|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385318) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385318) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385318) |
|  | |
| 7. Arrays  7.1. Arrays: Elements and Indexes  *Copyrighted Material - subject to fair use exception*  An array is an **ordered**collection of elements.  We can create arrays by using an array literal which is a just list of elements enclosed in square brackets.  >>> var myArray = [ "Alice", 89, true, 500.2 ];  **Array elements need not all have the same type**.  Each element has a position in the array, known as its **index.** JavaScript uses **zero-based indexing**.  We can access individual elements of an array by their index as follows:  >>> myArray[0]  "Alice"  >>> myArray[1]  89  >>> myArray[2]  true  >>> myArray[3]  500.2  >>> myArray[4]  undefined  Array elements may themselves be arrays:  >>> var myGrades = [ 86, 90, [ 100, 85 ], 99 ];  >>> myGrades[0]  86  >>> myGrades[1]  90  >>> myGrades[2]  [100, 85]  >>> myGrades[3]  99  >>> myGrades[4]  undefined  **To access an element in the nested array, we use the square brackets twice: [ ][ ]**  >>> myGrades[2][0]  100   >>> myGrades[2][1]  85  JavaScript arrays are dynamic: they grow or shrink as needed.  >>> var myArray = [ ];    // we can start with an empty array  >>> myArray  [ ]  >>> myArray[0] = 2;  // add a numeric element at index position 0.  >>> myArray  [ 2 ]  **JavaScript arrays may be sparse:** the elements don’t need to  have contiguous indexes and there may be gaps.  >>> myArray[3] = 1; // add a numeric element at index position 3  >>> myArray  [2, undefined, undefined, 1]  We can also create sparse arrays by including extra commas inside the array literal:  >>> var myArray = [ 'zero',  ,  , 'three' ];  >>> myArray  ["zero", undefined, undefined, "three"]  7.2. Array Length  *Copyrighted Material - subject to fair use exception*  Arrays have a length property.  For nonsparse arrays, this property specifies the number of elements in the array.  >>> var myNumbers = [ 'zero',  'one', 'two' ];  >>> myNumbers.length  3  >>> var myGrades = [ 86, 90, [ 100, 85 ], 99 ];  >>> myGrades.length  4  For sparse arrays, the length is one more than the largest index.  >>> sparseArray = [ ];  // start with an empty array  >>> sparseArray[ 100 ] = 'Alice';   // add an element at index position 100  >>> sparseArray.length  101  >>> var myArray = [ 'zero',  ,  , 'three' ];  >>> myArray.length;  4  You can also create an array by using the array constructor and specifying the length:  >>> var myArray = new Array(5);  >>> myArray.length  5  **The length is not an upper bound.** If you add an element with an index that is greater than or equal to the current length, the length will increase to contain the new element. No out of bounds error is generated.  You can assign a new value to the length. Making the length larger does NOT allocate more space for the array.  >>> var myArray = [ 'zero',  ,  , 'three' ];  >>> myArray.length;  4  >>> myArray.length = 6;  // assign a new larger value to the length  >>> myArray;  ["zero", undefined, undefined, "three", undefined, undefined]  **Making the length smaller will truncate the array**.  The elements with an index greater than or equal to the new length will be deleted:  >>> myArray.length = 1;  // assign a new smaller value to the length  >>> myArray  ["zero" ]  7.3. Rearranging an Array  *Copyrighted Material - subject to fair use exception*  We can rearrange the elements in an array.  We can **reverse**an array:  >>> var students= [ 'Alice', 'Charlie', 'Bob' ];  >>> students.reverse();  >>> students  [ "Bob", "Charlie", "Alice" ]  **The array is reversed in place.  The elements are rearranged in the original array.**  We can **sort**an array:  >>> students.sort();  >>> students  [ "Alice", "Bob", "Charlie" ]  **The array is sorted in place.  The elements are sorted in the original array.  The default sort is alphabetical, even if the elements are not strings.**  >>> var myNumbers = [ 100, 6, 30 ];  >>> myNumbers.sort();  [ 100, 30, 6 ]  In the alphabetical sort, the first character of the elements is compared and 1 comes before 3 which comes before 6.  We can specify a different sort by passing an optional **comparison function** to the sort method.  The elements are then sorted according to the return value of the comparison function.   The comparison function looks like this:  function compareFunction(a, b) {     // perform our own comparison of a and b and return one of the following:    // a number less than 0 if a is less than b in our sort    // 0 if a is a is equal to b  in our sort    // a number greater than zero if a is greater than b in our sort  }  The sort method will call compareFunction on two elements a and b of the array at a time:  If compareFunction(a, b) is less than 0 -> a comes before b.  If compareFunction(a, b) is 0 -> leave a and b unchanged with respect to each other.  If compareFunction(a, b) is greater than 0 -> b comes before a.  So to compare numbers instead of strings, we have several options for the comparison function.  One possibility is to define it as follows:  function compareNumbers(a, b) {      if (a < b) {          return -1;      } else if (a > b) {          return 1;      } else {          return 0;      }  }  Another possibility is to take advantage of the fact that for numbers, a - b is negative when a < b, 0 when a and b are equal and greater than 0 when a > b.  So we can write:  function compareNumbers(a, b) {      return a - b;  }  To sort the numbers in the array above, we can then write:  var myNumbers = [ 100, 6, 30 ];  myNumbers.sort( **compareNumbers** );  [ 6, 30, 100 ]  We can also use an **anonymous function expression** as the parameter to sort and write:  myNumbers.sort(**function (a, b) {**  **if (a < b) {**  **return - 1;**  **} else if (a > b) {**  **return 1;**  **} else {**  **return 0;**  **}**  **}**);  Or:  myNumbers.sort(**function (a, b) {**  **return a - b;**  **}**);    7.4. Splicing an Array  *Copyrighted Material - subject to fair use exception*  The splice method lets us perform complicated surgery on an array.  It can delete some elements and replace them with other elements in one step.  The first argument to splice is an index in the array. The second argument is the number of elements to remove.  The elements are removed from the given index on.  The removed elements are also returned by the method.  Any additional arguments get inserted into the array at that point:  >>> var colors = [ 'red', 'white', 'blue' ];  >>> colors.splice(1, 1, 'green', 'yellow');  // start at index 1, remove 1 element and return it; insert 'green' and 'yellow'.  ["white"]  >>> colors  ["red", "green", "yellow", "blue"]  >>> colors.splice(2, 2, 'magenta');  // start at index 2, remove 2 elements and return them; insert ‘magenta’.  ["yellow", "blue"]  >>> colors  ["red", "green", "magenta"]  >>> colors.splice(2, 0, 'cyan');  // start at index 2, remove 0 elements and return them; insert 'cyan'.  []  >>> colors  ["red", "green", "cyan", "magenta"]  7.5. The Array as a Stack  *Copyrighted Material - subject to fair use exception*  The push(), pop(), shift() and unshift() methods allow us to use an array as a first in last out stack.  **The push() method adds the given elements to the end of the array and returns the new length of the array.**  >>> var colors = [ 'red', 'white', 'blue' ];  >>> colors.push('green', 'yellow');  5  >>> colors  ["red", "white", "blue", "green", "yellow"]  **The** **pop() method removes the last element** **from an array and returns that element**.  >>> colors.pop();  "yellow"  >>> colors  // the original array is mutated  ["red", "white", "blue", "green"]  >>> colors.pop();  "green"  >>> colors  // the original array is mutated  ["red", "white", "blue"]  If you call pop() on an empty array, it returns an undefined value.  >>> [ ].pop();  undefined  The **unshift()** and **shift()** methods are similar to push() and pop() but they operate on the **beginning**of the array instead of at its end.  As a result, **they are usually slower than push() and pop().**  **The unshift() method inserts one or more elements to the beginning of an array and returns the new length of the array.**  >>> var colors = ['red', 'white', 'blue'];  >>> colors.unshift('green', 'yellow');  5  >>> colors  ["green", "yellow", "red", "white", "blue"]  **The shift() method removes the first element from an array and returns that element.**  >>> colors.shift();  "green"  >>> colors.shift();  "yellow"  If you call shift() on an empty array, it returns an undefined value.  >>> [ ].shift();  undefined  7.6. Iterating and Mapping  *Copyrighted Material - subject to fair use exception*  **The forEach() method iterates over an array and executes a given function once for each element.**  The function you supply is passed three arguments: the value of the array element, the index of the array element, and the array itself.  If you only need the value and index, you can write a function with only two parameters — the additional arguments will be ignored:  function showElement( value, index) {      console.log (index + '--->' +  value);  }  var myArray = [ 'red', 'white', 'blue' ];  **myArray.forEach(showElement);**  0--->red  1--->white  2--->blue  You can also use forEach() to modify the array you are invoking forEach() on.  function extraPoints( value, index, array) {      array[index] = value + 5;  }  var grades = [ 85, 94, 82, 90 ];  grades.forEach(extraPoints);   // add 5 to each element of the array  console.log(grades);    // the array is modified.  [ 90, 99, 87, 95 ]  **The map()** method is similar to forEach() but it **returns a new array with the results of calling the given function on each element in this array**.  In this case, the given function should have a return value.  function square (number) {      return number \* number;  }  var myArray = [ 1, 2, 3, 4, 5 ];  var mySquares = myArray.map(square);  console.log(mySquares);  [ 1, 4, 9, 16, 25 ]  console.log(myArray);  // the original array is unchanged,  [ 1, 2, 3, 4, 5 ]  forEach() and map() were introduced in ECMAScript 5.  7.7. Filtering and Testing  *Copyrighted Material - subject to fair use exception*  **The filter() method creates a new array with all elements that pass the test implemented by the given function.**  Here's an example:  function isA(grade) {      if (grade >= 90) {          return true;      } else {          return false;      }  }  var classGrades = [ 60, 90, 87, 100, 86 ];  var aGrades = classGrades.filter(isA);  console.log(aGrades);  //aGrades will have the elements of classGrades that are >= 90.  [ 90, 100 ]  console.log(classGrades); // The original array is unchanged.  [ 60, 90, 87, 100, 86 ]  If we need to pass an additional parameter to the filtering function, we can do that using 'this' as follows:  var **myA** = { threshold: 90 }  // we create an object that will hold the threshold for an A.  var aGrades = classGrades.filter(isA, **myA**); // **we pass the object as a filter argument**.  And now we can use the object inside the filtering function as follows:  function isA(grade) {      if (grade >= **this.threshold**) {   // 'this' refers to the myA object.          return true;      } else {          return false;      }  }  Note that we can also use an **anonymous function expression** as the argument to filter as follows:  var classGrades = [ 60, 90, 87, 100, 86 ];  var threshold = 90;  var aGrades = classGrades.filter(**function (grade) {**  **if (grade >= threshold) {**  **// the anonymous function has access to the variable threshold**  **return true;**  **} else {**  **return false;**  **}**  **}**  );  console.log(aGrades);  [ 90, 100 ]  **The every() method tests whether all elements in the array pass the test implemented by the given function.**  classGrades.every(isA);  false  aGrades.every(isA);  true  **The some() method tests whether some element in the array passes the test implemented by the given function.**  classGrades.some(isA);  true  The filter(), every() and some() methods were introduced in ECMAScript 5.  7.8. What else can we do with an array?  *Copyrighted Material - subject to fair use exception*  The **indexOf()** method returns the first index at which a given element can be found in the array, or -1 if it is not present.  The **lastIndexOf()**method returns the last index at which a given element can be found in the array, or -1 if it is not present.  These two methods were introduced in ECMAScript 5.  >>> var myArray = [ 100, 85, 90, 100  ];  >>> myArray.indexOf(100);  0  >>> myArray.lastIndexOf(100);  3  >>> myArray.indexOf(100, 1);  // start searching from index 1 on.  3  >>> myArray.lastIndexOf(100, 1);  // start searching backward from index 1 on.  0  >>> myArray.indexOf(65)  -1  **The join() method joins all elements of an array into a string and returns that string**. You can specify a separator.  If you don’t, the default separator is a comma.  The original array is unchanged.  >>> var colors = [ 'red', 'white', 'blue' ];  >>> colors.join();  // no separator is specified.  It will default to a comma.  "red,white,blue"  >>> colors  // the original array is unchanged.  [ "red", "white", "blue" ]  >>> colors.join(' and ');  // we specify ‘ and ‘ with extra spaces as the separator.  "red and white and blue"  **The slice() method returns a shallow (one level deep) copy of a portion of an array** **from a given index position up to but not including a second index position**.  We’ll examine the distinction between shallow and deep copy in the next section.  >>> var myArray = [ 'zero', 'one', 'two', 'three' ];  >>> myArray.slice(1, 3)  [ "one", "two" ]  We get a copy of a portion of the array from a 1 up to but not including 3.  The original array in unchanged.  >>> myArray  [ "zero", "one", "two", "three" ]  >>> myArray.slice(1)  [ "one", "two", "three" ]  When the end index is not specified, the slice goes to the end of the array.  >>> myArray.slice()  [ "zero", "one", "two", "three" ]  When neither index is specified, we get a copy of the whole array.  When we specify a negative index, we start counting from the end of the array.  >>> myArray.slice(-3);  [ "one", "two", "three" ]  **The concat() method returns  a new array containing a shallow copy of the array on which it  is invoked followed by each of the arguments to concat().** If an argument is an array, then each of its elements is concatenated individually.  If one or more of these elements is an array (nested array), then it is concatenated as an array.  >>> var grades = [ 100, 86 ];  >>> var homework = [ 95, 98 ];  >>> var newGrades = grades.concat(homework, 90);  Each of the elements of the array homework is concatenated individually.  [ 100, 86, 95, 98, 90 ]  >>> grades;  // The original array is unchanged  [ 100, 86 ]  In the example below, the argument to concat() is an array, [ homework, 90 ] containing an array, homework:  >>> newGrades = grades.concat(**[ homework, 90 ]**);  [ 100, 86,**[95, 98], 90** ]  7.9. Shallow vs Deep  *Copyrighted Material - subject to fair use exception*  Assignment with an **= on objects does not make a copy**. Instead, the assignment makes the two variables point to the same object in memory.  Consider two array variables alice and bob.  We create these variables as follows to hold Alice’s and Bob’s grades on various assignments.  >>> var alice = [ 98,87,100 ];  >>> var bob = alice;  >>> alice  [ 98, 87, 100 ]  >>> bob  [ 98, 87, 100 ]  What happens if we now change one element in the array alice?  >>> alice[1] = 0;  >>> alice;  [98, 0, 100]  >>> bob  [98, 0, 100]  What if we wanted bob to have a**different copy**of the array, one that initially has the same values as alice but that is not affected by future changes to alice.  If the array we are copying **is not nested**, the following **slice assignment** will work.  >>> bob = alice.slice();  // slice from 0 till the end of the array  >>> bob  [ 98, 0, 100 ]   >>> alice[1]=5;  >>> alice  [ 98, 5, 100 ]  >>> bob  [ 98, 0, 100 ]  However with nested arrays the slice assignment will NOT work.  It is a **shallow copy.**  >>> var charlie = [ 100, 98, [ 85, 90 ], 85 ];  >>> var diana=charlie.slice();  >>> diana  [ 100, 98, [ 85, 90 ], 85 ]  >>> charlie[1] = 0;  >>> charlie  [ 100, 0, [85, 90], 85 ]  >>> diana  [ 100, 98, [ 85, 90 ], 85 ]  >>> charlie[2][1] = 0;  >>> charlie  [ 100, 0, [ 85, 0 ], 85 ]  >>> diana  [ 100, 98, [ 85, 0 ], 85 ]  We'll later see how to use JSON to get around this problem.  7.10. Array or Object?  *Copyright (c) 2014, Rula Khayrallah*  Arrays are objects.   The typeof operator, when used on an array, returns "object".  >>> var grades = [ 86, 90, 99 ];  >>> typeof grades  "object"  However there are times when we need to figure out whether a given object is an array or not.  The function Array.isArray() introduced in EcmaScript 5 allows us to do just that.  Here's how we use it:  >>> Array.isArray(grades)  true   >>> Array.isArray([])  true   >>> Array.isArray({})  false  >>> Array.isArray({make: 'Honda', mileage: 0})  false  Here's an example where Array.isArray() comes in handy.  Let's say that we have a nested array of grades, grouped by components:  a nested array contains homework grades, another nested array contains  midterm grades ([80, 100]) and the final exam grade is just an element (90).  var grades = [[100, 90, 95, 85, 100], [80, 100], 90];  Our task is to calculate the average of all these grades, assuming equal weight.  The following code traverses the array recursively to add and count the grades.  Note that we take advantage of the forEach() method as well as the Array.isArray function:   // Compute the average of a nested list of grades  'use strict';  var grades = [[100, 90, 95, 85, 100], [80, 100], 90];  var sum = 0;  var count = 0;    function add(item) {      // add array item or its nested items recursively      if (**Array.isArra**y(item)) {          item.**forEach**(add);      } else if (typeof item === "number") {          sum += item;          count += 1;      }  }    add(grades)  if (count) {      console.log('Average: ', sum / count)    Average: 92.5 |  |