Javascript-Refresher (Summary)

2026-01-19

0.1 Key Takeaways

- More specifically using import and export allow us to import and export variables, functions, objects, etc from one .js file into another for example:
- Primitives cannot be modified (immutable), modifying a primitive doesn't change the underlying variable but rather generates a brand new variable of the same type
- Similar to other programming languages, functions can be defined inside another function (making it a private function - therefore not callable outside of the parent function)
- We can directly export values by using the `default` keyword: `export default 'thing'`.
- map - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- filter - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter
- 2.1 If returning a JS object then parentheses must be wrapped around the object defintion: `number => ({'age' :  number})`
- `const` : creates a immutable variable (i.e: cannot be reassigned after creation)

1 Section 2 - Javascript Refresher

1.0.1 Import and Export

```
<script src="../path/to/js" type="module"></script>
```

More specifically using import and export allow us to import and export variables, functions, objects, etc from one .js file into another for example:

```
|-- file1.js
|-- file2.js
|   |--folder
|   |   |-- file3.js
```

```

/* File 1 */
export let var = "string";

/* File 2 */
import { var } from 'file1.js'

```

```
/* File 3 - React */
import { var } from 'file1'
```

We can directly export values by using the `default` keyword: `export default 'thing'`. To import said 'thing', we simply remove the curly braces. If you are exporting multiple things from a .js file, instead of explicitly importing by seperating with a comma, we can do:

```

/* import file */
export default 'apikey';
export let password = '123';
export let username = 'user';

/* import file */
import * as util from 'path/to/importfile.js';

func(util.password) /* or util.username or util.default etc
```

1.0.2 Variables, Operators

`const` : creates a immutable variable (i.e: cannot be reassigned after creation)
`let` : creates a mutable variable

1.0.3 Functions

Defining Standard Functions

```
function somefunc(param1, paramN) {
  console.log(`Some func with ${param1}, and ${paramN}`);
}
```

```
//Arrow functions are particularly useful with anonymous functions (functions that
↪  do not have a 'name')

export default (param1, paramN) => {
  console.log(`Some func with ${param1} and ${paramN}`);
};
```

- If a arrow func has one parameter the parentheses can be omitted: `param1 => {}`
- If a arrow func only contains a return statement and no other logic, you can omit the curly braces: `param1 => param1 + 2`

2.1 If returning a JS object then parentheses must be wrapped around the object defintion: `number => ({'age' :  number})`

1.0.4 Objects & Classes

- Console logging objects outputs them in a JSON-like format:

```
treasure = "$1000"
date = "Dec 1st 2025"
const records = {
        key : treasure,
        date : date
}
console.log(records)

-----------------------------------
key : "$1000"
date : "Dec 1st 2025"
```

JS objects can also contain methods. Methods created in JS objects do not need the function keyword. Methods defined within a JS object can access the values, and/or other methods within the object using the `this` keyword

```
const records = {
    age : 24
    money : "$1000"

    strRepr(){
        console.log(`Age: ${this.age}, Net-worth: ${this.money}`);
    }
}
```

blueprints for JS objects can be defined using the `class` keyword, which can then later be used to create objects. Constructors for classes can be defined using the `constructor(){ }` method. In a constructor, object variables can be defined using the previously mentioned this keyword (similar to self in Python)

1.0.5 Arrays

Arrays are defined with square brackets: `const arr = [val1, val2, ..., valn]`

```
// @run-each
const array = ["tree", "house", "dog"];

array.push("cat"); // Similar to .append
const val = array.findIndex((param) => {
  /*
  findIndex can use a arrow function which takes at least one input parameter,
  ↪  after-which
  the function body can then perform logic on it
  */
  return param === "tree";
});

console.log(val);

// @run-each
```

```
array.map((item) => {
  /*
  Iterates through each item through an array and operate logic on it, useful for API parsing + JSX
    - Returns a new array
  */
  return item + "!";
});

// @run-each
//Can also remove the curly braces as such:
const newArray = array.map((item) => item + "!");

console.log(newArray);
/*
Map can map types to other types i.e: string in original array to JS objects
*/
const newArray2 = array.map((item) => ({ text: item }));

console.log(newArray);
```

Say we have an array defined as: `userData = ['Max', 'Schwarz']`, instead of assigning the array values to variables as such:

```
userData = ["Max", "Schwarz"];
const firstName = userData[0];
const lastName = userData[1];

// We can =>

const [firstName, lastName] = ["Max", "Schwarz"];
```

```

userDetail = {
  name : 'Max'
  age : 24
}

userName = userDetail.name
userAge = userDetail.age

//We can do this =>   *Note, that the variable names must match the field names of the object
const {name, age} = {
  name: "Max"
  age: 24
}

//An alias can be assigned to the variable name by: {fieldName : alias}

const {name: userName, age} = {
```

```
  name: "Max"
  age: 34
}

console.log(userName) // => "Max"

/*
Similarly, an object can be destructured within an object => Say we have a function: func(object), instead of accessing its attributes
with the dot notation we can destructure the object and use the fields as **local variables**
*/

function storeOrder(order) {
  console.log(`ORDER: ${order.id} : ${order.quantity} `)
}

function storeOrder({id, quantity}) {
  console.log(`ORDER: ${id} : ${quantity}`)
}
```

1.0.6 Spread Operator

The spread operator can be used to merge arrays elements:

```
const oldHobbies = ["reading"];
const newHobbies = ["sports"];

const mergedHobby = [...oldHobbies, ...newHobbies];
```

The spread operator can also be used on traditional objects:

```
const user = {
  name: 'Max'
  age: 34
}
const adminPerms = {
  isAdmin : true
}
const extendedUser = {
  ...user,
  ...adminPerms
}
```

1.0.7 Control Structures

Iterating through elements of an array has different syntax in JS then expected, assume we have the array: `const array = ['element1', 'element2', etc]` - We can iterate through the elements in the array like: `for (const arr of array) {some logic}`

```
names = ["Jax", "Ben", "Den"];
```

```
for (name of names) {
  console.log(name);
}
```

1.0.8 Passing Functions as Values

- A function can be passed as a value to a another function. To do this correctly, the function being passed as a value should have its parantheses omitted

```
// Using the setTimeout function

const handleTimeout = () => {
  console.log("shout here");
};

setTimeout(handleTimeout, 2000); //execute handleTimeout after 2000ms

/* We can also do this with anonymous functions */

setTimeout(() => {
  console.log("shout here");
}, 2000);

/* This can obviously also be done with non-built-in functions */

function log(logging) {
  logging();
}

log(() => console.log("logging something"));
```

Similar to other programming languages, functions can be defined inside another function (making it a private function - therefore not callable outside of the parent function)

```
function init() {
  function greet() {
    console.log("Hi");
  }

  greet();
}

init();

//=> 'Hi'
```

1.0.9 Reference vs. Primitive Values

Primitives cannot be modified (immutable), modifying a primitive doesn't change the underlying variable but rather generates a brand new variable of the same type

- Quick note on primitive: imagine we have an array defined as such `const arr = ['1', '2']`

We are able to modify this const using methods such as `arr.push('4')`, this won't raise an error because the const keyword defines a constant reference to the object and not the value of the object itself. All objects are mutable in Javascript therefore the properties of the object can be changed just not change the address the variable is referencing.

Good Analogy From Gemini

```
Imagine of a const object as a house with a street address
- You cannot change the address of the house but you can...
- Change the furniture, the occupants, etc.
```

1.1 Reference

Useful JS array methods:

- map - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map
- find | findIndex - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Ar
- filter - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter

More can be found on docs found on MDN