

CS355 - Topic 2: JavaScript Memory Management

Dates: February 2, 2026 and February 11, 2026

February 2:

Guess the Output Questions:

Question 1:

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

Output:

```
31
```

Explanation: The following table is a visualization of the *Call Stack*

Identifier	Address	Value
n1	0x01	31 32
n2	0x02	31

- Note that all primitives always get stored in the call stack
- The value assigned to an identifier and its designated address **before** assignment is TDZ
- TDZ - A temporal dead zone (TDZ) is the area of a block where a variable is inaccessible until the moment the computer completely initializes it with a value (definition from codesweetly.com)
- Declaration: `let n1;`
- Assignment: `n1 = 31;`
- Re-assignment: `n1 = 32;`

Question 2:

```
let f1 = { num: 3, den: 4 };
let f2 = f1;
f1 = { num: 1, den: 2 };
console.log(f2);
```

Output:

```
{ num: 3, den: 4}
```

Explanation: The first table is a visualization of the *Call Stack* and the second table is the *Heap*

Identifier	Address	Value
f1	0x01	0xA1 0xA2
f2	0x02	0xA1
Address	Value	
0xA1	{ num: 3, den: 4 }	
0xA2	{ num: 1, den: 2 }	

- Note that objects get stored on the Heap
- The value of a variable storing an object is the memory address on the heap where the object is stored
- Key thing to note for this question is that `f1 = { num: 1, den: 2 }` is creating a new object rather than updating an existing one

Question 3:

```
let f1 = { num: 3, den: 4};
let f2 = f1;
f1.num = 1;
f1.den = 2;
console.log(f2);
```

Output:

```
{ num: 1, den: 2 }
```

- The main difference between this question and question 2 is that by using the dot operator in the following lines `f1.num = 1` and `f1.den = 2` we are updating the existing object rather than creating a new one

Question 4:

```
let f1 = { num: 3, den: 4 };
let f2 = f1;
f1.ToDecimal = function() { return this.num / this.den };
console.log(f2.ToDecimal());
```

Output:

0.75

Explanation: The first table is a visualization of the *Call Stack* and the second table is the *Heap*

Identifier	Address	Value
f1	0x01	0xA1
f2	0x02	0xA1
Address	Value	
0xA1	{ num: 3, den: 4, toDecimal: 0xA2 }	
0xA2	function() { return this.num / this.den }	

- Note that just like objects, function definitions also go on the heap
- `f1.toDecimal = function() { return this.num / this.den }` assigns this function to the object f1, we know the object f1 is stored in the heap at memory address 0xA1, so the object itself gets modified with an additional key value pair with the key being the function name and the value being the address on the heap where the function is defined
- In this specific case the code works when we run the function on variable f2 because the value of f2 on the stack is the memory address to the object on the heap that has the function

Question 5:

```
let f1 = { num: 3, den: 4 };
let f2 = { ...f1 };
f1.num = 1;
f1.den = 2;
console.log(f1);
console.log(f2);
```

Output:

```
{ num: 1, den: 2 }
{ num: 3, den: 4 }
```

Explanation:

Identifier	Address	Value
f1	0x01	0xA1
f2	0x02	0xA2
Address	Value	
0xA1	{ num: 3, den: 4, toDecimal: 0xA2 }	
0xA2	function() { return this.num / this.den }	

Address	Value
0xA1	{ num: 3 1, den: 4 2 }
0xA2	{ num: 3, den: 4 }

- The spread operator used in line 2 is creating a new object
- Remember: spread operator removes the outer most braces
- Using the spread operator we are creating what is called a shallow copy of the object meaning any changes to the copy don't effect the original and any changes to the original won't be reflected in the copy

Question 6:

```
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);
```

Output:

```
{ num: 3, den: 4, inverse: { num: 2, den: 1 } }
```

Explanation:

Identifier	Address	Value
f1	0x01	
f2	0x02	

Address	Value
0xA1	{ num: 4 2, den: 3 1 }
0xA2	{ num: 3 1, den: 4 2, inverse: 0xA1 }
0xA3	{ num: 3, den: 4, inverse: 0xA1 }

- This one is a bit tricky but just note that nested objects gets added to memory before the outer object. In this case the inverse object gets a space in memory first and then a space in memory for f1 as a whole and the value for the key inverse is the memory address where the inverse object is stored
- We get a bit of an unexpected output for this question due to the fact shallow copies do *not* work on nested objects. Thus, we don't have the re-assigned values for the num and den of f1 but we do have the re-assigned values of the child (inverse) because we use a reference to a memory address to denote its value

- To make a hard copy instead of a shallow copy you could opt for `let f2 = structuredClone(f1)` which copies the object *plus* any of its associated children

Scopes:

1. Global Scope (avoid)
2. Function Scope (avoid)
3. Lexical / Block Scope

Global Scope:

- Variables defined with no key words at declaration (`let`, `const`, or `var`)
- It is not safe to use this in practice as that variable will be available throughout the entirety of the program Example:

```
function scope(){
  x = 10;
}
scope(); // without this line the program would result in an error as x
would be undefined
console.log(x); // prints out 10
```

Function Scope:

- Variables defined with the keyword `var` can be used anywhere inside of the function Example:

```
function scope(){
  var x = 10;
}
scope();
console.log(x); // Error: x is not defined
```

- You can see the staggering difference adding the keyword `var` made to this code block as compared to the code block in global scope. Since `x` is defined as a function variable it is **NOT** accessible outside of the function even if the function is called.

Lexical / Block Scope:

- Variables declared using the keywords `let` or `const` only exist within the scope of the curly braces they were defined within Example:

```
function myFunction(){
  if(true){
    let x = 10;
  } // x stops existing at this line as this is the closing curly brace
of the scope in which it was defined
```

```
    console.log(x); // Error: x is not defined
}
```

- It is advised to use block / lexical variables when writing code in JavaScript

Higher Order Functions:

1. A function that takes another function as input
2. A function that returns another function as its output
3. Both 1 and 2

Higher Order Function Example:

```
function createAdder(x){
  return function(y){
    return x + y;
  }
}
```

- The function create adder takes a number as its input and returns to us a new function
- Essentially we are able to use the createAdder function to create functions

```
const addTen = createAdder(10);
console.log(addTen(50)); // outputs 60
```

- Breaking this down: `const addTen = createAdder(10)` we are calling the `createAdder` function to create a new function called `addTen`. When we called `createAdder` we gave a value for `x`. Now when we use the function `addTen` we are really passing in a value `y` for the original `createAdder` function.

Higher Order Function Memory Management:

Stack Frame Basics:

- Function call **invokes** a new stack frame
- First thing on the new stack frame is function arguments, followed by any variables declared inside the function
- A placeholder is set for the return pointer but its value gets updated last
- Treat the return line as an assignment (i.e. `return 7 === return = 7`) and then treat the right hand as you usually would when it comes to memory allocation

Sample Code:

```
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);

```

Code Explanation:

1. The function call `outer(10)` creates a stack frame with `a = 10`, `b = 20`, and `unused = 50`
2. When the `outer()` function is called, the `inner()` function is created, AND captures `a` and `b` in its closure (stored in the heap). `unused` is not captured as it is not used by the `inner()` function.
3. Once the `outer()` function call is complete, the corresponding stack frame for `outer` is popped from the stack, but closure keeps `a` and `b` alive
4. When the `inner()` function is called by `i(30)`, this creates a new stack frame with `c = 30` and `d = 40` and accesses variables `a` and `b` through closure
5. Returns the template string, so the final output would be `10,20,30,40`, and then the stack frame corresponding to `inner` is also popped from the stack

February 11:

Scope Review

- Global scope variables are declared without keywords and can be used anywhere in the program
- Function scope variables are declared using the keyword `var` and can be used anywhere within the function it was declared in
 - Note: Arguments passed into a function also are function scope variables
- Block / Lexical scope variables are declared using keywords `const` or `let` and its scope is between the set of curly braces it was declared in
- Lexical declaration cannot exist in a single line context. This is due to the fact `const` and `let` can only exist inside a block nested by curly braces. Without braces, JavaScript has no scope to give these variables thereby throwing a `SyntaxError`. Thus the following code throws an error:

```

if (true)
  let x = 5;

```

- The fix to the above error is very simple, just wrap the body of the if-statement in a set of curly braces like so:

```

if (true) {
  let x = 5;

```

```
}
```

What happens when we make a function call?

- JavaScript allows nested child functions to access needed parent variables by closure
- A function call invokes a new stack frame

Higher-Order Functions

- Case 1: A function that takes another function as input
- Example of Case 1:

```
[1, 2, 3, 4].map(x => x * 2); // [2, 3, 6, 8]
```

- The function `map` is a higher-order function as it takes a function as input
- Case 2: A function that returns another function as output
- Example of Case 2:

```
function adder(x){
    return function(y){
        return x + y;
    }
}
```

- The function `adder` is a higher-order function because it returns another function as its output
- Case 3: A function that takes another function as input AND returns another function as output
- Example of Case 3:

```
function not(func){
    return function(...x){
        return !func(...x);
    }
};

let isEven = x => x % 2 == 0;
let isOdd = not(isEven);
```

- The `not` function is a higher-order function. The input for the `not` function is another function with boolean return value and the output is a function that behaves the exact opposite of the original

Side note: The `isOdd` function is NOT "dependent" on `isEven`, meaning if after this code block we added `isEven = null;`, we can still use `isOdd()` without a problem. This is due to how the pointers work in the memory management. `isEven = null` is not removing the function definition

from the heap, so `isOdd()` is still able to access the function definition of `isEven()` using a pointer to its memory address

Memory Management

- Closure: Preservation of access to variables that a function depends on to exist. This is crucial when you have a nested function and the inner child function, needs parameters or variables declared in the outer parent function to execute
- The way we learn closure is that when the inner child function gets added to the heap memory, after the function definition there is an object called `env` where the property names are the variables the inner child function needs from the outer parent function and the values are pointers to these specific variables memory addresses. It is important to note we are not storing the value itself, but rather the memory address of the variable so that if it ever updates we get the most up to date value
- Garbage Collection: When we create variables, objects, or functions, JavaScript allocates space in the memory for them whether it be on the stack or the heap. When they're no longer reachable (nothing in your program can access them anymore), the garbage collector frees that memory automatically. We don't have to call the garbage collector, it runs by itself at random times.
- The garbage collector starts from what is referred to as the roots which are global variables, and any identifiers on the main call stack
- Everything that is referenced from the root, whether it be directly or through a chain of calls, is preserved. Everything else is garbage collected to free up memory