Javascript Notes

"this" keyword:

1. in global space value of "this" depends on global object which can be acccessed through "globalThis" keyword. Now value of globalThis depends on different environments, like in DOM it is window.

2. Inside functions it depends on strict or non strict mode.

    - in strict mode its "undefined".
    - in non strict its globalThis. Why? because of "this substitution". "This substitution"??: whenever the value of "this" in "undefined" or "null" it becomes globalThis, but, but... only in non strict mode. So this substitution works only in non strict mode.
    - but in both strict and non strict mode if we provide a context, the value of this will depend on that context.

    ```
    "use strict";
    function a() {
        return this;
    }
    a(); // undefined
    window.a(); // window object.
    ```

3. inside objects "this" depends on the object

    ```
    const user = {
        name: "One",
        printName: function {
            return this.name // this refers to the user object
        }
    }
    console.log(user.printName) // One
    ```

    - but if we want to use different object as context use "call", "apply", "bind" methods.

    ```
    // example of "call"
    const user = {
    name: "One",
    printName: function {
    return this.name // this refers to the user object
    }
    }
    const anotherUser = {
    name: "Two"
    }
    ```

```
            console.log(user.printName.call(anotherUser)) // Two
        ```
    so we called the user's printName function but used anotherUser
    object as the context.
```

4. Arrow function don't have their own this, they borrow the value of this from their lexical context. Lexical context means where the function was defined. So when the function was defined what was the value of "this", that will be the value of "this" inside the arrow function.

```
const person = {
    name: "Alice",
    y: this,
    speak: () => {
        console.log("arrow", this);
    },
    speak2: function () {
        console.log("regular", this);
    },
    speak3: function () {
        // new lexical context
        const andarWala = () => {
            console.log("andar wala arrow", this);
        };
        andarWala();
    },
};

console.log(person.y); // window
person.speak(); // window -> coz value of "this" was window when the
function was defined.
person.speak2(); // person
person.speak3(); // person -> coz lexical context changed
```

Symbol:

1. A "Symbol" is a unique identifier.
2. To create it we use "Symbol" keyword

```
let sym = Symbol();
```

3. We can give it a description:

```
let sym = Symbol("sym");
```

4. They are guaranteed unique even if they share the same description.
```

```
let sym1 = Symbol("sym");
let sym2 = Symbol("sym");
console.log(sym1 === sym2); // false
```

5. Use it as hidden properties in objects. In this way they can't be overwritten in any way.

```
let sym = Symbol("sym");
user[sym] = "some kind of sym value";
// if someone else has their own `sym` Symbol, it wont overwrite
yours.

// below a different example
let user = { name: "Sarthak" };
user.sym = "Original sym value";
user.sym = "New sym value";
// overwritten
```

6. They are skipped in for...in loops

```
let sym = Symbol("sym");
let user = {
    name: "Sarthak",
    [sym]: 123,
};

for (let key in user) console.log(key); // name but no symbols

console.log("Suymbol only", user[sym]); // 123
```

7. What is we want same-named symbols? we use Global Symbols. A global symbol registry exists, we create a symbol there and access it by the same name and it will return the same symbol. To do this we use Symbol.for("key") syntax.

```
let sym = Symbol.for("sym");
let sameSymbol = Symbol.for("sym");
console.log(sym === sameSymbol); // true
```

Symbol.for tries to return the symbol, if not found it creates it first and then returns it.

Now the above returns the symbol name, what if we want to return a name by global symbol, we use Symbol.keyFor(name) syntax

```

```

```
const symOne = Symbol.for("one");
const symTwo = Symbol.for("two");

console.log(Symbol.keyFor(symOne), Symbol.keyFor(symTwo)); // "one",
"two"
```

Garbage collection:

1. It happens automatically in javascript, unlike some low level languages.
2. Its all about reachability. If some value is not reachable in the object tree it is garbage collected.
3. Any value is considered reachable if it can be reached from root by a reference or by chain of references.

   ```
   const user = { name: "one" };
   const anotherUser = user;
   user = null;
   console.log(anotherUser); // {name: "one"}
   ```

   now above the object is still reachable through anotherUser variable so its not garbage collected.
4. Main algorithm which is used in called "Mark-and-Sweep". As the names suggests, it starts from root and mark (remembers) it, then it goes to every node which can be visited through reference and marks them as well. It keeps doing this untill every reachable node is marked. Then the remaining unmarked are removed from the memory.
5. There are some optimizations done by javascript engines to make the process of garbage collection faster and not introduce any delays into the code execution. Below are some of those optimizations:
   - **General collection:** Objects are grouped as "old" and "new". New objects are checked often because most die quickly, old ones are checked less often to save time. New ones, if used again ana again is then put in that old category so they are also checked less often.
   - **Incremental Collection:** Memmory is cleaned in small parts, not all at once. This avoids big pauses during the code execution. So we, user, don't notice any pause.
   - **Idle-time Collection:** Cleaning happens when CPU is idle. This reduces any slowdown in running code.

     There are more optimizations, but these are main.

Transpilers and Polyfills

New syntax and features of languages don't work with older engines or browsers, to tackle this issue there are 2 tools that are used:

- **Transpilers**
- **Polyfills**

1. **Transpilers** are the piece of softwares that translates the source code to another source code. It means it takes modern syntax and rewrite it using older syntax so that older browsers can understand it. Eg. 'nullish coalescing operator' didn't existed in older version or Javascript, so it takes that and converts it into something else.

```
// before running the transpiler
const height = height ?? 100;

// after running the transpiler
const height = height !== undefined && height !== null ? height : 100;
```

   Usually programmers runs the transpilers in their own computer and deploys the transpiled code to the server.

   Babel is one of the most used transpilers.

2. **Polyfills** are softwares that take new features and rewrites them in older ways. Eg. `Math.trunc` don't work in older engines so a polyfill will be created such that it will convert it to older syntax.

```
if (!Math.trunc) {
    // if no such function
    // implement it
    Math.trunc = function (number) {
        // Math.ceil and Math.floor exist even in ancient JavaScript
engines
        // they are covered later in the tutorial
        return number < 0 ? Math.ceil(number) : Math.floor(number);
    };
}
```

Object to primitive conversions

1. functions like `alert(...)` or `String(...)` expects string to be passed, but happens when we pass object instead of strings? It gets auto converted to primitive values.
2. There are 7 primitive types: `string`, `number`, `bigint`, `boolean`, `symbol`, `null` and `undefined`.
3. All objects are `true` when converted to boolean.
4. Numeric happens when we apply mathematical functions, like subtracting dates to get the difference.
5. String conversion happens when we try to output the object with like `alert(user)`. How does javascript decide which conversion to apply?: **Hints** There are 3 hints "string", "number", "default".
    1. "string" is used when we do operations that expects "string" argument like alert
    2. same thing with "number", when math is being done.

3. "default" its a rare case, happens when operator is not sure what to use. Eg. "+" can be used with string and numbers both, same goes with `==`, `> <.

6. All built-in objects except for one case (Date object, we'll learn it later) implement "default" conversion the same way as "number".

7. To do the conversion, JavaScript tries to find and call three object methods:

    1. Call `obj[Symbol.toPrimitive](hint)` – the method with the symbolic key Symbol.toPrimitive (system symbol), if such method exists.
    2. Otherwise if hint is "string", try calling obj.toString() or obj.valueOf(), whatever exists.
    3. Otherwise if hint is "number" or "default", try calling obj.valueOf() or obj.toString(), whatever exists.

8. **Symbol.toPrimitive:**

```
let user = {
    name: "John",
    money: 1000,

    [Symbol.toPrimitive](hint) {
        console.log(`hint: ${hint}`);
        return hint == "string" ? `{name: "${this.name}"}` :
this.money;
    },
};

// conversions demo:
console.log(user); // hint: string -> {name: "John"}
console.log(+user); // hint: number -> 1000
console.log(user + 500); // hint: default -> 1500
```

9. If no `Symbol.toPrimitive` is there then javascript tries to find `toString` and `valueOf` methods.

    1. For the "string" hint: call toString method, and if it doesn't exist or if it returns an object instead of a primitive value, then call valueOf (so toString has the priority for string conversions).
    2. For other hints: call valueOf, and if it doesn't exist or if it returns an object instead of a primitive value, then call toString (so valueOf has the priority for maths).
    3. By default, a plain object has following toString and valueOf methods:
        - The toString method returns a string "[object Object]".
        - The valueOf method returns the object itself.

We can overwrite the `toString` and `valueOf`. Below eg. we will overwrite it to make sure it works like it normally works

```
let user = {
    name: "John",
    money: 1000,
```

```
    // for hint="string"
    toString() {
        return `{name: "${this.name}"}`;
    },

    // for hint="number" or "default"
    valueOf() {
        return this.money;
    },
};

console.log(user); // toString -> {name: "John"}
console.log(+user); // valueOf -> 1000
console.log(user + 500); // valueOf -> 1500
```

10. A conversion can return any type, like `toString` don't actually have to return "string". Then only mandatory thing it these methods must return a primitive.

11. As we know already, many operators and functions perform type conversions, e.g. multiplication * converts operands to numbers.

If we pass an object as an argument, then there are two stages of calculations:

1. The object is converted to a primitive (using the rules described above).
2. If necessary for further calculations, the resulting primitive is also converted.

```
let obj = {
    // toString handles all conversions in the absence of other methods
    toString() {
        return "2";
    },
};

console.log(obj * 2); // 4, object converted to primitive "2", then
multiplication made it a number
```

```
let obj = {
    toString() {
        return "2";
    },
};

console.log(obj + 2); // "22" ("2" + 2), conversion to primitive returned a
string => concatenation
```

Map

1. Map are like objects but main difference is it allows keys of any types. Below are the methods that can be applied to a Map.

    - `new Map()` – creates the map.
    - `map.set(key, value)` – stores the value by the key.
    - `map.get(key)` – returns the value by the key, undefined if key doesn't exist in map.
    - `map.has(key)` – returns true if the key exists, false otherwise.
    - `map.delete(key)` – removes the element (the key/value pair) by the key.
    - `map.clear()` – removes everything from the map.
    - `map.size` – returns the current element count.

```
let map = new Map();

map.set("1", "str1"); // a string key
map.set(1, "num1"); // a numeric key
map.set(true, "bool1"); // a boolean key

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
console.log(map.get(1)); // 'num1'
console.log(map.get("1")); // 'str1'

console.log(map.size); // 3
```

2. Using objects as keys

```
let john = { name: "John" };

// for every user, let's store their visits count
let visitsCountMap = new Map();

// john is the key for the map
visitsCountMap.set(john, 123);

console.log(visitsCountMap.get(john)); // 123
```

3. Iteration:

    - `map.keys()` – returns an iterable for keys,
    - `map.values()` – returns an iterable for values,
    - `map.entries()` – returns an iterable for entries [key, value], it's used by default in `for..of`.

```
let recipeMap = new Map([
    ["cucumber", 500],
    ["tomatoes", 350],
    ["onion", 50],
]);
```

```
// iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) {
    console.log(vegetable); // cucumber, tomatoes, onion
}

// iterate over values (amounts)
for (let amount of recipeMap.values()) {
    console.log(amount); // 500, 350, 50
}

// iterate over [key, value] entries
for (let entry of recipeMap) {
    // the same as of recipeMap.entries()
    console.log(entry); // cucumber,500 (and so on)
}
```

Map also have inbuilt `forEach`

```
myMap.forEach((value, key, map) => {
    console.log(value, key, map); // cucumber: 500 etc, map is the map
itself here.
});
```

4. Creating map from object

```
let obj = {
    name: "John",
    age: 30,
};

let map = new Map(Object.entries(obj));
```

5. Creating Object from map:

```
const myMap = new Map([
    ["one", 1],
    ["two", 2],
]);
const myObj = Object.fromEntries(myMap);
```

But as we know map can have object as keys, so if we create object from such a map, key will be `[object Object]`.

Set

1. In a Set a value can accur only once. Its main methods are

    - `new Set([iterable])` – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
    - `set.add(value)` – adds a value, returns the set itself.
    - `set.delete(value)` – removes the value, returns true if value existed at the moment of the call, otherwise false.
    - `set.has(value)` – returns true if the value exists in the set, otherwise false.
    - `set.clear()` – removes everything from the set.
    - `set.size` – is the elements count.

    ```
    const mySet = new Set();
    mySet.add("one");
    mySet.add("two");
    mySet.add("one");
    for (const val of mySet) {
        console.log(val); // one -> two.
    }
    // one won't be repeated again.
    ```

    Alternative to above is using array, but we would need to check the value inside itby `arr.find(...)` so the performance would be bad.

2. Iteration of `Set`:

    ```
    let set = new Set(["oranges", "apples", "bananas"]);

    for (let value of set) console.log(value);

    // the same with forEach:
    set.forEach((value, valueAgain, set) => {
        console.log(value, valueAgain, set);
    });
    ```

    In above 3 arguments are passed, set returns the set itself. But why we have value twice? Its to make it compatible with `Map`. This is strange, but in some scenarios this helps replacing `Map` with `Set` and vice-versa.

    - `set.keys()` – returns an iterable object for values,
    - `set.values()` – same as set.keys(), for compatibility with Map,
    - `set.entries()` – returns an iterable object for entries [value, value], exists for compatibility with Map.

WeakMap

1. We know if object is reachable it won't be garbage collected.

```
let a = { name: "john" };
const b = new Map();
b.set(a, "something");
a = null;
```

still john object is in memory but weak map differs here. They don't stop garbage collection.

2. In WeakMap keys must be objects, else it throws error.
3. If object that is being used as key has no reference left, it will be removed from the memory.

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // overwrite the reference

// john is removed from memory!
```

So if john only exists as the key of WeakMap, it will be removed from memory, thus removing it from the WeakMap too.

4. WeakMap don't use iteration and methods like `keys()`, `values()`, `entries()`. It has only following methods.
    - weakMap.set(key, value)
    - weakMap.get(key)
    - weakMap.delete(key)
    - weakMap.has(key)

    Why though? Because as javascript engine don't do garbage collection immediately, it might wait to do it. So current count of elements inside WeakMap is not known.

5. Use cases:
    - If we are using some other data like a third party library, and we want to associate some data with it but only untill the third party library data is present, then we can use WeakMap. As if the third party library data is removed, its associated value will also be removed from the WeakMap.
    - In Caching. Using Maps, lets say we have an object and we want to associate some data with it. But in the future that object is removed, but it wont be garbage collected and its association will still be present in the ccache. On the other hand if we use WeakMap for caching, once the object is garbage collected, its associated data will also get removed from cache.

WeakSet

1. `WeakSet` are kinda similar to `WeakMap` such that.

- It is analogous to Set, but we may only add objects to `WeakSet` (not primitives).
- An object exists in the set while it is reachable from somewhere else.
- Like Set, it supports `add`, `has` and `delete`, but not `size`, `keys()` and no iterations.

```
let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john);
visitedSet.add(pete);
visitedSet.add(john);

console.log(visitedSet.has(john)); // true
john = null;
console.log(visitedSet.has(john)); // false
```

JSON

1. We have 2 main methods to convert things to JSON and vice-versa.

   - `JSON.stringify(...)`
   - `JSON.parse(...)`

   JSON.stringify(...):

   ```
   const user = { name: "John", age: 20 };
   const str = JSON.stringify(user); // {"name": "John", "age": 20}
   ```

   Now JSON converts everything to string and it wraps them in double quotes. It supports following data types:

   - Objects
   - Arrays
   - primitives:
     - strings
     - numbers
     - booleans
     - null

   JSON is data only language independent specification, so some javascript object properties are skipped, mainly

   - functions
   - Symbolic keys and values
   - properties that store undefined.

```
let user = {
    sayHi() {
        // ignored
        alert("Hello");
    },
    [Symbol("id")]: 123, // ignored
    something: undefined, // ignored
};

console.log(JSON.stringify(user)); // {} (empty object)
```

Nested objects are automatically converted. The only limitation? Circular references are skipped.

```
let room = {
    number: 23,
};

let meetup = {
    title: "Conference",
    participants: ["john", "ann"],
};

meetup.place = room; // meetup references room
room.occupiedBy = meetup; // room references meetup

JSON.stringify(room); // Error: Converting circular structure to JSON
```

Full syntax: `JSON.stringify(value[,replacer, space])`

- **value:** value to encode
- **replacer:** array of properties to encode or mapping functions `function(key, value)`
- **space:** amount of space to use for formatting

```
let room = {
    number: 23,
};

let meetup = {
    title: "Conference",
    participants: [{ name: "John" }, { name: "Alice" }],
    place: room, // meetup references room
};

room.occupiedBy = meetup; // room references meetup

console.log(JSON.stringify(meetup, ["title", "participants"]));
// {"title":"Conference","participants":[{},{}]}
```

The property list is applied to the whole object structure. So the objects in participants are empty, because name is not in the list. To include "name" pass "name" in array also.

Now lets say object is too big and we just want to remove couple of properties, then we can use replacer function.

```
let room = {
    number: 23,
};

let meetup = {
    title: "Conference",
    participants: [{ name: "John" }, { name: "Alice" }],
    place: room, // meetup references room
};

room.occupiedBy = meetup; // room references meetup

alert(
    JSON.stringify(meetup, function replacer(key, value) {
        alert(`${key}: ${value}`);
        return key == "occupiedBy" ? undefined : value;
    })
);

/* key:value pairs that come to replacer:
   :             [object Object]
   title:        Conference
   participants: [object Object],[object Object]
   0:            [object Object]
   name:         John
   1:            [object Object]
   name:         Alice
   place:        [object Object]
   number:       23
   occupiedBy: [object Object]
*/
```

The first call is special. It is made using a special “wrapper object”: {"": meetup}. In other words, the first (key, value) pair has an empty key, and the value is the target object as a whole. That’s why the first line is ":[object Object]" in the example above.

Custom toJSON:

```
let room = {
    number: 23,
};

let meetup = {
    title: "Conference",
```

```
    date: new Date(Date.UTC(2017, 0, 1)),
    room,
};

alert(JSON.stringify(meetup));
/*
{
   "title":"Conference",
   "date":"2017-01-01T00:00:00.000Z",  // (1)
   "room": {"number":23}               // (2)
}
*/
```

Here we can see that date (1) became a string. That's because all dates have a built-in toJSON method which returns such kind of string.

Now let's add a custom toJSON for our object room (2):

```
let room = {
    number: 23,
    toJSON() {
        return this.number;
    },
};

let meetup = {
    title: "Conference",
    room,
};

alert(JSON.stringify(room)); // 23

alert(JSON.stringify(meetup));
/*
{
   "title":"Conference",
   "room": 23
}
*/
```

JSON.parse(...):

- Syntax: `let value = JSON.parse(str[, reviver]);`

```
const obj = { one: 1, two: 2, three: 3, four: 4 };
const s = JSON.stringify(obj);

const j = JSON.parse(s, (key, value) => {
```

```
    return key === "three" ? undefined : value;
});
```

Above skips the "three" key value.

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

console.log(meetup.date.getDate()); // Error!
```

Gives error because date comes out as string. To fix this we use reviver.

```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function (key, value) {
    if (key == "date") return new Date(value);
    return value;
});

console.log(meetup.date.getDate()); // now works!
```
