| Arrow Functions |
| --- |

| <script>  let addStuff = function(x,y){  let sum = x + y;  return sum;  }  document.getElementById("show").innerHTML = addStuff(2,3);  </script>  Result is 5 |
| --- |

| Of course we could just return the expression itself  <script>  let addStuff = function(x,y){  return x + y;  }  document.getElementById("show").innerHTML = addStuff(2,3);  </script>  Result is 5 |
| --- |

| With arrow functions, remove the function keyword and replace it with the arrow symbol, however the arrow symbol appears on the right side of the parameter.  <script>  let addStuff = (x,y) => {  return x + y;  }  document.getElementById("show").innerHTML = addStuff(2,3);  </script>  Result is 5 |
| --- |

| Because the return expression is so short we could move it to the top so it is all in one line.  <script>  let addStuff = (x,y) => {return x + y;}  document.getElementById("show").innerHTML = addStuff(2,3);  </script>  Result is 5 |
| --- |

| Because the function is performing just one thing on just one line, we could remove the curly braces and the return keyword. Note, if the return keyword is necessary, then we must leave the curly braces.  <script>  let addStuff = (x,y) => x + y;  document.getElementById("show").innerHTML = addStuff(2,3);  </script>  Result is 5 |
| --- |

| If the function accepts just one parameter, we don’t need the parenthesis around it.  <script>  let addStuff = x => x \* x;  document.getElementById("show").innerHTML = addStuff(2);  </script>  Result is 4 |
| --- |

The real power of arrow functions appear in situations where we are working with some sort of collection

| In this scenario, the filter method is being called on an array. Filter works just like map. Filter will take a function that does something to the elements of the myArray array. In this case if the element leaves a remainder, we push that element into the new array called rem.  <script>  let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];  let rem = myArray.filter(function(remainder) {  return remainder % 2;  });  document.getElementById("show").innerHTML = rem;  </script>  Result is: 1,3,5,7,9 |
| --- |

| We could shorten the code using arrow functions like this:  <script>  let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];  let rem = myArray.filter((remainder) => {return remainder % 2;});  document.getElementById("show").innerHTML = rem;  </script>  Notice we got rid of the function keyword, placed the arrow after the parenthesis and moved the return statement up to the top line.  Result is: 1,3,5,7,9 |
| --- |

| But we could go further. We only have one parameter, so remove the parenthesis around remainder, and remove the return keyword and the curly braces. Result is the same.  <script>  let myArray = [1, 2, 3, 4, 5, 6, 7, 8, 9];  let rem = myArray.filter(remainder => remainder % 2);  document.getElementById("show").innerHTML = rem;  </script>  Result is: 1,3,5,7,9 |
| --- |

Map method

The map() method works with arrays.

It will iterate or loop through the original array, pull out each value and pass that value to a function. The result is a new array with the new values after being passed through the function you provided.

Lets apply the map function to this array called nums. We will use a JS built in function called pow()

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3];  </script>  </body>  </html> |
| --- |

Remember that the map() method returns an array, so lets define that array, basically this is the result of doing something with map()

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3];  **let nums\_squared;**  </script>  </body>  </html> |
| --- |

Apply the map method to nums array.

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3];  **let nums\_squared = nums.map();**  </script>  </body>  </html> |
| --- |

Now lets print out the result of nums\_squared

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3];  let nums\_squared = nums.map();  **document.getElementById("show").innerHTML = nums\_squared;**  </script>  </body>  </html> |
| --- |

This wont work, it produces an error, the map() mehtod expects a function, so lets provide one

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3];  let nums\_squared = nums.map( x => x);  document.getElementById("show").innerHTML = nums\_squared;  </script>  </body>  </html> |
| --- |

Here we are providing a function, which takes each element of the array and returns that element back to the new array, so basically we did nothing

Now lets provide a JS built-in function called pow()

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3];  let nums\_squared = nums.map( x => Math.pow(x, 2));  document.getElementById("show").innerHTML = nums\_squared;  </script>  </body>  </html> |
| --- |

Object destructuring

Using the same example, there is now a way to identify just part of an array, one, two or how many elements you wish to work with. It is possible to destructure just what you want and disregard the rest (of the array)

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3,4];  document.getElementById("show").innerHTML = "";  </script>  </body>  </html> |
| --- |

From the example above, what if we just wanted to work with the first two elements, so we need a way to identify just the 1 and 2, but forget about the rest of the elements in this array.

Well we will need first create a new array using the destructuring syntax.

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3,4];  let [a,b];  document.getElementById("show").innerHTML = "";  </script>  </body>  </html> |
| --- |

Now we can point the new array to the old one in order to transfer the values 1 and 2 into a and b

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3,4];  let [a,b] = nums;  document.getElementById("show").innerHTML = "";  </script>  </body>  </html> |
| --- |

Now we can use a and b as we please

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  let nums = [1,2,3,4];  let [a,b] = nums;  document.getElementById("show").innerHTML = a + b;  </script>  </body>  </html> |
| --- |

Destructuring objects

Usually though, destructuring objects is more common. Remember objects can contain nested name value pairs.

In this scenario we have *skills* that represent three name value pairs.

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  const skills = {  primary: 'JavaScript',  secondary: '.Net',  backup: 'Python'  };  document.getElementById("show").innerHTML = "";  </script>  </body>  </html> |
| --- |

What if we wanted to work with just the secondary value, we would have to create a new object and point it to the original *skills* object

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  const skills = {  primary: 'JavaScript',  secondary: '.Net',  backup: 'Python'  };  let { secondary } = skills;  document.getElementById("show").innerHTML = "";  </script>  </body>  </html> |
| --- |

Now we can use secondary directly in our code for display

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  const skills = {  primary: 'JavaScript',  secondary: '.Net',  backup: 'Python'  };  let { secondary } = skills;  document.getElementById("show").innerHTML = secondary;  </script>  </body>  </html> |
| --- |

Similarly we could do this

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  const skills = {  primary: 'JavaScript',  secondary: '.Net',  backup: 'Python'  };  let { secondary, primary } = skills;  document.getElementById("show").innerHTML = primary + " " + secondary;  </script>  </body>  </html> |
| --- |

Notice that the order does not matter

JavaScript objects can have different levels

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  const skills = {  primary: 'JavaScript',  secondary: '.Net',  backup: 'Python',  other:{  music:"Opera",  cooking:"French",  painting:"baroque"  }  }; let { secondary, primary } = skills;  document.getElementById("show").innerHTML = primary + " " + secondary;  </script>  </body>  </html> |
| --- |

If we wanted to get the value *French* for example we would have to de-structure and use int this way

| <html>  <body>  <div id="show" style="font-size:3em;"></div>  <script>  const skills = {  primary: 'JavaScript',  secondary: '.Net',  backup: 'Python',  other:{  music:"Opera",  cooking:"French",  painting:"baroque"  }  };   let { secondary, other } = skills;  document.getElementById("show").innerHTML = other.cooking + ", " +secondary;  </script>  </body>  </html> |
| --- |

Synchronous/asynchronous programming

The code below will work as expected, line 10 will execute before line 13.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. console.log('I was first...'); 11. }; 12. doFirst(); 13. console.log('OK I am second'); 14. </script> 15. </body> 16. </html> |
| --- |

However if we introduce a delay to the “I was first…” log line, then the output is reversed, this may or may not be a problem.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. setTimeout(function(){ 11. console.log('I was first...'); 12. }) 13. }; 14. doFirst(); 15. console.log('OK I am second'); 16. </script> 17. </body> 18. </html> |
| --- |

The problem is that the delay caused by *setTimeout* forces linen 11 to execute **after** line 15. This can happen in the real world when we hit APIs, database or other web services. There is always a delay when we work with stuff over some kind of network. JavaScript introduced asynchronous programming principles to address this concern. One way is to use promises. A promise is an object that is time based. It can only be resolved or rejected, meaning the object will either return what you asked of it or not.

We can begin to wrap the setTimeout function into a promise. We return a new promise object which will execute the setTimeout function. Notice that the Promise function has 2 parameters, if the setTimeout function is successful it will return its payload in the resolve parameter, otherwise an error object will be loaded into the reject parameter.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. console.log('I was first...'); 13. },2000) 14. }); 15. }; 16. doFirst(); 17. console.log('OK I am second'); 18. </script> 19. </body> 20. </html> |
| --- |

Lets use the parameter resolve to pass back as a promise return, the phrase we want to return. In other words if successful we want to pass this phrase “I was first…” to some other function to handle, that string is our data.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. doFirst(); 17. console.log('OK I am second'); 18. </script> 19. </body> 20. </html> |
| --- |

Promises can only be handled by then() functions. So when we call the doFirst() function on line 16, the code is executed inside of the promise object and then return something back to doFirst(). We would need to supply this then() method to handle that return.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. doFirst().then(); 17. console.log('OK I am second'); 18. </script> 19. </body> 20. </html> |
| --- |

That then() method will then need to have an inner function (callback) with a parameter to accept the return phrase and do something with it. The idea is that we now have control over what is printed when.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. doFirst().then(function(data){ 17. }); 18. //console.log('OK I am second'); 19. </script> 20. </body> 21. </html> |
| --- |

Now we can execute code based on this promise being completed first. Only when the promise is resolved, should we go ahead and run the second logical phrase.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. doFirst().then(function(data){ 17. console.log(data); 18. console.log('OK I am second'); 19. }); 20. </script> 21. </body> 22. </html> |
| --- |

Using Async/Await

We can create an async/await type of function to handle the same situation instead of using then(), first create a new function which we decorate as **async, see line 16.**  (you should remove the diFirst() call)

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. async function printMessages(){ 17. } 18. </script> 19. </body> 20. </html> |
| --- |

Within the printMessage body we can call the doFirst() function, but we decorate the call with an **await** statement. Also remember that doFirst() returns something, a statement, so pass that on to a named variable (line 17).

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. async function printMessages(){ 17. let firstMessage = await doFirst(); 18. } 19. </script> 20. </body> 21. </html> |
| --- |

Now we call printMessages() instead of calling doFirst() directly

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. async function printMessages(){ 17. let firstMessage = await doFirst(); 18. } 19. printMessages(); 20. </script> 21. </body> 22. </html> |
| --- |

The code will not do anything at this point, we still have to call the log function

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. async function printMessages(){ 17. let firstMessage = await doFirst(); 18. console.log(firstMessage); 19. } 20. printMessages(); 21. </script> 22. </body> 23. </html> |
| --- |

If you run the code now, after 2 seconds the *I was first…* message will appear in the console window. So now we can complete the application by printing the second statement, the point is now we control which statement gets printed first, then second etc.

| 1. <!DOCTYPE html> 2. <html> 3. <head> 4. <meta charset="utf-8"> 5. </head> 6. <body> 7. <div id="show" style="font-size:3em;"></div> 8. <script> 9. const doFirst = function() { 10. return new Promise(function(resolve, reject){ 11. setTimeout(function(){ 12. resolve('I was first...'); 13. },2000) 14. }); 15. }; 16. async function printMessages(){ 17. let firstMessage = await doFirst(); 18. console.log(firstMessage); 19. console.log('OK I am second'); 20. } 21. printMessages(); 22. </script> 23. </body> 24. </html> |
| --- |