

GYMNASIUM

JAVASCRIPT FOUNDATIONS

Lesson 2 Transcript

More JavaScript Basics

ABOUT THIS DOCUMENT

This handout is an edited transcript of the JavaScript Foundations lecture videos. There's nothing in this handout that isn't also in the videos, and vice versa. Some students work better with written material than by watching videos alone, so we're offering this handout to you as an optional, helpful resource. Some elements of the instruction, like live coding, can't be recreated in a document like this one. We encourage you to use this handout alongside the videos, rather than as a replacement of them.

INTRODUCTION

Welcome to lesson two. Here, I'm going to cover functions, a couple of new data types, arrays and objects, and then, move on to events, and advanced debugging. When you finish this lesson, you'll be able to create small programs, with more complex behavior that can react to user input. You'll still be using the console for output in this lesson. I understand you're probably eager to start writing code that interacts with a real web page, but bear with me. That's what the entirety of lesson three is all about.

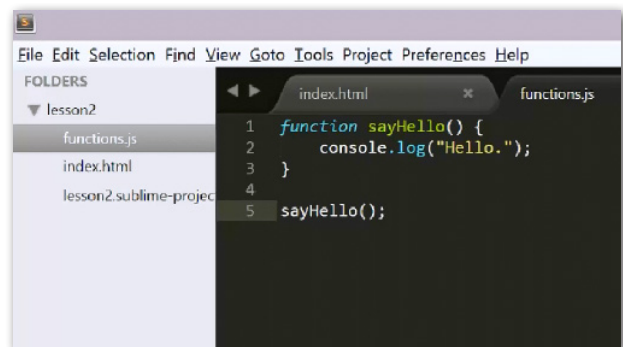
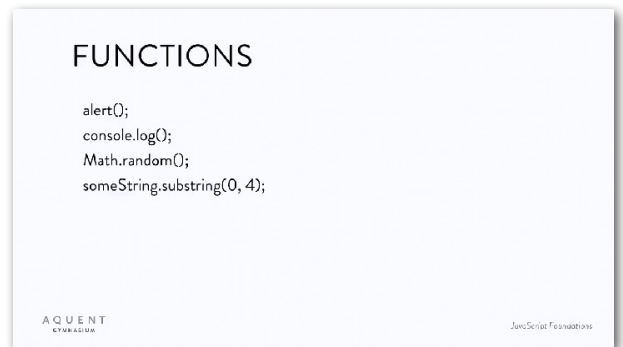
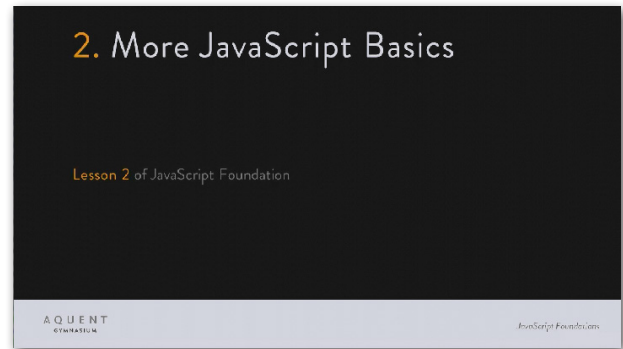
For now, let's get started with functions. You've already used many functions throughout lesson one: `alert`, `console.log`, `Math.random`, and the various string functions, to name a few. Now, you'll learn how to create your own functions that perform other custom actions.

The main reason for writing a function is to prevent you from writing the same code, multiple times. If you find yourself performing the same actions more than once or twice, it's a good idea to create functions for those steps. Then, if you need to do it again, just use the function, instead of typing out those steps again. In the same way that variables are declared using the `var` keyword functions are declared using the `function` keyword.

Here are all the parts to a bare minimum function. You have the function keyword, the name of the function, followed by a pair of parentheses, and a pair of brackets. Inside the brackets, you put the code that defines what you want the function to do. Valid function names follow all the same rules that variable names do.

So, let's go ahead and create this function. I've started a new Sublime Text project for this lesson. The HTML file loads a JavaScript file called, `functions.js`. Right now, that's a blank file. Pause here if you need to, and go ahead and set up a project yourself, with the same two files. You can also download the files from the course's website. Once you're done with that, just type in the `sayHello` function. And, in the body of the function, there's just one line, `console.log hello`.

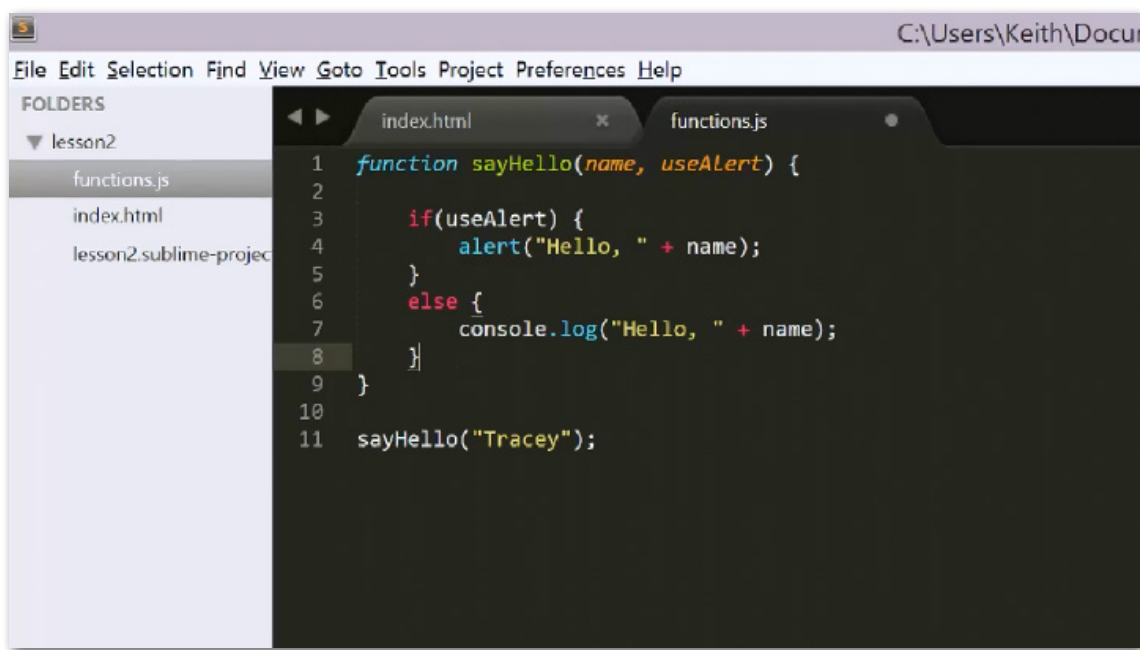
Now, to execute a function, you just type the name of the function, followed by a pair of parentheses, and a semicolon, of course. This is known as calling a function. When you call a function, the code inside the brackets of the function gets executed. So, if we save this, and load `index.html` into a browser, sure enough, it is logged hello.



Now, when you call a function, you can pass data to it. You specify what that data is by placing it in the parentheses of the function definition, like so. Here, name is called a parameter of the function. Parameter names follow all the same rules as variable names. And, in fact, what you're doing here is creating a variable, called name, that is only available to the body of this function. That variable's value is set when you call the function.

Now, you can use that parameter in your function's code. Here, we're just concatenating name onto hello. Don't forget the space. Now, when you call the function, you can pass in a name. Save, and run that, and it says hello, Tracey.

Your function can have multiple parameters. Just separate them with commas. I'll add another one called, useAlert and in the body of the function, I'll use this in an if statement, to either use the alert to output the message, or use the console. Now, if use alert is true, or evaluates to true, then, alert will be used. And, if use alert is false or evaluates to false, console.log will be used. The evaluates part is important.



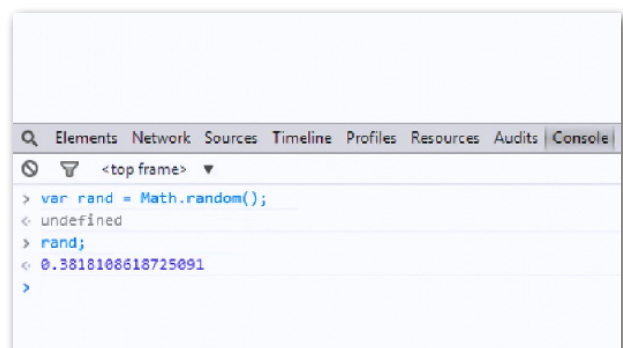
The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a folder named 'lesson2' containing 'functions.js' and 'index.html'. The code editor has two tabs: 'index.html' and 'functions.js'. The 'functions.js' tab is active, showing the following code:

```
1 function sayHello(name, useAlert) {  
2  
3     if(useAlert) {  
4         alert("Hello, " + name);  
5     }  
6     else {  
7         console.log("Hello, " + name);  
8     }  
9 }  
10  
11 sayHello("Tracey");
```

Remember type coercion. Right now, we're just calling the function with a name parameter. In this case, use alert will be undefined. This will evaluate, or be coerced, into false, so the console will be used. This winds up being very useful because it makes that second parameter optional. If you just type sayHello with a name, you get the default console behavior. But, if you type sayHello name, comma, true, then you get the alert behavior.

The last major thing you need to know about functions is that a function can send data back to the line of code that called it. You've seen this with many functions before, such as math.random.

Here, the math.random function is generating a random number, and sending that back to the line of code



The screenshot shows a browser console with the following code and output:

```
> var rand = Math.random();  
← undefined  
> rand;  
← 0.3818108618725091  
>
```

that called it. That value is assigned to the `rand` variable. We say that a function returns a value. You specify the data a function will return with the `return` keyword.

Let's keep going with `math.random` for our next function. As you know, `math.random` returns a decimal number between zero and one. And, I mentioned in lesson one that if you wanted a larger random number, like between zero and 100, you could multiply this by 100. But, this will still give you a decimal number, something like 65.49929.

But, sometimes, you need a random whole number between zero and 100, for example, something like 37, or 48, or 93. Well, you can generate a random number, multiply it by the maximum value, like 100, and then use `math.round` to round it to the nearest integer.

Well, that's three steps you'd have to do every time you wanted a random integer. And again, multiple steps that you need to do multiple times are perfect candidates for functions. So, let's define a `randomInt` function, like so. This function has one parameter, `max`, which represents the maximum number that will be returned. Then, generate a random number, and multiply it by `max`, storing this in a variable called, `rand`.

Next, say, `rand` equals `math.round rand`. This rounds the value of the `rand` variable, and stores the result back in `rand`. Finally, return `rand`. You can then call the `randomInt` function, passing in a `max` number. In fact, you can wrap that right in a `console.log` statement, and have it output to the console.

Now, we already have some code here, that's calling the `sayHello` function. I don't want that to happen right now, but I don't really want to delete it, either. So, this is a good time to introduce comments in JavaScript.

You're probably already aware of comments from HTML or CSS. These have formats like the one shown here. The browser will just ignore anything inside of those comments. JavaScript also uses the slash star type comments that CSS uses. In addition, you can turn any single line into a comment by starting it with a double forward slash. When JavaScript sees this slash slash, it completely ignores that line. You can use this for temporarily disabling a line of code.

```
var rand = Math.random()
rand = rand * 100;
rand = Math.round(rand);
```

Round it to get an integer like 65.

AQUENT
EXPERIMENT

JavaScript Foundations

index.html
lesson2.sublime-project

```
3  if(useAlert) {
4      alert("Hello, " + name);
5  }
6  else {
7      console.log("Hello, " + name);
8  }
9  }
10
11 function randomInt(max) {
12     var rand = Math.random() * max;
13     rand = Math.round(rand);
14     return rand;
15 }
16
17 console.log(randomInt(100));
18
19 sayHello("Tracey", true);
```

```
<html>
<!-- This is an HTML comment. -->
</html>

p {
  /* This is a comment
  in CSS
  */
}
```

Comments in HTML and CSS.

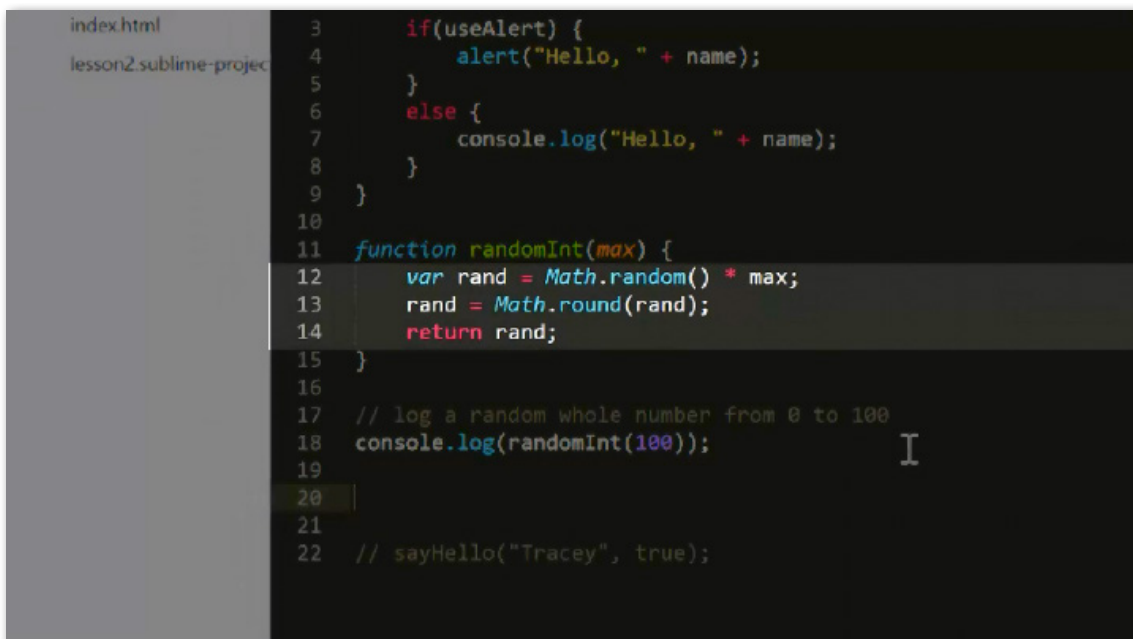
AQUENT
EXPERIMENT

JavaScript Foundations

In Sublime Text, Control or Command plus the forward slash is a shortcut for creating a comment. Note, that it also turns all the text on that line gray for a visual indication that this is no longer functioning code. Of course, the original purpose of comments was for adding actual comments to your code that help explain what's going on, like so.

So, now, you can save this, and run it in a browser. You'll see that the sayHello function is not running because it's now commented out. But, we do get a random integer. Refresh the page a few times, and verify that you get other random numbers, all from zero to 100.

Now, one very important thing to realize is that the rand variable is only available inside the body of that randomInt function. The variable is created only when the function is called. It is assigned a value, and can be used within the function, but once that function ends, that rand variable disappears.



```
index.html
lesson2.sublime-project
3     if(useAlert) {
4         alert("Hello, " + name);
5     }
6     else {
7         console.log("Hello, " + name);
8     }
9 }
10
11 function randomInt(max) {
12     var rand = Math.random() * max;
13     rand = Math.round(rand);
14     return rand;
15 }
16
17 // log a random whole number from 0 to 100
18 console.log(randomInt(100));
19
20
21
22 // sayHello("Tracey", true);
```

Variables declared within functions are called local variables, because they don't exist outside the confines of that function. We also talk about the scope of a variable. The scope defines the parts of that code where a variable is visible, and can be used. Here, the scope of the animal variable is the doSomething function. And, in our code, the scope of the rand variable is the randomInt function.

Let's try to access rand outside of the function, and see what happens. We'll just say console.log rand. And, you can see that rand is not defined outside that function. In fact, just by trying to access it, we're getting an error.



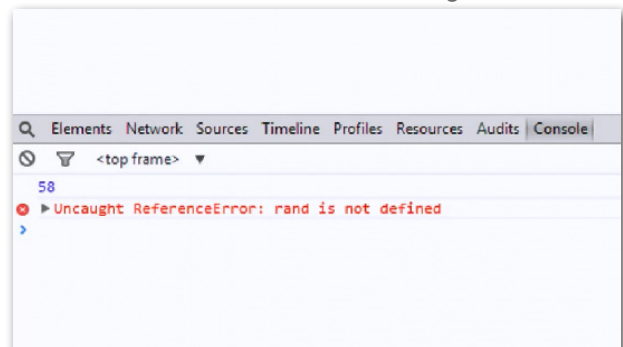
```
function doSomething() {
    /* This is a comment
       in JavaScript.
    */
}

function doSomethingElse() {
    // This is a single line comment in JavaScript.
}
```

You can also use // for a single line comment.

AQUENT
STRAVIA UN

JavaScript Foundations



Elements Network Sources Timeline Profiles Resources Audits Console

<top frame>

58

Uncaught ReferenceError: rand is not defined

>

Okay, so, that's the basics of what functions are, and how to create, and use them. You'll be seeing plenty of them as the course progresses, though, so before long, they'll become one of your best friends.

ARRAYS

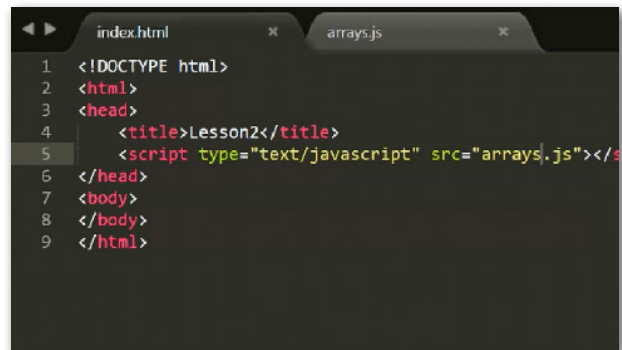
Next up, let's talk about arrays. An array is a way of storing multiple values in a single variable. The best analogy I can think of for an array, is one of those weekly pill organizers, with a little compartment for each day. An array has a nearly unlimited amount of slots you can store data in. These are called the array's elements. The elements are specified by an index, which is a whole number. The first element in an array is at index zero, the second one at one, and et cetera. Square brackets are the key to arrays. You can create an array with two square brackets. You can insert any data into an element of an array, or read the value that is stored in any element, by specifying an index inside those brackets. It can then be used just like any other single variable.

Let's try it out. I've created a new file called arrays.js. And, I've changed index.html, so, that it loads this arrays.js file. Here's that js file. It's currently empty.

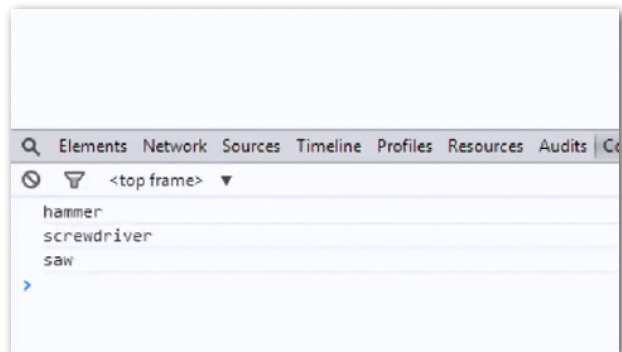
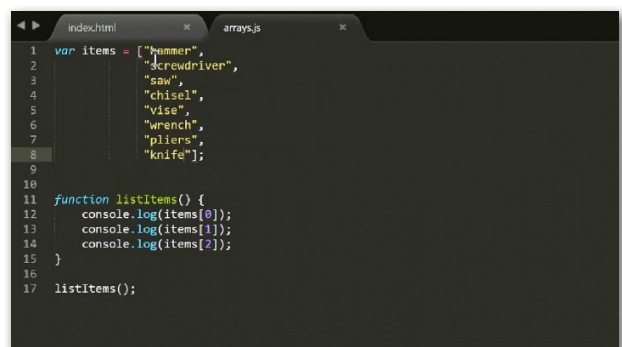
Now, imagine you're creating a site for a hardware store, and you want to create a function that lists all the items that are for sale. Without arrays, you'd need a separate variable for every item they sell. Item zero equals hammer. Item one equals screwdriver. Item two equals saw. Then, you could create a list items function that logged all of those items. So, we're just saying console log item zero, console log item one, and console log item two. So, we can call that function, and then, check the console. And, there are your items all listed out.

But, a real hardware store would contain hundreds, if not thousands of items. Having thousands of separate variables to keep track of would be a nightmare. And, your list items function would be thousands of lines long. Not to mention that there'd be no way to sort the items, or filter them to display only certain ones, or easily add, or remove items without manually rearranging them all. Yuck! Well, arrays can handle all that.

To start, let's get rid of the individual variables, and create an array called items. Then, assign each item to an element in the array. So, items zero equals hammer, items one equals screwdriver, et cetera. Then, in a list items function, log each element in the same way; console log items index zero, items index one, et cetera. That's marginally better. At least we've eliminated a bunch of variables.

A screenshot of a code editor showing the contents of index.html. The code is as follows:

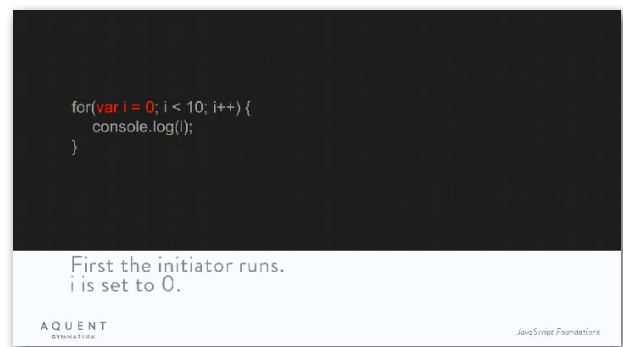
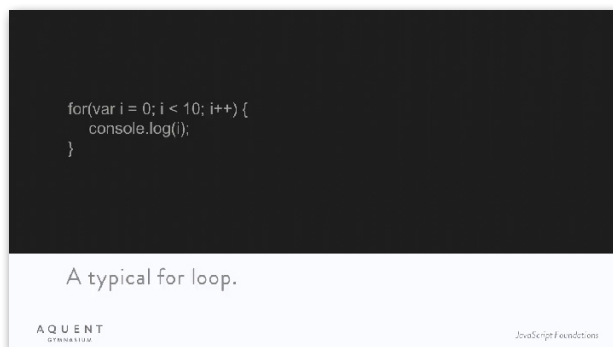
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <title>Lesson2</title>
5   <script type="text/javascript" src="arrays.js"></script>
6 </head>
7 <body>
8 </body>
9 </html>
```

A screenshot of a web browser's developer console. The 'Elements' tab is selected, showing a list of items: 'hammer', 'screwdriver', and 'saw'. The console also shows a blue prompt character '>' at the bottom.A screenshot of a code editor showing the contents of arrays.js. The code is as follows:

```
1 var items = ["hammer",
2             "screwdriver",
3             "saw",
4             "chisel",
5             "vise",
6             "wrench",
7             "pliers",
8             "knife"];
9
10
11 function listItems() {
12   console.log(items[0]);
13   console.log(items[1]);
14   console.log(items[2]);
15 }
16
17 listItems();
```

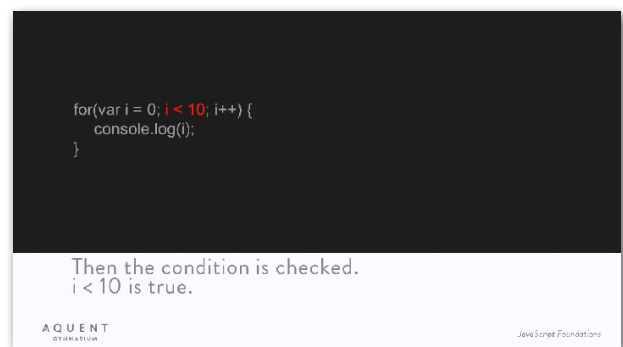

But, let's keep going. You can actually assign the elements of an array when you create it. Just list them inside the square brackets that creates the array, separated by commas, like this. So, we have "hammer", "screwdriver", "saw." Most developers space them out, like so, for easier reading. While I'm in here doing that, I'll add a few more elements.

But, what about this list items function? You don't want to just keep adding lines in here. Well, here's where one of the real powers of arrays come in. In addition to simply holding multiple values, arrays can be used within a special structure, called a for loop, that lets you access every single element in the array, with just a couple lines of code. So, let's take a look at this magical for loop. The for loop is a construct that allows code to be executed repeatedly, hence, the name loop. Its structure is similar to an if statement. It has the "for" key word, then a pair of parentheses, and a block of executable code within brackets. It's what goes within these parentheses that makes it so useful. Rather than a single conditional statement like an if statement, the for loop gets three expressions, separated by semicolons.



Now here's a real-life example of how the vast majority of for loops are used. The first expression is called the initiator. This runs a single time when the for loop is first encountered. This one declares a variable *i*, and sets it to zero. In lesson one, I said that a single letter variable is not a good idea. But, this is an exception. Here, *i* doesn't have any real meaning, other than keeping track of how many times a loop has been run. And, it's one of the most commonly used variables in for loops, so, everyone just kind of expects an *i* there.

The next expression is called, the condition. This is executed right after the initiator. If the condition evaluates to true, then the code inside the brackets is run. If the condition evaluates to false, then the code is skipped, and the for loop is complete. Here *i* is less than 10. So that evaluates to true, and the code runs logging *i*, zero.



After the code runs, then the last expression is executed. This is sometimes called the incremter, because it usually contains a plus plus, or a plus equals increment operation. Here, it's incrementing *i* by one, making it one. Then, the conditional is checked again. *i* is still less than 10, so the code runs again, logging one. This whole cycle continues to repeat, logging two, three, four, et cetera. Eventually, *i* will get up to 10.

This time the conditional will fail. The code will not be run and the for loop exits. Any code that occurs after the for loop, now continues to run. The result is number zero through nine are logged. This is how the for loop is most commonly used; to run a certain block of code, a specific number of times. It's very often used in conjunction with arrays to run the same chunk of code, with every single element in an array.

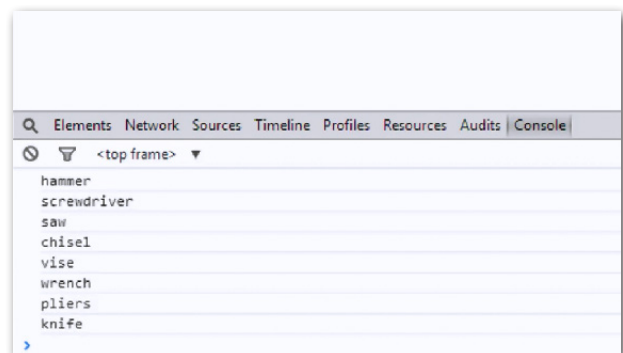
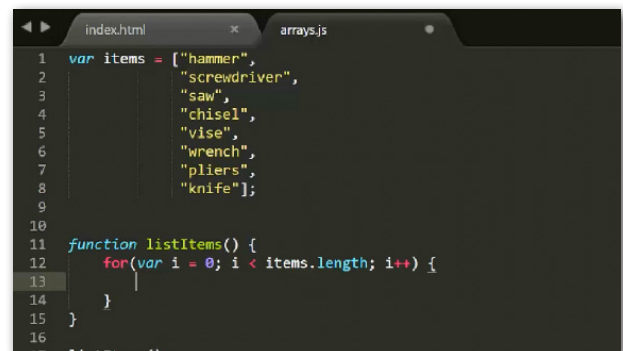
So, let's see how we can use that in our code. The items array currently has eight items in it; elements zero through seven. In the list items function, create a loop that starts with i zero, a condition of i less than eight, and an incrementer of i plus plus. Actually, it turns out that an array has a length property that will tell you how many elements that array currently has. So, the conditional can be changed to i is less than items.length.

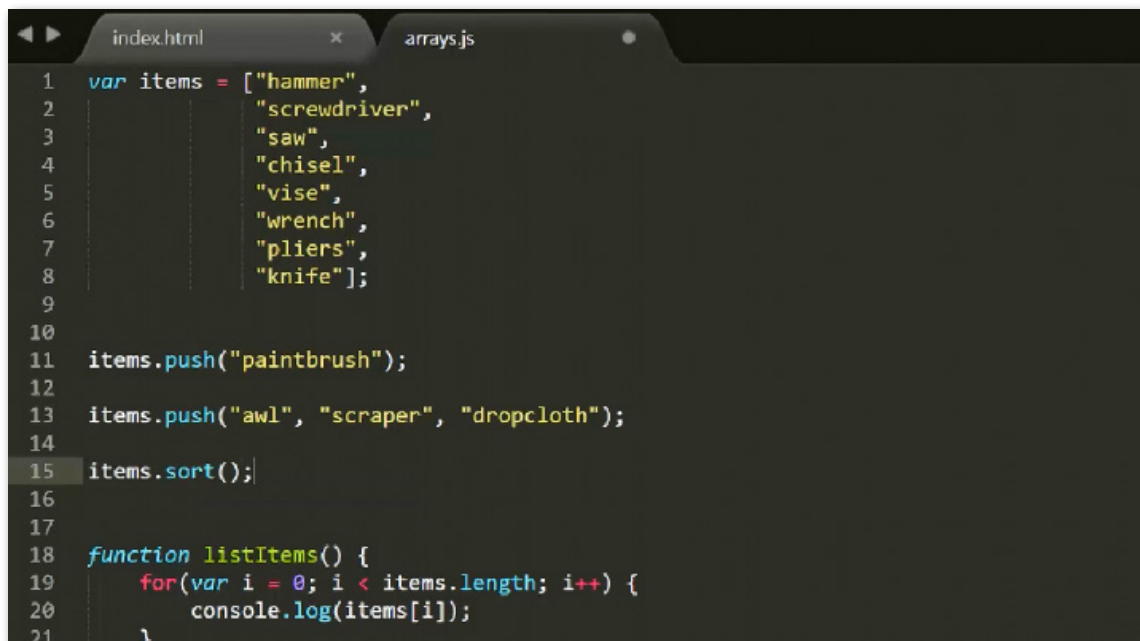
Now, that code block should be run eight times. Each time i will be different. The first time, it'll be zero. And, the last time, it will be up to seven. Well, those are the index numbers we need to access all the elements in the items array. So, we can reduce this to a single line; console log items, index i. The first time through the loop, i will be zero. So, this will log the value in items elements zero. The next time, i will be one. So, items element one will be logged, and so on, up to items element seven. When i equals eight, the condition will be false. And, the for loop will complete, skipping the code inside.

Save this. Run it in a browser. And, sure enough, all the items are displayed.

I suggest that you pause the video here, and play around with this. Try adding any number of new items to the array, or delete a few. Change the order around. Run the code each time, and see that the list items function continues to work perfectly. No changes are needed. In addition, there are lots of built-in functions that give arrays all kinds of power. For instance, to add one or more items to the end of array, you can use the push function. You can add a single item, saying items.push "paintbrush." Or add a list of items: items.push "awl", "scraper", "dropcloth".

And, then, there's a sort function that will sort the array alphabetically. Just say list items sort. There are lots of options to the sort function. So, you should check out the documentation for that one. If we run this now, you see all the new items are in there, and it's all sorted alphabetically.





```
1  var items = ["hammer",
2              "screwdriver",
3              "saw",
4              "chisel",
5              "vise",
6              "wrench",
7              "pliers",
8              "knife"];
9
10
11  items.push("paintbrush");
12
13  items.push("awl", "scraper", "dropcloth");
14
15  items.sort();
16
17
18  function listItems() {
19      for(var i = 0; i < items.length; i++) {
20          console.log(items[i]);
21      }
```

So, that's a whirlwind look at arrays. You'll be seeing plenty more of these as we go through the rest of the course, too. Be sure to check out the documentation to see what else you can do with them.

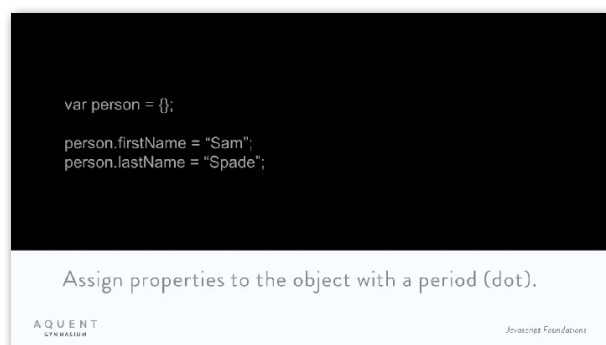
We have one more powerful data type to look at next: objects.

OBJECTS

So far, we've covered almost all the data types in JavaScript: numbers, Boolean, strings, null, undefined, and arrays. Technically, even functions are a type of data in JavaScript, as well.

But, I saved the best for last, objects. Objects are basically a composite type of data. An object is composed of other pieces of data, or even functions. You can create your own objects, store them in variables, and give them various properties, and behaviors, and then, use those objects in your code. It's quite powerful.

Similar to an array, you can create an empty object using curly brackets. Now, you can add properties to this object, and read those properties as well. Just type the name of the object, plus a period, plus the name of the property. Then, you can use that property, just like any other variable. Those property names follow the same rules as variables, functions, and just about any other identifier in JavaScript follows.



```
var person = {};
```

```
person.firstName = "Sam";
person.lastName = "Spade";
```

Assign properties to the object with a period (dot).

AQUENT
© 2018 AQUENT

JavaScript Foundations

One way of looking at objects is that, it's a variable that contains other variables. But, objects are a great way to group information together in ways that would be very difficult to do, otherwise.

Let's go back to the hardware store example we created in the last video. I've copied this over into a new file called, "Objects.js", which is now loaded by index.html. Now, in addition to item names, you'll probably want to keep track of the cost of each item, its ID, and maybe a URL to an image of that item. How would you do that?

Well, I've seen a lot of new developers make another array for each new property. Let's try that. You can follow along with this section, or just sit back, and watch how this goes. Let's just keep track of cost, for now.

We have an items array with the names of eight items in it. I'll create another array called "costs", and populate that with the cost of each item in the array. I need to be very careful to make sure that the first element in costs, represents the cost of the first element in items, et cetera.

Okay, I've got those in. But, here, I've pushed a few more elements onto the items array. So, I'll have to also push their prices of those in here, too. Now, this is starting to feel a little shaky to me. There's nothing really connecting these two arrays, other than the fact that they're very carefully held in the same order. Every time I add something to, or somehow change one array, I have to be sure to make the exact same change in the other one. Now, imagine additional arrays like this for ID, picture URL, maybe a description, et cetera. It starts to feel like a house of cards, just about to collapse. But, I've been very careful, here. So, you should be okay, right?

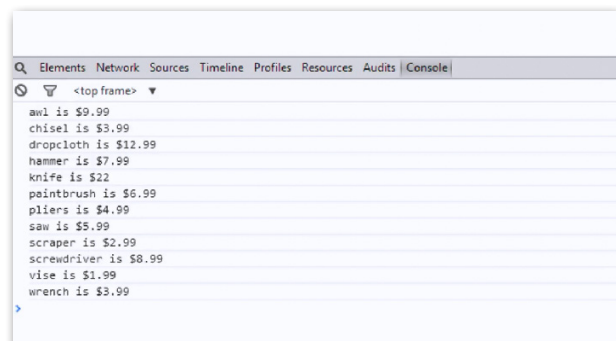
Let's change the list items function, so that it prints out the name and the cost. We'll just concatenate those two values onto one long string. And, now we test this. Uh oh, that's not right. I specifically remember the hammer being \$9.99. And, the vice was the most expensive item. Now, it's the cheapest. Nothing else looks quite right, either. So, let's go back to the code.

Well, what happened was this line here, items.sort. This totally rearranged the order of the items array, so that it's no longer in sync with the costs array, at all. And, there's no easy way to keep these two things in sync. If we sort costs, it will sort that array differently than it sorts items. And, it will just make things even more random. The solution, as you might have guessed, is to use objects. This will let you consolidate all the data about a single item into a single package. Let's convert this back to where we were, before we tried multiple arrays.

You might want to open up your editor and start following along, again. First, get rid of all these item names, and restart with an empty items array. Now, let's create the first item, an object. We'll give it some properties;

```
index.html  objects.js
1  var items = ["hammer",
2      "screwdriver",
3      "saw",
4      "chisel",
5      "vise",
6      "wrench",
7      "pliers",
8      "knife"];
9
10 var costs = [9.99,
11     3.99,
12     12.99,
13     7.99,
14     22.00,
15     6.99,
16     4.99,
17     5.99];
18
19 items.push("paintbrush");
20
21 items.push("awl", "scraper", "dropcloth");
```

```
10 var costs = [9.99,
11     3.99,
12     12.99,
13     7.99,
14     22.00,
15     6.99,
16     4.99,
17     5.99];
18
19 items.push("paintbrush");
20 costs.push(2.99);
21
22 items.push("awl", "scraper", "dropcloth");
23 costs.push(8.99, 1.99, 3.99);
24
25 items.sort();
26
27 function listItems() {
28     for(var i = 0; i < items.length; i++) {
29         console.log(items[i]);
30     }
31 }
32
```



```
Q Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
awl is $9.99
chisel is $3.99
dropcloth is $12.99
hammer is $7.99
knife is $22
paintbrush is $6.99
pliers is $4.99
saw is $5.99
scraper is $2.99
screwdriver is $8.99
vise is $1.99
wrench is $3.99
```

first, the name, and the cost. Then, also let's add an ID, and a string for holding an image URL. Just to show you how you can consolidate all kinds of information on one object. Then, we push that object onto the array.

Now, items index zero will be this hammer object. You could continue to do this for each item. But, there's another shortcut. Remember that you were able to specify all the elements in an array, when you created the elements by putting them inside the square brackets. Well, you can do the same thing with objects. You can specify the properties when you create the object, by placing the properties inside the curly brackets.

Here's how that looks. Inside the brackets, I've listed each property name, followed by a colon, and then the value that I want that property to have. Here, I've created a name property with the string value hammer, a cost property with the number value 9.99, and so on. Now, you can actually even combine populating the array with populating each item. Just list each object inside the square brackets of the array, like so.

So, we start by opening the square bracket. Then, we create our first object, then a comma, then our second object, another comma, and the third object. So, these are the three elements in the items array, now.

This may seem complex at first, but once you get the hang of it, it's a very concise and elegant way to create a bunch of objects. All you need to do now is fix up the list items functions, so that it uses these new objects. Because each item is an object now, rather than just a string, you'll need to specify, which property you want to log. Here, I'll use the item's name property, plus a cost, and concatenate those into one long string. And, let's see how that looks, perfect. And, because each object contains all the properties of that specific object, the list will never get out of sync. So, now, you know the very basics of objects, how to create them, and give them properties, and access those properties. Next, we'll look at a few more advanced operations with objects.

```
index.html arrays.js
1 var items = [];
2
3 var item = {
4   name: "hammer",
5   cost: 9.99,
6   id: 101,
7   picture: "hammer.jpg"
8 };
9
10 items.push(item);
11
12
13 function listItems() {
14   for(var i = 0; i < items.length; i++) {
15     console.log(items[i]);
16   }
17 }
18
19 listItems();
20
```

```
index.html arrays.js
1 var items = [
2   {
3     name: "hammer",
4     cost: 9.99,
5     id: 101,
6     picture: "hammer.jpg"
7   },
8   {
9     name: "screwdriver",
10    cost: 4.99,
11    id: 102,
12    picture: "screwdriver.jpg"
13  },
14  {
15    name: "saw",
16    cost: 12.99,
17    id: 103,
18    picture: "saw.jpg"
19  }
20 ];
21
```

Elements Network Sources Timeline Profiles Resources Audits Console

<top frame>

```
hammer is $ 9.99.
screwdriver is $ 4.99.
saw is $ 12.99.
```

```
var messenger = {};
messenger.sayHello = function() {
  console.log("Hello.");
};
```

An anonymous function has no name.

AQUENT

JavaScript Foundations

OBJECT METHODS

In the last section, you learned how to make objects and manipulate properties on them. But, that's only half the story. In addition to properties that can be numbers, strings, Booleans, or other types, objects can have functions attached to them. I briefly mentioned in the last video that functions are just another data type, so you can just add another property to your object, and assign it a function. This looks like this.

I've created a messenger object, and we've created a new property called, `sayHello`, and we assign a function to that. This is almost exactly the same way we were creating functions before, but notice that this time you're not giving the function a name.

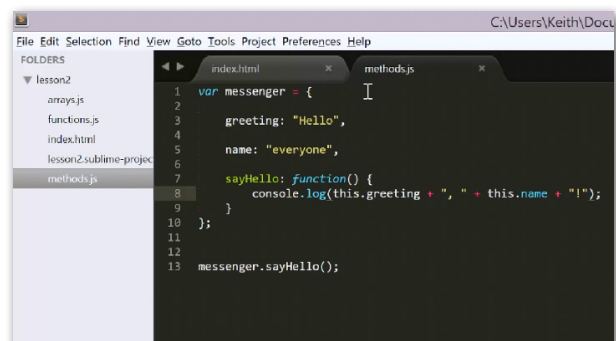
You just use the `function` keyword, directly followed by parentheses. Because it doesn't have a name, it's called an anonymous function. But, you've assigned it to the `messenger.sayhello` property, so you can access it from there.

Functions that are attached to objects like this are often called methods, to distinguish them from other functions that are not part of objects you created. So, don't get confused by the term method. It just means a function that's part of an object. Well, let's try this. I've created a new empty `methods.js` file that is loaded from the `index.html` file. And here, I'll just create the messenger object and give it the `sayHello` function that logs "Hello."

And, now, you can call that function by saying, `messenger.sayHello`. And, we check the console, and that worked. Now, this is exactly what's going on when you type `console.log` or `math.random`. `Console` and `math` are objects that are built into the JavaScript interpreter in the browser. `Log` and `random` are methods of those objects. They're just functions. It's the same when you create a string, and call a function like, `substring`. That string you created with quotes is converted to a JavaScript string object. That string object has all kinds of methods on it that will manipulate that string.

Like other object properties, you can assign functions when you create the object. Again, just list the function name within the brackets, followed by a colon, and then the function itself. Object methods have one other, very powerful feature. They're able to access the object that they're attached to, with the keyword, "this." Let's add a couple of other properties to the messenger object, `greeting` and `name`.

Now, within the body of the `sayHello` method, you can use the keyword, "this" to refer to the messenger object. So, you can change that `console.log` function to `this.greeting` plus comma, space, plus `this.name`. This `greeting` is really `messenger.greeting`. And, `this.name` is really `messenger.name`.



So, running this, we see “Hello, everyone!” in the console. After creating the messenger object, you can change either, or both of those properties. Because a `message.greeting` equals “Konichiwa,” and `message.name` equals “minasan.” Run it again, and we’re saying, “Hello, everyone!” in Japanese.

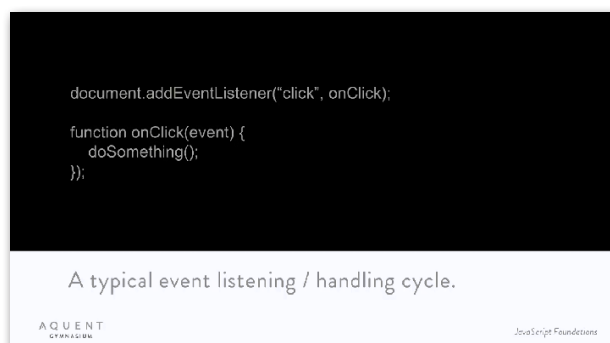
You can go pretty deep into this subject of objects. You’ve most likely heard of the subject of object oriented programming. It’s a vast subject concerning the creation, manipulation, and use of objects to perform various programming tasks. But, at its heart, it’s all about objects, properties, and methods. Okay, now we’ve covered all the data types in JavaScript, and we can start doing some really interesting stuff. Next up, I’m going to cover events, and let you introduce a bit of interactivity into your programs.

EVENTS

So far, in almost all of the examples you’ve worked with, you’ve been writing some code. That code gets loaded in, and runs. It does something, usually involving an alert or a console log. That’s the end of the story. Program is complete, and the web page just sits there in a static state. In this section, I’m going to show you how to add interactivity to your programs. This involves using events to wait until something happens, and then run some bit of code, when it does. The events we’ll be covering here will include mouse events, and keyboard events. But, in the rest of the course, other types of events will play a big part as well.

In JavaScript, an event means pretty much what it means in English, something’s happened. There are all kinds of events in JavaScript; events that occur when pressing down, or releasing keys on the keyboard, for moving the mouse, or pressing and releasing mouse buttons, touch events for devices with touch screens, and events for when something’s loaded, to name a few of the most used ones. We say that you listen for events, which means that you indicate that you want to be informed, when a certain type of event occurs. And, then, you handle those events, which generally means that you write a function that will be executed when that event occurs. Event handling usually looks something like this.

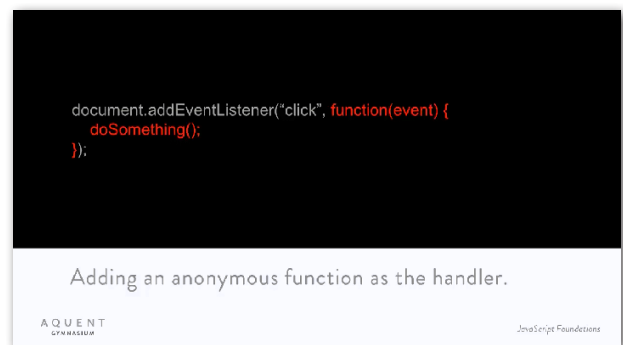
So, let’s break this down. First we have an element. Most events are associated with some HTML element on the page. In this section, we’ll just be dealing with the document element, which represents the HTML tag itself. But in lesson three, you’ll learn how to access any element on any HTML page. Associating events with a specific element is vital. This is what allows you to do different things when the user clicks on different buttons, for example.



Then, we have the add event listener. This is a method that exists on any element object. It has two parameters; first is the event that you want to listen to. This is just a string like, press down, or key up, or click, or load, et cetera. The second parameter lets you specify a function that will get called when that event occurs. Finally, there's the function itself. This is just a regular old function. The common way to name these functions is using the word, "on" plus the name of the event. So, we have onClick. Other variations specify the element that's the source of the event, such as onStartButtonClick.

Again, this function will execute when the specified event occurs. When this function gets called, it will be past a single parameter. This is an object that contains specific information about the event that just occurred. For example, on a keyboard event, it will contain the specific key that was pressed, whether or not the Control, Alt, or Shift keys were pressed, and some other information. In a mouse event, it will contain the mouse's position on the screen, as well as info about what mouse buttons might have been pressed.

So, when you create your handler function, you should usually define that function with a single parameter. You can call that parameter whatever you want. But, I usually name it, event, so I remember what it is. Now, here I've created a named function, and passed the name of that function to the add event listener method, but, very often, you'll see it written like this. Here, we have another anonymous function defined directly within the parentheses of the add event listener method. This may seem very strange at first, but it's perfectly valid. And, you'll see it often enough that you should get used to it. Either way, the function is created, and it gets passed to the method. In most cases, it doesn't really matter whether you use a separate name function, like I did at first, or an anonymous function.

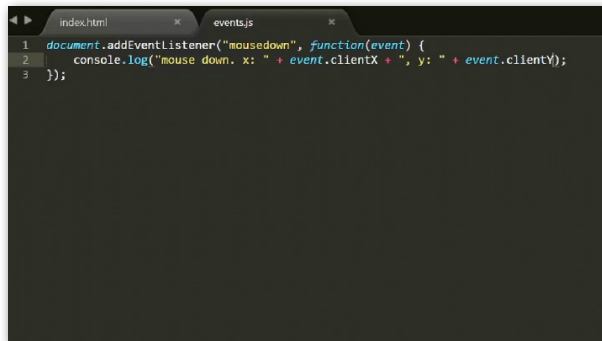


And, it's just a matter of what the developer who writes the code prefers. Some people think separate name functions are more readable. Others think anonymous functions are more concise. You should be able to understand either one when you see them, though.

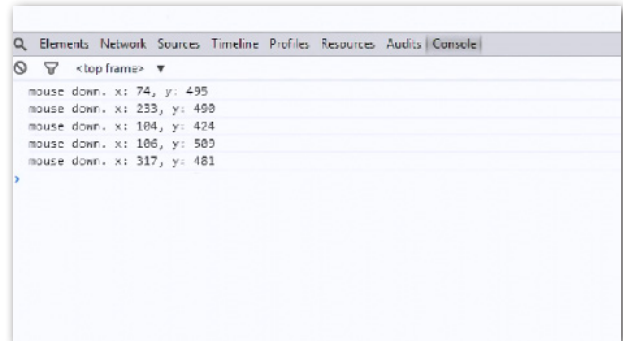
Now, let's deal with some real events. For the mouse, you'll mostly be dealing with events: mouse down, mouse up, click, and mouse move. And, for the keyboard: key down, and key up. For all of these, it should be pretty obvious what they represent. Mouse down and mouse up means that the user pressed and released the mouse button, while over a particular element. Click means that the user pressed and released the mouse button, while over the same element. Mouse moved means that the mouse moved, while it was on top of a particular element .

Key down and key up are pretty straightforward. But, know that if the user holds the key down, you'll probably get a continuous stream of key down events, as long as the key's held down. But, let's just write a quick program that listens for some of these events, and logs when one of them happens, along with a bit of info about that event. First mouse down; the element we're listening on is the current document. So no matter where the mouse is, we should get the event. We'll just log the string mouse down. Let's test it. Yup, that works. Now,

we can use the event parameter to get the current x and y position of the mouse. Again, event is an object. The mouse position will be on properties client(x), and client(y). So, we can say, mouse down with an x of event.Client(x) and y of event.client(y), concatenating that onto one huge string.

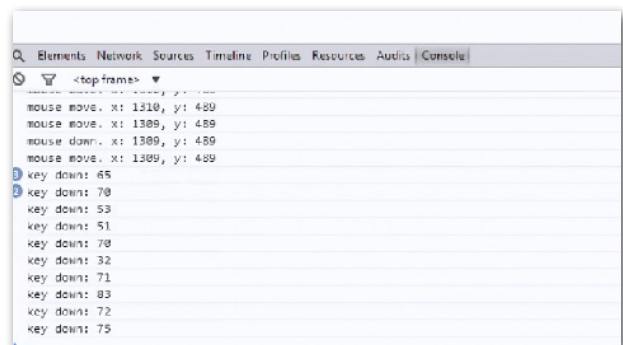


```
1 document.addEventListener("mousedown", function(event) {
2   console.log("mouse down. x: " + event.clientX + ", y: " + event.clientY);
3 });
```



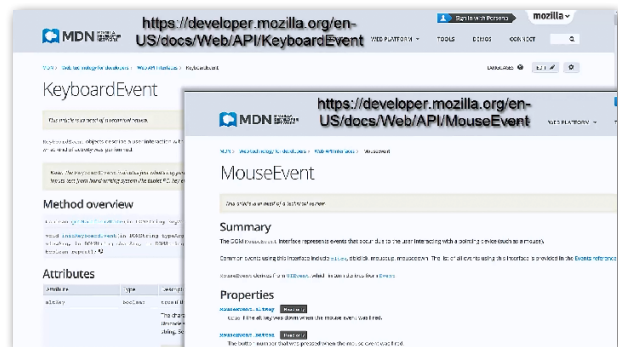
So, we test that, and sure enough, the mouse position is being logged with the click. Now, we can copy and paste that code, and change the event name to mouse move, and we test that. And, now, we're getting a mouse move event whenever the mouse moves, along with the current position; not too surprising. Let's try some keyboard stuff without a key down listener.

Here, I'm appending the key code property of the event object. When I test this, you can see that it logs a number for each key that I press. This number is an integer that represents which key was pressed. This number is generally the same as the ASCII code of the character of that key. So, we can use a special string method called, fromCharCode that will convert that number into the actual character that it represents. So, we say key down plus string fromCharCode(event.keyCode).



We'll try that. Here, you can see that this works for most of the alphanumeric keys, but can be unpredictable for other keys, such as left square bracket, right square bracket, and backslash.

Here's a link to the page that lists the different key codes, and what keys they refer to. You might want to keep that handy. And, that's about all there really is to handling events. Go ahead, and add listeners for mouse up, click, and key up, if you want to see those in action. It's good practice. You should also check out the documentation for keyboard events, and mouse events, at these two links. Look at some of the other properties on those events, and try to write some code that reads, and logs some of those properties. Or, if you're really ambitious, try writing an if, or if else statement that does different things based on different properties, such as logging one message if one key is pressed, and another message when another key is pressed.



ADVANCED DEBUGGING

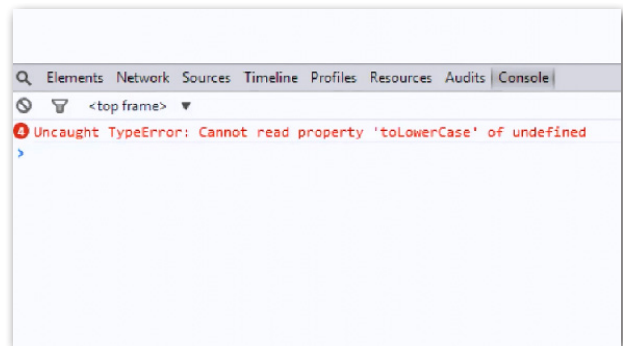
For demonstration purposes, I've set up a small program that has a bug in it. You can find this program as `debugging.js` along with a `debugging.html` file in this lesson's materials. Take a second to download it, and get it up in your editor. I'll quickly go through the program, to explain what I think it should do.



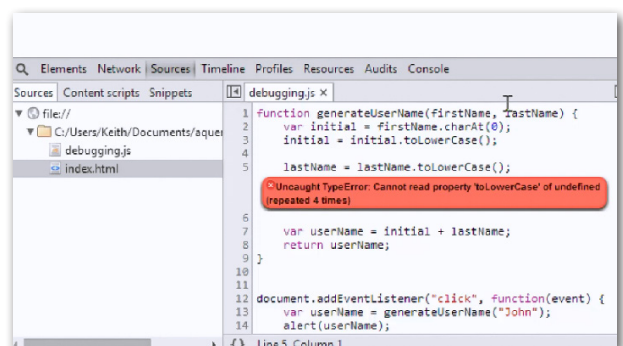
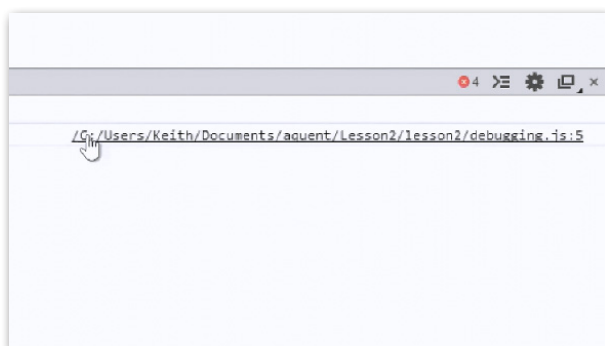
First, we have this generate Username function. It gets two parameters, first name, and last name, and should generate a username based on the first initial plus the last name, all in lowercase. It gets the first initial by using `charAt(0)` of the first name, and then converts that to lowercase. It then converts the last name to lowercase, and creates a username by concatenating the two. And finally, it returns that username.

Then, we have a click event listener on the document. In the handler function, we call generate username, passing in the name, John. And, then call up an alert box with the generator username. So, what I expect to happen, is that I click on the web page, a username is generated, and that's displayed in an alert.

Let's test it, and see what happens. So, here we are and I click. I click again. Keep clicking. Nothing's happening. Well, let's open up the console. Aha! We have some kind of an error, something about two lowercase, and undefined. Hmm. Well, that might tell you all you need to know, or it might not. Well, remember that over here on the right, it points us to the exact line, in the exact file where the problem is.



What I didn't show you before is that you can actually click on that bit of text. This will open up that file in the sources tab of the developer tools. And here, it even highlights the exact line, and again, shows you that error. In a lot of cases, this should be enough for you to see the issue.

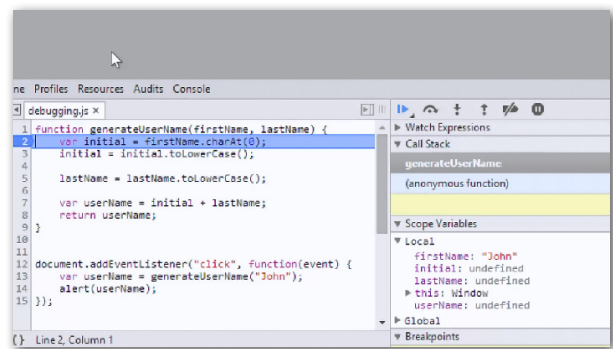


But, let's say this doesn't turn on any light bulbs, just yet. At least we know that the error is something within this function, here. What we can do is click over here in the margin, right at the first line of the function. Now, see that a blue arrow appears there. This is called a breakpoint. A breakpoint will actually stop the execution of JavaScript right at that point. Let's try it.

Refresh the page in the browser, and then, click again. Now, you see that the line with the breakpoint is totally highlighted in blue. This means that JavaScript has stopped right there. This line has not been executed yet. We also get a little notice of the fact that we're paused up here on the page.

Now, in the sources tab, we have a panel over here on the right with a bunch of information. The key area is this scope variable section. Here we can see all the local variables in the function where JavaScript has

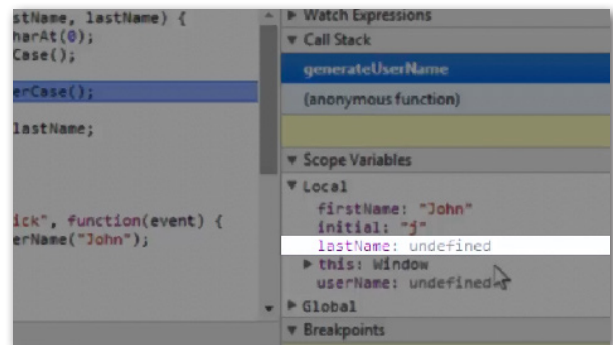
stopped. Remember, that local variables are variables that are only defined within that function. So, we can see first name, initial, last name, and username. And, most of these are undefined right now. Notice that we also have some controls up at the top here. This arrow here will unpause the script, and let the program continue to run, as usual. This one lets us step over the next line, executing it.



There's some others you can learn more about on your own. But, this step over button is the one you'll use most. If we click that, it executes that highlighted line, and the highlight moves on to the next line. Now, over here, we see that the local variable initial has changed from undefined into a capital J. That's good. It shows that this line executed correctly. Click that button again, and we move to the next line, and we see that initial has changed to a lowercase j. So far, so good. Click again, oh, crash! Alright, so yeah, definitely something wrong with this line, here.

Let's remove this breakpoint, and put a new one right on that troublemaker. We'll refresh the page again, and we'll click. And, we stop right here. We're just about to execute that line. But, I know that if we do, we're going to crash again. So, let's see what the state of the code is at this point. We know that we're about to call the two lowercase method of the string that's in the last name variable. And, we're going to assign the result of that back to last name.

What is the value of last name, at this point? Oh, it's undefined. Well, that makes sense. Undefined won't have a two-lowercase method. Only a string will. But, how did last name become undefined? Well, here's another neat feature. If you mouse over the parameters of the function here, you'll see the values that actually got passed in.



So, here we can see the first name is, John. That's good. And last name, undefined. So, let's look where this generate username function is called. Aha! We didn't pass it in a last name. I think we can fix this now. We'll go back to the code, and we'll add a last name. And, we'll save this, go back to the browser, remove this breakpoint, refresh the page, and click. And, there's our alert with the username, J Smith. Program debugged, ship it.

Now, many of your debugging sessions may be a lot more difficult than that. And, many will be a whole lot simpler. But, to be honest, what we just did is pretty representative of what goes on in the day of the life of a developer debugging code. Being good at debugging is a vital skill for any developer in any language. Go through these steps whenever your code isn't doing exactly what you want, and before long, you'll be a pro.

And, that brings us to the end of lesson two. And, now for your assignments.

The first assignment, as usual, is the quiz that you'll find on this course's web page. Take the quiz. It will enforce the principles that you're learning. And, it will help you gauge your progress. Now, the rest of the assignments are all programming problems. Do your best on them. But if anything goes wrong, or it doesn't work exactly like you expect it, use the debugging techniques we learned in this lesson to narrow down what the problem is, and fix it.

Assignment two: fill in the body of this function. The `containsString` function should return a value of `true`, if search string is part of main string, and `false` if it's not. For example, `containsString("independent", "depend")` should return `true`, because `depend` is part of `independent`. But, `containsString("independent", "indy")` should return `false`, because that string is not part of `independent`. As a hint, check into the index of method of a string.

Assignment three: create an array containing the 12 months of the year, January through December. Create a function called `getMonth` name that takes a single number as a parameter. That function should return the name of that month. For example, `getMonth(3)` should return `March`. Remember, that arrays are indexed starting with zero, but here, month one should be January. So, you'll have to account for that difference, somehow.

Assignment four: create a JavaScript object named `"user"` that has properties first name, last name, and email. Populate those with your own name and email. Then, write a function that takes this user object as a parameter, and logs that user's info to the console. Something like, `log user info user`, which should log something like, `John Smith: jsmith@company.com`.

And, the last assignment, number five, write a program that listens for a `keyup` event. In the handler for that event, check which key was released. If it was `Y`, log the word `"Yes"`. If it was `N`, log the word `"No"`. And, if it was any other key, log `"I don't understand."` When you're done with all that, I'll see you on Lesson 3, where we'll break free of the console, and actually start doing things with real web pages.

ASSIGNMENT #2:

```
function containsString(mainString, searchString) {  
    
}  
  
containsString("independent", "depend"); // true  
containsString("independent", "indy"); // false
```

AQUENT
GYMNASIUM

JavaScript Foundations

ASSIGNMENT #3:

Create an array containing "January" through "December".

Create a function called `getMonthName` that takes a single number as a parameter and returns the name of that month. For example:

```
getMonth(3); // will return "March"
```

Remember that arrays are indexed starting with 0, but here, month 1 should be January. So you'll have to account for that somehow.

AQUENT
GYMNASIUM

JavaScript Foundations

ASSIGNMENT #4:

Create a JavaScript object named `"user"` that has properties: `firstName`, `lastName`, `email`. Populate those with your own name and email.

Write a function that takes this user object as a parameter and logs that user's info to the console. Something like:

```
logUserInfo(user);
```

Which should log something like `"John Smith: jsmith@company.com"`

AQUENT
GYMNASIUM

JavaScript Foundations

ASSIGNMENT #5:

Write a program that listens for a `keyup` event.

In the handler, check which key was released.

If it was `"Y"`, log the word `"Yes"`.

If it was `"N"` log the word `"No"`.

If it was any other key, log, `"I don't understand."`

AQUENT
GYMNASIUM

JavaScript Foundations