***toUpperCase()* and *toLowerCase()* Methods**

**toLowerCase()** **Method**

* The **toLowerCase()** method converts a string to lowercase letters
* The **toLowerCase()** method does not change the original string.

string.toLowerCase()

|  |  |
| --- | --- |
| let text = "Hello World!";  let result = text.toLowerCase();  console.log(result); |  |

**toUpperCase()** **Method**

* The **toUpperCase()** method converts a string to uppercase letters.
* The **toUpperCase()** method does not change the original string.

string.toUpperCase()

|  |  |
| --- | --- |
| let text = "Hello World!";  let result = text.toUpperCase();  console.log(result); |  |

***length()* Method**

The **length()** property sets or returns the number of elements in an array.

array.length()

|  |  |
| --- | --- |
| const fruits = ["Banana", "Orange", "Apple", "Mango"];  let length = fruits.length;  console.log(length); |  |

***reverse()* Method**

* The **reverse()** method reverses the order of the elements in an array.
* The **reverse()** method overwrites the original array.

array.reverse()

|  |  |
| --- | --- |
| const fruits = ["Banana", "Orange", "Apple", "Mango"];  console.log(fruits.reverse()) |  |

***replace()* Method**

* The **replace()** method searches a string for a value or a regular expression. Only replaces the first string it finds by default.
* The **replace()** method returns a new string with the value(s) replaced.
* The **replace()** method does not change the original string.
* The **replace()** is case sensitive

string.replace(searchValue, newValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| searchValue | Required.  The value, or regular expression, to search for. |
| newValue | Required.  The new value (to replace with). |

|  |  |
| --- | --- |
| let text = "Visit Microsoft!";  let result = text.replace("Microsoft", "W3Schools");  console.log(result); |  |

If more than one string matches you have to use a global replacement:

|  |  |
| --- | --- |
| let text = "Mr Blue has a blue house and a blue car";  let result = text.replace(/blue/g, "red");  console.log(result); |  |

***slice()* Method**

* The **slice()** method **returns selected elements** in an array, as a **new array**.
* The **slice()** method selects from a given start, up to a (not inclusive) given end.
* The **slice()** method does not change the original array.

array.slice(start, end)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| start | Optional.  Start position.  Default is 0.  Negative numbers select from the end of the array. |
| end | Optional.  End position.  Default is last element.  Negative numbers select from the end of the array. |

|  |  |
| --- | --- |
| const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  const citrus = fruits.slice(1, 3); |  |
| const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  const myBest = fruits.slice(-3, -1); |  |

***join()* Method**

* The **join()** method returns an array as a string, separated by the specified separator.
* The **join()** method does not change the original array.
* Any separator can be specified. The default is comma (**,**).

array.join(separator)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| separator | Optional.  The separator to be used.  Default is a comma. |

|  |  |
| --- | --- |
| const fruits = ["Banana", "Orange", "Apple", "Mango"];  let text = fruits.join();  console.log(text);  text = fruits.join('');  console.log(text);  text = fruits.join(' ');  console.log(text); |  |

***map()* method**

**map()** **creates a new array** from **calling a function for every array element**.

* It calls a function once for each element in an array.
* It does not execute the function for empty elements.
* It does not change the original array.
* It returns an array with the results of a functions for each array element.

array.map(function(element, index, array){  }, this);

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function() | Required.  A function to be run for each array element. |
| currentValue | Required.  The value of the current element. |
| index | Optional.  The index of the current element. |
| arr | Optional.  The array of the current element. |
| thisValue | Optional.  Default value **undefined**.  A value passed to the function to be used as its **this** value. |

The example below multiplies every element in the original array by 4. The output will be **[650, 440, 120, 40]**

const numbers = [65, 44, 12, 4];

const newArr = numbers.map(myFunction)

function myFunction(num) {

  return num \* 10;

}

console.log(newArr)

For example, you may have an array of objects that stores **firstName** and **lastName** values as below.

You can use **map()** method to iterate over the array and join the values of **firstName** and **lastName** as below. This would return **['Malcom Reynolds', 'Kaylee Frye', 'Jayne Cobb']**.

const persons = [

  {firstName : "Malcom", lastName: "Reynolds"},

  {firstName : "Kaylee", lastName: "Frye"},

  {firstName : "Jayne", lastName: "Cobb"}

];

persons.map(getFullName);

function getFullName(element) {

  return [element.firstName,element.lastName].join(" ");

}

See another example below, where the **map()** method is used to create another array with just the names.

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  const itemNames = items.map((item) => {    return item.name  })  console.log(itemNames) |  |

***find()* method**

The **find()** method returns the value of the first element that passes a test.

* It executes a function for each array element.
* It returns **undefined** if no elements are found.
* It does not execute the function for empty elements.
* It does not change the original array.

array.find(function(currentValue, index, arr),thisValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function() | Required. A function to run for each array element. |
| currentValue | Required. The value of the current element. |
| index | Optional. The index of the current element. |
| arr | Optional. The array of the current element. |
| thisValue | Optional.  Default **undefined**. A value passed to the function as its **this** value. |

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  const foundItem = items.find((item) => {    return item.name === 'Book'  })  console.log(foundItem) |  |

See the example below, where we it is used to find the “Book” element in the array:

***findIndex()* method**

* The **findIndex()** method executes a function for each array element.
* The **findIndex()** method **returns the index (position) of the first element that passes a test**.
* The **findIndex()** method returns -1 if no match is found.
* The **findIndex()** method does not execute the function for empty array elements.
* The **findIndex()** method does not change the original array.

array.findIndex(function(currentValue, index, arr), thisValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function() | Required.  A function to be run for each array element. |
| currentValue | Required.  The value of the current element. |
| index | Optional.  The index of the current element. |
| arr | Optional.  The array of the current element. |
| thisValue | Optional.  Default **undefined**.  A value passed to the function as its **this** value. |

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  const foundItemIndex = items.findIndex((item) => {    return item.name === 'Book'  })  console.log(foundItemIndex) |  |

***forEach()* method**

* The **forEach()** method executes a provided function once for each array element.
* **forEach()** return **undefined**.
* It is not executed for empty elements.

array.forEach(function(currentValue, index, arr), thisValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function() | Required.  A function to be run for each array element. |
| currentValue | Required.  The value of the current element. |
| index | Optional.  The index of the current element. |
| arr | Optional.  The array of the current element. |
| thisValue | Optional.  Default **undefined**.  A value passed to the function as its **this** value. |

For example, in the example below, for each item in the items array, it will log the price of the element.

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  items.forEach((item) => {    console.log(item.price)  }) |  |

**Differences between map() and forEach()**

These two methods are used to **iterate on an array**, more technically they invoke the provided callback function for every element of an array.

If you want the benefits of the **return value** or **you don’t want to change the original array** then proceed with the **map()**. As you can see below, we have stored the results in a new array and the original array remains not changed.

|  |  |
| --- | --- |
| **map()** | |
| //an array of numbers  let numberArray = [1, 2, 3, 4, 5];  //output the square of each number  let newArray = numberArray.map((num) => num \* num)  //view the results  console.log(numberArray);  console.log(newArray); |  |

Otherwise if you are just interested to **iterate** or perform the **non-transformation process on the array**, **forEach()** could be the better choice.

|  |  |
| --- | --- |
| **forEach()** | |
| //an array of numbers  let numberArray = [1, 2, 3, 4, 5];  //output the square of each number  let newArray = numberArray.forEach(num =>    (num \* num)  );  //the array hasn’t changed  console.log(numberArray);  console.log(newArray); |  |

***some()* method**

The **some()** method checks if **any array elements pass a test** (provided as a callback function).

* executes the callback function once for each array element.
* returns **true** (and stops) if the function returns **true** for any array element.
* returns **false** if the function returns **false** for all of the array elements.
* does not execute the function for empty array elements.
* does not change the original array.

array.some(function(value, index, arr), this)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function | Required. A function to run for each array element. |
| value | Required.  The value of the current element. |
| index | Optional.  The index of the current element. |
| arr | Optional.  The array the current element belongs to. |
| this | Optional.  Default **undefined**.  A value passed to the function to be used as its "**this**" value. |

For example, the code below checks if any of the items has a price below 100, which it has, so it returns **true**.

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  const hasCheapItems = items.some((item) => {    return item.price <= 100  })  console.log(hasCheapItems) |  |

***every()* method**

* The **every()** method executes a function for each array element.
* It returns **true** if the function returns **true** for all elements.
* It returns **false** if the function returns **false** for one element.
* It does not execute the function for empty elements.
* It does not change the original array

array.every(function(currentValue, index, arr), thisValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function | Required. A function to run for each array element. |
| value | Required.  The value of the current element. |
| index | Optional.  The index of the current element. |
| arr | Optional.  The array the current element belongs to. |
| this | Optional.  Default **undefined**.  A value passed to the function to be used as its "**this**" value. |

For example, not every element passes the test given by the anonymous function, so it will return **false**.

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  const hasCheapItems = items.every((item) => {    return item.price <= 100  })  console.log(hasCheapItems) |  |

***reduce()* method**

* The **reduce()** method executes a reducer function for array element.
* **It returns a single value: the function's accumulated result.**
* It does not execute the function for empty array elements.
* It does not change the original array.

array.reduce(function(total, currentValue, currentIndex, arr), initialValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function | Required.  A function to be run for each element in the array. |
| total | Required.  The *initialValue*, or the previously returned value of the function. |
| curretValue | Required.  The value of the current element. |
| currentIndex | Optional.  The index of the current element. |
| arr | Optional.  The array the current element belongs to. |
| InitialValue | Optional.  A value to be passed to the function as the initial value. |

In the example below, the reduce method uses the callback function to subtract the numbers in the array, starting at the first value (175). So, the return value will be 100 (175-50-25=100).

|  |  |
| --- | --- |
| const numbers = [175, 50, 25];  newArr = numbers.reduce(myFunc);  function myFunc(total, num) {    return total - num;  }  console.log(newArr) |  |

***filter()* method**

* The **filter()** method **creates a new array** filled with elements that pass a test provided by a function.
* It does not execute the function for empty elements.
* It does not change the original array.

array.filter(function(currentValue, index, arr), thisValue)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| function() | Required.  A function to be run for each array element. |
| currentValue | Required.  The value of the current element. |
| index | Optional.  The index of the current element. |
| arr | Optional.  The array of the current element. |
| thisValue | Optional.  Default **undefined**.  A value passed to the function as its **this** value. |

The following array returns an array with the numbers that pass the criteria > 18.

|  |  |
| --- | --- |
| const ages = [32, 33, 16, 40];  const filteredAges = ages.filter(checkAdult);  function checkAdult(age) {    return age >= 18;  }  console.log(filteredAges) |  |

See another example, where the **filter()** method is used to create a new array only with the items which the price is below 100.

|  |  |
| --- | --- |
| const items = [    {name: 'Bike',      price: 100  },    {name:'TV',         price: 200  },    {name: 'Album',     price: 10   },    {name: 'Book',      price: 5    },    {name: 'Phone',     price: 500  },    {name: 'Computer',  price: 1000 },    {name: 'Keyboard',  price: 25   }  ]  const filteredItems = items.filter((item) => {    return item.price <= 100  })  console.log(items)  console.log(filteredItems) |  |

***includes()* method**

* The **includes()** method returns true if an array contains a specified value.
* It returns **false** if the value is not found.
* It is case sensitive.

array.includes(element, start)

For example, the code below checks if the number 5 is included in the array. And it is, so it will log **true**.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| element | Required.  The value to search for. |
| start | Optional.  Start position. Default is 0. |

|  |  |
| --- | --- |
| const fruits = [1,2,3,4,5]  const isNumberIncluded = fruits.includes(5);  console.log(isNumberIncluded) |  |

Using **includes()** with conditional statements

One of the useful ways to use includes method is in if statements, where you want something to happen if an array or string contains a certain element.

const numbers = [1,2,3,4,5]

if (numbers.includes(2)) {

console.log(`the number is included`)

}

**Useful case of String *includes()* method**

We can combine includes with other methods such as **filter()** to, for example, create an array with elements that include a certain letter.

|  |  |
| --- | --- |
| const products = [    {      title: 'high-back bench',      company: 'ikea',    },    {      title: 'albany table',      company: 'marcos',    },    {      title: 'accent chair',      company: 'caressa',    },    {      title: 'wooden table',      company: 'ikea',    }  ]  const filteredProducts = products.filter((product) => product.title.includes('ac'))  console.log(filteredProducts); |  |

***startsWith()* method**

* The **startsWith()** method returns **true** if a string starts with a specified string. Otherwise it returns **false**.
* The **startsWith()** method is **case** **sensitive**.

string.startsWith(searchValue, start)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| searchValue | Required.  The string to search for. |
| start | Optional.  Start position. Default is 0. |

|  |  |
| --- | --- |
| const person = 'Peter Smith';  const employee = '23456-EMP-PETER-SMITH';  console.log(person.startsWith('Pet')); |  |

You can also specify the index that you want it to start reading as a second parameter:

|  |  |
| --- | --- |
| const person = 'Peter Smith';  const employee = '23456-EMP-PETER-SMITH';  const manager = '23456-MAN-JOHN-DOE';  console.log(employee.startsWith('EMP',6)); |  |

***endsWith()* method**

* The **endsWith()** method returns **true** if a string ends with a specified string. Otherwise it returns **false**.
* The **endsWith()** method is **case** **sensitive**.

string.endsWith(searchValue, length)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| searchValue | Required.  The string to search for. |
| length | Optional.  The length of the string to search.  Default value is the length of the string. |

|  |  |
| --- | --- |
| const manager = '23456-MAN-JOHN-DOE';  console.log(manager.endsWith('DOE')); |  |

You can also specify The length of the string to search as a second parameter:

|  |  |
| --- | --- |
| const manager = '23456-MAN-JOHN-DOE';  console.log(manager.endsWith('MAN',9)); |  |

***repeat()* method**

The **repeat()** method returns a string with a number of copies of a string.

string.repeat(count)

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| count | Required.  The number of copies. |

|  |  |
| --- | --- |
| let text = "Hello world!";  let result = text.repeat(2); |  |

***for of* method**

The **for of** statement loops through the values of an **iterable** object (like Arrays, Strings, Maps, NodeLists, and more).

Unlike **forEach**, it lets you **break** (using **break**) or **skip** (using **continue**) while iterating through the object.

In the example below, we want to create a **shortName** variable with the same content of **longName** but without spaces. Each letter corresponds to an index in the **longName** string. So, we use continue to look for the spaces and skip them. If it is not a space, it will add the letter to the **shortName** variable.

|  |  |
| --- | --- |
| const longName = 'John Smith Pepper III';  let shortName = ''  for (const letter of longName) {    if(letter === ' ') {      continue;    } else {      shortName += letter;    }  }  console.log(shortName); |  |

You can also break the loop as soon as it finds a certain element of the array:

|  |  |
| --- | --- |
| const fruits = ['apple', 'orange', 'banana', 'peach'];  for (const fruit of fruits) {    if(fruit === 'banana') {      break;    } else {      console.log(fruit);    }  } |  |

Or you can skip over it:

|  |  |
| --- | --- |
| for (const fruit of fruits) {    if(fruit === 'banana') {      continue;    } else {      console.log(fruit);    }  } |  |

***for in* method**

The **for in** statement iterates over all **enumerable string properties** of an object including inherited enumerable properties.

It is not advised to use **for in** in arrays. Use for **of or** the other array methods instead.

In the example below, we use **for in** to iterate through each of person’s property names and assign a value to it.

const person = {

  firstName: 'john',

  age: 25,

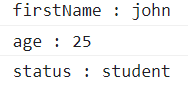
  status: 'student'

};

for (const propertyName in person) {

  console.log(`${propertyName} : ${person[propertyName]}`);

}



***Array.of* method**

The **Array.of()** method **creates a new Array instance** from a variable number of arguments, **regardless of number or type of the arguments** – items can be strings, numbers or Booleans.

In the example below, we pass 3 elements into the **Array.of** method to create an array with those elements.

|  |  |
| --- | --- |
| const example = Array.of('john', 2, true);  console.log(example); |  |

***Array.from* method**

The **Array.from()** static method creates a new, shallow-copied Array instance from an iterable or array-like object (e.g. a string, nodeList, Set, arguments, etc).

**Array.from with Strings**

|  |  |
| --- | --- |
| console.log(Array.from('john')); |  |

**Array.from with arguments**

See the example below where we use the keyword **arguments** to extract the arguments passed when calling the function. The goal is to **convert it to an array** object so we can then use array methods like **reduce** to calculate the total.

|  |  |
| --- | --- |
| function countTotal() {      console.log(arguments);  }  countTotal(67, 89, 54, 100); |  |

As you can see below, by turning the object into an array object we were able to use the **reduce** method on it.

|  |  |
| --- | --- |
| function countTotal() {      console.log(Array.from(arguments));      let total = Array.from(arguments).reduce(          (total, currNum) => (total += currNum),0)      console.log(total);  }  countTotal(67, 89, 54, 100); |  |

**Array.from with NodeList**

Node Lists are something that we get back when we select multiple elements, like with querySelectorAll.

For example, consider the HTML below:

|  |  |
| --- | --- |
| <h1>ES6</h1>      <p>john</p>      <p>peter</p>      <p>susan</p>      <h2 id="result"></h2>      <h1 id="second"></h1> |  |

We cab use **Array.from** to convert the Node List (in the case all the paragraphs) so we can use array method, like map, which in this case we use to join the paragraphs.

const p = document.querySelectorAll('p');

const result = document.getElementById('result');

const second = document.getElementById('second');

let newText = Array.from(p);

newText = newText.map(item => `<span>${item.textContent}</span>`).join(' ');

result.innerHTML = newText;

***Object.key* method**

**Object.key convets** property names into arrays.

|  |  |
| --- | --- |
| const person = {    firstName: 'john',    age: 25,    status: 'student'  };  const keys = Object.keys(person);  console.log(person);  console.log(keys); |  |

***Object.key* method**

**Object.value** converts property values into arrays.

|  |  |
| --- | --- |
| const person = {    firstName: 'john',    age: 25,    status: 'student'  };  const values = Object.values(person);  console.log(person);  console.log(values); |  |

***Object.entries* method**

**Object.entries** is a combination of **Object.keys** and **Object.values** because it returns both names and values of the objects into an array.

|  |  |
| --- | --- |
| const person = {    firstName: 'john',    age: 25,    status: 'student'  };  const result = Object.entries(person);  console.log(result); |  |

As you can see it returns an array with 3 arrays inside. These arrays contain 2 elements each, one is the key and other one is the value. You can use array methods to split them into items.

|  |  |
| --- | --- |
| const person = {    firstName: 'john',    age: 25,    status: 'student'  };  const result = Object.entries(person);  const newResult = result.map((item) => {    const [first, second] = item;    console.log(first);    console.log(second)  }) |  |

Or, a shorter and more efficient version would be using a **for of** method.

|  |  |
| --- | --- |
| const person = {    firstName: 'john',    age: 25,    status: 'student'  };  const result = Object.entries(person);  for (const [first, second] of result) {    console.log(first);    console.log(second)  } |  |