ES6 & TypeScript

Symbols

- Symbols are a new primitive type in ES6.
- Symbols are tokens that serve as unique IDs.
- They are created via a factory function Symbol() as follows:

  *const mySymbol = **Symbol**('mySymbol');*

- Every time you call the factory function, a new and unique symbol is created. It means two Symbols can never be equal.

Using Symbols as enumeration constant

```
var COLOR_RED    = 'Red';
var COLOR_ORANGE = 'Orange';
var COLOR_YELLOW = 'Yellow';
switch (color) {
      case COLOR_RED:        return 1;
      case COLOR_ORANGE:  return 2;
      case COLOR_YELLOW:  return 3;
}
```

Using Symbols as unique property keys

Symbols are mainly used as unique property keys – a symbol never clashes with any other property key (symbol or string).

```
const MY_KEY = Symbol();

let obj = {};

obj[MY_KEY] = 123;

console.log(obj[MY_KEY]); // 123
```

Iterators

- Iterators are used to traverse a collection.
- JavaScript developers use for..in loop to iterate. However, in ES6 we use for..of loop.

```
let aryNames = ['Tom', 'Isabela', 'Emil'];
for(let name of aryNames) {
    console.log(name);
}
```

- Note that **for..in** iterates over property names while **for..of** iterates over property values.
- We can also iterate using Iterator object.

```
let itr = aryNames[Symbol.iterator]();
console.log(itr.next()); //{value: "Tom", done: false}
console.log(itr.next()); //{value: "Isabela", done: false}
console.log(itr.next()); //{value: "Emil", done: false}
console.log(itr.next()); //{value: undefined, done: true}
```

Custom Iterator

You can also define iterator for user defined object. Code for Fibonacci series using iterator:

```
let fibonacci = {
    [Symbol.iterator]() {
        let pre = 0;
        let cur = 1;
        return {
        next() {
            [pre, cur] = [cur, pre + cur];
            return {done: false, value: cur};
        }
    }
}
let itr = fibonacci[Symbol.iterator]();
console.log("Fibonacci no ", itr.next()); //call itr.next() multiple times…
```

Generators

- Generators are functions that can be paused and resumed.
- A function declared as **function*** returns a **Generator** instance.
- Generators are subtypes of iterators that include additional **next** and **throw** functions.
- Generators provide **yield** keyword to pause a function.

Simple Generator

```
function* myGenerator() {
        console.log("1st line");
        yield(555);
        console.log("2nd line");
        yield("End");
}
export function testGenerators() {
        let gen = myGenerator();
        console.log(gen.next());
        console.log(gen.next());
}
```

Output:

1st line

Object {value: 555, done: false}

2nd line

Object {value: "End", done: false}

Generator for Fibonacci series

```
function* fibonacci() {
    let pre = 0;
    let cur = 1;
    for(;;){
        [pre, cur] = [cur, pre + cur];
        let reset = yield cur;
        if(reset) {
            pre = 0;
            cur = 1;
        }
    }
}
let seq = fibonacci();
console.log(seq.next().value); //call multiple times
console.log(seq.next(true).value); //resets the Fibonacci series to one.
```

Output:

1

2

3

5

1

2

3

Map & Set

- ES6 has given support for collections in JavaScript. Now we have two data structures:
  1. Map &
  2. Set

Map

Map data structure allows us to create data with key-value pairs.

```
let map = new Map();
map.set('foo', 123);
map.set(‘bar', 222);
console.log(map.get('foo'));
console.log(map.has('foo'));
console.log(map.delete('foo'));
console.log(map.has('foo'));

map.clear();

console.log(map.size);
```

Iterating over a Map

- **Iterating over keys:**

```
for (let key of map.keys()) {
        console.log(key);
}
```

- **Iterating over values:**

```
for (let value of map.values()) {
        console.log(value);
}
```

- **Iterating over entries (key, value both)**

```
for (let entry of map.entries()) {    console.log(entry[0], entry[1]);  }
for (let [key, value] of map.entries()) {    console.log(key, value);  }
```

Spreading a Map

```
let map = new Map([
[1, 'one'],
[2, 'two'],
[3, 'three']
]);
let arr = [...map.keys()]; //Map spreading
console.log(arr);
```

Output: [1,2,3]

WeakMap

- WeakMap is a map that doesn't prevent its keys from being garbage-collected. That means that you can associate data with objects without worrying about memory leaks.
- WeakMap is a data structure whose keys must be objects.
- WeakMap has same API as Map.
- You should not perform following operations on WeakMap:
  - You should not iterate over contents i.e. keys, values or entries.
  - You should not clear WeakMap.

Set

- Set data structure stores data with no duplicate value.

```
let set = new Set();
set.add('red');
console.log(set.has('red'));
set.delete('red');
console.log(set.has('red'));
set.clear();
console.log(set.size);
```

- Set can also be created as follows:

```
let set = new Set(['red', 'green', 'blue']);
```

- Converting set to an array:

```
let set = new Set(['red', 'green', 'blue']);
let arr = [...set]; //Spread operator
console.log(arr);
```

WeakSet

- WeakSet is a set that doesn’t prevent its elements from being garbage-collected.
- WeakSet doesn’t recommend for iteration, looping, or clearing.

Thank You!

US – Corporate Headquarters

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

20 Broadwick Street
Soho, London
W1F 8HT, UK

89 Worship Street
Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

India

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076

Pune
5th Floor, Amar Paradigm
Baner, Pune 411 045

Kolkata
2B, 12th Floor, Tower 'C'
Rajarhat, Kolkata 700 156

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road,
Koramangala 1st Block,
Bengaluru (Bangalore) 560034

Gurgaon
A/373rd Floor, Sigma Center
Gurgaon, Haryana 122 011s