**var, let and const**

**Defining a variable with *var***

Defining a variable with **var** allows you to **update** and **redefine** the variable.

|  |  |
| --- | --- |
| //define  var number = 100;  console.log(number);  //update  number = 200;  console.log(number);  //redefine  var number = 'orange';  console.log(number); |  |

**Defining a variable with *let***

Defining a variable with **let** allows you to **update**, but **not** **redefine** the variable.

|  |  |
| --- | --- |
| //define  let number = 100;  console.log(number);  //update  number = 200;  console.log(number); |  |
|  |  |
| //redefine  let number = 'orange';  console.log(number); |  |

**Defining a variable with *const***

Defining a variable with **const** doesn’t allow you to **update** or **redefine** the variable.

|  |  |
| --- | --- |
| //define  const number = 100;  console.log(number); |  |
| //update  number = 200;  console.log(number); |  |
| //redefine  const number = 'orange';  console.log(number); |  |

However, you can update the variable **non-primitive type** (like string, or number).

If we have a **reference type** (like object or array) the values within the object or array can be mutated.

const person = {name: 'bob'};

person.name = 'john';

**Function Scope and Block Scope**

Another key difference between **var**, **let** and **const** is going to be the fact that **var** has only **functions** **scope**, but **let** them **const** are **blocked** **scope**.

You can think of blocked of being anything within the curly braces.

**Function scope**

* **var** / **let** / **const**: a variable defined outside a function can be accessed from inside the function. However, a variable defined inside the function cannot be accessed from outside the function.

|  |  |
| --- | --- |
| //global scope  var amount = 100;  function display() {      //local scope      var random = 'some random value'      console.log(amount + random)  }  display(); |  |
| //global scope  var amount = 100;  function display() {      //local scope      var random = 'some random value'      console.log(amount + random)  }  console.log(random) |  |

**Block scope**

If **var** is defined inside curly braces is still a global variable though. As you can see below, the value of the number variable was redefined inside the curly braces, and JS was able to access it from outside.

|  |  |
| --- | --- |
| var number = 1000;  var test = true;  if (test) {      //local scope      var number = 2000;  }  console.log(number); |  |

However, for **let** and **const** that is not possible.

|  |  |
| --- | --- |
| let number = 1000;  let test = true;  if (test) {      //local scope      let number = 2000;      let amount = 3000;  }  console.log(number);  console.log(amount); |  |

**Variable Initialization**

While var allows the variable to be accessed before initializing the variable (although the value won’t be accessed, because it wasn’t assigned at that point), let and const don’t.

|  |  |
| --- | --- |
| console.log(number)  var number=100; |  |
| console.log(number)  let number=100; |  |
| console.log(number)  const number=100; |  |

**Template Strings**

Before template strings, we had to use concatenation to join text and variables.

Template Strings made it much easier to do that by just using backstricks `` and ${}.

const firstName = "bob";

const lastName = "sanders";

const age = 25;

const phrase = "My full name is " + firstName + " " + lastName + " and I'm " + age + " years old.";

const phrase2 = `My full name is ${firstName} ${lastName} and I'm ${age} years old`;

console.log(phrase);

console.log(phrase2);



**Dynamically inserting HTML**

Consider the following HTML file just with a div like this:

<div id="result"></div>

We could use concatenation to add HTML to our file thought JS:

const person = {

  name: "kyle",

  job: "developer",

  hobbies: ["surfing", "baking", "bowling"],

};

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

result.innerHTML = "<h2>" + person.name + "</h2>" + "<p>" + person.job + "</p>";

However, it looks much tidier with template strings. And also, we can use template strings inside template strings. Below, we also use the map function to create an unordered list from the items in the array.

const person = {

  name: "kyle",

  job: "developer",

  hobbies: ["surfing", "baking", "bowling"],

};

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

result.innerHTML = `

<h2>${person.name}</h2>

<p>${person.job}</p>

<ul>

${person.hobbies

  .map(item => {

    return `<li>${item}</li>`;

  })

  .join("")}

</ul>

`;

**Tagged Template Literals**

Another interesting thing we can deal with them is the literals is **tagging** them, and essentially it just means that we can run our template literals through a function and have more control over our template.

Consider the following HTML file just with a div like this:

<div id="result"></div>

And the following JS:

const firstName = "Bob";

const lastname = "Sanders";

const fullName = `My name is ${firstName} ${lastname}`;

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

We can create a function and tag it, like below. This function should have the text as the first argument (in this case My name is), and then every argument as the following arguments.

const firstName = "Bob";

const lastname = "Sanders";

const fullName = highlight`My name is ${firstName} ${lastname}`;

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

function highlight(text, arg1, arg2) {

    console.log(text);

    console.log(arg1);

    console.log(arg2);

}



For simplification (specially in cases where the template string has a lot of variables) we use the **rest operator**, creating a new variable and storing the variables inside that array. The text array is always going to have one more value than the values array.

const firstName = "Bob";

const lastname = "Sanders";

const fullName = highlight`My name is ${firstName} ${lastname}`;

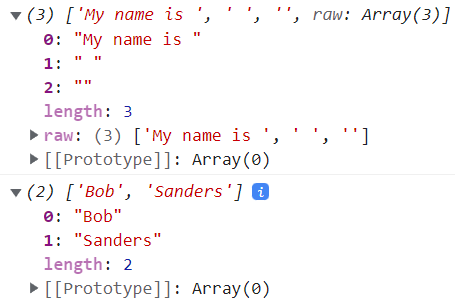
const result = document.getElementById("result");

function highlight(text,...vars) {

    console.log(text);

    console.log(vars);

}



Now, if we want to, for example, apply a bold effect to the variables only, we can use a **map** function the create a strong tag around each variable.

The idea is to run through the text array and every time we loop through an item in it, we add the variable with the same index but with the strong tags.

const firstName = "Bob";

const lastname = "Sanders";

const fullName = highlight`My name is ${firstName} ${lastname}`;

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

function highlight(text,...vars) {

    const strongText = text.map(function(item, index) {

        return `${item} <strong>${vars[index]}</strong>`;

    });

    return strongText.join('')

}



However, we still get an undefined because the **vars** array is one element shorter than the **text** array. So, we need to tell JS to just return an empty string if **vars[index]** is undefined.

const firstName = "Bob";

const lastname = "Sanders";

const fullName = highlight`My name is ${firstName} ${lastname}`;

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

result.innerHTML = fullName;

function highlight(text,...vars) {

    const strongText = text.map(function(item, index) {

        return `${item} <strong>${vars[index] || ""}</strong>`;

    });

    return strongText.join('')

}



**Arrow Functions**

Arrow functions allow us to write shorter function syntax:

|  |  |
| --- | --- |
| **Regular Function** | **Arrow Function** |
| hello = function() {    return "Hello World!";  } | hello = ()=> {    return "Hello World!";  } |

It gets shorter! If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

hello = () => "Hello World!";

If you have parameters, you pass them inside the parentheses:

hello = (val) => "Hello " + val;

In fact, if you have only one parameter, you can skip the parentheses as well:

hello = val => "Hello " + val;

Let’s see more examples or regular functions converted into arrow functions:

|  |  |
| --- | --- |
| **Regular Function** | **Arrow Function** |
| function sum(a,b) {    return a + b  } | let sum = (a,b) => a + b |
| Notes:   * The **function** keyword creates a sum variable, but since we don’t have it in the Arrow Function format, we need to define it with **let**, **const** or **var**. * The arrow **=>** denotes that **a** and **b** are parameters. * We just have one statement, so return can be removed. | |
| function isPositive(number) {    return number >= 0  } | let isPositive = number => number >=0 |
| Notes:   * When the function only has one single parameter, we can remove the parenthesis around the arguments. | |
| function randomNumber() {    return Math.random  } | let randomNumber() = () => Math.random |
| Notes:   * When the function doesn’t have any parameter, it starts with **()** | |
| document.addEventListener(‘click’, function {console.log(‘click’)}) | document.addEventListener(‘click’, () => console.log(‘click’)) |
| Notes:   * When an anonymous function is used without parameters and with just one lune, it can be as simple as **() =>** | |

**Arrow Functions and *this* keyword**

In **regular** **functions**, **this** refers to the **parent**.

In **arrow** **functions**, **this** **inherits the scope of its enclosing scope.**

In the example below, where we use a **regular** **function**, you can see that **this** refers to the object **bob**, which is the parent, because **this.firstName** and **this.lastName** belong to **sayName**, which has **bob** as parent.

|  |  |
| --- | --- |
| const bob = {    firstName: "bob",    lastName: "smith",    sayName: function() {      console.log(this);      console.log(`Hello, my name is ${this.firstName} ${this.lastName}`);    },  };  bob.sayName(); |  |

However, when the method uses the **arrow** **format**, you can see that it refers to the **window** object, because it inherits the scope of his parent. His parent is the object **bob**, and **bob** is defined inside the global scope.

|  |  |
| --- | --- |
| const bob = {    firstName: "bob",    lastName: "smith",    sayName: () => {      console.log(this);      console.log(`Hello, my name is ${this.firstName} ${this.lastName}`);    },  };  bob.sayName(); |  |

**Arrow Functions and *this* in callback functions**

This comes very handy in, for example, **callback** **functions**.

In the example below, we use a **regular** **function** as a **callback** **function** inside the **setTimeout** function, which is inside the **sayName** function.

The callback function where **this.firstName** and **this.lastName** are mentioned, don’t have the object **bob** as direct parent anymore, so it will point at the global object.

Thus, when we call **sayName**, the values of the variables **firstName** and **lastName** are undefined, as these properties don’t exist in the global object.

|  |  |
| --- | --- |
| const bob = {    firstName: "bob",    lastName: "smith",    sayName: function() {      console.log(this);      setTimeout(function() {        console.log(this);        console.log(`Hello, my name is ${this.firstName} ${this.lastName}`);      }, 1000);    },  };  bob.sayName(); |  |

However, arrow functions inherit the **this** reference of the **enclosing** **scope**, which in this case is the function **sayName**.

In **sayName**, **this** refers to the **bob** object, so the anonymous function will too.

|  |  |
| --- | --- |
| const bob = {    firstName: "bob",    lastName: "smith",    sayName: function() {      console.log(this);      setTimeout(() => {        console.log(this);        console.log(`Hello, my name is ${this.firstName} ${this.lastName}`);      }, 1000);    },  };  bob.sayName(); |  |

**Arrow Functions – Select Elements and *this***

Consider we have a button defined in our HTML and we want to target it and change its style.

<button class="btn">click me</button>

We can simply use a regular function, which would point at it, because the button is its parent.

|  |  |
| --- | --- |
| const btn = document.querySelector('.btn');  btn.addEventListener('click', function(){    this.style.color = "black";    console.log(this);  }) |  |

Actually, an **arrow** **function** would point at the **window**, so the code wouldn’t really work. This is because its **enclosing** **scope** is the **global** **object**, so it would inherit the **this** reference of the global object, which is **window**.

|  |  |
| --- | --- |
| const btn = document.querySelector('.btn');  btn.addEventListener('click', () =>{    console.log(this);    this.style.color = "black";  }) |  |

However, it would work if we had a timeout function, because this time it’s **enclosing** **scope** would be the first anonymous function. In the first anonymous function, **this** refers to the **btn**, so the second **this** would inherit its reference.

|  |  |
| --- | --- |
| const btn = document.querySelector('.btn');  btn.addEventListener('click', function(){    console.log(this);    setTimeout(()=>{      console.log(this);      this.style.color = "black"},1000);  }); |  |

**Hoisting in Arrow Functions**

Unlike what happens with regular functions, arrow functions need to be defined before they are called.

**Destructuring**

The **Destructuring Assignment** syntax is a JavaScript expression that makes it possible to unpack values from **arrays**, or **properties** from objects, into distinct variables.

**Destructuring** makes it easy to extract only what is needed. We may have an array or object that we are working with, but we only need some of the items contained in these.

**Destructuring Arrays**

A destructured array looks like example below:

let a, b, rest;

[a, b] = [10, 20];

console.log(a);

// expected output: 10

console.log(b);

// expected output: 20

While before we used array indexes to access the elements of an array, we can now assign variables to each of those elements and store the value in that variable. In both cases, if we **console.log(car)**, the output will be **mustang**.

|  |  |
| --- | --- |
| **Before** | **After** |
| const vehicles = ['mustang', 'f-150', 'expedition'];  // old way  const car = vehicles[0];  const truck = vehicles[1];  const suv = vehicles[2]; | const vehicles = ['mustang', 'f-150', 'expedition'];  const [car, truck, suv] = vehicles; |

And if we want to skip the **truck** variable (i.e. don’t want to access it):

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car,, suv] = vehicles;

**Destructuring Objects**

We can also destructure objects, as you can see below.

While in arrays the key name doesn’t need to match in order to access the element, when **destructuring objects the key name needs to match the property**.

|  |  |
| --- | --- |
| **Before** | **After** |
| const bob = {    firstName: 'bob',    lastName: 'sanders',    city: 'chicago',    siblings: {      sister: 'jane',    }  }  const firstName = bob.firstName;  const lastName = bob.lastName;  const city = bob.city;  const sister = bob.siblings.sister;  console.log(firstName, lastName, city, sister); | const bob = {  firstName: 'bob',  lastName: 'sanders',  city: 'chicago',  siblings: {  sister: 'jane',  }  }  const {firstName, lastName, city, siblings:{sister}} = bob;  console.log(firstName, lastName, city,sister); |

And you can also change the property names when you destructure it:

|  |  |
| --- | --- |
| const bob = {    firstName: 'bob',    lastName: 'sanders',    city: 'chicago',    siblings: {      sister: 'jane',    }  }  const {firstName:fName, lastName, city, siblings:{sister}} = bob;  console.log(fName, lastName, city, sister); |  |

**Destructuring Objects in Arrays**

See the example below where we have an array of 2 objects:

const animals = [

  {name:"cat", sound:"meow"},

  {name:"dog", sound:"woof"}

]

const [cat, dog] = animals

console.log(cat)

//expected output {name: 'cat', sound: 'meow'}

So, **cat** stores the properties object with index 0 {name:"cat", sound:"meow"} of the array animals, and the variable **dog** stores the properties of the object with index 1 {name: "dog", sound:"woof"}

**Destructuring Function Parameters**

You can also pass the properties of an object as parameters using destructuring:

|  |  |
| --- | --- |
| **Before** | **After** |
| const bob = {    firstName: 'bob',    lastName: 'sanders',    city: 'chicago',    siblings: {      sister: 'jane',    }  }  function printPerson(person) {    const firstName = person.firstName;    const lastName = person.lastName;    const city = person.city;    const sister = person.siblings.sister;    console.log(firstName, lastName, city, sister)  }  printPerson(bob) | const bob = {    firstName: 'bob',    lastName: 'sanders',    city: 'chicago',    siblings: {      sister: 'jane',    }  }  function printPerson({firstName, lastName, city, siblings:{sister}}) {    console.log(firstName, lastName, city, sister);  }  printPerson(bob) |

**Spread Operator**

The **Spread** **Operator** **…** allows an iterable object to spread/expand individually inside a receiver. Or basically, to split it into single items and copy them.

**Spread Operator in Strings**

As you can see below, we can use the **Spread** **Operator** to store each element from a string and store it in an array.

|  |  |
| --- | --- |
| const udemy = 'udemy';  const letters = [...udemy];  console.log(letters); |  |

**Spread Operator in Arrays**

Imagine we want to combine the elements of the following array in a single new array. The following syntax wouldn’t work because it would create arrays inside arrays.

|  |  |
| --- | --- |
| const boys = ['john', 'peter', 'bob'];  const girls = ['susan', 'anna'];  const bestFriend = 'arnold';  const friends = [boys, girls, bestFriend];  console.log(friends); |  |

To do that, you have to use the **Spread** **Operator** as below:

|  |  |
| --- | --- |
| const boys = ['john', 'peter', 'bob'];  const girls = ['susan', 'anna'];  const bestFriend = 'arnold';  const friends = [...boys, ...girls, bestFriend];  console.log(friends); |  |

**Spread Operator in Objects**

We can use **Spread** **Operator** to **add** **properties** to an object.

|  |  |
| --- | --- |
| const person = {  firstName: 'john',  job: 'developer'};  const newPerson = {  ...person,  city: 'chicago'  };  console.log(newPerson); |  |

And you can also overwrite them as well.

|  |  |
| --- | --- |
| const person = {firstName: 'john', job: 'developer'};  const newPerson = {  ...person,  city: 'chicago', firstName:'peter'  };  console.log(newPerson); |  |

**Spread Operator in DOM Elements**

**Spread** **Operator** is very useful when we want to **turn Node Lists into arrays**, so we can then use all the useful array methods after, like **map**.

For example, consider we have the following HTML with 4 headers and a **result** **div**:

<h1>ES6</h1>

<h1>Javascript</h1>

<h1>HTML</h1>

<h1>CSS</h1>

<h2 id="result"></h2>

Then we to use a **map** function to iterate through all of them and join. As you can see below, when we select the elements with **querySelectorAll** it returns a node list. But, if we use the Spread Operator, it turns it into an array

|  |  |
| --- | --- |
| const headings = document.querySelectorAll('h1');  const headingsArray = [...headings];  console.log(headings);  console.log(headingsArray); |  |

We can now use the **map** function to do join the headings:

|  |  |
| --- | --- |
| const headings = document.querySelectorAll('h1');  const result = document.getElementById('result');  const headingsArray = [...headings].map(item => `<span>${item.textContent}</span>`  ).join(' ');  console.log(headingsArray);  result.innerHTML = headingsArray; |  |

**Spread Operator in Function Arguments**

A lot of times we need to receive data in a array format. But many functions don’t work with that. They are unable to extract the information of the array – unless we use the spread operator to separate them.

For example, the **Math.max** function would work with the format below, because all the inputs are numbers:

Math.max(1,2,3,4,5)

However, if those numbers came in a array format, it wouldn’t work.

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

Math.max(numbers);

However, if we use the spread operator, we can separate those numbers and access them:

|  |  |
| --- | --- |
| const numbers = [1,2,3,4,5];  console.log(numbers);  console.log(...numbers);  console.log(Math.max(...numbers)); |  |

This obviously also works with normal functions. See below the function sayHello. We want to extract the elements from the john array and pass it as parameters.

const john = ['john','sanders'];

const sayHello = (firstName, lastName) => {

    console.log(`Hello ${firstName} ${lastName}`);

};

To do that, we could simply call it using:

sayHello(john[0],john[1])

But we can also use the spread operator:

sayHello(…john)

**Rest Operator**

The **rest** **operator** does the **opposite of the spread operator** and even has the same syntax **…**

Rest element **must be the last element**.

The rest element doesn’t need to be “rest”, it can be anything as long as it is in front of the **…**

There are two cases where we use the rest operator:

* When you’re want to **destructure** **arrays** or **objects**.
* When you want to **gather parameters in a function**.

**Rest Operator in Arrays**

The **rest** **operator** gets all the items in an array that where now already destructured.

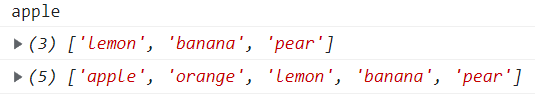
const fruit = ['apple', 'orange', 'lemon', 'banana', 'pear'];

const [first, second, ...rest] = fruit;

console.log(first);

console.log(rest);

console.log(fruit);



**Rest Operator in Objects**

Like what happens in arrays, the **rest** **operator** gets all the properties in an array that where now already destructured.

|  |  |
| --- | --- |
| const person = {      firstName:'john',      lastName:'smith',      job:'developer'  };  const {job, ...rest} = person;  console.log(job);  console.log(rest); |  |

**Rest Operator in Functions**

As said before, another use of the rest operator is to **collect the parameters when we are passing in the function**. That is one of the main differences between the spread operator and the rest operator:

* **Spread** **Operator**: **spreads the items** in an array so they are all as an **argument** in a array format, they are passed as individual items.
* **Rest** **Operator**: allows to **gather the items to pass it as parameters**.

In the example below, we use the rest operator to **pick up the arguments from the function calling** and place them in a array, so then it can be passed as a parameter in the function definition.

const person = {name:'john', lastName: 'smith', job: 'developer'}

function getAverage (firstName, ...scores) {

    console.log(firstName);

    console.log(scores);

}

getAverage(person.name, 78, 90, 56, 43)



Now that we have access to the array, we can use array method, such as **for of** and calculate the average.

const person = {name:'john', lastName: 'smith', job: 'developer'}

function getAverage (firstName, ...scores) {

    console.log(firstName);

    console.log(scores);

    let total = 0;

    for(const score of scores){

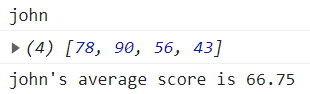
        total += score

    }

    console.log(`${firstName}'s average score is ${total / scores.length}`)

}

getAverage(person.name, 78, 90, 56, 43)



When we call the function, we can use the spread operator to get the items of an array, instead of passing the values in the function calling itself, and it will give the same result.

const person = {name:'john', lastName: 'smith', job: 'developer'}

function getAverage (firstName, ...scores) {

  let total = 0;

  for(const score of scores){

      total += score

  }

  console.log(`${firstName}'s average score is ${total / scores.length}`)

}

testScores = [78, 90, 56, 43];

getAverage(person.name, ...testScores);

**new Set()**

The **Set** object **stores a collection of unique values of any type**.

It comes with methods like:

* **add(***value***)**
* **delete(***value***)**
* **clear()**
* **has(***value***)**

In the example below, we use the **add()** method to add elements into the Set object we created. As you can see, we can add variables, numbers or strings. Notice also that we have tried to add the same element twice and it didn’t work, because it only adds unique values.

|  |  |
| --- | --- |
| const unique = new Set();  random = 'thirdValue';  unique.add('firstValue');  unique.add(2);  unique.add(random);  unique.add(2);  console.log(unique); |  |

We can also **delete()** items. You can also get **true** or **false** depending if that item was deleted or not.

|  |  |
| --- | --- |
| const unique = new Set();  random = 'thirdValue'  unique.add('firstValue');  unique.add(2);  unique.add(random);  unique.add(2);  unique.delete();  const deleteConfimation = unique.delete(random);  console.log(deleteConfimation);  console.log(unique); |  |

Or **clear()** the whole items.

|  |  |
| --- | --- |
| const unique = new Set();  random = 'thirdValue'  unique.add('firstValue');  unique.add(2);  unique.add(random);  unique.add(2);  unique.clear();  console.log(unique); |  |

Or check if the **Set** object has that **element** you are looking for.

|  |  |
| --- | --- |
| const unique = new Set();  random = 'thirdValue'  unique.add('firstValue');  unique.add(2);  unique.add(random);  unique.add(2);  const hasValue1 = unique.has('thirdValue');  const hasValue2 = unique.has(random);  console.log(hasValue1);  console.log(hasValue2);  console.log(unique); |  |

**Useful examples of Set**

Imagine we have an array of objects like below and we want to create a filter, which basically is an array of unique items, so we can select which company we want to see. We would also have an option of “all” in case we wanted to see items from every company.

const products = [

  {

    title: 'high-back bench',

    company: 'ikea',

  },

  {

    title: 'albany table',

    company: 'marcos',

  },

  {

    title: 'accent chair',

    company: 'caressa',

  },

  {

    title: 'wooden table',

    company: 'ikea',

  },

]

So because it is an array, we could use the map method to iterate over the elements and check the properties. The, Set will only return the properties with unique values.

|  |  |
| --- | --- |
| const products = [    {      title: 'high-back bench',      company: 'ikea',    },    {      title: 'albany table',      company: 'marcos',    },    {      title: 'accent chair',      company: 'caressa',    },    {      title: 'wooden table',      company: 'ikea',    }  ]  const result = new Set(products.map((item) => item.company));  console.log(result) |  |

We can then make result an array, add “all” element to it and then use the spread operator to add the rest of the array elements.

|  |  |
| --- | --- |
| const products = [    {      title: 'high-back bench',      company: 'ikea',    },    {      title: 'albany table',      company: 'marcos',    },    {      title: 'accent chair',      company: 'caressa',    },    {      title: 'wooden table',      company: 'ikea',    }  ]  const result = ['all',...new Set(products.map((item) => item.company))];  console.log(result) |  |