available in all of the target engines. Now this project is sponsored by some big-name tech companies, and that is kind of the way that most open source JavaScript projects work. The tech companies that utilize a tool make sure it stays up to date for anyone that uses it. Now, this isn't a learning path on TypeScript or Babel or even core-js. And to be honest, to use any of these tools, you need to understand JavaScript first. With that being said, I wanted you to understand how JavaScript versioning has led to an entire collection of tools to enable developers to utilize new capabilities even before those capabilities are available everywhere your code will run. Now at this point, I've covered most of what you need to know to get started in JavaScript. So coming up next, I'm going to review what you can do with this information. Let's chat about next steps.

Next Steps with JavaScript

Diving into JavaScript

Now, really quick, let's review where we've been and where you're going. When we started this course, we didn't even have a working definition of JavaScript. And in that initial module, we gave it a definition. And we ended up saying that JavaScript is a dynamic interpreted scripting language that can be used to create web applications, backend services, games, mobile applications, and just about anything else you can imagine. So, after this, we covered many of the different places where you could find JavaScript in the world from the browser to the server, from our phones to IoT devices, from the cloud to the computers on our desk. JavaScript is truly the most ubiquitous programming language in existence. And next, I walked you through the evolution of JavaScript. We talked about its history, how it's governed, and how versions worked. We even talked about how you can extend the capabilities of JavaScript with tools like polyfills and transpilers. Now what will you do with this information? You might be convinced that you will be a front-end developer using a framework like React. Or maybe you want to build applications in the cloud with Node.js. Or it could be that you've learned about TypeScript in the previous module, and you're ready to dive all in on that. I realize you might be tempted to jump over to any of those paths here, on Pluralsight. But I would encourage you to hang out in this path for a while because to be successful in any of those areas, you will need to have a good working knowledge of JavaScript. In this path, we want you to know how to configure your development environment, how to learn the language fundamentals, how to debug the code that you have, and even try out building your own JavaScript experiences. So after covering the information in this learning path, I believe you can be successful in any of those other areas I mentioned earlier. Finally, I want to remind you about the additional resource I've included with this course. I've compiled a collection of links to various JavaScript resources on my personal blog, and you can access the page with this link. If I mentioned any platforms, frameworks, or tools during this course, I have included those corresponding links on that page. So if you are interested in diving deeper into anything I've covered, you can do it from there. Now, thanks for joining me for JavaScript: The Big Picture. If you've enjoyed learning from me, you can catch me later in this learning path where I'll be covering JavaScript fundamentals. So, feel free to connect with me on LinkedIn or Twitter if you want to continue the discussion on JavaScript. I'm excited to see what you do with JavaScript.