Functional Programming: The Big Picture

by Nate Taylor

Course Overview

Course Overview

Hi everyone. My name is Nate Taylor, and welcome to my course, Functional Programming: The Big Picture. I am a solutions architect at Aviture in Omaha, Nebraska. As computers continue to run on multi-core CPUs, and as distributed computing in the cloud continues to gain traction, functional programming will continue to see more and more usage in the software world. In this course, we're going to first of all learn what makes functional programming different from other programming paradigms, secondly, see how functional programming can help reduce the number of bugs in your application, third, get a glimpse of how functional programming simplifies horizontal scaling, and fourth, discover a list of possible functional programming languages to learn. By the end of the course, you'll see why functional programming matters, and you'll be ready to start learning a functional language in more depth. Before beginning the course, it would be helpful, but not required if you're familiar with an object-oriented programming language. I hope you'll join me on this journey to learn functional programming with the Functional Programming: The Big Picture course, at Pluralsight.

What Is Functional Programming?

What Is Functional Programming?

If you're watching this course then you've most likely already heard of functional programming. It could be that you've seen some tweets or blog posts, or possibly you've overheard someone talking about it at a conference or around the office. And while those people sounded convinced of the value of functional programming, they didn't exactly explain what it is. In this module, you'll start to get a basic understanding of functional programming that will be expanded on throughout the rest of the course.

A Quick Roadmap

I am assuming you're coming to this course with a few questions. Perhaps you find yourself asking how is this different than what I already do. Can I use functional programming in a line of business-type application, or even the the most common, why should I learn functional programming? Over the next hour or so you'll see these questions answered throughout the course. The goal is that by the end of the course, you should have an answer for why you would want to continue learning more about functional programming. You'll also be able to see places where you might use functional programming in place of what you're doing today. Hopefully all of this will make you excited to continue your journey into functional programming. This course has a pretty straightforward path. Starting with this module, you'll learn what functional programming is. Then you'll learn some of the basic philosophy of functional programming in the module Do One Thing Well. After that, you'll see how functional programming can improve code quality in the Reducing Bugs with Immutable Data module. In the module Why Functional Programming Matters, you'll see some of the areas of programming that functional programming excels at and even simplifies in some cases. And in the final module, Where To Go, you'll see next steps that you can take in your journey to learn functional programming. Before getting to any of that though, the next clip starts with a definition of functional programming, so that you and I can be on the same page.

Defining Functional Programming

In order to be able to understand what functional programming is, it will be good to start with a definition. This will ensure that there's a common vocabulary for the rest of the course. According to Wikipedia, functional programming is programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. This requires some unpacking, but doing so will help make the rest of the course easier. The first part of the definition is that functional programming is a paradigm. Simply put, functional programming is a way or style of solving problems. It's not limited to a specific language. Additionally, more and more languages are becoming multi-paradigm languages. For example, both C# and Java have incorporated function programming concepts with LINQ and lambda functions respectively. Chances are you already know a few other paradigms. One common paradigm is object-oriented. A lot of formal programming education starts with C++ or Java. Both of these are object-oriented languages. As such, they focus on concepts like classes, inheritance, and encapsulation. While C++ and Java are both object-oriented languages, the way that they are often taught actually starts with a different paradigm, procedural. When I first learned C++ back in college, the first few months didn't even cover any of the object-oriented side. In fact, it just as easily could have been a C class, which is a procedural language. Additionally, if you've ever written any SQL statements against a relational database, you've taken advantage of a declarative paradigm. That is you programmed by telling the database what you wanted without worrying about control flow. The point here is that if you've already done some programming, you've likely already spent time in different paradigms. So hopefully being able to place functional programming in the same mental category as object-oriented, procedural, or even declarative will help eliminate some of that mental block. After establishing that functional programming is another paradigm, it's important to look at what makes it different from other paradigms. The definition includes two key components that differentiate functional programming from other paradigms. The first of those is that it treats computation as the evaluation of mathematical functions. That statement makes it seem like functional programming is not for everyone. It also is a common reason people ask if they can use functional programming in a line of business application, but this is a case where the precision needed for the definition causes some undo concern. The key phrase there is mathematical functions. A function in mathematics is something in which the output maps one to one to the input. For example, the equation of a straight line is y = mx +b, where m is the slope and b is the offset. Once those are established, a single value of x will only ever produce a single value of y, and it produces a graph like the one you see on the screen. In this graph, the mathematical function is simply y = x. That is every value of y is determined by the value of x passed in. As you can see here with the dot that when you pass in 1 for x, it produces a value of 1 for y. But how does that relate to this paradigm? In the same way that a mathematical function

maps the results one to one with an input, in functional programming, the individual functions will always return the same result given the same inputs. Functions like these are called pure functions. You might be thinking to yourself that your object-oriented functions also always return the same result when given the same input, but that's not entirely true. For example, take a quick look at this code in which Math. random is called two different times. In this code you call the same function twice, but you get two completely different results. This is true even though you passed in the same arguments, and in this case you passed in no arguments. In fact this is one way to know that a function isn't pure, if it doesn't have any arguments, it can't possibly be pure. Other paradigms including object-oriented do have pure functions. For example, this code in C# demonstrates a pure function. Math. sqrt of 81 will always return 9 no matter how many times you call it. However the difference here is that while you can do this in other paradigms, you must do it in functional programming. These pure functions actually contribute to the second key element in the paradigm, and that is they avoid changing-state or mutable data. This can be a bit tricky because it's impossible to write any meaningful program that doesn't change data. Thinking through the average program, you'll realize all it's doing is taking some kind of input, doing some kind of calculation, and delivering output, but in order to do this the data must be able to change. An example here will be helpful. As with pure functions, start by looking at some object-oriented style code. Here you have shoppingCart. AddBook and shoppingCart. GetTotal. Even if you've never worked on an ecommerce site, you can probably guess what this code is doing. It's taking the book You Are What You Love, and it's placing it in the user's shopping cart. After that when getting the total, it's summing up all the elements in that cart. If you were to dig down a little bit deeper into the GetTotal function, it might look something like this. GetTotal returns an Items. sum of the price. Here Items is an array of the items that have been added. Any time AddBook was called, it placed a new item into the Items list, that is the AddBook function is updating the state on the shoppingCart class. This is a fairly common approach in object-oriented land where one method will update the state and another method will perform calculations on that new state. In functional programming, these two operations would look slightly different. For example, to add a new book to the shopping cart, you might write code that looks like this, updatedCart =add shoppingCart, You Are What You Love. And to get the total price of the current shopping cart, you might call price = getTotal updatedCart. In both of these functions, you're supplying all of the data needed to make the calculation, and every time you call getTotal updatedCart, you'll get the same total because updatedCart will have the same data in it. That was a pretty long definition of what functional programming is and some of how it differs from other paradigms. If you don't yet quite get it 100%, don't worry, the rest of the course will go into more detail on these points. The main takeaway at this point is that functional programming by its nature is a different approach

to writing software. It does not mean that you can't do things like add a book to a shopping cart. Instead, it simply means that you would do so in a different way. There are some benefits to writing software this way, and you'll learn more about those in the modules Do One Thing Well, Reducing Bugs with Immutable Data, and Why Functional Programming Matters.

Contrasting with Object-oriented Programming

Through the examples that you've just looked at, you've already seen some differences between functional programming and object-oriented programming, but it'll probably be helpful to take a step back and look at some of the fundamental differences between these two paradigms. Martial artist Bruce Lee once said I fear not the man who practiced 10, 000 kicks once, but I fear the man who has practiced 1 kick 10, 000 times. His point here is that if you practice a single type of kick 1 time, you're not going to be very proficient with it, and sure, you may know 10, 000 kicks, but you don't know any of them very good. On the other extreme, you may only know 1 kick, but if you've done that kick 10, 000 times, there's a very good chance that you will be good, if not even great with that kick. You might be asking what does kicking have to do with functional programming? Well computer scientist and professor Alan Perlis had an observation similar to Bruce Lee's. He remarked it's better to have 100 functions operate on 1 data structure than 10 functions on 10 data structures. Similar to Bruce Lee's quote, at the end of the day there are 100 functions whether it's on a single data structure or spread across multiple. While the quantity does not change, the quality does. And this is one of the fundamental differences between object-oriented programming and functional programming. You might be asking how does this play out in functional programming languages? The answer is that functional programming has fewer data structures than object-oriented programming. For most functional programming there are two major data structures that are used. The first is a list. A list is similar to an array, it holds multiple values. These could be primitives such as integers or strings, or more complex data structures. The other data structure is a map. This is a data structure that has a collection of key value pairs. The combination of lists and maps will allow you to accomplish quite a lot in functional programming. Contrast this with object-oriented in which the norm is to create new classes to handle new functionality. Your class will consist of data such as fields and properties, as well as functions that execute on the data. This is where Dr. Perlis' quote comes into play. Object-oriented programming is the case where you would have 10 functions on 10 data structures, but functional programming would give you the 100 functions on a single data structure. In practice, this seems to be the secret to adopting functional programming. Other aspects such as pure functions and immutability can occur in object-oriented programming, but the difference in how

data is processed in the two paradigms can cause confusion and lead to heartburn as someone tries to move from object-oriented programming to functional programming. I completely understand that all of this might be a little overwhelming or confusing at this point. You might even feel a bit of tension as your brain tries to take a functional programming concept and somehow map it to the programming concepts you've learned in object-oriented paradigms. If that describes you, I want to tell you first of all that's pretty normal. A few minutes ago we talked about how functional programming is a new paradigm. So the tension or overwhelming feeling that you're experiencing is likely a result of your brain adjusting to new ideas. It's causing you to think about this concept you thought you new so well, which is programming, in a brand-new way, and that's one of the benefits of learning another paradigm. Secondly, I want to encourage you to keep going. As this course unfolds, these points of tension will ease. As you're exposed to this new idea in repeated modules and clips, your brain will start to make connections to concepts you already know. I know this first hand that this can be difficult, it took me several attempts to get past that overwhelming feeling, but I'm really glad that I did. It helped me understand so much more about programming. With that basic explanation of what functional programming is, the next step is to look at how this paradigm focuses on doing one thing really well.

Do One Thing Well

Do One Thing Well

In the last module, you heard Bruce Lee's quote that he feared the man who had practiced 1 kick 10, 000 times. You also heard Alan Perlis' statement that it's better to have 100 functions operate on one data structure than 10 functions on 10 data structures. Both of these quotes emphasize the importance of focus. You also saw a quick explanation of how that type of focus applies to functional programming. Now it's time to move beyond the cursory and see the principle in more depth as you start to learn about the special role functions have in functional programming.

Unix Philosophy

There are some similarities between functional programming and how Unix operates. That might not be too surprising as often Unix can be as intimidating as functional programming. However, there are a couple of key tenets of the philosophy that will help you wrap your head around

functional programming. In 1978, Doug McElroy documented the four main aspects of the Unix philosophy. Those four principles first of all make each program do one thing well, second, expect the output of every program to become the input of another yet unknown program, third, design software to be tried early, and four, use tools over unskilled labor to lighten the load, that is write a script instead of using a brute-force method to transform data in a file. Again McElroy was talking specifically about how to write programs for Unix, but at least the first two and probably even the first three apply also to functional programming. Applying these principles to functional programming might look something like this. First of all make each function do one thing and only one thing well. Second, expect the output of every function to become the input of yet another unknown function. And third, design functions to be tested early. The rest of this module will cover how these three principles apply to functional programming. Looking at functional programming in this way will help you understand some of the fundamentals of the paradigm without getting bogged down in any one language's specific syntax.

Focus on One Thing

The first principle to look at states that you should make each function do one thing and only one thing well. If you're thinking well that seems like good coding in general, you're not wrong. Keeping your code focused is a good practice and it'll help eliminate errors, but you might be surprised to find out that often times functions that appear to do one thing are actually doing multiple things. The following example is going to parse a text file and find the courses that belong to a specific author. For example, the text file might look like what you see on the screen, Postman Fundamentals by Nate Taylor, Getting Started with Elixir by Nate Taylor, Advanced AngularJS Workflows by Jonathan Mills, and React: The Big Picture by Cory House. If you passed Nate Taylor to the function, you'd expect to get an array with the first two courses, and you'd like them to be title cased like they are on the screen. The code on the screen now is a function inside a class called courseList. Before continuing on, note this is more or less pseudo code. The functions and syntax likely don't exactly match any one language. However, the concepts should be familiar if you've done C++, Java, C#, or something like that. The first thing to note is that is that FindCoursesForAuthor is taking in a string for the author name to look up. At first blush, this function looks like it's doing just one thing, it's finding the courses you asked for. But as you dig a little deeper, it's doing a bit more. Starting on the first line of the function, it's opening a file and reading the contents. The next line has a function splitting that text into an array so that it can iterate over the array of courses. Finally, inside the loop it starts finding the courses that you requested, those that contain the author's name. But if you look a little bit deeper at the line

authorCourses. push, you can see it's doing one more thing. Before it pushes contents into the array, it's converting the string ToTitleCase. All of this is ultimately accomplishing what you want, it's giving you a list of title case strings for courses created by an author that you specified. But this function is doing more than one thing, in fact this function is doing at least four things. First, it's opening a file, then it's splitting the contents, next it's filtering the courses, and finally it's title casing the strings. If you're used to writing code in this manner, you likely didn't even think about items number one and two being separate tasks. I know that when I first started playing around with functional programming, that's exactly what I thought. I thought well of course I opened the file and split the contents, how else am I supposed to filter the courses? You might also be thinking how could functional programming do this in a more focused way? That is how can the function that you previously saw be re-factored to do one thing and only one thing well? Well much like the object-oriented code, the code block on the screen is also pseudo code. I tried to use a different syntax so it would be more apparent in this course when we're playing with object-oriented versus functional. Here the findAuthorCourses function is only doing on thing. It's taking a list of courses and it's passing them to a filter function. That filter function is calling a separate anonymous function. In this case it's x. contains author. Often times this function will be called a test, and anything that passes the test will be returned. Anything that does not pass the test will not be. After looking at this function, you might be saying come on Nate, that seems like you're cheating. After all the functional findAuthorCourses doesn't even return the same result as FindCoursesForAuthor. It's a completely different function, not a refactor. And if you had unit tests for the first function, they'd all probably fail on the second function. While you might think this is just a contrived example to prove a point during a demo, I can guarantee you it's not. Yeah, the function only does one thing, but that's the point. Additionally do one thing well is just the first of our principles for functional programming. As this module continues, you'll see how this function fits into the larger scheme of things and will eventually accomplish the same results as the object-oriented version.

Type Signatures

Before adding in the other functionality such as opening the file or title casing, allow me to take a brief detour to talk about type signatures. Start by looking at the functional code from the last clip. Look at the inputs here. What type are they? Well it's kind of hard to tell because I've made this pseudo-functional code dynamic, but from context, you can tell that courses is likely an array of strings, or at the very least it's an array of a type that has a contains function. Author is likely a string or again at the very least it's a type of whatever is in the array that's passed in. Besides the

variables names, there's nothing here that is specific to courses or authors. That is this same function would work for every combination of string and string array that you passed in. If this code were written in C# as an example, you could make both author and courses be generic by using the IEnumerable of T. This tells the. NET compiler that the course list and result list are both made up of elements of the same type, T. And that type is also the same as the individual author. Further the type T here is a placeholder, meaning C# doesn't care what type it is, as long as all four of these types are the same. In functional programming, you'll often see type signatures instead of method signatures. For the function findAuthorCourses, you might see a type signature like what's on the screen. This'll often appear in the docs, as well as immediately above the function definition, for example, it might look something like this if you were to be reading a code base for functional programming. The signature can be broken down pretty quickly. Before the double colon is the function name, in this case findAuthorCourses. The parentheses represent the input parameters, in this case that's a string and an array of strings. The skinny arrow indicates what will be returned, in this case an array of strings. If you had a similar, but slightly different function like the one you see on the screen, your type signature could be modified as well. Here instead of courses, the list is a list of authors. Your type signature might look like this. Notice it's been genericized to use a instead of string. In functional programming, a simply represents some type. This signature is telling you that if you pass in some type and an array of the same type, it'll return an array of the same type. That is where you would see t in C#, you'd likely see an a in functional programming. In contrast, if you saw the type signature that's now on the screen, you would know that the following function takes an input of some type and an array of some other type, and it returns an array that has the same type as the first parameter you passed in. Type signatures like this help highlight how generic a function is, similar to templates in other languages. For our function finding courses by the author name, it doesn't make much sense to genericize it anymore, but it's always something to pause and think about as you're writing the code. I'll often ask myself will this only work on the data type I'm using right now, or will any type work? And when you're focusing on only doing one thing well, it becomes easier to think about a function working generically. Why was this little detour important to take? Well there's a few reasons. First as you continue your journey and read docs for libraries and languages in the functional realm, you'll see type signatures more and more. Second, it again highlights Alan Perlis' philosophy, that is a lot of people writing functional code will tend to make their functions generic. Sure you can do that in other paradigms, but it seems to be more prevalent in functional programming. These type signatures are a reminder and sometimes even an encouragement. Now you want to have those 100 functions on a single data structure instead of 10 functions on 10 different data structures. And finally it'll simplify the second principle. The next clip will dive into

the second principle of functional programming in more depth, and by the end of that clip, you'll see how type signatures can assist you in composing your application.

Expect Output to Become Input

The second point of the Unix philosophy, which is being applied here to functional programming is that you should expect the output of any program or function to become the input of another yet to be written program or function. Applying type signatures can help implement this principle. By examining the type signatures, you can begin to see how various functions will relate to each other. For example, the type signature on the screen is for a function that takes a list of one data structures and returns a list with a different data structure. The new type signature that's on the screen for funcB shows that it's a function that takes a list of data structures and returns a single instance of that data structure. Looking at the two signatures, you can see that the output of funcA can easily be passed in as the data to funcB. Don't get hung up on the fact that funcA returns a list of B. In this case B is not a specific data type, but instead indicates that funcA is transforming the data from one type to another. That is A can simply be thought of as the first data type, and be can be thought of as the second data type. But in this case, the second data type can become the first data type of another function. You've no doubt taken the output of one function and passed it into the inputs of another function in various paradigms. Going back to the object-oriented approach, you've likely done something like the code that you see on the screen. Where you call funcA and assign it to a tempResult, and then you call funcB, passing in tempResult. Of course you could also eliminate the temp variable, and it would accomplish the same result. However, the nice thing about temp variables is that it can help improve the readability of the code. This option that's on the screen now, to me anyway, is harder to read because there's multiple parentheses, square brackets, and curly brackets, and it's all on one line. Since functional programming focuses on allowing the results of one function to flow into another function as it's input, several of them have developed an operator called pipe. For the purpose of the pseudo-functional code, we'll use the pipe operator, which is on the left-hand side of the screen. Not coincidentally, this is the same operator that's used in more than one language. This operator tells the language to take the output of one function and use it as the input to another. So chaining or piping those same function calls would look like the code you see on the screen. And notice that there's not a parameter passed in to funcB, that's because the pipe operator will do that for you automatically. It's time to apply this principle back to the function that was discussed in the focus on one thing clip. If you remember, we left that clip with the function that you see on the screen now. At the time I remarked that while you might think I was cheating by

simplifying this function so dramatically that eventually you'd see how it came together. Well the time to see that is now. By using the pipe operator in our pseudo code, you could write code that would return the same result as the original object-oriented code. Walking through this code, you can see that the first line reads a text file in a single string. This long string is passed into the second function, which splits the string on new lines, and it returns an array of strings. That list of strings is then passed into the findAuthorCourses function, along with the name of the author. This function returns a list of strings. Remember that the language is piping the data into this function for us, so the pipe syntax is the same as the code you see on the screen, that is the pipe supplies the courses for us. Finally, the last function takes a list of strings and it returns a list of the same length, but this time those strings are title cased. At this point you have four individual functions, each one only doing one thing well. These four functions are chained together to produce the same result as the object-oriented approach earlier. That is by having functions that do one thing well and who return data that they expect to be inputs to another function, they've decomposed a harder problem into small succinct steps. There's one more benefit to these four functions. That is because they focus only on doing one thing well, they can be used by other functions. That is you can likely imagine a lot of scenarios where the SplitOnNewLine function could be used, or even the titleCase function. Had they just been wrapped up in a larger findCoursesByAuthor function, then every time you wanted to split on a new line, you'd have to write that code again. By following the first two principles of functional programming, you now have well-defined utility functions that can be used in other places.

Test Early

The third principle from the Unix philosophy that applies to functional programming is to test early. To be fair, in my opinion testing early applies to all software regardless of the language or paradigm that it's written in. When you test early, you're better equipped to be able to catch errors in missed cases before it's too late. It can be quite frustrating to spend a significant amount of time working on a piece of software to only realize days or even weeks later that it's just not quite right. Even worse, if you realize those changes that you need to make are now more difficult because of how long you waited to test. It's important to figure out what test early means. In the Unix philosophy, the original author intended software to be tested within weeks, and then throw away the clumsy parts or parts that didn't work. But I think that's way too long, particularly if you're working on a team. Think about how much code can be generated in just a single week. Waiting until that week is over to test the code could produce an overwhelming amount of work. I recommend testing within minutes. The easiest way to test your function within minutes, not

weeks is to have good unit tests. Unit tests are not unique to functional programming. In fact, I hope that you're writing them in whatever language you're currently using. While unit tests are not unique to functional programming, it does offer some advantages when it comes to testing. First of all, setup is way easier. Often times when you're testing code, it can be hard to determine all the different mocks and fakes that are necessary for your test to work. It can often require setting several properties on an object or a series of objects just to be able to have the right data to test with. However with functional programming, that's greatly reduced because the only data a function will operate on will be the data that you pass in. As a result, there's often less need to mock or fake out any function calls, and you can typically pass in the exact right data easier to a single function. Second, verification is easier. As with setup, it can often be difficult to determine if your code did what you thought it was going to do. It can often require checking that some other function was called with the correct data or was even called at all. However with functional programming, every function returns a value. As a result to test if a function did the correct thing, you simply needed to check its return value. And since functional programming focuses on each function doing one thing well, you should be able to determine what the output is supposed to be without much hassle. Imagine testing the function splitNewLine from the last clip. You know you would need to pass in a string with a new line, and that you should get a list with two items back as the result. And while this may seem like a trivial case, remember that function is embedding the principles of functional programming. The output is only dependent upon the input, and the function is focused on doing one thing well. Granted not every function you write will be that simple, but if you keep those principles in place, testing code from a functional code base should be pretty straightforward.

Complexity through Simplicity

This module could be summarized by saying that functional programming solves complex problems via simple methods. Functional programming attempts to decompose the harder problems into simpler problems, and then to decompose those simple problems into even simpler ones, until all that's left is a series of functions that each do one thing particularly well. After identifying those simple solutions, functional programming encourages you to use those building blocks to solve your more complex problem. It accomplishes this by chaining those single-purpose functions together. Composing functions in this way allows you to create more complex functions. And because those functions are pretty straightforward, they can be more easily tested. This means that you can have a higher degree of certainty that they'll work on the various data that you pass in. And that is how Alan Perlis can achieve his 100 functions on 1 data

structure. By building up lots of small functions that apply to any list or any map, you begin to eliminate the need of defining custom data structures. At the end of the day, doing functional programming isn't so much learning new syntax, but shifting it how you look at problems. And one way to shift how you look at problems is how you utilize data. In the next module, you'll learn how functional programming's emphases on immutable data helps eliminate bugs.

Reducing Bugs with Immutable Data

Reducing Bugs with Immutable Data

Bugs are the enemy of every program and every programmer. In the previous module, you learned about making sure each function only focused on one thing. This is often credited as one of the ways that functional programming makes it easier to reason about the problem. However, pure functions are not enough to reduce bugs in and of themselves. Another major factor in simplifying reasoning about software is immutable data. As the name implies, immutable data refers to data that does not or cannot change. Since most developers have not worked with immutable data, this module will cover what immutable data is and show how it can help reduce bugs in your code.

What Is Immutable Data?

Sometimes talking about programming concepts in the abstract can make it harder to wrap your head around. So before digging into exactly what immutable data is, a code example might be helpful. After watching the last module, you started writing code in a more functional style. So you created the pure function that you see on the screen. It takes a cart and a single item. It then adds the single item to the list of items in the shopping cart. Next, it updates the total price of the cart. Finally, it returns the cart. This function only depends on what is passed in and only alters what it returns. That is based on the last module, it's a good example of functional programming. However, that code is not an example of functional programming because it has mutable data. That is you're changing the values of both cart. items and cart. total. Looking back at the definition of functional programming from earlier in the course, it highlighted that functional programming will avoid changing state and mutable data. Digging into what immutable data

means, Wikipedia states it is simply an object whose state cannot be modified after it's created. This definition in and of itself is pretty straightforward. Once you create an object or a data structure, the data cannot be changed. While the definition is pretty straightforward, it does raise some questions. Most notably, how can my program do anything if the data can't change? Revisit the code example from earlier. If you were to eliminate all of the lines that change data, your function would look drastically different. It would just be like the code on the screen. Your function now doesn't actually do anything, except return the car that you already passed in, which means you'll never be able to add any items to be purchased. However, I've told you that functional programming is as valid as a paradigm as object-oriented programming. I've promised that later in the course you'll learn about some of the areas where functional programming really thrives. So there must be some way to change data. And yeah, the previous example was a bit of hyperbole, but I wanted to drive home the point that we often change data without even thinking about it. So how would a functional programming language handle the addToCart? The code on the screen shows an example of how functional programming could accomplish this update to the shopping cart. To start, the percent with the curly braces syntax is how our pseudo code is going to show a map. Also there's no return statement because this pseudo-functional language will simply return the value of the last expression. So in this case, addToCart is returning the new map that is created. The next difference is subtle. The cart. items. concat function here does not update the cart. items list, but rather it takes that list, adds the newly-supplied item, and returns a new list. This new list is then set to the items key of the map. Finally a similar operation is performed for the total. Again instead of attempting update the cart. total, it takes that value, adds the price of the new item, and assigns that to the total of the new cart map. As I've already said in this course, the challenge of functional programming likely won't be the new syntax. Instead, it'll likely be the change in mindset and approach. Immutability can be one of the bigger shifts in mindset, and while the past few minutes have not been enough for you to become an expert in immutability, they should've shown you that at its core, it's really not that difficult of a concept. You likely still have plenty of questions about immutability, for example, how would you call functions with the understanding that the data is immutable? That'll be covered next.

How Does Immutability Reduce Bugs?

Looking at how immutability works throughout a series of function calls will help show how it reduced bugs. Earlier you saw the function that's on the screen now as an example of a function in a functional programming language, specifically how it deals with immutable data, but this is just one function and it'll likely be used with other functions. This code that's now on the screen

shows the addToCart function as part of a larger process. It starts by adding the book map to an empty cart. What is returned is the cart variable, which now has one single item and has a total of 10. 00. The value of cart is what you see on the screen. Items is that list of 1 item, and the total is 10. 00. After it's been added to the cart, the customer decided that they wanted 2 copies of the book. The function updateQuantity is called with the new cart value, as well as the book that they want to update and the new quantity. This function returns another new cart. It has all the data of the original cart, but now the quantity for the book is 2, and so the total is doubled as well. At this point, the value of cart is unchanged. However, cartWithMultipleCopies has the value that you see on the screen now. It has the same title, but now the quantity is 2 and the total is 20. 00. Finally the customer's applied a 20% off discount. They accomplish this via the applyDiscount function and the code 20Off. After calling this function, cart still remains unchanged, cartWithMultipleCopies is also unchanged, but the new variable cartWithDiscount now has the value that you see on the screen. It has the same number of items, but now the total is 16. 00. Throughout this walkthrough, you can see how cart, cartWithMultipleCopies, and cartWithDiscount are each different states of the system's shopping cart. However it still might not be obvious how this can help reduce bugs. Because of the fact that functional programming uses immutable data, it's not possible for a function to change your data without you being aware. And in fact even then it's not changing data, it's giving you a new struct with the changes included. In contrast, mutable languages have functions that update an object, perhaps without you even being aware. A common case of unaware immutability is in arrays. An example can be seen with the code on the screen. Before continuing on, do you know what the output is going to be of that Console. WriteLine? It's going to be lower case a, capital A, then b, then c. Why are other letters sorted in this order? Because in this case sort is not immutable. That is it doesn't return a new sorted list, instead it sorts the current list in place. And because otherLetters was set to the same reference as letters, they both get sorted. If this was the block of code you had, it might not be too hard to detect this bug, but imagine instead of all the code being contained in about 6 lines, it was spread over a function that was 50 or even 100 lines long. By the time you got down to that last line, it's quite possible you wouldn't have been able to easily predict the output because you would've seen that otherLetters was assigned a reference further up the stack, and that the sort happens somewhere in the middle. In addition to that example, in a lot of languages the native list or array is not considered thread safe, therefore it's possible that thread A could be iterating over the list, while thread B is adding to that same list, which means your expected results could be completely thrown off on thread A. These types of bugs become particularly nasty to track down because it's not a single function updating the list, but instead an entirely different thread. This would not happen with functional programming though because

with immutable data, thread B could be creating all the new lists it wants, but the list being operated on in thread A simply isn't going to change. By keeping your data immutable, you'll be able to avoid an entire class of bugs, bugs that can be often painful to track down and diagnose. Knowing that the data isn't going to change allows your code to be easier to read and reason about.

Performance of Immutability

Now that you've seen some examples of how immutability can be used by functional programming, you might be thinking with all those copies of the data running around, won't that negatively impact performance? Well the short answer is no, but it's important to understand why the answer is no. This is the code that was shown earlier in the course to highlight how immutable data works. It was the entire process of interacting with our shopping cart. The code on the screen is the original cart. The code on the screen is the original cart variable. This is what our shopping cart looked like immediately after the first book was added. The items property in that map simply has 1 book. If the customer added a second book, the updatedCart variable might look like what you see on the screen. That is the items list would now have two maps inside of it, and the total would be updated. Another way to look at the updatedCart is to consider the hash values of the pieces of data. Please note that this is for illustrative purposes, actual hash values would likely look completely different. However note that the first line contains the hash of the entire cart. The second line shows the hash of the items list that belongs to the cart. More than that, it shows the list contains two items, and it shows each of their hashes. And finally it shows you the hash where the total is stored. Now imagine that your user has added a free item to their shopping cart. The cartWithFreeItem would look something like the data structure you see on the screen now. From earlier in the course you know that this cart must be a different structure than the previous carts because the data is immutable. However the first two items, as well as the total are still the same. So what would the hashes look like for that? The hashes on the screen will highlight the differences. In this case there's a new item in the items list, additionally the item's hashes and the cartWithFreeItem both have new hashes. This is because the compiler has done some optimization. It knows that item 1, and item 2, and total have not changed. So it can reuse those items, and as a result, there is no reason for their hashes to have changed. The reuse also reduces the memory needed to hold both updatedCart and cartWithFreeItem. Additionally, this can make checking for equality much easier for the language. If you wanted to see if the items lists were equal, it would only need to check their hashes. If the hashes are identical, it knows that every element inside that list is also identical. But since the hash for updatedCart. items and

cartWithFreeItems. items are different, it knows that there are some differences. It does not need to navigate into each item in the list to compare it. However even if it did need to traverse the entire list, each item in the list also has a hash, so it'd be able to tell that item 1 and item 2 are the same in both lists. It does not need to actually compare the title of item 1 in both carts. Since the hashes are the same, all the values inside that map are the same. I can't stress this enough, but this is a simplified illustration of how immutable data can still be performant both in memory storage and equality comparisons. The actual ways that a functional programming language accomplishes this are much more details and vary from language to language. The takeaway though is that even though immutable data can lead to more copies of the data, it does not necessarily impact performance. In many cases, it can even be more performant. As a result, not only can immutable data help reduce bugs, but it can also do so in a performant way. This is just a small example of ways that functional programming can improve your applications. The next module will examine a couple of other reasons that functional programming matters, and how it can help simplify specific issues in programming.

Why Functional Programming Matters

Why Functional Programming Matters

Up until now, the focus of the course has been some of the paradigm shifts that developers must take in order to grasp the core concepts of functional programming, but until the previous module, which talked about reducing bugs with immutable data, there likely wasn't a reason why you should be pursuing functional programming. That'll change with this module, as the focus changes from the paradigm shift to some of the applications of functional programming. In my opinion, the following applications are glowing examples of why functional programming will become more and more important, and also why more and more languages are bringing in functional programming concepts.

Caching

A common problem in software development is how to increase performance of an application, particularly when the same data is being requested. A common solution to this problem is to

cache data. With functional programming, this can be accomplished rather easily and at a finer-grain level than is normally thought. About 6 or 7 years ago, I was working on a dashboard for a large corporation that had lots of different regions and districts. This dashboard was to track and display the various performance metrics that the management team was interested in. The problem was the systems behind the scenes weren't exactly streamlined. As a result, a query for a particular district might take almost a minute to run. However, the data for this system only changed about once a day, and for some of the queries, things like sales last month, the data shouldn't be changing at all once the new month had started. As a result, the development team I was on decided that we should start caching the results. That way the website would appear to be more performant. In fact we could even possibly write a script to execute a few common queries, so those would always be cached. However, we quickly ran into problems. For the most part, the cache worked; however, busting the cache wasn't always as simple as we would have thought. There were times that changing the parameters would not always bust the cache. In the end, we had to work up an entirely different solution. But to this day when I talk to my former teammates and mention caching, I get a knowing look as we all remember the nightmare we experienced on that project. In contrast to that nightmare, functional programming provides ways to more easily cache the results of functions. And as is often the case in functional programming, this caching comes with a new word, memoization. Memoization is an approach to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Memoization is just one subcategory of caching, and in this case it deals specifically with caching the results of expensive functions. However, expensive functions can be kind of a relative term. How does functional programming take advantage of memoization? By going all the way back to the basics. Functional programming uses pure functions, functions that only depend on their inputs. If a function only depends on the values that are passed in, it's much easier to memoize the results. The code that's on the screen now shows a function that will take longer to run the more items that are in the shopping cart. It'll iterate over each one and add that item's price to the total price for the cart. In reality, it's probably not a good approach, but it's also one that you wouldn't be surprised to find in a code base that you are working on. Since this is a pure function, the only thing this calculation depends on is the list of items that are passed in. As a result, this function would be a candidate for memoization. With a slight change to the previous function, this function now has its results cached. Any time a user passes in a list of items that's already been calculated, the function will automatically return the cached item. Remember from the previous module because functional programming uses immutable data, determining equality is fairly quick because it can check the hash of the list and determine if it's the same list or not. In addition to this pattern, some

functional programming languages have built-in utilities or helper functions that can make memoization even simpler by not requiring you to write that if/else statement, but instead just making a call to a single function. A common example of memoization comes from Redux. Redux is a pattern that's most often associated with React on the front end. In Redux there's a store of data. This is information that comes back from your API, and it's combined with information that your application has already set. The way to access data in a store is to use a selector, and a selector pulls information out of the store. In some ways it's kind of like a getter in an object-oriented language. However, selectors often depend on other selectors in order to travel deep into the store object. The nested selectors take advantage of memoization. The selectors being pure functions know that if an object passed into the function is the same as the object passed in the previous time, then the data is the same. So it only needs to return the data instead of recalculating the data. This pattern and the libraries it uses allow expensive operations to only be performed when absolutely necessary. This discussion just scratches the surface of memoization. In reality, there are several different ways that your code could benefit from memoization beyond the Redux selector pattern. However, the point here is that functional programming uses memoization not because all the functional programming languages have decided to implement memoization libraries, but rather because at its core, functional programming depends on pure functions, and pure functions greatly simplify memoization.

Laziness

As you just saw, memoization is simplified with pure functions. By only depending on the inputs to the function, the function can be easily cached or memoized, and it's based solely on the arguments passed in. Pure functions offer another advantage to functional programming, and that is they enable lazy evaluation. Returning once more to Wikipedia, lazy evaluation delays the evaluation of an expression until its value is needed. As is often the case, at least for me, the definition from Wikipedia can make sense, but there's still some unanswered questions. Perhaps an example will illuminate the concept. In his book Functional Thinking, Neal Ford shows the code on the screen when discussing lazy evaluation. The output that you would see depends on if you're doing lazy or eager evaluation. With eager evaluation, the code would throw some kind of DivideByZeroException. This comes from the third element in that list. Eager evaluation would create the array 3, 6 divide by 0, 1 before it would calculate the length. That is it would perform each of the operations in the list. In contrast, if the language you're using is lazily evaluated, the output would simply be 4. That's because a lazy language would not care what the individual elements of the lists are, or at least not yet, since the line on the screen is only concerned with

what the length is. Further, it would be possible with lazy evaluation to be use values from the list, provided it never needed the third element. That is if the code using this list stopped after the second element, there's be no problem. Sometimes grasping the value of lazy evaluation can be harder with lists. Often times when people discuss functional programming in laziness, they'll talk about infinite lists such as streams. And while it's true that laziness allows for that situation, I often found it hard to wrap my head around that. Instead what I like to do is I like to think of a really large database because for all intents and purposes, that's kind of infinite. The code on the screen shows an example of how a database could be used in a language that does lazy evaluation. It begins by constructing a query, including an instruction to sort the records, but at this point nebraskaTaylors does not hold any data because it's not yet needed. The code in fact won't even be evaluated until there's a line like what now appears on the screen. At this point, nebraskaTaylors will be populated with all of the data that matches the query from the database. But the key here is that it did not do multiple queries, that is it didn't go out and query users where the last name equals Taylor, and then a separate query where the state equals nebraska, and then finally a third query where it sorts all of that. Instead it did a single query because it waited to do the query until everything was needed. And that's one of the major benefits of lazy evaluation. It defers execution, including expensive execution like hitting a database or running a long-running query until the time that it's actually needed. And if it's never needed, then the expensive operation will never happen. All of this is enabled by the power of pure functions. If a function does not depend on anything other than what is passed in, then it doesn't matter when it's executed, provided it's executed by the time the result is needed. It's important to note that laziness is a possible benefit of functional programming. However, if you remember back to the module What Is Functional Programming, you'll remember that laziness was not included in the definition of functional programming. There are a few prominent functional programming languages that do not use lazy evaluation. However, it's more common in functional programming than in other paradigms.

Parallelism and Concurrency

While the previous benefits have been made possible by how functional programming uses functions, this last benefit will come about in a large part because of immutability. Have you ever watched your task manager while executing code on your machine? If so, you might've noticed that it looks similar to the one that's on the screen now. That is you might see multiple cores, in this case 8, but none of them really appear to be doing too much. In fact, only half of them are doing anything at all, and even then the first core, which is the one working the most is only at

about 30%. For most of your applications, that's probably okay. After all with each successive release of a CPU, the performance of your application improves, but sometimes you have an application that is a very processor intense. For example, file processing. Reading in a file is not very CPU intense, but depending on what you are doing once that file is read in, the CPU usage could escalate. As an example, I once wrote a program that would take a large text file, and by large here, I mean several copies of the King James Bible, and sum up the usage of each word in that file. The first attempt I didn't do anything special, I streamed in the file and tried to sum up each word. The CPU usage looked about like what you would expect, 4 cores, 1 or 2 or them doing more work than the rest, but none of them really doing much work. The total execution time of this approach took about 23 seconds, and that was on an 8 core 16-gig computer processing multiple copies of this King James file. For the next iteration though, I parallelized the work. I chose the same programming language, performed the same task, and got the same output. In fact I actually even used the exact same module, I simply used a different function on that same module. However, the parallelized version utilized all 8 of my cores, and the total execution on that same 8 core, 16 Gb computer took about 7 seconds. What made this performance gain possible? Well a major contributor was that the data was immutable. How did immutable data make this possible? Well take a quick look at how the program flow happened. To start, look at an example sentence from the text. In the beginning, God created the heaven and the earth. This is a good example sentence, as you can see the same words occur three times, the word the. The flow of the program then splits the entire text on spaces and new lines. So at the end it would've had a list that looks something like in, the, beginning, etc., basically one word per item in the array. Once it had that list, it partitioned the list into smaller lists. The function did not instruct how the list was to be partitioned, only that it needed to do it. As a result, one possible way would've been to give each core one-eighth of the list. So it might look something like this where core 1 got the words in and the, and core 2 got the word beginning, all the way up to the last core getting the word earth. After I had all those lists partitioned, each core could do a sum of the word count of the list that it had. Ultimately each core passed back the results, and the final totals were tallied. Because the entire list was immutable, the program knew that there was no other code manipulating the list, that is the list on core 3 was only concerned with its own data. It knew there was no process that was going to add data or subtract data from its own list. As a result, it never needed to stop and check if the list changed, it only needed to move as fast as possible to count the words. Beyond immutability, the example code also takes advantage of lazy evaluation. That is it treats the file as a stream and only executes on that stream when it needs the data. While this example was contrived solely to show off the power of parallelization, I hope you can start to see that power. By leveraging pure functions to do the calculation, laziness

to load the data and using immutable data with functional programming, parallelization naturally becomes easier. In fact this benefit is probably my favorite application of functional programming because it opens the door to making scaling an application almost easy. Any time I have a set of functions that I know aren't relying on some internal state, and I have data that I know isn't going to change, I know I have the opportunity and recipe to split that out into different cores or even entirely different systems. In fact, functions as a service like AWS Lambda take advantage of this very principle. You hand lambda some data, and it performs a function and hands back a result. If you need to spin up several lambdas to process your data, you're able to scale your solution horizontally without much difficulty. In addition to parallelism, this module has shown you two other reasons for why you might want to use functional programming. And while these each have their own place, the bigger picture is that as we continue to produce more and more data, functional programming is equipped to handle that amount of data, whether it's by caching, delaying execution, or through processing the data in a distributed fashion. If you're interested in learning how to put functional programming to use in the real life, the next module will examine some steps on where you can go after this course.

Where to Go from Here?

Where to Go from Here?

My hope is that by this point you've started to see some of the benefits of functional programming. Perhaps it's memoization, parallelism, or reducing the bugs in your code, or maybe those final applications aren't even your primary motivation. You simply like the ability to use small, well-defined pure functions in your code. Whatever it might be, by this time I hope that you're at least excited to start writing more code using functional programming. It's quite possible that even though you've seen things in this course to peek your interest, you don't feel any closer to being productive with functional programming than when you started. This final module will help you get on the right path by looking at the next steps you can take in order to master functional programming.

Starting the Transition to Functional Programming

The good news is that you can start moving towards functional programming without adopting a new language. The first step you can take toward functional programming goes back to the

module Do One Thing Well. That is there's nothing stopping you from writing pure functions in your current language. As you go back to your current language, try to eliminate void functions, and try to write wherever possible functions that only depend on the parameters that are passed in. While any language can take advantage of pure functions, some languages have other constructs that make adjusting to functional programming even easier. For example, since. NET 3. 5, C# and VB have had LINQ. LINQ stands for Language Integrated Query, and for most. NET developers, the first time they encounter it is when working with databases using ORMs like Entity Framework or LINQ to SQL, which is now obsolete. But LINQ doesn't have to just to be for database queries. The functions such as Where or Select demonstrate some of the concepts that were discussed earlier. For example, LINQ functions are pure. They don't depend on any external state, they only depend on the data that's passed in. LINQ functions also don't update the data, they instead return new instances of the data, that is they don't mutate state, so therefore they're immutable. And finally LINQ functions are designed to be piped into other functions. The code on the screen starts with a list of users and then filters it down to the list of active users. After that, it selects only the LastName, and then it creates a unique list of LastNames. Finally it takes 5 LastNames from that list. Each of these functions only depends on the data that's passed in, and each of these functions returns a new list each time. In effect, LINQ is allowing you to do functional programming inside of C#. By taking advantage of LINQ, you can begin to incorporate several functional programming concepts even while using. NET. And as you begin to utilize LINQ beyond just data queries, you'll find yourself thinking more and more like a functional programmer. C# is not the only multi-paradigm language. More and more languages that have an object-oriented background have been adopting functional constructs. Included in that list is Java. Starting in Java 8, Java introduced lambda expressions. These functions have much of the original motivation as LINQ, that is they focus on bringing some of those functional programming concepts to a traditionally object-oriented language. C# and Java are just two examples of languages that you might already be using today that can assist you in the transition to functional programming. Most of the time this is the best approach in an existing code base. It rarely makes sense to rewrite an existing code base into a new language or paradigm, but that doesn't mean that you shouldn't try to reap the benefits of functional programming when you can.

Learning a Functional Language

As was just mentioned, it almost never makes sense to rewrite from one language to another purely to switch paradigms. However, if you're starting a new project or want to experiment with more full-blown functional programming, there are some good languages for you to experiment

with. Recall the definition of functional programming from the module What is Functional Programming. There it was defined as you see on the screen. Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state in mutable data. That definition covers a lot of topics, but one topic that it doesn't cover is typing. That is a functional programming language can be a statically-typed language. When a functional programming language or any language for that matter is statically typed, it means that type safety is verified at compile time. C++ is an excellent example of a statically-typed language. When a C++ program is compiled, if there are any type mismatches, you'll be notified at that time. In contrast to static typing is dynamic typing, and just as there are static functional programming languages, there are dynamic functional programming languages as well. A dynamic programming language will verify the type safety at runtime. Perl, Python, JavaScript, and Ruby are all examples of dynamically-typed programming languages. The distinction of static type and dynamic is a good initial grouping for functional programming languages. It's one in which people tend to have strong opinions. Personally despite growing up with static languages, I found that I enjoy working with dynamic languages much more. However, I have coworkers who are just the opposite. They truly enjoy the static nature of their favorite functional programming language. My goal here is not to persuade you to one side or the other, but instead to help you see that no matter which way you prefer, there's options for you to start with. I also hope that your curiosity will get the best of you, and you'll try at least one language from the other side. That is if you really like dynamic languages, give a static one a shot and see how it can be beneficial. The first two static functional programming languages that will be discussed can be seen as gateway languages. That is you're likely already familiar with their ecosystem, but not their syntax. This can greatly aid in your learning as you don't have to learn a whole new series of tools and systems. You can focus on syntax and of course the paradigm shift that I've referenced so much. In these cases, the ecosystem almost acts as a safety net because you're already familiar with a lot of the necessary pieces. The first of these languages is F#. F# is truly a multi-paradigm language. That is it's possible to do imperative, functional, object-oriented, or even metaprogramming using F#. And while it is a multi-paradigm language, it's most often associated with functional programming. This is because it's taken a different route to being a multi-paradigm than other languages. For example, C# started with an object-oriented language and added more functional concepts. In contrast, F# started as a functional programming language and added in concepts from other paradigms. F# is in the. NET family of languages, and that's why I say it's a language you can use if you're already familiar with the. NET ecosystem. If you're already familiar with C#, then the transition to F# can be made with the help of blogs, and videos, and Stack Overflow questions that discuss how something that would be done in C# can

be done in F#. If you're interested in learning F#, I would suggest you check out the course F# Fundamentals. It's just one of several F# courses in the Pluralsight library. Much like F# is part of the. NET family, there's another multi-paradigm language that often operates as a functional programming language that is part of the JVM, and that language is Scala. Scala was created in part to address some of the early criticisms of Java. Like F#, it was also influenced by several functional programming languages and has numerous functional programming concepts included in the language. For example, Scala has both immutability and lazy evaluation. Two concepts touched on earlier in the course are baked right into the language. Also much like how F# will compile down to the. NET runtime, Scala is designed to be compiled into Java bytecode so that it can run wherever the JVM can run. This provides those of you familiar with Java the ability to write code in a new language, but use some of the same tools and libraries that you're already familiar with. If you're interested in learning Scala, there's a course in the library named Thinking Functionally in Scala, as well as a few other Scala-specific courses. The third statically-type programming language Haskell is not one that has its roots in either the. NET nor Java ecosystems. Instead it's a language that influenced several other programming languages including F# and Scala, and it's a statically-typed programming language. Haskell first appeared almost 30 years ago in 1990 in largely academic circles. However, since then its use has grown and expanded outside of academia to include commercial applications. Because Haskell is not a multi-paradigm language, it's truly on a functional language, of the languages mentioned in this course, it'll require the most thorough paradigm shift to truly get behind. However, that's not all bad. As a common concept in the Haskell community is that once you get your program to compile, you have a very, very high certainty that it'll work as you expect it. If you'd like to learn more about Haskell, there is a two-part course in the library named Haskell Fundamentals that'll walk you through the syntax and use of the language. This is just a small sample of the statically-typed functional programming languages that exist. There is also dynamically-typed functional programming languages. And as with the statically typed, there are a few that would be good to be aware of. One dynamic language that continues to grow in usage and popularity is JavaScript. Much like Scala or F#, it is a multi-paradigm language; however, it has definitely be influenced by functional programming. JavaScript users have been using functional concepts, for example, higher-order functions, and do so often without even realizing that these are functional programming concepts. JavaScript as a language doesn't have immutability, but there are libraries that can provide that functionality, as well as libraries that can help assist with other aspects of functional programming such as Ramda or lodash/fp. If you're already using JavaScript and would like to see how it can be used in a functional way, I'd suggest you check out the Fundamentals of Functional Programming in JavaScript course, which is in the library.

Another multi-paradigm dynamic programming language that allows for functional programming is Python. Much like JavaScript, there are already millions of developers that have been using Python, probably without ever realizing that it can be used in a functional way. Also like Haskell, Python was released in the early '90s, in fact it's only one year younger than Haskell. The fact that it's close to 30 years old should provide you with confidence that it's not going away any time soon. Additionally with the ever-increasing demands for data science solutions, more and more work is being done in Python as it and R compete for the two prominent languages to solve data science issues. If you're interested in how you could use Python as a functional programming language, I'd suggest Functional Programming with Python in the course library. The final language to look at in the dynamic category is my personal favorite, Elixir. In fact Elixir greatly influenced the pseudo-functional code that you saw earlier in this course. But unlike JavaScript and Python, it is not a multi-paradigm language. Instead it is only a functional programming language. Of the languages mentioned in this course, it's the youngest, as it was first released in 2011. And while it's relatively new to the scene, it does have an advantage in that it's built on top of the Erlang VM. Erlang is another functional programming language that is used for massively-scalable and distributed systems. Further it's been around since the mid '80s. And while it's built on top of the Erlang VM, it also has been influenced by Ruby. In fact the creator of Elixir spent several years working in Ruby, and some of the concepts are quite familiar to Ruby developers. If you have a familiarity with Ruby, the transition would be similar to going from C# to F#, as there are numerous blogs and articles on how to do something in Elixir that you already know how to do in Ruby. However, even if like me, you have never spent much time or possibly any time in Ruby, Elixir is still a very accessible language to learn. If you'd like to give Elixir a shot, I'd recommend the Getting Started with Elixir course in the Pluralsight library. These six languages, F#, Scala, Haskell, JavaScript, Python, and Elixir are but a small representation of the functional programming languages that are out there. As time goes by, they'll continue to grow and develop, and new languages will continue to pop up. Returning to the goals set out in the module What is Functional Programming, my hope is that you can now see how functional programming is different than what you already do, although it's quite likely you've learned that you're already doing some functional-style programming in the language you're currently using. I also hope that the code samples throughout the course, as well as seeing languages such as Scala and JavaScript allow you to see that functional programming can be used in a line of business application. And most importantly, I hope this course helped answer the question of why you should learn functional programming. There's not a single answer for that question, it's typically different for every person. For me, it was about the possibility of concurrency in parallelism combined with the idea of small, succinct functions. But those don't have to be your why.

Regardless of why you would like to learn functional programming, you now have a good basic understanding, as well as several possible next steps that you can take. Finally, as always, I want to say thank you. I'm glad you made it to this part of the course, and I hope that it was helpful for you. I hope that you enjoyed watching it as much as I enjoyed preparing it for you. If you'd ever like to reach out, I'd love to hear from you on Twitter or in the discussion section of this course.

Course author

Nate Taylor

Nate's first program was written in QBasic on an 8086 clone his dad built. Since then he's written applications in C++, .NET, and Node.js. He spent the first 12 years of his career writing Windows...

Course info

Level	Beginner
Rating	★★★★★ (103)
My rating	★★★★★
Duration	1h 12m
Released	21 Nov 2018

Share course