

Week 2: JavaScript Memory Management

Ross Emile Aparece

Class 3

02/04/2025

- Standard syntax for an object is key value pairs

```
let myfrac = {num:1, den:4};
```

- Keys can be any type of string value but there are restrictions
- Do not need quotes for a property
- Values can be any data type

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

- Functions are called using ()

```
myfrac.toDecimal;  
returns f(){return this.num / this.den}  
  
myfrac.toDecimal();  
returns 0.25;
```

- Deconstructing assignment

```
let downloaded = {a:5, b:7, c:8, z:89};  
  
console.log(b);  
console.log(z);
```

- Spread operator convert from container type to parameter list

```
let arr1 = [1, 2, 3, 4 , 5];
let arr2 = [8, 9, 10]

[...arr1, 6, 7, ...arr2, 10, 11, 12,13]
returns (13) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10 , 11, 12, 13]
```

- Informally removes the outermost braces and turns it into a comma seperated list
- Works with any container type, common use case:

```
let downloaded = {a:5, b:7, c:8, z:89}
let augmented = {...downloaded, d:100}
```

- Spread operator cannot be used anywhere

```
...arr1
let arr1 = [1, 2, 3, 4 , 5];
let arr2 = [8, 9, 10]

//but this works:

[...arr1]
let arr1 = [1, 2, 3, 4 , 5];
let arr2 = [8, 9, 10]
```

- Assume downloaded is a massive array

```
Math.max(downloaded);
//Returns NaN because it wants a parameter list

Math.max(...downloaded);
//Converts the array to a parameter list allowing the
function to work properly
```

- Rest operator ... (Same symbol as spread operator)
 - Context lets us determine the difference between the two
- Converts from parameter list to array
- Scenario: assume download is a massive array

```
let download = [5, 6, 7, 8, 9, 10, 11];
let [first,second, ...r] = download;
```

```

console.log(first);
console.log(second);

console.log(r);
//returns the array containing the rest of the numbers

```

- Second use case:

```

//Use a rest operator for an infinite amount of numbers
function myMax(...n){
  let max_candidate = n[0];
  for(let i = 1; i < n.length; i++){
    if(n[i] < max_candidate){
      max_candidate = n[i];
    }
  }
  return max_candidate;
}

myMax(1, 2, 3, 4, 5, 6, 7, 8, 9);
//Should return 9;

```

- Allows a function to accept any amount of inputs

- Topic 2 start

```

let n1;
n1 = 31;
let n2;
n2 = n1;
n1 = 32;
console.log(n2);

```

	ID	ADD	VAL
STACK	n1	0x01	undefined 31 32
	n2	0x02	undefined 31

Class 4

02/06/2025

- Shallow Copy example 2

```
let f1 = {num: 3, den: 4, inverse:{num:4, den:3}}
let f2 = {...f1};
f1.num = 1;
f1.den = 2;
f1.inverse.num = 2;
f1.inverse.den = 1;

console.log(f2);
//returns 3/4 and 2/1
```

- We want 3/4 and 4/3
- Not a true copy hence SHALLOW copy

	ID	ADD	VAL
STACK	f1	0x01	undefined 0xA2
	f2	0x02	undefined 0xA3

	ADD	VAL
HEAP	0xA1	{num:4 2, den:3 1}
	0xA2	{num:3 1, den:4 2, inverse: 0xA1}
	0xA3	{num:3, den: 4, inverse: 0xA1}

- True copy

```
let f2 = structuredClone(f1);
```

- structuredClone should be used in the server environment
- However not every browser supports structureClone

- How do we target browsers on an older version?

```
let f2 = JSON.parse(JSON.stringify(f1));
```

- Converts object to a JSON string and converts it back into an object
- Useful to target legacy systems
- Doesn't preserve a lot of data types
 - * Functions may be dropped

* Complex data types may be dropped

- Scopes:
 - Global scope, function scope, block scope
 - Lexical scope
 - Module scope (not discussed)
- Lexical is more of theory than actual scope
- Block scope will be used within this class
- Global Scope:
 - Not recommended due to namespace pollution
 - There is no global keyword

```
function myscope(){  
    x = 5;  
}  
myscope();  
console.log(x);
```

- x is a global variable in this scenario
- Not using let, var, const makes the variable global

```
function myscope(){  
    var x;  
    x = 5;  
}  
myscope();  
console.log(x);
```

- First use of the variable decides if its global or not
- Global variables are bad because namespace pollution where variable name is no longer available
- Makes it hard to maintain even if you are working alone
- There are some specific usecases: building a language, building a library, etc.
 - If you are building for a developer and not an end user
- Function Scope:

- Exists anywhere within the function

```
function myscope(){
  var x = 5;
}
myscope();
console.log(x);
//prints out undefined
```

```
function myscope(){
  console.log(x);
  var x = 5;
}
myscope();
```

```
function myscope(){
  console.log(x);
  //var x = 5;
}
myscope();
```

```
function myscope(){
  if (false) {
    var x = 5;
  }
  console.log(x);
}
myscope();
//prints out undefined
```

```
function myscope(){
  if (false) {
    //var x = 5;
  }
  console.log(x);
}
myscope();
//doesn't work
```

- Variables defined in function scope have preprocessing
 - Moves the declaration to the top of the function
 - The variable is accessible anywhere within the function
- Block Scope (Recommended):

- Using let or const

```
function myscope(){
  if (true) {
    let var x = 5;
  }
  console.log(x);
  //x cannot be accessed outside the braces
}
myscope();
```

- Scoped within the braces
- No hoisting at all

```
function myscope(){
  if (true) {
    let var x = 5;
    console.log(x);
  }
}
myscope();
```

- You don't even need anything before braces

```
{
  let var x = 5;
  console.log(x);
}
```

- Not a real use case

- Lexical scope:

```
function a(){
  let data1 = 1;
  function b(){
    let data2 = 2;
    function c() {
      let data3 = 3;
      console.log(data1, data2, data3);
    }
    c();
    console.log(data1, data2);
  }
  b();
}
```

```
}  
a();
```

- You can access the ancestors variables
- Only works inner to outer

- High Order Functions and Closures

- Either/or/and:
 - * Function that takes another function as an input
 - * Function that returns a function as output
- Most common is map function

```
[1,2,3,4,5,6,7].map(someFunction);
```

- Maps the function to each element of the array and returns a new array with the function applied to the elements

```
[1,2,3,4,5,6,7].map(x => x + 1);
```

- Returns an array with each element + 1
- Create adder function

```
function create_adder(x){  
  return function(y){  
    return x + y;  
  }  
}  
const add12 = create_adder(12);  
add12(10);
```

```
function create_adder(x){  
  return function(y){  
    return x + y;  
  }  
}  
const add5 = create_adder(5);  
add12(100);
```

- Sends back a function that binds x and has a floating variable y that we set

- We can have both a function that accepts a function as an input and returns a function as an output

```
function outer(a){
  let b = 20;
  let unused = 50;
  return function inner(c){
    let d = 40;
    return `${a}, ${b}, ${c}, ${d}`;
  }
}
let i = outer(10);
let j = i(30);
console.log(j);
```

- Returns 10, 20, 30, 40
- We are accessing something outer when we are no longer within that scope
- Closure is the preservation of access to variable that another function (or block of code) depends on
 - * Does not preserve the data

```
return function inner(c){
  let d = 40;
  return `${a}, ${b}, ${c}, ${d}`;
}
Env:{a:0x03, b:0x05}
//environmental variables that lets us maintain access that
  allows us to break scope
```

- Closure ensures that the garbage collector does not delete those elements