**ECMAScript 7 (2016)**

**New operators**

* **\*\*** // power

**New methods in arrays**

* array**.includes(item)** // It returns **true** or **false** if an item or a substring exists in an array or string. Before this, we had to use **indexOf,** which wouldn't return true or false, which sometimes it's the only thing you need. Almost all browsers support it now. You can check that on the website [**https://caniuse.com/**](https://caniuse.com/).

**ECMAScript 8 (2017)**

**New methods in strings**

* **str.includes(item)** // like above. Returns true or false.

**String padding**

* **str.padStart(length, char)** // If str.length is smaller than the length given, adds the char to the start of the string to fill the gap. Always to the left, no matter what language. The default char, if not given, is space.
* **str.padEnd(length, char)** // If str.length is smaller than the length given, it adds the character to the end of the string to fill the gap. Always to the right, no matter what language. The default char, if not given, is space.

'Derry'.padstart(10, 'ab')

Output: 'ababaDerry'

**New methods in Objects**

* **Object.values(obj)** // returns an array of the values. Before this, we had to use **Object.keys(obj)**, which was messy and long.
  + With Object.keys(obj), in forEach and for-of

Object.keys(obj).forEach(key => console.log(**obj[key]**))

* + With Object.values(obj)

Object. values(obj).forEach(value => console.log(**value**))

* **Object.entries(obj)** // returns an array of arrays with key, values pairs.

Object. entries(obj).forEach(**(**[key, value]**)** => console.log(key, value)) //array destructuring

**Async function and await**

**Callbacks had problems:**

* By having too many callbacks, readability is affected, and debugging is hard.
* Declaring callbacks makes the code long

**Promises had problems:**

* Promises solved the first issue, but it was long and confusing while return a promise in a function.

**Async**

* Async resolves both issues: async keyword comes before function definition and uses return and throws new Error() to return value and catch data. It is a promise, so since async returns a promise, the part where we catch data and errors is the same as promises.

const fetchData = **async**(url) => { // It returns a promise

let data = { name : 'hesam' , family : 'mousavi'};

if(true) {

**return** data;

} else {

**throw new Error**('error URL');

}}

const saveDataToDB = **async**(data) => {

if(true) {

**return** true;

} else {

**throw new Error**('error DB')

}}

fetchData('roocket.ir') // Like promises

.then((data) => saveDataToDB(data))

.then((status) => console.log(status)) // output: true

.catch((err) => console.log(err));

**Await**

* Await is similar to above, with the difference that the await expression causes **async function** execution to **pause** until a Promise is settled (that is, fulfilled or rejected) and resume execution of the async function after fulfillment. When resumed, the value of the await expression is that of the fulfilled Promise. Based on the situation we can choose which of these two are better. Await is always used inside an **async function**.

const fetchData = async(url) => {

let data = { name : 'hesam' , family : 'mousavi'};

if(true) {

return data;

} else {

throw new Error('error URL');

}

}

const saveDataToDB = async(data) => {

if(true) {

return true;

} else {

throw new Error('error DB')

}

}

let executeFetchData = **async**() => { // an async function

**try** {

let data = **await** fetchData('roocket.ir');

let statusDB = **await** saveDataToDB(data);

console.log(statusDB);

} **catch**(err) {

console.log(err);

}

**console.log('hi')** // this won't be executed until fetchDAta and saveDAtaToDB are executed

}

executeFetchData();