Java script memory management

- **Program stack**: Where static variables and assignments live.
- **Call stack**: Where pirmitive variables live (Strings, int, float, boolean...)
- **Heap**: Where non-static or dynamic variables live.
    - Objects and arrays are allocated in the heap.

`Example 1`

```
  let f1 = { num: 3, den: 4 };
  //f1 variable is stored in callstack

  // right hand side of f1 gets stored in heap because we dont know how large its gona
be

  let f2 = f1;

  //f1 gets coppied as a pass by reference to f2 but not the actual reference itself
  f1.num = 5;

  console.log(f2);
  //output should be {num: 5, den: 4}

  f2.num = 5;

  console.log(f1);
  //output should still be {num: 5, den: 4}
```

`Example 2`

```
  let f1 = { num: 3, den: 4 };
  let f2 = f1;
  f1 = { num: 75, den: 100 };

  console.log(f2);
  //this should output {num:3, den:4};
```

`Example 3`

```
  let f1 = { num: 3, den: 4 };
  let f2 = f1;
  f1.num = 1;
  f1.den = 10;
  f1.toDecimal = function () {
    return this.num / this.den;
  };

  console.log(f2.toDecimal());

  //this should print out 0.10 by following pointers and references from the callstack
to the heap
```

So how can we just copy an object without copying the entire pointer?

`Example 4`

```
let f1 = { num: 3, den: 4 };
let f2 = { ...f1 };

f1.num = 1;
f1.den = 10;

console.log(`f1 = ${f1}`);
console.log(`f2 = ${f2}`);
//this will print the seperate objects because the memory addreses are different in
this case
/*
f1 = {num:1, den:10};
f2 = {num:3, den:4};

*/
```

Gotcha when using the spread operator...

`Example 5`

```
let f1 = {
  num: 3,
  den: 4,
  inverse: {
    num: 4,
    den: 3,
  },
};

let f2 = { ...f1 };
f1.num = 1;
f1.den = 10;
f1.invserse.num = 10;
f1.inverse.den = 1;

console.log(f1);
console.log(f2);

/*
You would expect this to print out f2 as it was before we changed f1 because we used
the spread operator. However, the objects inside of the object still gets referenced
as a pointer to a memory location in the heap instead of an object and thus, gets
changed.

*/
```

So how can we copy nested objects without copying the pointer to the memory location?

> ***Method 1:*** *Deep clone*

`Example 6`

```
/*
Instead of what we did in `Example 5`, we set f2 to the serialzed f1 object by using a
node module called ('v8');
*/

const v8 = require("v8");

const deepClone = (x)=>{
    return v8.desirealize(v8.serialize(x));
}
let f2 = deepClone(f1);
```

> ***Method 2:*** *JSON stringify*

`Example 7`

```
//This method is widely used in javascript but only works in javascript

const deepClone = (x) => {
  return JSON.parse(JSON.Stringify(x));
});

let f2 = deepClone(f1);

/*
This basically copies a json string instead of the memory locations or pointers
allowing you to succesfully copy nested objects without copying the nested pointers.
After turning it into a string using JSON.stringify you parse it back into its object
notation with JSON.parse(x)
*/

console.log(f2);
//This outputs the object without any conflicting pointers
```