ECMAScript 6

Lecture 2

Agenda

- Sets
- Computed Properties
- Map, Filter, Find, Reduce
- Promise
- Async / Await
- Questions !

Sets

- In a mathematical sense, a Set is a group of values that are unique.
- ES6 has introduced a new object that can only contain unique values called a Set it's an Iterable object.
- We can create a Set by creating a new instance of the object Set()

```
let mySet = new Set();
console.log(mySet); // mySet {}
```

Sets

- We can pass an array when I am creating a set and this will remove the duplicate items (used a lot)

```
const names = ['Ahmed','Ali','Ibrahim','Ali'];
let mySet = new Set(names);
console.log(mySet); // Set(3) {'Ahmed', 'Ali', 'Ibrahim'}
```

Modifying Sets

- We can add items to a set using the function add, If you attempt to .add() a duplicate item to a Set, you won't receive an error, but the item will not be added to the Set

```
const names = ["Ahmed", "Ali", "Ibrahim"];
let mySet = new Set(names);
mySet.add("Mohamed");
console.log(mySet); // Set(4) {'Ahmed', 'Ali', 'Ibrahim','Mohamed'}
mySet.add("Ahmed");
console.log(mySet); // Set(4) {'Ahmed', 'Ali', 'Ibrahim','Mohamed'}
```

Modifying Sets

- We can delete items to a set using the function delete, if you try to .delete() an item that is not in the Set, you won't receive an error, and the Set will remain unchanged.

```
const names = ["Ahmed", "Ali", "Ibrahim"];
let mySet = new Set(names);
mySet.delete("Ahmed");
console.log(mySet); // Set(2) {'Ali', 'Ibrahim'}
```

Modifying Sets

- We can use for … of loop to go through the items of a set (we can also use forEach)

```
const names = ["Ahmed", "Ali", "Ibrahim"];
let mySet = new Set(names);
for (let name of mySet) {
  console.log(name);  // 'Ahmed', 'Ali'
}
```

Working with Sets

- .size property gets us the number of items in a set
- .has() check if a value exist or not in the set and return a boolean

```
const names = ["Ahmed", "Ali", "Ibrahim"];
let mySet = new Set(names);
mySet.size; //3;
mySet.has("Ahmed"); //True
mySet.has("Mohamed"); //False
```

Working with Sets

- .size property gets us the number of items in a set
- .has() check if a value exist or not in the set and return a boolean

```
const names = ["Ahmed", "Ali", "Ibrahim"];
let mySet = new Set(names);
mySet.size; //3;
mySet.has("Ahmed"); //True
mySet.has("Mohamed"); //False
```

Computed property names

- It can be a more complex expression and not a simple variable call.

```
let param = "size";
let config = {
  [param]: 12,
  [`mobile${param.charAt(0).toUpperCase()}${param.slice(1)}`]: 4
};
console.log(config);
```

Map

- Map is a method used to loop through an array in a functional style while modifying the array items.
- It takes one parameter: a callback function.
- That callback function takes one parameter: the current item.
- It return a new array with the applied modifications.

```
const newArray = array.map(function (item) {
  return item;
})
```

Map

- The logic you might apply to modify the item goes before the return keyword

```
const array = [1, 2, 3, 4];
const newArray = array.map(function (item) {
  item = item * 2;
  return item;
});
console.log(newArray); //[2,4,6,8]
```

Map

- Because of the new arrow notation, map functions allow us to neatly write code in one line:

```
const array = [1, 2, 3, 4];
const newArray = array.map((item) => (item = item * 2));
console.log(newArray); //[2,4,6,8]
```

Map

- We also get in the arguments of the callback, the index of each loop and the original array.

```
const array = [1, 2, 3, 4];
const newArray = array.map((item, index, array) => {
  return { item, index, array };
});
console.log(newArray); //(4) [{...},{...},{...},{...}]
```

Filter

- Filter loops through each item of an array and return a new array that fulfills a certain condition.

```
const numbers = [0, 1, 2, 3, 4, 5];
const eventNumbers = numbers.filter((item) => item % 2 === 0);
// Condition that should fulfilled
console.log(eventNumbers); //(3) [0,2,4]
```

Find

- Find worked the same way as filter except that it returns the first item that matches the condition.
- It’s used to look for a specific item.

```
const employees = [
  { id: 1, name: "Ahmed" },
  { id: 2, name: "Ali" },
  { id: 3, name: "Hassan" },
  { id: 4, name: "Magdy" },
];

const certainEmployee = employees.find((person) => person.id === 4);
console.log(certainEmployee); // {id: 4, name: "Magdy"}
```

Reduce

- Reduce is a function used to get one output from a series of inputs (array)

```
const people = [
  { id: 1, name: "Ahmed", age: 20},
  { id: 2, name: "Ali", age: 30},
  { id: 3, name: "Hassan", age: 35},
  { id: 4, name: "Magdy", age: 25},
];

const sumOfAges = people.reduce((acc,item) => acc + item.age, 0);
console.log(sumOfAges); // 110
```

Reduce

- It takes a call back function with 2 main params:
  - The accumulator
  - The current Item
- It takes a second param the initial accumulator value.
- The function loops through the array and each time, it returns a value, it saves it into the **accumulator**.
- The value that is returned in the end is the last value saved in the accumulator.

Reduce

- Array Flatten Example

```
const array = [
  [1, 2],
  [3, 4],
  [5, 6],
];
const flatArray = array.reduce((acc, item) => {
  return [...acc, ...item];
}, []);
```

Reduce

- Group Instances Example

```
const names = ["Ahmed", "Ali", "Alaa", "Mohamed"];
const nameInstances = names.reduce((acc, item) => {
  acc[item] = Object.keys(acc).includes(item) ?
acc[item] + 1 : 1;
  return { ...acc };
}, {});
```

Chaining Functions

```
const employess = [
  { fname: "Ahmed", lname: "Zaki", salary: 100, gender: "male" },
  { fname: "Mohamed", lname: "Mahmoud", salary: 300, gender: "male" },
  { fname: "Sara", lname: "Wael", salary: 200, gender: "female" },
  { fname: "Ehab", lname: "Hassan", salary: 150, gender: "male" },
  { fname: "Islam", lname: "Ali", salary: 400, gender: "male" },
  { fname: "Mona", lname: "Ahmed", salary: 400, gender: "female" },
];
const fullNameAndMaxSalary = employess
  .map((person) => ({
    ...person,
    fullname: `${person.fname} ${person.lname}`,
  }))
  .filter((person) => person.gender === "female")
  .reduce(
    (acc, person) => {
      acc = person.salary > acc.salary ? person : acc;
      return acc;
    },
    { salary: 0 }
  );
```

Promises

- JS is a single thread language, every line must be resolved before it movies to the next.
- You can either make a promise or consume a promise (most probably you will promises).
- In promise there are 3 states:
  - Pending
  - Resolved (Fulfilled)
  - Rejected

Promises

- let’s take an example:
    - https://www.digitalocean.com/community/tutorials/understanding-javascript-promises

```
const myPromise = new Promise((resolve, reject) => {
  if (something) {
    resolve(paramSuccess); // success
  } else {
    reject(paramFailure); // failure
  }
});
```

Consuming a promise

- Then() and Catch():

```
myPromise
  .then((success) => {
    console.log(success); // Success Param
  })
  .catch((error) => {
    console.log(error); // Failure Param
  });
```

Built-in Methods: All - Race

- promise.all([promise1, promise2]) is a function that take an array of promises to handle them one time.
- If one of these promises has a rejected then the all execute the catch () function.
- Race takes the result of the first promise in account.

Async / Await

- There’s a special syntax to work with promises in a more comfortable fashion, called ‘async/await’.
- It’s surprisingly easy to understand and use.
- The word ‘async’ before a function means one simple thing:
  - A function always return a promise.
  - Even if a function actually returns a non-promise value.

Async / Await

- The keyword await makes JavaScript wait until that promise settles and returns its result.
- It can only be used inside an async functions.

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Done!"), 1000);
  });
  let result = await promise;
  // Wait till the promise resolves(*)
  alert(result); // Done!
}
f();
```

Async / Await

- You can combine it with the arrow notation to implement really neat logic.

```
const getUsers = async () => {
  const response = await fetch("API_URL");
  const users = await response.json();
  console.log(users);
};
getUsers();
```

ANY QUESTIONS ?

LAB 2

1- Create an array of food (arrayOfFood)

['burger', 'pizza', 'donuts', 'pizza', 'koshary', 'donuts', 'seafood', 'burger']

A. Create a Set with values of this array.
B. Add 'pasta' to the set and log the set to the console.
C. Remove 'burger' from the set and log the set to the console.
D. Write a function that takes the set as a parameter and clear the set if it has more that 2 items,

LAB 2

2- Using the fetch API and async / await, get the data from that link:

https://www.json-generator.com/api/json/get/cfrfjgAOIy?indent=2

- Tip: using fetch function.

A. Format the array of the users by adding an extra **full_name** attribute to each item.
B. Only get the male users who are **older than 30**.
C. group the the filtered users by **nationality**.
D. Get the oldest person with a nationality of **'tur'**.