Node js dependencies

- When using the exercise files, you may find it necessary to install the dependencies before any of the files will work. In chapter eight, lesson four, I'm looking at the finished folder in my exercise files. Any folder that has a package.json may have some dependencies that need to be installedfor our app to work. So if I just try to run this app, node app, I will get an error.We cannot find module 'express'. So before we can run any of our finished exercises and before we can actually start iterating on previously built lessons in the start folder, you may find it necessary to run an npm install. This will install the dependencies that are required for the application in your finished folder to run. So now when I node app, we can see that our application is in fact running because we've installed all of the dependencies.

How Node js works

 I'm going to demonstrate how Node.js works and why it is so fast by taking a look at two restaurants. The first restaurant is Apache Steaks and Chops.It is a big, nice, fancy restaurant. In this restaurant, every new guest represents a new user, and making an order is like making a request. If I place an order for a salad, the manager will need to hire a new waiter to take care of me. In this restaurant, our waiter represents a thread. We are going to have our own waiter, our own thread, and they will handle all of our orders. This is similar to how Apache works. Every request is single-threaded. After placing the order, the waiter will take the order to the kitchen and give it to the chef. And now the waiter just waits. He won't do anything else until the chef is finishedmaking the food. I would like to order a glass of water, but I can't order anything until the chef finishes making that salad.The chef is blocking me from being ableto simply order a glass of water. In this analogy, the chef represents the file system or a data store. In Apache, the single thread waits for the file system to finish reading files before it can do anything else. We refer to this as blocking. Finally, my salad is ready. My waiter brings me the food. I can order my glass of water, and my waiter also brings me that, too. My request has been served. And now the manager is firing my waiter because they are not needed anymore. Now, when this restaurant gets busy for dinner service,every guest has their own waiter, which is pretty nice. That is pretty good service,but the waiters are mostly hanging around the kitchen and waiting for the chef to make the food. If this restaurant gets really popular, it requires a lot of space to expand because more guests means more waiters. Now, let's take a look at this other cafe, Chez Node. At this cafe, there is only one waiterbecause Node.js is single-threaded.Here, we can order some crepes. We can see that our waiter places the order for the food, then moves on to take an order from another new table. Hmm, this single thread services all of the restaurant guests. That is pretty cool.When my crepes are ready, the chef rings a bell, and our waiter goes and gets the crepes and delivers them to me.He then proceeds to take another order from a new table. When their food is ready, the waiter will bring it to them as soon as he can. We can say that this waiter behaves asynchronously.Everything this waiter needs to do represents a new event, a new table, placing orders, delivering orders. These are all events, and they will get handled in the order that they are raised. Our waiter does not wait. There is no blocking. Our single waiter is busy, busy, busy, but he is killing it because he can multitask. This is what it means when we say nonblocking, event-driven IO. We have a single thread that will respond to events in the order that they are raised.This thread behaves asynchronouslybecause it does not have to wait for resources to finish doing what they're doing before our thread can do anything else. If this cafe gets popular, we can simply franchise it. Chez Node can easily be expanded by simply duplicating or forking the restaurant into a neighboring space. And this is precisely how we host Node.js applications in the cloud. Now, remember, Node.js is single-threaded.All of the users are sharing the same thread. Events are raised and recorded in an event queue and then handled in the order that they were raised. Node.js is asynchronous, which means that it can do more than one thing at a time. This ability to multitask is what makes Node.js so fast and one of the reasons so many developers are building their web applications with Node.js.

Installing Node js on Pc

- If you have not installed Node.js or you installed an older version of Node.js awhile back, you will need to go and install it. The easiest way to install Node.js is to navigate to nodejs.org and choose from one of the installers that you see on the first page. We're going to install version 4.2.1. During the recording of this Node.js class, version 5 just became available. But version 5 is just available and we're going to continue to work with 4.2.1, assuming that it is more stable for our files.Another way that you can select a download, is to go to the Downloads tab. You can actually download a specific installer. I'm just going to go back to the homepage and select the 4.2.1 from the homepage. Once you have the installer downloaded, you will need to run it. The basic options are absolutely okay for most developers. If you wanted to change where Node.js gets installed, you could do so. If you wanted to pick which options you want to have installed, you can do so. But I'm just going to select the defaults and go ahead and install it.Once we are finished, we have successfully installed Node.js. We will just click Finish. We can go ahead and test our installation. Once I get to the Start screen, I'm just going to go aheadand search for the command prompt. So you can do a search for cmd. There it is the command prompt. We're gonna go ahead and open that up. From the command prompt we can see which version of Node we have installed by typing node -v. This shows that we have version 4.2.1 presently installed.

Global Object

- If you're use to using JavaScript in the browser, then you're probably pretty use to the window being the global object.In node.js, the global object is global.Here we're looking at the node.js api.These are the objects that are available to us globally on the global name space.Let's go ahead and take a look at what we can use immediately in a javascript file without having to require anything.I'm going to find the exercise fileschapter three, lesson one, under the start folder. There is an empty javascript file called global.js. I'm going to go ahead and open up this empty javascript file. Available to us globally, we have a console object that will allow us to log messages to the console. So, I can type console.log and then, log a string to the console. The hello world string, will now show up in my terminal. So, I'm going to go ahead and save this first node.js javascript file, and I am going to switch to our terminal. Now, you've run a node.js file with node and then the file name. So, node global.js will run that javascript file using node js, and we will see Hello World printed to the terminal.Let's go ahead and navigate back to our files. So, here we can see we used the console object. The console object is available to us globally because it is a part of the global name space. So, by adding the global name space before this console log, this will actually work the same. So, if I save that and go back over into our terminal and run this file again, we will see that it says Hello World. So, this works pretty similar to the browser. Because the global name space is assumed, we do not have to always include it. Now, I see that this works pretty similar to the browser, but it doesn't work exactly like the browser.Let's see what happens when I create a variable. So, here I've created a variable for Hello, and that variable will be set to a string. If you're use to using javascript in the window, you know that these variables are added to the global object.That means, we should be able to see our Hello variable by typing global.hello.I will go ahead and save this, and we can go back out to the terminal to run it to see what happens. So, from the terminal, this time, I'm going to run node global.js, and this time when I run it, I see an undefined. That is because node js works a little bit different than the browser when it comes to storing variables. Let's go back to our code.Every node js file that we create is it's own module. Any variable that we create in a node js file, is scoped only to that module. That means that our variables are not added to the global object the way that they are in the browser. We can see this variable simply by removing that global name space. I will save this, and go back into the terminal and run this file again. There we can see our Hello World from node js variable. If we go back to our code, you also notice that you have available to you, anything that you're use to typically doing in javascript. Anything with primitives, objects, arrays. If I wanted to get a substring from this string, I could do so with the string slice method. Var justNode will be equal to hello. slice. We will slice out the first 17 characters of this string. That should give us, just node. Since node js four and above, now support some ES six, I can use this justNode variable in a template string.Instead of logging hello down here on line five, I'm going to log the justNode variable in a template string. So, to do that, I'm going to use to two backtick characters. These are not single quotes.These are the backtick character. This is usually found under the escape key in the upper left hand corner of your keyboard. I can write a string in between these backticks. If I want to include the value of a variable, all I need to use is a dollar sign and a couple of curly brackets. In between the curly brackets, we can place the variable that we want to use. So, justNode. We will go ahead and add this substring here using a template string. Let's go ahead and save this, and navigate back to the terminal and run our file node global.js. We can see Rock on World from justNode js that was cut out of our initial string. If we go back to our code, there are many otherthings that are available to us globally in our code. One of the things that's nice to use is a reference to the current directory that we're in and a reference to the current node module that we're using. If I were to log \_\_dirname. We would get the full path to the current directory where this module is located. If I were to log\_\_file name, we can get the full path to the current file as well as this file's name. So, I'm going to ahead and save this, and navigate back to the terminal, and let's run our file again,node global.js, and we can see our first console log Rock On World from Node js. The second console log, is the full path to the directory that this file is located in. The third console log, is the full path to the directory, that this file is located in with the file name. Now, if I wanted to just pluck the global.js file name from that second string, I can use some tools that are available to us, with our install. Let's go back to the code.Another thing that's available to us globally, is the require function. The require function, is what we're going to use to import other node js modules. So, here at line one, I'll go ahead and drop my cursor and hit enter a few times.Now, I'm going to create a variable for the path. I will load this path instance using require. So, I'm going to require the path module. The path module, is a module that is available to you with your installation of node js. It gives us some tools for working with paths. Now, let's delete these last console logs down here. I'm also going to delete the Hello, and justNode. This time, what we're going to do is, in this template string Rock on World from justNode, we're going to use the path module .base name function to pluck the base file name from a full path. I'll put in \_\_filename, and that should pluck justthe global.js file name from that full path. Let's go ahead and save this.Navigate back to the terminal, and runour file one more time. Now, this time, when I run the file, I'm going to leave off that js extension. When we run a node js file, we can include that js extension, or not. It's assumed that the file is going to be js, so when we node global, node js will find the file that's marked global.js and run that. So, here we can see Rock On World from global.js. So, the node js global object contains those objects and functions that are available to us globally. Meaning, that we can start adding these objects to our node js code immediately. Now, in the next couple lessons, we're going to continueto work with this global object, and we will discover the process object, as well, as the timing functions. Then, we'll take a deep dive into the common js module pattern. Which includes module, exports, and require.

Argument variables with Process.argv

- One important object that is available to us globally is the process object. It can be accessed from anywhere, and it contains functionality that allows us to interact with information about the current process instance. We can use the process object to get environment information, read environment variables,communicate with the terminal, or parent processes, through standard input and standard output. We can even exit the current process. This object essentially gives us a way to work with the current process instance. One of the the things that we can do with the process object is to collect all the information from the terminal, or command prompt, when the application starts. All of this information will be saved in a variable called process.argvwhich stands for the argument variablesused to start the process. So let's go ahead and navigate to our finder. In the start folder there is an empty JavaScript file called app.js, I'm going to go ahead and open up that file. And the first thing I'm going to do is just log the process.argv variable. Now process is available to us in any module globally.So we can use the process object wherever we like. Let me go ahead and save this. And then navigate out to the terminal. And we can run our file simply by typing "node app.js". And when we do so, we can see that the process.argv variable is an array. It contains a path to node, and a path to app.js. We started the app by running node. User/local/bin is the directory that that executable is found in. We also started this app.js fileand we can see that we have the full path to that here. So, what happens if we start the process again and send some more information? Like I might send a user flag and a greeting. So now, when I start this process, I am sending some more information. Now we can see that our process.argv array has grown. It also contains our user flag and a value for the user, and a greeting flag,and a value for the greeting. Now, because this is an array, we can work with it the same way we would work with JavaScript arrays. Let's go back to our code. And let's create a function that will help us grab the values from the process.argv array from their flag name.So I'm going to create a function called "grab". And this function is going to take in the flag as an argument, either the greeting or the user. And we want it to return the value of that flag. Well since process.argv is just an array, I can create a variable called "index", and I can set it to process.argv, that's an array, so I can use indexOf and then we can go ahead and find the index of the flag in the array. Now, what we want to do is returnthe value, or the next variable in the array. So we're going to have to use the index plus one in order to get that value.So now we want to make sure that we are able to find that flag within the array.I'm going to add a return statement here. And the first thing that we want to do, is we want to test this index to make sure it doesn't equal negative one. So if this index is equal to negative one, that means that we did not find the flag within the array. So we will just return null from this function. Now, if this index is not equal to negative one, it means that we did find the flag in the array. So we're going to use this index to return the next item from the array.Process.argv is an array so we can return index+1, and that should be the value found after the flag. Let's go ahead and use our new grab function to set some variables from those flags that we used to start the application. I will set a variable for greeting. And we will grab the greeting from the process.argv array,simply by using that grab function and sending the greeting flag. Let's create a variable for our user. We will grab the user simply by sending the user flag to our grab function, and that should give us the value for the user. Now, let's go ahead and say if we do not have a user,or we do not have a greeting, let's log a message to the console. We will go ahead and log "You Blew it!". Now, if we do have the user and we do have a greeting, we can use that information and log it back to the console. I'm going to use a template string for this. And, remember, to create our template stringwe want to use this backtick characters.So I will add two backticks. And what I'm going to go ahead and do is simply say "Welcome", and we will use the value of the user variable by placing it in between the curly brackets found after the dollar sign for the template string.And we'll go ahead and also output their greeting. So if you send us a user and a greeting, we will display that information back to you in the terminal. Let's go ahead and save this. And go back out to the terminal and run our new app. So I'm going to run this app first by typing node app. And remember I can use the js, or not. And it says "You Blew it!". And that's because we didn't send the greeting and a user. So let me try this again. Node app; this time I'm going to send a user, Alex, and a greeting, "Hello Hello Hello". And this time when we run it, it says "Welcome Alex, Hello Hello Hello". So the process.argv array gives us a way to get information into the app when we start it. This is a great tool if we want to specify certain ports, or specify certain file folders for our app to use at the very beginning.

Standard input and Standard output

- Another feature of the process objectis standard input and standard output.These two objects offer us a way to communicate with a process while it is running. For now, we will use these objects to read and write data to the Terminal. Later on in the course, we're going to use the standard input and standard output objects to communicate with a child process. Let's go ahead and navigate to our Exercise Files. Under the Start folder, there's an empty JavaScript file marked ask.js, let's go ahead and open up that file. Now, we've already really been using the standard output object. The console log uses the standard output object to actually log messages to the console. It logs the message and also controls the line spacing of a message to give you a new line. Let's go ahead and write a message to the console just using standard output. process.stdout is our standard output object and we can use the write method to write some strings to the Terminal. So I'll go ahead and write Hello, and then I'll go ahead and open this up again, and this time I'm going to go ahead and write World, but I'm going to go ahead and throw some line breaks in there, so we'll throw some \ns, which will give us a new line. Process standard output will write strings to the terminal,however it does not give you new lines automatically. Let's go ahead and save this and then let's navigate to our Terminal and run this file. So we can run this file simply by typing node ask. You will also notice that we have two process.stdout.write calls that are making this one Hello World string, so we have to actually control the line spacing when using standard output.Let's go back to our code. What I'm going to do now is I'm just going to create an array of questions to ask.questions = and we will set up a new array, and I'm just going to go aheadand ask three questions in this array.What is your name will be the first question, What is your fav hobby? And then we will also ask What is your preferred programming language?There we go. So we're going to ask these three questions from an array and we're also going to store the answers in an array. So I'll create another array down here for our answers. And that'll be an empty array. Alright, now what I'm going to do is create a function that will allow us to ask a question, function ask,and what we're going to do is take in the index of the question that you'd like to ask. If you would like to ask the first question of this array you'll pass the ask function zero, if you would like to ask the last question in this array you would pass the ask function two. So what we're going to do now is we are actually going to copy these stdout.write calls here and place them inside of the function. So I'm going to go ahead and cut them from there and paste them inside of our function, give it a little bit of spacing,and instead of writing Hello World what we actually want to write is the question.So we want this to ask one of the questions from the array. I'm going to use another template string, so we're using those ``, and in this template string I'm going to go ahead and throw a couple of new lines there, and then I will go ahead and use the ${ questions, that's the array, and I'm going to use the i that's passed to this function to index the question that we should ask. Now after we ask the question I'm also going to add a little prompt here, so instead of writing World we'll just write a couple of spaces, a prompt, and then a couple more spaces. Great. So now we have a function for asking a question, let's use it. Let me try to ask question 0. There we go. So I can go ahead and save this, and then navigate out to my Terminal, and from the Terminal I can go ahead and clear the screen by typing clear. If you're on a PC you can do this in the command prompt with cls. And now I'm going to run our app, so I'll node ask. Now notice, using process standard output has screwed up the way our Terminal is working. So it does ask the question,and then our process exits, and now we can see that we are in the start folderand we see another prompt right here after the question. So as you can see our standard output object is just running everything together, it doesn't even put out new lines when our program ends.Let's go ahead and add a listener to our program to listen for the answers to these questions. So I'm going to go ahead and go back to Sublime, and then from Sublime what I want to do is right under this function here I'm going to wire up an event listener for data on the standard input object. So process.stdin, standard input, on data. So when a data event is raised, this means that when the user types some data into the Terminaland hits enter we will raise this data event here. So I'm going to go ahead and add a callback function to handle this. So when the user enters some data,or when any data is sent to our application through the standard input object this callback function will fire and that data that is sent to our applicationwill come in as an argument. So let's just go ahead and echo our data. What I'm going to do is say process.stdout.write,and we will go ahead and first write a new line, and then we will add that to data.toString. So we'll take our data, we will convert it to a string, and we'll also trim it. And the reason that we want to trim this data string is it's going to come with some leading and trailing \ns and slash \rs, so we want to get rid of those.And I will also just add my own \n there at the end, there we go. So when there's any data sent to this application through the Terminal this callback function will fire and we will echo the data back by writing it to the Terminal and displaying it to our user. Let's go ahead and save this and navigate out to the Terminaland see what's happening. Now I can run my app ask. And as we can see we're being asked for our name. Notice that the application didn't stop. The reason that this process isn't over is because it's still running. This is the first time we're using node.js asynchronously. We are waiting for some input and when we add that input it will be handled with that asynchronous callback. So what is my name? Alex. Oh, we see that echoed.Hello world. Every time I hit enter I'm sending some data to the application,raising a data event on process standard input, and then we are writing that data back to you using standard output. So when you have a running application it is going to continue to run until it encounters an error or you stop it. The way that we stop these applications is by hitting control C. So this is a running application here in the Terminal, hitting control C will kill this application and exit us back to the Terminal. I'll go ahead and clear this. Let's go back into our code and what we want to do from our code this time is actually save your answers and ask the next question, as apposed to just echoing your answer back to you. So from the process.stdin.on data listener that we have here I'm going to delete this line that echos the data that you've entered back to you, and instead of echoing the data, what I want to do is save it to the answers array. answers.push will push an item into the array. I'm going to push data.toString .trim back into the array.So that's your answer, you've filled out the answer and hit enter, we're going to collect that and push it into the answers array. Now I want to check to see if there are any more questions. If answers.length is questions.length then you still have some questions left to ask.So what we're going to go ahead and dois just ask the next question. We will ask answers.length. And what this does is if you've answered one question the answers.length will be one. So we will now ask you the next question in the questions array, What is your favorite hobby? If you've answered two questions the answers.length will be two, which we can use to index the question in the questions array, What is your preferred programming language?So as so long as there are more questions to ask we will ask them. Now if there aren't any more questions to ask,meaning that the answers.length is greater than questions.length we will force this process to exit. process.exit will cause the process to exit from within. So when we hit line 21 we should be kicked back out to the Terminal prompt. I'm going to go ahead and save this and navigate to the Terminal and let's run our application again. node ask will ask What is your name? Go ahead and type in Alex. What is your favorite hobby? Skiing. What is your preferred programming language? JavaScript. And you can see that after I answer that third question we are kicked backed out to the Terminal prompt. That's because the process.exit was invoked. We had answered all the questions, there weren't anymore questions to ask, so we exit the process. What we want to do instead of exiting the process is display our answers back to our user. So we can add another listener here just before the first question is asked on line 28 and in this listener what we're going to do is listen for an exit event on the process object.process.on exit will listen for an exit event on the process object. And when the process.exit is invoked this callback function will fire. So when we exit the process we can do a couple of thingsjust before the process exits. What I want to do is display the answers back to the user. So the first thing I'm going to do is give it a little bit of padding.process.stdout.write and I'm just going to write some padding in the form of new lines, so I'll go ahead and add a couple new lines there. There's some padding at the top. I'm going to go ahead and copy this and paste it just to have padding at the top and the bottom. So between this process.stdou.write new line sandwichwe want to actually write your answers to you. So now I'm going to process.stdout.write and this time because I want to display the answersI'm going to use another template string, so we're going to use those `` characters and I'm going to say Go and I can find your hobby through answers 1,because it's the second question that you ask, and I can find your name through answers 0, because it is the first question I ask, and then we'll go ahead and find your preferred languagethrough answers 2, or the last question that you asked, so just before the process exits we are going to echo back to you your question answers. Let's go ahead and save this and go back to the Terminal and again, we can clear the previous screen by typing clear, or cls if you are on a PC. And now I can node askand here's the first question, my name is Alex, the second question, skiing is my favorite hobby, and JavaScript is my favorite programming language. And we can see that we echo back to our userGo Skiing Alex you can finish writing JavaScript later. And then we are exited out of the process. So process.stdin and process.stdout are ways that we can communicate with a running process.

Global timing functions

- In the last lesson we started working with Node.js asynchronously by using event listeners. Another way we can work with Node.js asynchronously is through using the timing functions. The timing functions setTimeout,clearTimeout, setInterval, and clearInterval work the same way they do in the browser and are available to you globally. Let's go ahead and use a setTimeout function. Let's navigate to our Exercise Files. In the start folder you will find an empty JavaScript file called timers.js. Let's go ahead and open that up in Sublime. And in this file I'm going to create a variable for a waitTime, and we will make the waitTime three seconds long. And then I'm going to go ahead and console.log a Wait for it message.And what we're also going to do is use our timing function setTimeout. So a setTimeout will create a delay of a certain time and then invoke a callback function. So what we're going to do is invoke this function after our delay. And our delay is our waitTime, or three seconds. So times are handled in milliseconds, which is why 3000 milliseconds will equal three seconds.What we're going to do is cause a timeout that will cause this application to wait for three seconds and then it will invoke this callback function where we will simply console.log a done. So during these three seconds while we're waitingour application is going to be running.Let's go ahead and go to the Terminal and test this out. So I will node timersour timers.js file. And what we see is Wait for it. And after about three seconds we see done and we are back out to our Terminal prompt. So after three seconds the application exits,which means that during that time the application is running. Let's go back to our code and take a look at setInterval.So what I'm going to do is also add an interval to this. We will add a couple more variables right here under our waitTime. I'm going to add a variable for the currentTime, which when we start this application will be 0 milliseconds.And then I'm also going to add a variable for the waitInterval. The waitInterval is going to be half a second.And what this means is I want to fire this interval callback function like a heartbeat, every half second. So setTimeout waits for a delay once and then invokes the callback function,where setInterval will fire the callback function over and over again on an interval time. So let me add a setInterval to my code. So what we're going to do every half second is one, we need to update our current time, so we want to add the waitInterval to the current Time.So this will add 500 milliseconds to the currentTime every 500 milliseconds. That means our currentTime will give us an accurate indicator of how long we've waiting. Now let's go ahead and log a message to the console using a string template. So that means we want to use those ``. And I will log waiting. And I can use the $ and the {} in my string template to display a variable,currentTime. And I'm going to divide this by 1000, so we get this currentTime in seconds, so our currentTime is stored in milliseconds, dividing it by 1000 should give us the currentTime is seconds. And I'd better make sure I put that on the inside of this ` here, there we go. So now we are going to wait three seconds and log done, but then every half second during the waiting we're going to log how long we've been waiting. Let's go ahead and save thisand navigate to the Terminal. And I'm going to run this file through node timers. And as you can see, Wait for it,and every half second we see the message. Notice that after the done logwe are still going. That interval is like a heartbeat, it will not stop. Every half second it's going to let us know how long we've been waiting. And as long as that interval is running our application is running. So we could literally sit hereand watch this timer time forever. In order to stop this application thoughwe're going to hit control C. And what we need to do is after three seconds we need to stop our interval from running.So let's go back to our code. The clearTimeout and clearInterval functionsare used to stop intervals or stop any timeouts that we might have currently in our application. So after three seconds, just before we're done, we need to clear this interval. So the way that we can clear an interval is we first need to set a variable instance to that interval. So on line 7 if I say var interval = setInterval now I have this interval instance saved to the variable. I can use that interval variable in a clearInterval. So clearInterval and then send it, the interval that we would like to clear will stop this interval from running. So it will stop that heartbeat from displaying that message on the page. And when there is no longer an interval running we will also exit the application, because we no longer have any listeners and we also no longer have any intervals. Let's go ahead and save this and see what happens. So we can go back out to the Terminal and I will clear these old messages by typing clear, cls on a PC. And now let's go ahead and node timers and now we can see we are waiting for it, but after three seconds when we are done we are actually going to display the done and the application stops. Let's take one more iteration on this code to display the time waiting in a percentage and also control the standard output, so that we overwrite the last line, meaning that we can see a percentage number grow.Let's go ahead and go back to our code.And the first thing that we're going to do is we are going to add a function for writing the percentage. So here on line 5 I'm going to delete this console.log that says Wait for it, and I'm going to add a function. And this function I'm going to use for writeWaitingPercent. And we will send it, that waiting percentage, as an argument. and then what we're going to do this time is we're going to use the process standard output object to clear the last line that we wrote. The clearLine function on standard output will clear the last line inside of the Terminal. And then I'm going to move the cursor backto the start of that line using the standard output object. cursorTo object will move our cursor back to the beginning of that line. Now I'm going to use the process.stdout object to write the percentage that we've been waiting.And I can, again, use a string template for this one, since we're going to use a variable. And I will write waiting... And now we can actually use that percentage variable right here. So if you send this writeWaitingPercent function a percentage it will write it over the last line in the Terminal. So now I'm going to add a variable to hold the percentage that we've been waiting. And I will call this variable percentWaited. And we will start that out as 0 as well. So the very first thing I want to do after we set up these intervals and after we set up the timeout is I want to write that to the console, so I'm going to use my function writeWaitingPercent and I'm going to send it the percentWaited, which should start out as zero. So when we start this application we should see that we've been waiting 0%. Right before we write that waiting percentage let's go ahead and add a couple new lines,process.stdout.write and we can go ahead and add a couple new lines there.There we go. Now in the setTimeoutonce we are totally finished with this application, after we clear the interval,let's go ahead and write the WaitingPercent as 100. So when we're done we will see that we have waited for 100% of the time. Now what we want to do is calculate the percent waited inside of the interval. So inside of this interval on line 13, right after we append the currentTime, what we're going to do is calculate that percentWaited. So I am going to say percentWaited is going to be = to Math.floor. This is JavaScript, so we have available to us the Math objecton the top level, and what we're going to do is take the currentTime and divide it by the waitTime, so the waitTime is our total time, the currentTime is the amount of time that we've waited, we can divide the currentTime by the waitTime and multiple that by 100, and we will get a percentage. So now instead of writing this message to the console,what we're going to do is let ourwriteWaitingPercent function do that for us. So I will call writeWaitingPercent and we will add the percentWaited. Alright.So now we've added the ability to show our users how long they're waiting as a percentage. Let's go ahead and save thisand go back out to the Terminal. And again, we can clear our previous Terminal screen with clear, or cls on a PC. And now I'm going to run our timers file and we can see that we are waitingand then eventually when we get to 100we are done. So there are just a couple more things that we want to add to this just to clean it up. It appears to be working, but our users do not know those are percentages, unless we throw a little % indicator in there. And the last thing is is before we log done we also probably want to log a couple of new lines and maybe just a couple of new lines after it to make it look good. There we go. So let's go ahead and save thisand try to run it one last time. Go back out to our Terminal, and we will node timers, and now we can see that our percentWaiting is being up. If we want to see that number go faster, if we want to see that waiting percentage be calculated faster all we need to do is up our waitInterval. So I'm going to go back to the application, I will change our waitInterval to 10 milliseconds instead of 500, meaning our waitInterval is now going to fire 10 times a second instead of twice a second. So let's go ahead and save this and go back out to our Terminal. And I will clear this. And let's run it one last time, node timers, and we can see that percentage is being calculated 10 times a second. So those are the timing functions. They're available to you globally in Node.js and they work the exact same way that they would work in the browser.

Core Modules

- In the last chapter, we took a look at the objects and functions that are globally available to us. One of those functions that is globally available to you is the required function. In the last chapter, we used the required functionto load the path module. Let's do it again. Let's go ahead and navigate to our files. In the start folder, there is an empty JavaScript file called "core JS".Let's go ahead and open that up with Sublime. And, in this file we will import a module into a local variable by requiring it. So I can create a variable for path and require the path module. The require function is what we use to load modules.Many of the modules that we use are hosted the NPM registry and need to be installed first. We'll take a look at those modules later on in the course. For this chapter, and the next few chapters,we're going to focus on those modulesthat you do not have to install with NPM. These modules were installed locally with your installation of node JS.We refer to these modules as core modules. And path is one of the core modules available to you with node JS.In the last lesson, we used the path module to pluck the base file name from a full path. We could do that again. I could log the path.base name. And that is a method that we can send a full file path with name to. And we have that available to us on underscore, underscore, file name. And it will pluck out the base file from that full path.Which, in this case, will be core JS. We can also use the path module to create path strings. The path dot join function can be used to join strings together in a path. Ver dirUploads will be a new path that I make to the Uploads directory using the path.join function So, I will join this to the current directory. And we will assume that it is in a www directory. And then under that, there is a files directory.And then under that, we will assume there is an uploads directory. So, what I have done is I've just created a long path string using the path join function,where I would send it each directory in the path and it will create the string for me. Let's go ahead and log that. So, I will save this and go back to the terminaland run it. And, we can run this just by typing node core.js. And you can see that the first path.basename plucks the core.js module from the full file path.You can also see that the path.join was used to create a very long path string,that includes the current directory, plus www/files/uploads. So, the path module has some great tools for working with paths. There are other modules that are available to you as a part of the node core. Let's go back to our code. We can use a utilities module. The utilities module is called util. The utilities module has several helper functions that we can use. One of which is a log. So instead of doing console logs, I can do util logs.And, the difference is we're still going to log this information to the console. The utilitie module log function also adds a date and time stamp. So, I can go ahead and save this, and go back over to the terminal. And we can run our file again.This time I will run it without the JS extension. There should be no difference between doing that. And here we see that Core JS was logged on the 26th of October at this exact time, and our full path to www, files, uploads was also logged with a time stamp. There are plenty of these modules that we can use.One module you can use to get the current details about memory usage.That's the V8 module. So, I'm going to go back to my code. Since, no JS is built on top of Google Chrome's v8 processor, we can use the v8 module to get information about memory. So, on line three I will create a variable that we will call v8. And I will require that core v8 module. And now, right after the directory uploads, I will also use the util log function to log v8.getHeapStatistics.And that will give us current information on our current memory statistics. So, let's go ahead and go back out to the terminal and run this file node core. And we will see all of our previous logs plus this last log is giving us a JavaScript literal of all of our memory usage statistics. So, there are a handful of modules that we can require out of the box with node js. For the next couple chapters, we're going to focus on these modules in particular.

Collecting Objects with Readline

- Readline is a module that allows us to ask questions of our Terminal user. It is a wraparound the standard input and standard output objects that allow us to easily control prompting a user with questions and saving those answers.Let's go ahead and go into our Exercise Files. In the start folder, you will find a promptPerson.js file. This is an empty JavaScript file that I'm just going to go ahead and open up in Sublime. We can use Readline. It is a module that is available to us with our installation of Node.js. So I'll create a variable and call it readline and then I will require readline. And once I have readline, I can create an instance of the readline objectwhich will create prompt for me by sending it the standard input and standard output objects. So var rl will be my readline instance and I will use readline to create an interface. With this interface, we're going to send it the process.stdin and process.stdout.Readline is going to control these objects for us asking questions and collecting information so that we don't have to control the process.stdin and stdout directly. In order to ask a question of Readline, all you need to do is invoke rl.question. And the first argument will be the question that will show up in the Terminal. We will ask, "What is the name of a real person?" The second argument is the function that will invoke once we have an answer from the Terminal. And in this function, the answer will be sent as an argument. We can go ahead and just log that answer to the console. Let's go ahead and save our file this far and then navigate out of the Terminal to run it. So I will go ahead and node promptPerson. We can see that our question is being asked right here in the Terminal. Our application is still running and we are being prompt for an answer. So as soon as I enter, we can see George Washington echoed back to us.Notice that this application is still running. That is because we have not told Readline to close. And because Readline is still listening, our application is still running. So in order to stop this now, I'm going to have to hit Control C in the Terminal. I will also go ahead and type it clear just to clear out my Terminal for me. Let me go back to my code and add another iteration to this project.What I want to do is ask the user for the name of a real person and then I want to ask the user for a collection of things of that real person might say. So up here where I create my Readline instance,right below that on line three, I'm going to go ahead and create a variable for a real person. This realPerson is going to be a JavaScript object. And I will go ahead and give the realPerson a name of empty to start with and a sayings of an empty array to start with. So this is where we're going to save our name of the person and the things that they might say. Now we can ask a question with rl.question but we can also use the Readline to set up a prompt so that we can ask a question over and over again.rl.setPrompt is the function that we can use to set the Readline prompt. I'm going to use a template string which means I'm going to use these backticksto ask, "What would", and then use the name of the real person,{realPerson.name} say. So we can use the answer of the real person's name to find out what they will say. But before we can use that answer, we're going to need to set it. So here on line 12, inside of my function that will callback once we have an answer to the name of the real person. I'm going to use that answer to set the real person's name. And now we can use that real person's name to ask the next question. We'll make sure I add a question mark here. And instead of logging this answer, what we're going to do here is delete this console.log and then we are going to call rl.prompt. So on line 14, we set the prompt. And on line 16, we are going to display the prompt asking our user a question. Now in order to listen for any of these new answers, we're going to listen for any new lines. rl.on('line') is an event that will fire when the user submits an answer.This callback function, the second argument of the rl.on('line', function()will also be invoked once we have that answer. And the answer will be passed to back to this function. Now at this point we're asking what would the real person say, so I'm going to call the answer a saying. And now we're just going to go ahead and echo back the saying. console.log(saying.trim()). So let's go ahead and save this and then navigate out to our Terminal and run this application, node promptPerson. And this is going to ask us for the name of a real person. This time I'm going to go ahead and use BenFranklin. And, "What would BenFranklin say?" Well, I think you'd say, "Well done is better "than well said." And when I click Enter, we can see that that line that we entered, the second question, has been echoed back to us. Now, notice our application is still running at this point because we have not told Readline to close. So I'm going to go ahead and hit Control C to stop our application and then we'll go ahead and type in clear just to clear out the Terminal. And then I'm going to go back to the code. I want to add another prompt so that we can continue asking the same question over and over againand collecting the sayings of a real person. So when we get the answer from what a real person would say in this callback function found here on line 18,what I'm going to go ahead and do is change the prompt one more time,rl.setPrompt(). And this time I'm going to change the prompt to read, "What else would ${realPerson.name} say?" And we're going to continue to ask this question over and over again until the user types exit. So we will also let them know that they need to type exit to leave. There we go. So that's the new prompt. And what we want to do now is display that prompt. So when we get answer, we're going to ask you another question, "What else would this person say?" And then we will display that prompt with rl.prompt(). So let's go ahead and save this file and navigate out to the Terminal. And let's run it again, node promptPerson. "What is the name of a real person?" Ben Franklin. "What would Ben Franklin say?" For now, we'll just say Red. What else will he say?Green, what else will he say? Blue, and it says type exit to leave but we actually haven't handled that. So if I type exit, I'm not leaving. In order to exit this application, I still have to hit Control C.So I will exit and then I will clear it. Let's go back into our code and add the exit command which will cause the user to leave. So here inside of this callback on line 18 where we are prompting our users for more questions, what we're going to do is surround this in an if statement. if saying, or their answer, .toLowerCase, just to make it case insensitive , .trim() === 'exit'. So if one of the answers is exit, what we're going to go ahead and do is call rl.close. This will close the Readline instance.Otherwise, we're going to go ahead and ask the person another question. So in the else block, I'm going to take these two statements here, the one that sets the prompt and prompts the user and I'm going to go ahead and add those into our else block. So now so long as our user doesn't type exit, we will continue to ask questions. When our user types exit, we're going to close this Readline instance. And the last thing that I'm going to do is just listen for our close on that Readline instance. So I'm going to come way down here to the bottom of this file and I'm going to wire up a listener for a close event on the Readline instance. rl.on('close') will listen for a close event. So on line 21, when we invoke the Readline close, we can fire this callback function just before closing the Readline. So when we invoke a Readline close, what we're going to go ahead and do is just log the details of the real person. We can log the real person's name and the real person's sayings. So we have collected their name and their sayings. We're going to log that back to the user. Placing a %s in the console log is a placeholder for a string.So what it will do is it will replace the second argument that we have added with that string. So here I will add realPerson.name. The next thing that we're going to go ahead and do is write a sentence, is a real person that says.And now I'm going to add a %j here.This %j will replace this variable with a JSON String. So we're going to take that real person's array of sayings and replace that %j with that. So realPerson.sayings represents that array of sayings that we have collected. The last thing that we need to do is I'm realizing we are not actually collecting these sayings so let's make sure we collect them. We're asking the question here on line 23. We're prompting the user whenever we have a new saying.We actually need to save that. So I'm going to call realPerson.sayings.push the saying, (saying.trim(). There we go. Just making sure that we are collecting all of those sayings that the user has answered right there. So just before we run this application, there's one more step that I'd like to take here on line 37 after we display the information, the real person's name and the things they said,I want to exit the process. So I'm going to invoke a process.exit which will cause our process to end. So let's go ahead and click Save on that and then let's navigate to our Terminal and let's run our application, node promptPerson."What is the name of a real person?"Ben Franklin. "What would Ben Franklin say?" "You may delay, but time will not.""What else would Ben Franklin say?""Well done is better than well said." And it's going to continue to ask us, "What else would Ben Franklin say?" "An investment in knowledge pays the best interest." And it will continue to ask mefor these Ben Franklin sayings until I type exit. As soon as I type exit, we see Ben Franklin is a real person that says and we see that array of sayings as a JSON String. As you can see, we can keep adding sayings until we are finished. Exit is what causes Readline to exit and then our process to exit. So Readline is a pretty cool tool that we can use to prompt the user without having to use standard input and standard output directly.

Handling Events with EventsEmmiter

- Another powerful feature that ships with Node.js is the Event Emitter. The Event Emitter is Node.js's implementation of the pub/sub design pattern, and it allows us to create listeners for an emit custom Events. In fact, every time we've used that on function to listen for a new Event, we've already been using an implementationof the EventEmitter. Let's go to our files.You will find an empty JavaScript filetitled BenFranklin.js. Go ahead and open up that file in Sublime. The EventEmitter is a part of the Events module. So, I'm going to go ahead and create a variable for events, and I will require the events module. Now, the EventEmitter itself is a constructor. So, I'm going to create a new instance of a variable called emitter.So, we'll create a new instance of the emitter by using the new keyword with the event module, .Event Emitter. There is the function that we want to use. So, the EventEmitter is constructor, and we're going to create a new instance of the EventEmitter. Now, this object that we created here on line 3 has on an emit. So, every time we use on, we can wire up a custom event. You can name an event whatever you like. In this case, I've just called this customEvent. The second argument that the on function takes is a callback function that will be invoked when the custom event is raised. In this case, our custom event is going to pass a message and a status to this function as arguments. So, when our custom event occurs, this callback function will be invoked asynchronously.Let's just go ahead and log the status in the message using template string. So, I'm going to use those ` characters. And we'll go ahead and grab the status. And now, I'm going to go ahead and add the value of the message to this template string. There we go. So, when a custom event is raised, we'll pass a message and a status to this callback function asynchronously, and we're just going to log that message in status. The next part of the EventEmitter is the ability to trigger or emit custom events. We can trigger or emit a custom event with the emit function. So, emitter.emit will fire our custom event. The first argument is the event name that we want to fire. And then, the next arguments are going to be the arguments that will passed to the callback function. So, the first argument in the callback function, the message, is actually the second argument of this emit function. So, for the message, I will send Hello World. And the third argument is going to be the second argument in the callback. So, I will send a status of 200. In this code, we've created a new instance of the EventEmitter object, and we wired up a listener to listen for custom events. Now, we are going to emit a custom event.Let's go ahead and save this and go over to the Terminal, and let's run our file,node BenFranklin. We can see here that this worked successfully, giving us a status of 200 and a message of Hello World returned in the Terminal. The EventEmitter is rarely used as a standalone object. We can really get mileage out of it by allowing our objects to inherit the EventEmitter. Let's change our code so that we have a person object that inherits the EventEmitter.Let's go back to our code in Sublime.The first thing that I want to do is instead of including just the events up here on line one, we can actually pull the EventEmitter out of events directly in this require statement here. So, I will use EventEmitter as a variable. And then, I can just chain on to the end of this require statement EventEmitter. And that will pull that constructor function out of the events module and set this variable to our new constructor function. The next thing that we're going to do is we're not going to just create an instance of the emitter. So, I'm actually going to delete all of this code. What I would like to do is create an object, and that object is going to be a Person. Now, in JavaScript, objects are functions. So, we will create a constructor function here for this object. In this constructor function, we're going to take in the Person's name. In this object, this Person's name will be set to the name value that is passed to this constructor function. So, we're going to create an instance of a Person. This instance of a Person is going to be an object that has a name property. Alright, so, what we want is our Person object to inherit the EventEmitter. We can actually use the utilities module to help us do this. Up here, on line two, I'm going to create a var called util, and I'm going to require our utilities module. The utilities module has an inherits function, and it's a way that we can add a object to the prototype of an existing object. That's how JavaScript handles inheritance. So, what I'm going to do is add util.inherithere on line eight. We want our Person object to inherit our EventEmitter. So, this one line here has just added the EventEmiiter to the Person's prototype,which means that the Person object inherits all of the functionality from the EventEmitter. If I create a new instance of a Person, it will have an on and emit function. So, let's create an instance of BenFranklin or ben with this new Person object. So, here on line 10, I'm going to add a variable for ben. That will be the new Person that we're going to create an instance of. I'm going to invoke my Person function with a new keywordbecause it is a constructor. When we invoke this new Person function, we're going to send it a name. So, there we have created a new instance of a Person.And the name of this new instance is Ben Franklin. So, because this Person inherits the EventEmitter, ben can listen for custom events. We could listen for, let's say, when ben speaks. If a speak event is raised, this callback function will be used to handle it. And whatever ben just said will be passed to this function.I'll go ahead and add that there. So, this callback function will take in said as the argument when the speak event raises.We'll go ahead and just log what ben is saying. I'm going to use a template string to do that. Let's use those ` characters. What we're going to do is use a $ and a couple of {s to display this or this current Person, ben's name. So, we can use the this keyword becausethis callback function knows that this is the current instance of your object. This is equal to ben. And then, we'll go ahead and also log what they said. Alright, so, ben also has an emit function, which means that this instance of ben can emit a speak event. You may delay, but time will not. There we go. So, whenever ben emits a speak event, any listeners that listen for the speak event, like the one we've wired up on line 12, will fire their custom callbacks. In this case, we are passing You may delay, but time will notto that callback function as the said variable. When this happens, we're going to log the name of the person,and we're also going to log what they just said with the recent speak event. So, let's go ahead and save this. Let's go back to our Terminal, and let's run our application again. We will node BenFranklin. We can see that Ben says You may delay, but time will not. The EventEmitter provides us a way to create custom objects that raise custom eventsthat can be handled asynchronously.And because the events are handled asynchronously, it is a very important tool in node.js.

Exporting custom Modules

- Another powerful feature that ships with Node.js is the Event Emitter. The Event Emitter is Node.js's implementation of the pub/sub design pattern, and it allows us to create listeners for an emit custom Events. In fact, every time we've used that on function to listen for a new Event, we've already been using an implementationof the EventEmitter. Let's go to our files.You will find an empty JavaScript filetitled BenFranklin.js. Go ahead and open up that file in Sublime. The EventEmitter is a part of the Events module. So, I'm going to go ahead and create a variable for events, and I will require the events module. Now, the EventEmitter itself is a constructor. So, I'm going to create a new instance of a variable called emitter.So, we'll create a new instance of the emitter by using the new keyword with the event module, .Event Emitter. There is the function that we want to use. So, the EventEmitter is constructor, and we're going to create a new instance of the EventEmitter. Now, this object that we created here on line 3 has on an emit. So, every time we use on, we can wire up a custom event. You can name an event whatever you like. In this case, I've just called this customEvent. The second argument that the on function takes is a callback function that will be invoked when the custom event is raised. In this case, our custom event is going to pass a message and a status to this function as arguments. So, when our custom event occurs, this callback function will be invoked asynchronously.Let's just go ahead and log the status in the message using template string. So, I'm going to use those ` characters. And we'll go ahead and grab the status. And now, I'm going to go ahead and add the value of the message to this template string. There we go. So, when a custom event is raised, we'll pass a message and a status to this callback function asynchronously, and we're just going to log that message in status. The next part of the EventEmitter is the ability to trigger or emit custom events. We can trigger or emit a custom event with the emit function. So, emitter.emit will fire our custom event. The first argument is the event name that we want to fire. And then, the next arguments are going to be the arguments that will passed to the callback function. So, the first argument in the callback function, the message, is actually the second argument of this emit function. So, for the message, I will send Hello World. And the third argument is going to be the second argument in the callback. So, I will send a status of 200. In this code, we've created a new instance of the EventEmitter object, and we wired up a listener to listen for custom events. Now, we are going to emit a custom event.Let's go ahead and save this and go over to the Terminal, and let's run our file,node BenFranklin. We can see here that this worked successfully, giving us a status of 200 and a message of Hello World returned in the Terminal. The EventEmitter is rarely used as a standalone object. We can really get mileage out of it by allowing our objects to inherit the EventEmitter. Let's change our code so that we have a person object that inherits the EventEmitter.Let's go back to our code in Sublime.The first thing that I want to do is instead of including just the events up here on line one, we can actually pull the EventEmitter out of events directly in this require statement here. So, I will use EventEmitter as a variable. And then, I can just chain on to the end of this require statement EventEmitter. And that will pull that constructor function out of the events module and set this variable to our new constructor function. The next thing that we're going to do is we're not going to just create an instance of the emitter. So, I'm actually going to delete all of this code. What I would like to do is create an object, and that object is going to be a Person. Now, in JavaScript, objects are functions. So, we will create a constructor function here for this object. In this constructor function, we're going to take in the Person's name. In this object, this Person's name will be set to the name value that is passed to this constructor function. So, we're going to create an instance of a Person. This instance of a Person is going to be an object that has a name property. Alright, so, what we want is our Person object to inherit the EventEmitter. We can actually use the utilities module to help us do this. Up here, on line two, I'm going to create a var called util, and I'm going to require our utilities module. The utilities module has an inherits function, and it's a way that we can add a object to the prototype of an existing object. That's how JavaScript handles inheritance. So, what I'm going to do is add util.inherithere on line eight. We want our Person object to inherit our EventEmitter. So, this one line here has just added the EventEmiiter to the Person's prototype,which means that the Person object inherits all of the functionality from the EventEmitter. If I create a new instance of a Person, it will have an on and emit function. So, let's create an instance of BenFranklin or ben with this new Person object. So, here on line 10, I'm going to add a variable for ben. That will be the new Person that we're going to create an instance of. I'm going to invoke my Person function with a new keywordbecause it is a constructor. When we invoke this new Person function, we're going to send it a name. So, there we have created a new instance of a Person.And the name of this new instance is Ben Franklin. So, because this Person inherits the EventEmitter, ben can listen for custom events. We could listen for, let's say, when ben speaks. If a speak event is raised, this callback function will be used to handle it. And whatever ben just said will be passed to this function.I'll go ahead and add that there. So, this callback function will take in said as the argument when the speak event raises.We'll go ahead and just log what ben is saying. I'm going to use a template string to do that. Let's use those ` characters. What we're going to do is use a $ and a couple of {s to display this or this current Person, ben's name. So, we can use the this keyword becausethis callback function knows that this is the current instance of your object. This is equal to ben. And then, we'll go ahead and also log what they said. Alright, so, ben also has an emit function, which means that this instance of ben can emit a speak event. You may delay, but time will not. There we go. So, whenever ben emits a speak event, any listeners that listen for the speak event, like the one we've wired up on line 12, will fire their custom callbacks. In this case, we are passing You may delay, but time will notto that callback function as the said variable. When this happens, we're going to log the name of the person,and we're also going to log what they just said with the recent speak event. So, let's go ahead and save this. Let's go back to our Terminal, and let's run our application again. We will node BenFranklin. We can see that Ben says You may delay, but time will not. The EventEmitter provides us a way to create custom objects that raise custom eventsthat can be handled asynchronously.And because the events are handled asynchronously, it is a very important tool in node.js.

Creating child processes with exec

- Node.js comes with a Child Process module which allows you to execute external processes in your environment.In other words, your Node.js app can run and communicate with otherapplications on the computer that it is hosting. In the next two lessons we're going to take a look at the two main functions used to create Child Processes:spawn and execute. So far we've been working with the terminal or the command prompt. We can actually type other terminal commands into this prompt. For instance open . will openmy current folder inside of the finder.These commands are going to bedifferent based on the system. In order to do this in Windows you would explorer . Some commands are the same for both systems. Opening a URL for instance will work the same in both systems; open http:// your URL will open that URL in your default browser. But most commands are unique per system.To list files on the Mac I would us the ls command where as I would use the dir command to list files on a PC. If I wanted to open an instance of the terminal I would open -a Terminal . and that would open a new instance of the terminal on my current folder. In Windows that is cmd . it will open a new instance of the command prompt in that folder. When can also use externally installed apps that we can communicate with through the command line. For instance, if I wanted to find out the version of git that I had installed I could type git versionand that would tell me that I have git version 2.3.8 installed. With the Node.js execute function we can actually execute these external commands from our Node.js modules. Let's go ahead and go to our file system and you will notice that under the start folder there is an exec.js file. This is an empty JavaScript file. Let's just go ahead and open it up.With the Child Process module we have an execute function. I'm going to go ahead and extract that from this module by creating a new variable and calling it exec, and then I'm going to require the"child\_process" module. The Child Process module contains the execute function, so I'm just going to chain this on the end so I can pluck it out into my execute variable. Now with the exec function I can execute commands. I can exec, for instance, an open, and now when I actually save and run this node module we will go ahead and open LinkedIn in our default browser. Let's go ahead and go to the terminal and I will clear this out and we will go ahead and node our file. So we will node our new file and we can see that when our file starts LinkedIn is also opened in our default browser window. Let's no back to our file in sublime and let's try and execute a different command. Let me try and open a terminal. This command is different in the dos environment. For me it's "open -a Terminal ." in the dos environment it's just cmd. to open a new instance of the command prompt. So I will go ahead and save this, go out to our files and run my file again. You can see that running that file opens another terminal instance. Let's go back into our file, and this time let's execute a processwhich would display some output to the terminal. We can actually collect that output as well. What I'm going to do is execute a ls or a directory listing. You can execute a dir if you are in a Windows environment. Now every time we've been executing any of these processes any data that gets returned by the process would be returned to the second argument in the execute function, a call back function. So in this call back function after we execute ls and we receive information back from lsit will be injected into this function via argument. So first, if there are any errorswhen executing a directory listing we pass those as the first argument. The next argument is the stdout, or standard output, that we've received from executing the ls. So I will add that there.What I'm going to do is in this functionif there was an error executing a directory listing I'm just going to throw it. If there is an error we can throw that error and that will cause our application to crash and our process to exit. But, this should execute fine, so we can just list the console output. So I'll just go ahead and log a ls finished, or a "Listing Finsihed", and then I will go ahead and log that output which would be found on our stdout object. There we go. So let's go ahead and save this and now let's go back into our terminal and let's run this file again. Now we are executing a directory listing and we can see it says Listing Finished and the only thing it found was exec.js in this file. Let's see what happens if we go in here, back to our code, and we try to execute that git command. I'm going to go ahead and type "git version". Down here instead of adding a "Listing Finished" we will just install "Git Version Executed". I will go ahead and save this and let's go back to our terminal and we can actually run our file again and we can see that we have executed the Git version in out stdout from executing that Git Version command. Displays git version 2.3.8. So the execute command is a nice tool that allows us to execute external processes found in our environment.

Creating a child process with spawn

- In the last lesson we used the execute function off of the child process module.In this lesson we're going to take a look at the spawn function, and also contrast the difference between these functions.Let's go ahead and take a look at our files. So in our files system under the Start folder, you'll notice that we have two JavaScript files now. One is call spawn.js, that's an empty JavaScript file,and the other one is alwaysTalking. Let's take a look at alwaysTalking really quick.So alwaysTalking, in this JavaScript file we just have an array of sayings that we declare here on line 1. And then on line 11 we are creating an interval, so every second we are going to randomly write one of these sayings to the standard output object. So on line 12 we get a random index that is no greater than the length of the sayings array, and then on line 13 we use that random index to randomly select one of the sayings, and using the stdout object we will write that saying along with a new line. On line 16 we wired up a listener on the standard input object to listen for a data event. So whenever data is sent to this module via standard input, the callback function will fire and we will see the data there. On line 17 in the console log we will just log the data that comes into this function,and then we will clear the interval, and then on line 19 we will exit the process.So we will stop the sayings from happening and also exit this process.Let's take a look at this file. So let's go ahead and node alwaystalking, and we can see that we see a bunch of Ben Franklin quotes every second. So this is an ongoing file. All of this data is being passed to the standard output object,that's why we're able to see it here on the terminal. As you notice this is a large amount of output. The longer we wait, we're going to get more output. Plus this process is ongoing, it doesn't end. It won't end until we send it some data, so I'm going to send it a stop command. So sending it stop causes the standard input data received message to get displayed, and we have the stop command, that's the data that was received, and then it also calls the application to exit. So this is a large bit of data, and it also is a long running process. So the difference between spawn and execute is spawn is made for processes like this, where execute was made for asynchronously synchronous processes, meaning that if I do a directory listing we run that process, we get the listing in standard output, and then the process is over. If I do a git version the process runs, and then we quickly get the standard output and the process is over. So if we have these processes that have small bits of data,they're perfect for execute. So spawn is made for longer, ongoing processeswith large amounts of data, like our alwaysTalking. So notice this process doesn't just give us some data and immediately end, which means that execute is not the function that we want to use with this. So let's go ahead and go back into our files, and from our files you will notice that we have a spawn.jswhich is just an empty file. What we're going to do this time is use the spawn function. So I'm going to create a variable for that function, and then I'm going to require our child process module. And I'm going to go ahead and pluck that spawn function out of there.So now I have my spawn function set up.This time I'm going to create a variable for the child process, and that will be returned by my spawn function. The first argument that I'm going to send to the spawn function is the command that I want to run in the terminal. That is node.The second argument is going to be an array of all of the things that I would runafter the node command. So in order to run the alwaysTalking app we will node alwaysTalking in the command line, so I'll just go ahead and put alwaysTalkingor the options that would show up in the command line after node into an array,which represents the second argument of the spawn function. So with the child process instance I can listen for data events on the stdout object. So cp.stdout is the stdout object of the child process. Whenever we have a data event, this callback function will fire. And this callback function will fire, and we will receive that data as an argument. So I'm just going to log that data. We'll go ahead and log that we received some data, stdout, and then we can actually log what that data is right here. Don't forget your last back-tick for that template string. There we go. Another thing we can do with the child process is we can listen for when they close. A close event will raised on the child process when it closes. When a close event is raised I'm going to invoke this callback function here. So when our alwaysTalking process ends I'm just going to go ahead and log that that has ended. And I will also exit this process.So when our child process alwaysTalking ends, we will also end this process. Now we can send data to this child process instance using the standard input object,so let's go ahead and set a timeout. And this function will be invoked after about four seconds, so we'll set a timeout, we'll let our child process run for about four seconds, and then after four seconds I'm going to send some data to the child process. And I'm going to go ahead and write stop. Any data that we send to the alwaysTalking application will cause it to stop the interval and also exit the application. So after about four secondswe should close that child process by sending it the stop command on the standard input. So I'm going to go ahead and save this, and let's go out to our terminal. And now we can run our app, and when we run this app we can see that we have started our alwaysTalking process. And whenever our alwaysTalking process writes data to standard output, we are receiving that data via our standard output listener.Also when the child process ends, notice that we see STDIN Data Received -> stop. So we are able to send the stop command to our child process, and that has caused it to end, which causes our process to log Child Process has ended,and then our process ends. So we were able to spawn the alwaysTalking JavaScript file from our spawn.js file.Again you're going to use spawn for ongoing, longer processes with more data, and you can communicate with those processes via standard input and standard output.

Listing Directory Files

- Node.js also ships with a module that allows us to interact with the file system.The fs module can be used to list files and directories, create files and directories, stream files, write files, read files, modify file permissions or just about anything that you need to be able to do with the file system. Let's get started by listing the contents of a current directory. Let's go ahead and navigate to our files. In start you will notice that we have lib directory. So we are going to list the contents of this directory. It contains a file called people.json, sayings.md and a sub-directory scripts, and in the sub-directory are the alwaysTalking.js file and the spawn.js file from the last lesson. So what we want to do is inside of our list.js, which is an empty JavaScript file,we are going to list the contents of the lib directory. So let's go ahead and open up list.js. The first thing we need to do is include the file system module, so, var fs, we will require the files system module.So the file system module is fs and through this fs module we can do just about anything with files and directories.I'm going to create a variable for filesand I'm going to set this variable to fs.readdirSync. So we're going to synchronously read the lib directory, so the contents of the lib directory. The next thing that we're going to go ahead and do is just output those contents by logging the files. So, let's go ahead and save and run this. We can go back over to our terminal. Now I can simply node list, and we can see that we get a list of everything that we find inside of the lib directory as an array. We'll notice that it also found that hidden .DS-Store file inside of that directory. You also may notice that our files have an extension but our sub-directories do not. Let's navigate back to our code. So notice that when we read these files we are using the readdirSync command. This means that I read the contents of the directory synchronously with a blocking request. By the time we get to line fivewe should actually have the files. When we use any methods of the fs modulewe are given the option to use themsynchronously or asynchronously. Be careful because reading files synchronously will block the single Node.js thread so all other connections will wait for this synchronous recall. We usually like to read configuration filessynchronously when we start the app.You want to take advantage of Node.js'sasynchronous nature and not read the files from the directory synchronously.We can do that simply by dropping the Sync. When we drop the Sync from any of these fs commands, also readdir is not going to return our files any longer,but this is an asynchronous command,so what it's going to do is put in a request to read the files from the library folder and when the file system is finished reading those files this call back will be invoked. Now, inside of this call back, if there was an error reading the files from the directory that will be the first argument that's passed to this call back. Otherwise, the files themselves will be passed as the second argument. In this case, if there is an error I'm just going to go ahead and throw it, and otherwise instead of logging the files down here I'm going to go ahead and log the files within the call back. So this happens asynchronously which means we can log down here "Reading Files..."So the single threaded nature of Node.jswill come in here and will place a command on line three to read the contents of the library directory. Then it will go on the doing other things. It will go down to line 13 andconsole.log("Reading Files...") Once the file system has actually read the files and we have results, the call back function that we added as the second argument to the readdir function will be invoked.This call back will pass an error if any errors occur during the process of reading files and the actual files from the directory that were read. Let's go ahead and save this and come back out to our terminal and we can run this again. And this time we can see Reading Files...occurs first and then we see the same array of files. So we're doing the exact same thing except we are doing this asynchronously. If we go to the api documentation for the file system what you will notice is that every single one of these commands has a synchronous version and an asynchronous version. If we look at this we can see that we can get file statistics synchronously and asynchronously. We can go ahead and un-link files synchronously and asynchronously. We can remove directories synchronously and asynchronously. We have the option to use the synchronous command or the asynchronous command. Again, we like to use the synchronous command when we start our applications. So if we're reading configuration details, so on and so forth, but when we are actually running our applications, we want to stay away from those synchronous commands and make sure we are putting in asynchronous commands.