1. **Create the DOM objects using CreateElement()**

**Steps:**

1. Create a Element using document.createElement().
2. Append the child using appendChild();
3. Add the content using textContent = ‘’;

function solve() {

const list = document.querySelector('ul');

const newel1 = document.createElement('li');

const newel2 = document.createElement('li');

list.appendChild(newel1);

list.appendChild(newel2);

newel1.textContent = 'Learn JavaScript';

newel2.textContent = 'Master JavaScript';

};

solve();

**Instead of** using a **appendChild()** we can use a **append()** for adding a text nodes to **element.**

1. Prepend() 🡪 For Adding the element in list as **First**

**Iterator (@@):**

**Array:**

1. Store (nested) data of an array kind and length.
2. Zero indexed to access the value, order is not automatically change and duplicate are allowed.
3. Iterable, also many special array methods are available.

**Adding and Removing Element:**

**Methods:**

**push() 🡪** Push an element into the **Last.**

**pop() 🡪** Pop an element from **end of the array.**

**unshift() 🡪** It **shifts all the element to the right end of an array.** It add the element in **first position of an array.**

**shift() 🡪** It **shifts all the elements in an array one place to left.** So, that the **first element** **drops from an array**.

**Slice() 🡪** It used to **Selecting the range from the array.** It **does not select in reverse direction. But we can use in negative index.**

**indexOf() 🡪 It is used to search the element in an array.**

**lastIndexOf() 🡪 It is used to search the element from the last index of an array.**

**find() 🡪 It actually takes a argument that argument is another function.**

**findIndex() 🡪 It gives the index of an array, that array has object.**

**forEach() 🡪 It is a method likes a find(), findIndex(). In forEach() it takes a pass something to the method but it doesn’t return anything.**

**Map() 🡪 one difference between it does not take any new array for the operation. Instead of it takes new element to its original array. Map() method takes place it pass something to the method and it return something. It returns the new elements to its original array. How it’s not true really. It does not replace it the original array is untouched. Instead of it return brand new array with brand new address in memory where every element has been changed.**

**Split() 🡪** It is used to **split the element with specific condition which we given.**

const data = 'newyork;10.29;200';

const tranformedSplitedArray = data.split(';');

tranformedSplitedArray[1] = +tranformedSplitedArray[1];

console.log('Splited array --> ', tranformedSplitedArray);

**output:**

**Splited array 🡪 (3) ['newyork', 10.29, '200'].**

**Join() 🡪** It is used to **join the value in an array.**

**Sort() 🡪** Normal sort method used to sort the string. If you need to sort the **number** then follow this type bellow,

const prices = [10.99, 5.99, 3.09, 6.59];

const sortedPice = prices.sort((a,b) => {

    if(a>b) {

        return 1;

    } else if(a == b) {

        return 0;

    } else {

        return -1;

    }

});

console.log(sortedPice);

**reverse() 🡪 It is used to reverse an array. Whenever we need to reverse the array that time we can use this reverse method. Else we need to change the condition of the sorting logic. For example; return 1 is changed into -1 and return -1 is changed into 1.**

**Filter() 🡪 filter the some conditions.**

**Spread operator (…) 🡪 Spread operator** is an operator that in the end pulls out all elements of an array and gives them to you as a standalone.

By using **spread operator to copy the array or object only the old array or object get affected not a copied array.**

**For example:**

const person = {name: ‘padmaja’, hobbies: [‘coding’, ‘cooking’]};

const person2 = {…person};

person.age = 21;

console.log(person); // name: ‘padmaja’, hobbies: [‘coding’, ‘cooking’], age: 21};

console.log(person2); // name: ‘padmaja’, hobbies: [‘coding’, ‘cooking’]};

**But adding a new element in objects of an array makes sense.**

Person.hobbies.push(‘music’);

console.log(person); // name: ‘padmaja’, hobbies: [‘coding’, ‘cooking’, ‘music’], age: 21};

console.log(person2); // name: ‘padmaja’, hobbies: [‘coding’, ‘cooking’, ‘music’]};

const p3 = {…person, print: 1, hobbies: […person.hobbies]};

console.log(p3);

person.hobbies.push(‘gaming’);

A screenshot of a computer program

Description automatically generated

**Splitting an array into an variable:**

Without access by using index of an array. Instead of that we can use **Destructing an array.**

const nameData = ['Padmaja', 'Srinivasan'];

// const firstName = nameData[0];

// const lastName = nameData[1];

// by getting the value separately there is another method to get like this

// right side of nameData is pull out the firstName and lastName variable on left hand side of an array

// splitting an array to an variable

const [firstName, lastName] = nameData;

console.log(firstName, '', lastName);

**Set:**

1. Store (nested) data of any kind and length.
2. **No** indexed to access the value, order is **not guaranteed** and duplicate are **not** allowed.
3. Iterable, also some special **set** methods are available.

**Maps:**

1. Store **key-value** data of any kind and length, any key is allowed.
2. Iterable, also some special **map** methods are available.
3. Order is **guaranteed,** duplicate keys are **not allowed** but **duplicate values are allowed, key based access.**

**Map methods:**

**clear() 🡪** It clear all the data from an maps.

**delete() 🡪** It delete a single entry by key from the map.

**forEach() 🡪** It allows you to go through all items.

**has() 🡪** It allows to check the certain key is part of map.

**set() 🡪** It allows to set the **values.**

**size🡪 how many items in the map at the moment.**

**Maps:**

1. Can use any values as key.
2. Best performance for large quantities of data.
3. Best performance for **adding and removing data** frequently.

**Objects:**

1. Only use strings, numbers, or symbols as key. It **does not take a Boolean as key or an object. You can’t use another** object or array.
2. Perfect for **small or medium sized sets of data.**
3. **Easier or quicker** to create (typically also with better performance).

**weakSet Method:**

1. **add**
2. **delete**
3. **has**

**There** is **no other method to get all the entries.**

**Objects:**

**Objects** are **real-world entities. 🡪 button, movies**

**Objects** are made up of **property and methods.**

**Property** these are these **key-value pairs. It i**

**Methods** in the end are also **key-value pairs** but **values is function in an object.**

**Add property and delete a property:**

**Delete 🡪 For example:** delete person.age.

**Add 🡪 For example:** person.age = 21.

**Variable name** is used to **Key name.**

**Spread operator in object:**

A screenshot of a computer program

Description automatically generated

**Another way to copy the object:**

A screenshot of a computer program

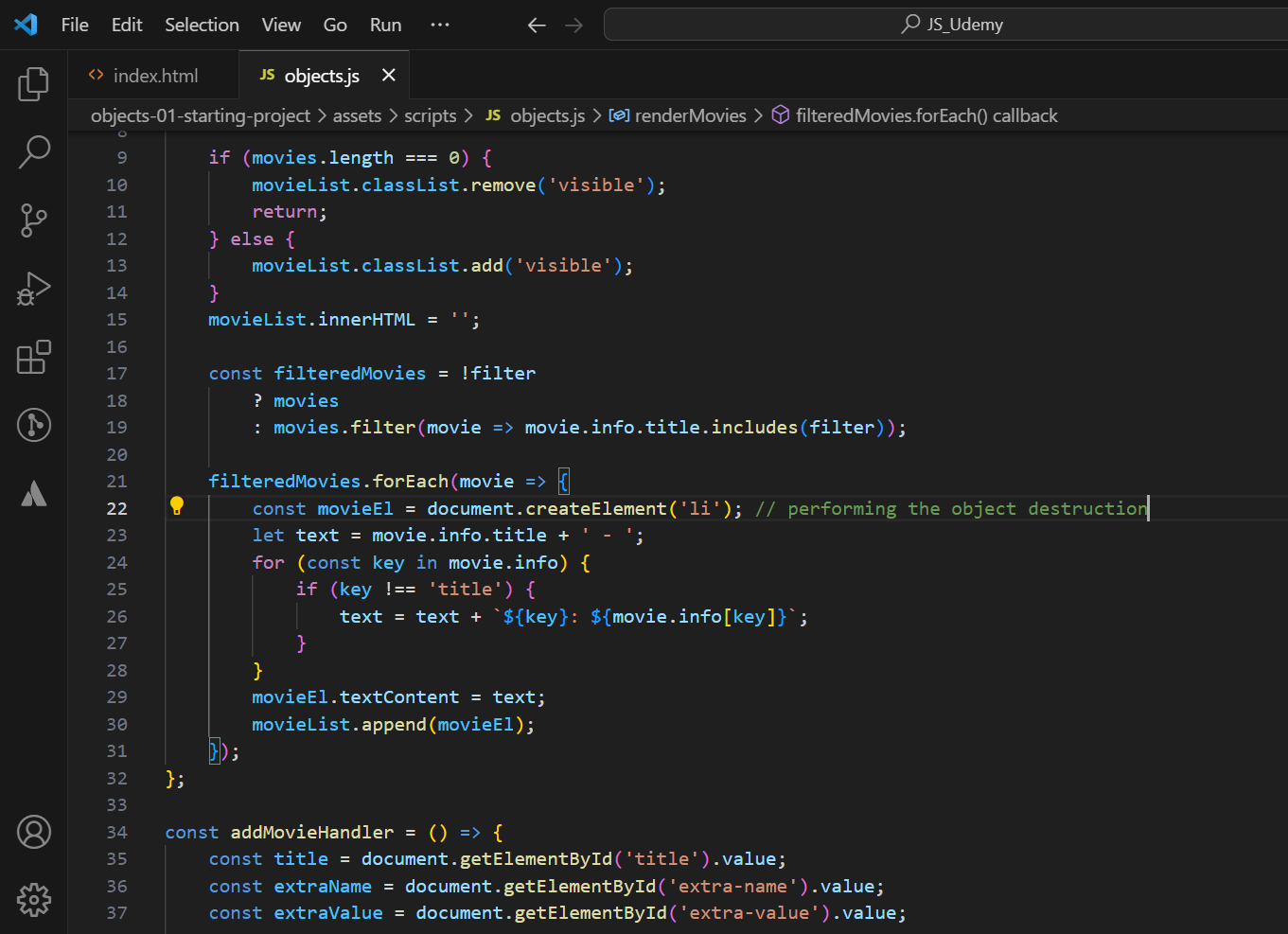
Description automatically generated

**Object.assign() 🡪 The target of the parameter** consisting of a **new object or existing object is the first parameter and** second parameter is **which object needs to copy.**

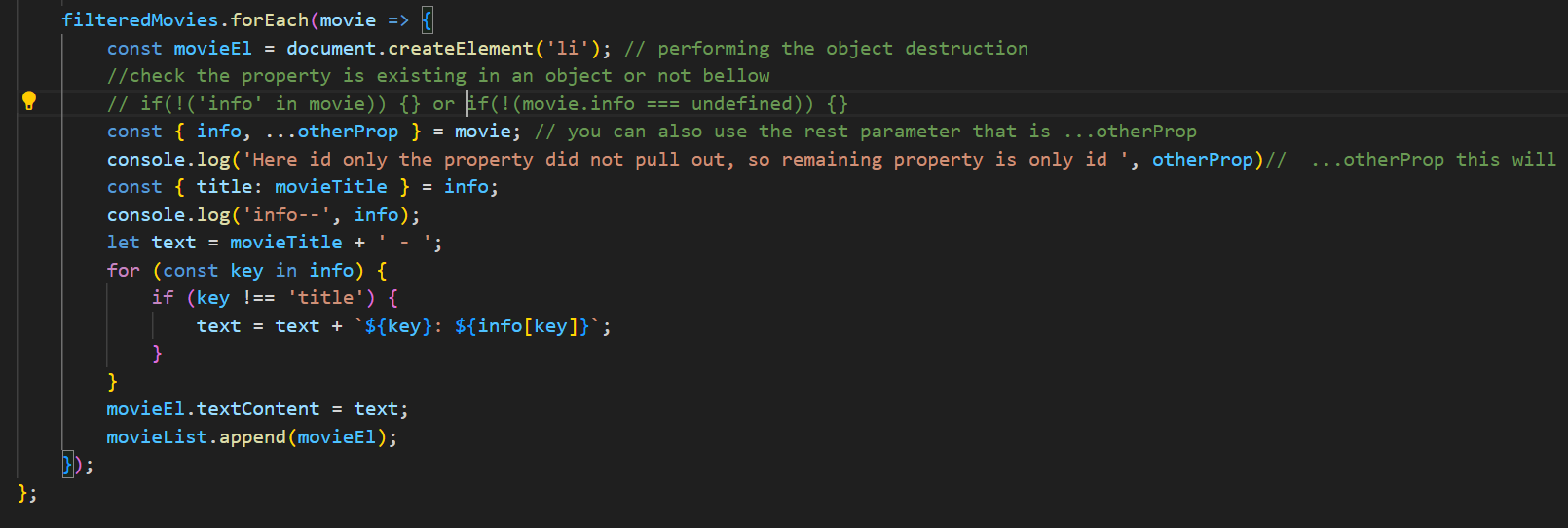
**Object Destructing:**

**Object destruction** done in **filteredMovies 🡪 for refer in codebase.**

**Before object destruction:**

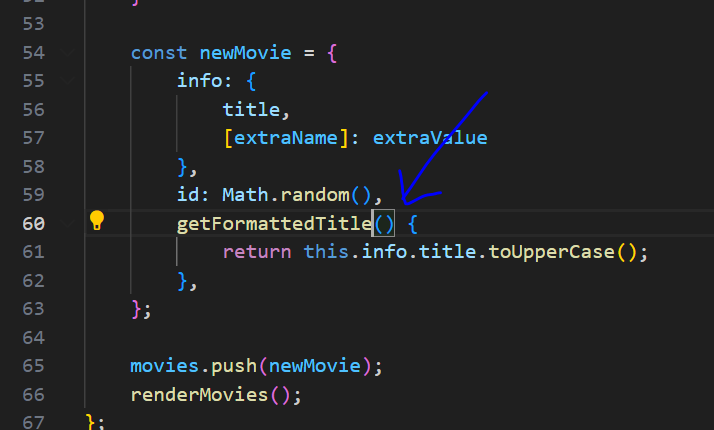


**After object Destruction:**



**this keyword: 🡪 refer codebase**

**shorthand function in object:**



**bind() method, an object can borrow a method from another object.**

**Besides bind() we can also have call() . call()** also takes multiple arguments.

Bind() prepare a function for **future execution, bind return a new function object in the end which we then store here getFormattedTitle,** but **call does not do that, call instead goes ahead and execute the function right away.**

A screenshot of a computer

Description automatically generated

**this keyword in 5 cases:**

**The first 3 cases are in a Global Context:**

1. **Outside of any function in the Global Context**
2. function something() {
3. console.log('something')
4. }
5. console.log('njsdnsaclkam', this);// logs global object (window in browser) - ALWAYS (also in strict mode)!
6. // something();
7. **In a function (Non-Arrow Function) in GC.**
8. // 2. In a function (Non-Arrow Function) in GC.
9. function inside() {
10. console.log(this); // logs global object (window in browser) in non-strict mode, undefined in strict mode
11. }
12. inside();

**3. In a function (Arrow Function) in GC.**

1. const something = () => {
2. console.log(this);
3. }
5. something(); // logs global object (window in browser) - ALWAYS (also in strict mode)!

**Objects 🡨 this keyword.**

1. **In a Method Arrow Function – called on object**
2. const person = {
3. name: 'Max',
4. greet: () => {
5. console.log(this.name);
6. }
7. };
9. person.greet(); // logs nothing (or some global name on window object), "this" refers to global (window) object, even in strict mode

this can refer to unexpected things if you call it on some other object, e.g.:

1. const person = {
2. name: 'Max',
3. greet() {
4. console.log(this.name);
5. }
6. };
8. const anotherPerson = { name: 'Manuel' }; // does NOT have a built-in greet method!
10. anotherPerson.sayHi = person.greet; // greet is NOT called here, it's just assigned to a new property/ method on the "anotherPerson" object
12. anotherPerson.sayHi(); // logs 'Manuel' because method is called on "anotherPerson" object => "this" refers to the "thing" which called it

**5. In a Method (Non–Arrow Function) 🡪 called on Objects.**

1. const person = {
2. name: 'Max',
3. greet: function() { // or use method shorthand: greet() { ... }
4. console.log(this.name);
5. }
6. };
8. person.greet(); // logs 'Max', "this" refers to the person object