UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 4: Closures and Mental Models

Wednesday September 24, 2025

UMassAmherst | Manning College of Information & Computer Sciences

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Midterm 1 is next Tuesday (9/30) at 7-9pm
  - Exams from previous semesters can be found under Modules on Canvas
- Homework 4 has 2 parts, with separate deadlines, both after the midterm.
  - Homework 4a has been released (due October 5th)
  - Come to office hours for help!
- If you need to miss lab and have a valid reason according to the syllabus (medical, other personal) please fill out the questionnaire on Canvas before the start time of your lab.
  - Waking up late, bus was late are NOT valid reasons to miss lab.

Today's Goals

- Closures
- Mental Models

Exercise 1: Closures

- Write a function that takes as argument an array of Boolean functions, all with the same argument type T and returns a Boolean closure with an argument x of type T. The closure should return true if and only if more than half of the functions in the array return true for x.
- Use reduce for implementation.

```
function mostTrue<T>(
    funarr: ((arg: T) => boolean)[]
): (arg: T) => boolean {
    // TODO
}
```

Exercise 1: Solution

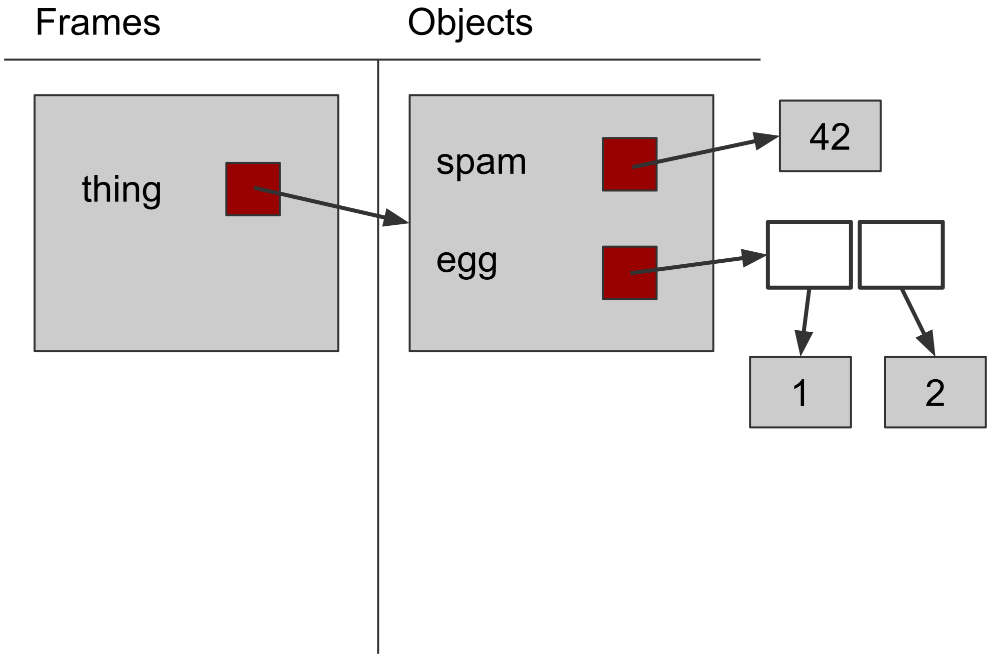
- Write a function that takes as argument an array of Boolean functions, all with the same argument type T and returns a Boolean closure with an argument x of type T. The closure should return true if and only if more than half of the functions in the array return true for x.
- Use reduce for implementation.

```
function mostTrue<T>(
    funarr: ((arg: T) => boolean)[]
): (arg: T) => boolean {
    return (x: T) => funarr.reduce((acc, f) => f(x) ? acc+1 : acc-1, 0) > 0;
}
```

Exercise 2: Mental Models

As you may have seen towards the end of Lecture 7, the memory diagram for the following code is depicted on the right.

```
let thing = {
    spam: 42,
    egg: [1, 2]
};
```

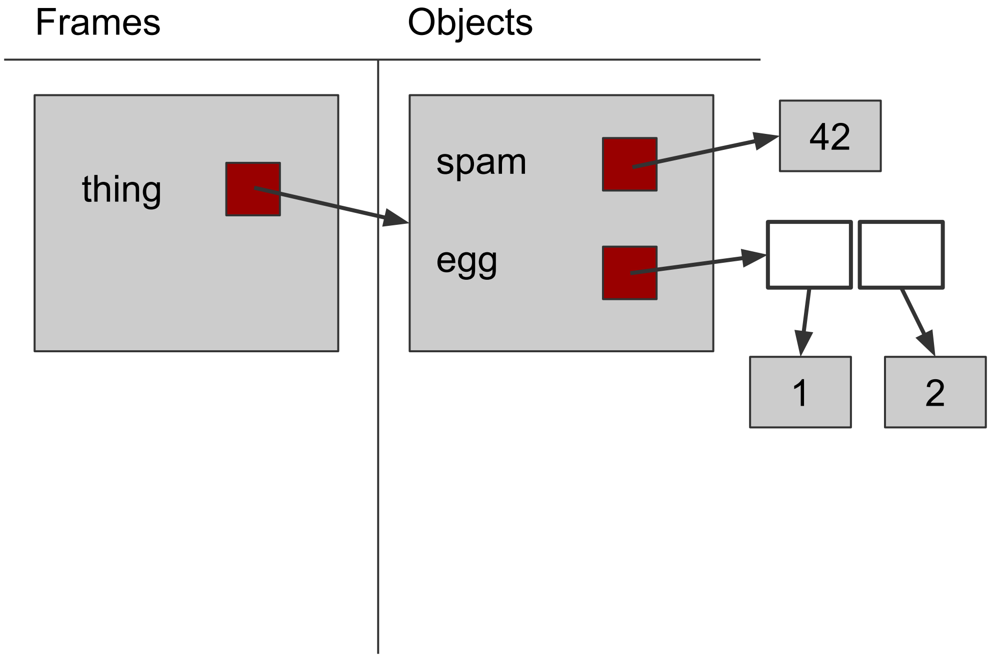


Exercise 2: Mental Models

```
1 let thing = {
2     spam: 42,
3     egg: [1, 2]
4 };
```

Draw the updated memory diagram after running lines 5-10 below:

```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```

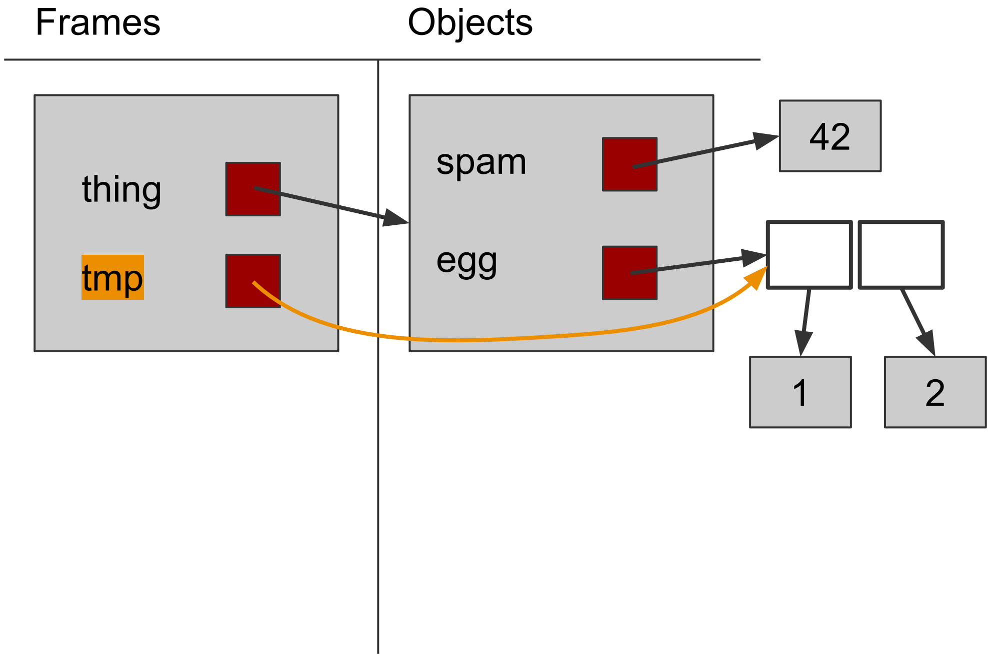


(lines 5-8 from lecture 7 exercise)

Exercise 2: Solution

Line 5: variable tmp is initialized and the value stored in thing.egg (reference to array [1,2]) is stored in tmp.

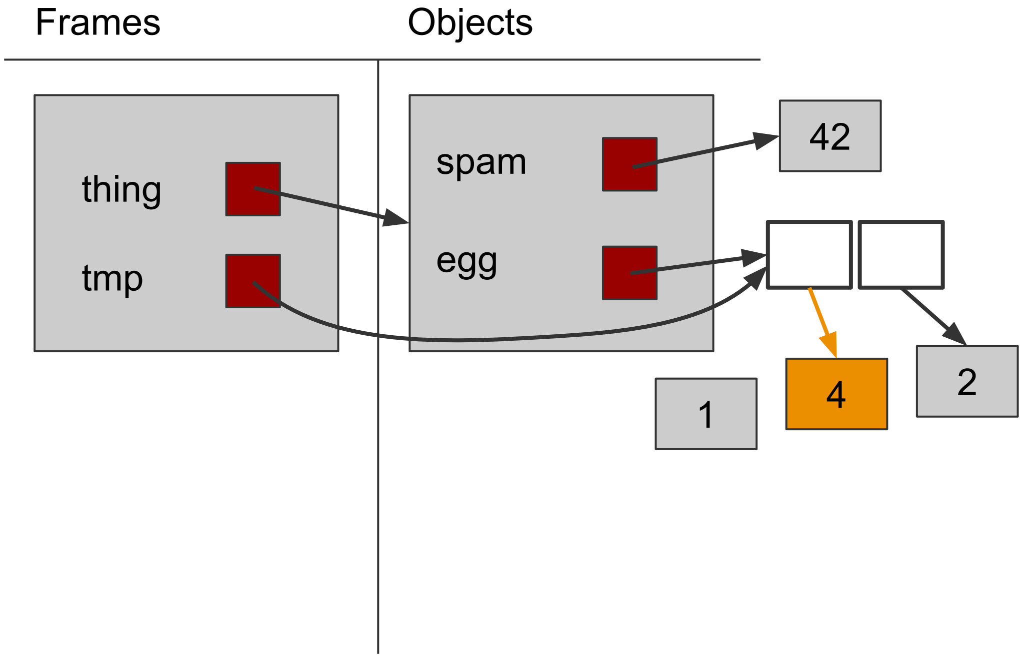
```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```



Exercise 2: Solution

Line 6: the element at index 0 in the array referenced by thing.egg is incremented by 3.

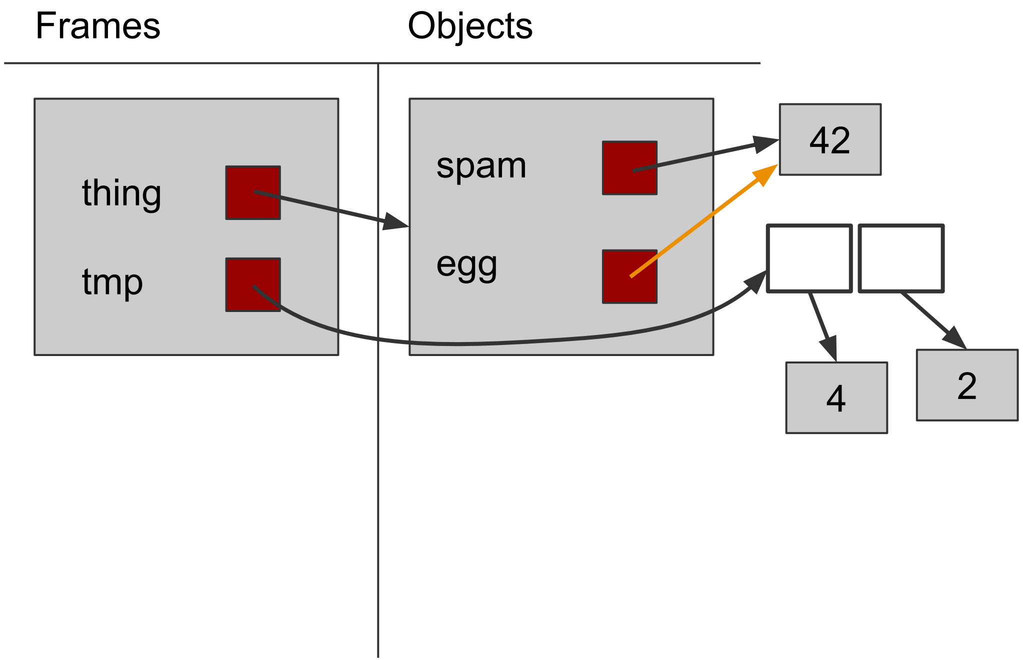
```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```



Exercise 2: Solution

Line 7: the egg property of object thing is set to the value stored in thing.spam, which is the number 42.

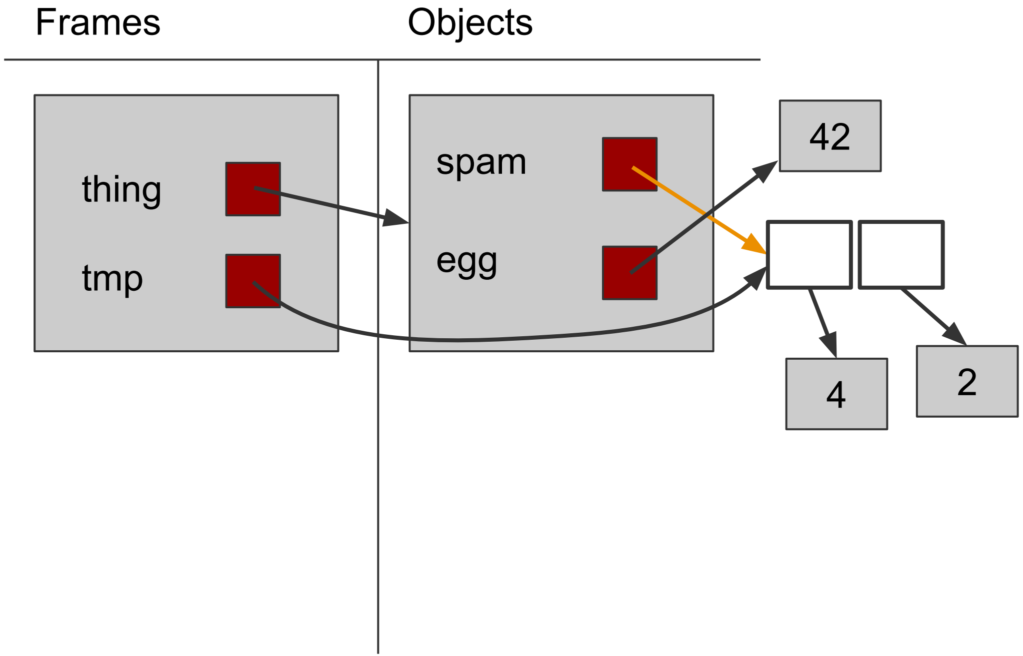
```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```



Exercise 2: Solution

Line 8: the spam property of object thing is set to the value stored in tmp, which is the reference to the array [4, 2].

```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```

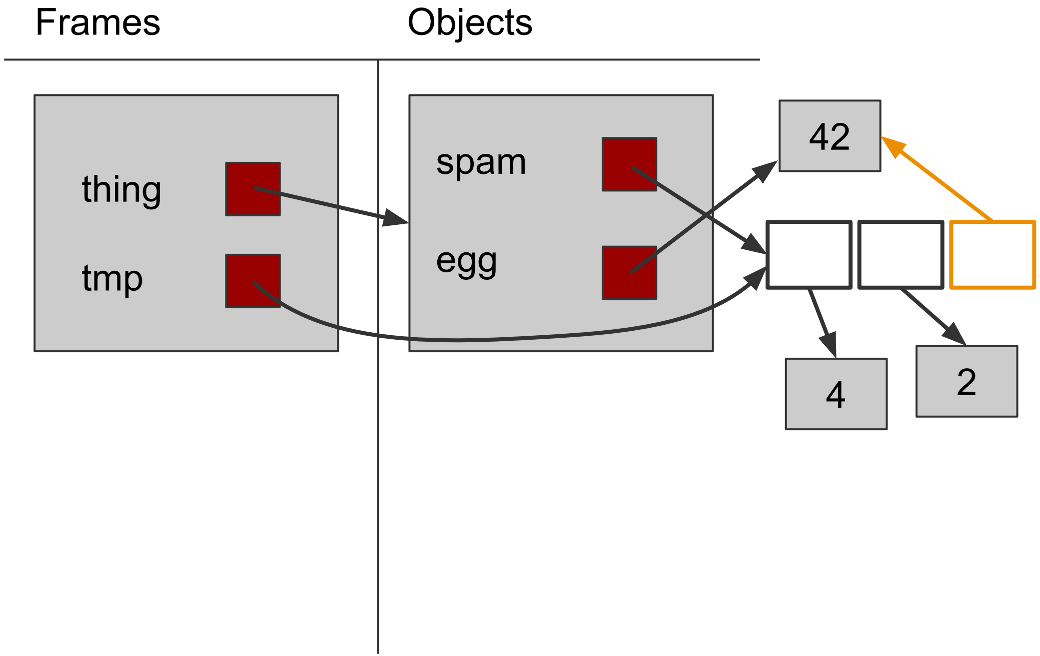


Note: lines 7 and 8 would error in Typescript!

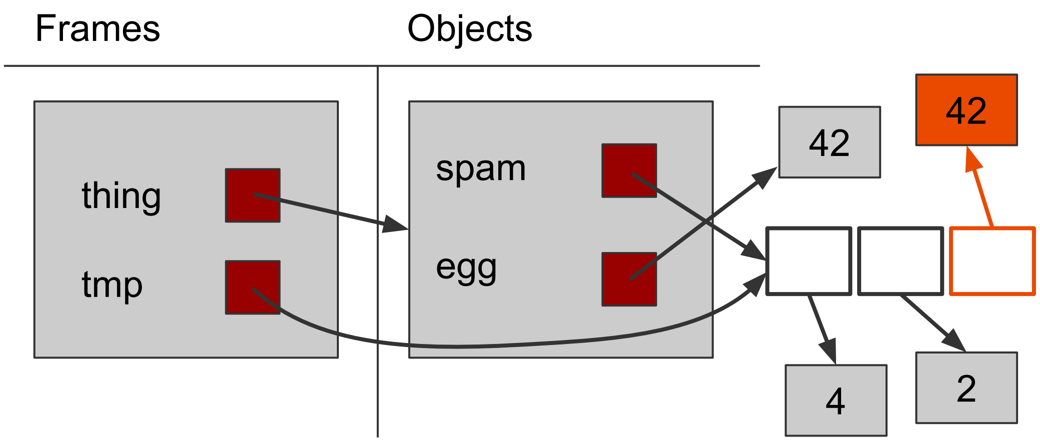
Exercise 2: Solution

Line 9: the value stored in thing.egg (42) is added as an element to the array referenced by thing.spam.

```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```



Exercise 2: Solution




```
let tmp = thing.egg;
thing.egg[0] += 3;
thing.egg = thing.spam;
thing.spam = tmp;
thing.spam.push(thing.egg);
thing.spam.filter(x => x < 5);
```

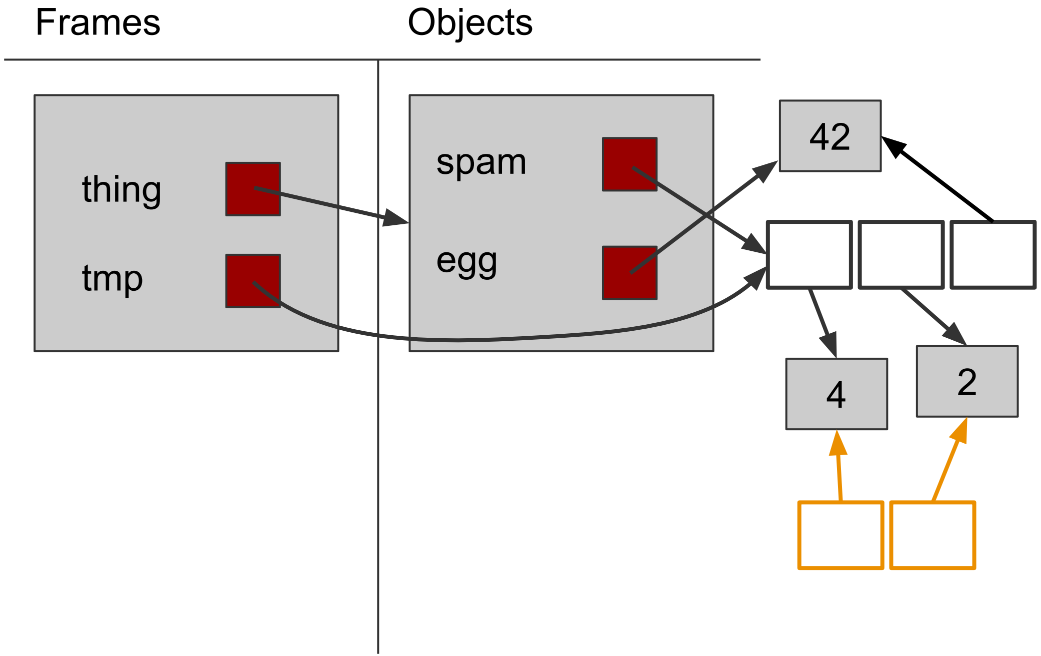
Why is the diagram on the left **incorrect**? Why is it **incorrect** for thing.spam[2] to point to a different 42?

thing.spam.push(thing.egg) means we are adding a new element to the array referenced by thing.spam, where this element points to the same value as thing.egg. So it should point to the **same** 42 as thing.egg.

Exercise 2: Solution

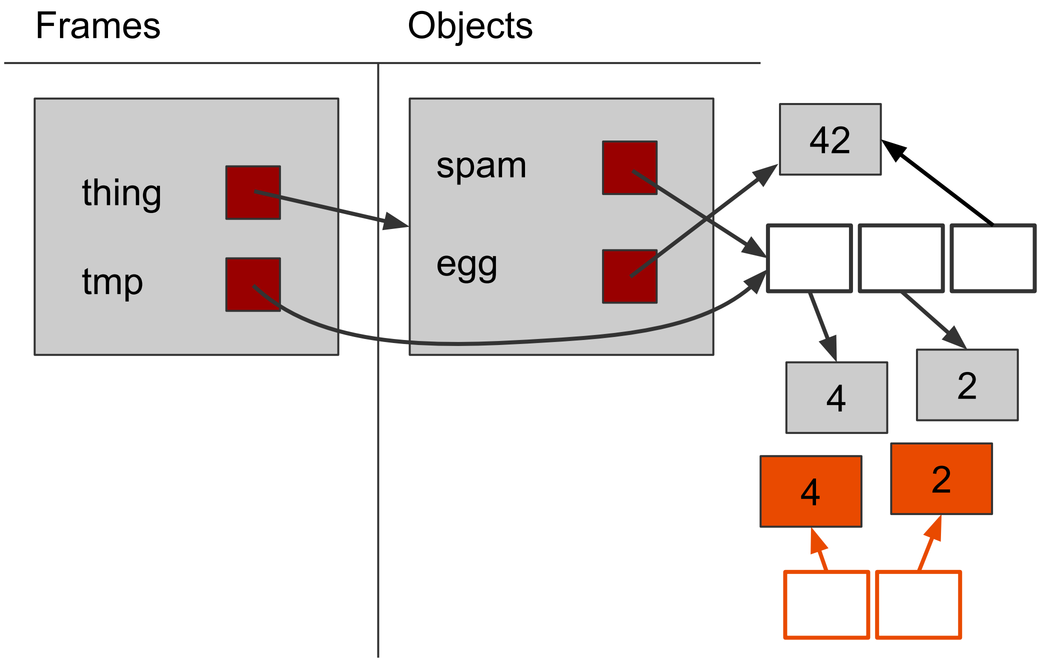
Line 10: filter is called on the array referenced by thing.spam ([4, 2, 42]). A new array is created, with only the elements less than 5 ([4, 2]). The reference to this array is never stored.

```
5  let tmp = thing.egg;
6  thing.egg[0] += 3;
7  thing.egg = thing.spam;
8  thing.spam = tmp;
9  thing.spam.push(thing.egg);
10 thing.spam.filter(x => x < 5);
```



UMassAmherst | Manning College of Information & Computer Sciences

Exercise 2: Solution

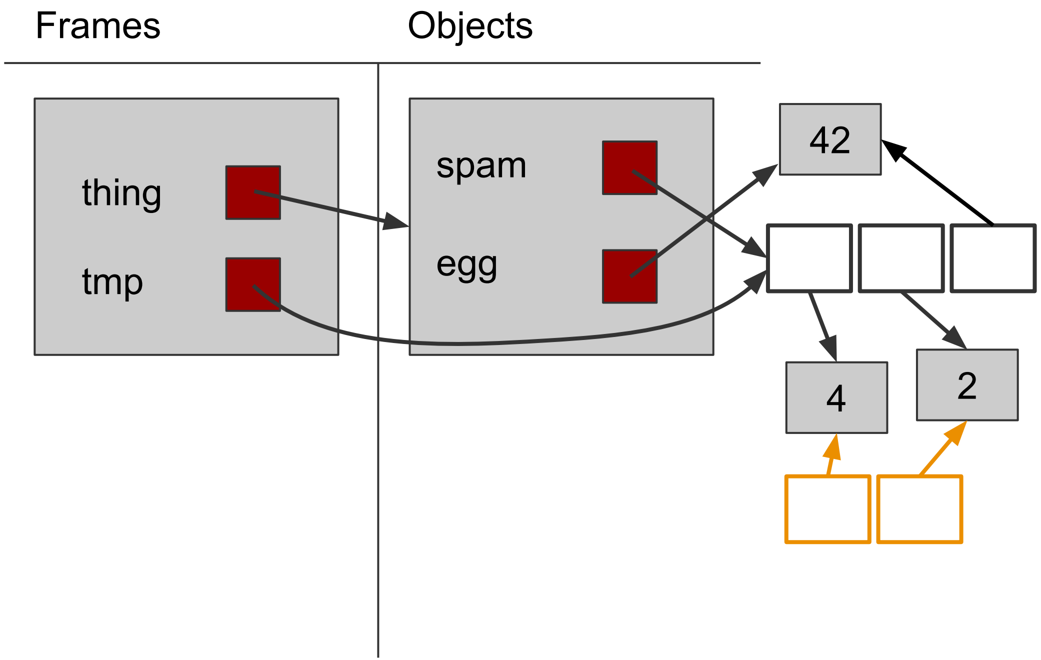


Why is the diagram on the left **incorrect**? Why is it **incorrect** for the elements of the new array to point to different numbers than those pointed to by the existing array?

With the way filter works, the newly created array will still point to the same numbers as the original array.

```
let tmp = thing.egg;
thing.egg[0] += 3;
thing.egg = thing.spam;
thing.spam = tmp;
thing.spam.push(thing.egg);
thing.spam.filter(x => x < 5);
```

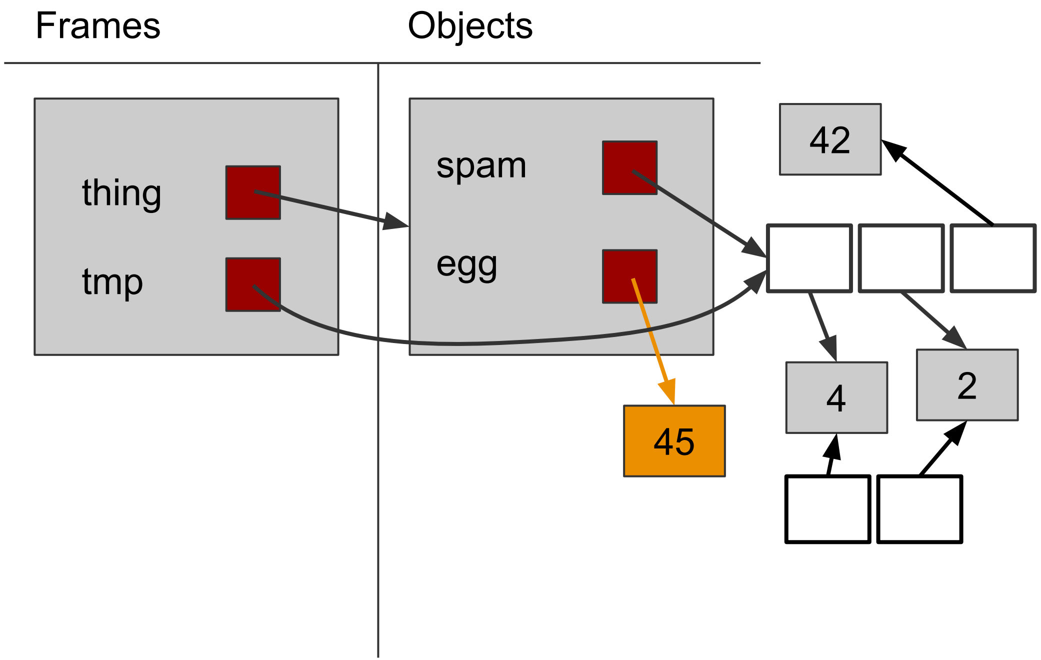
Exercise 2: Solution



What would happen if we now ran the following lines of code?

```
thing.egg += 3;
thing.spam[0] += 3;
tmp[1] += 3;
```

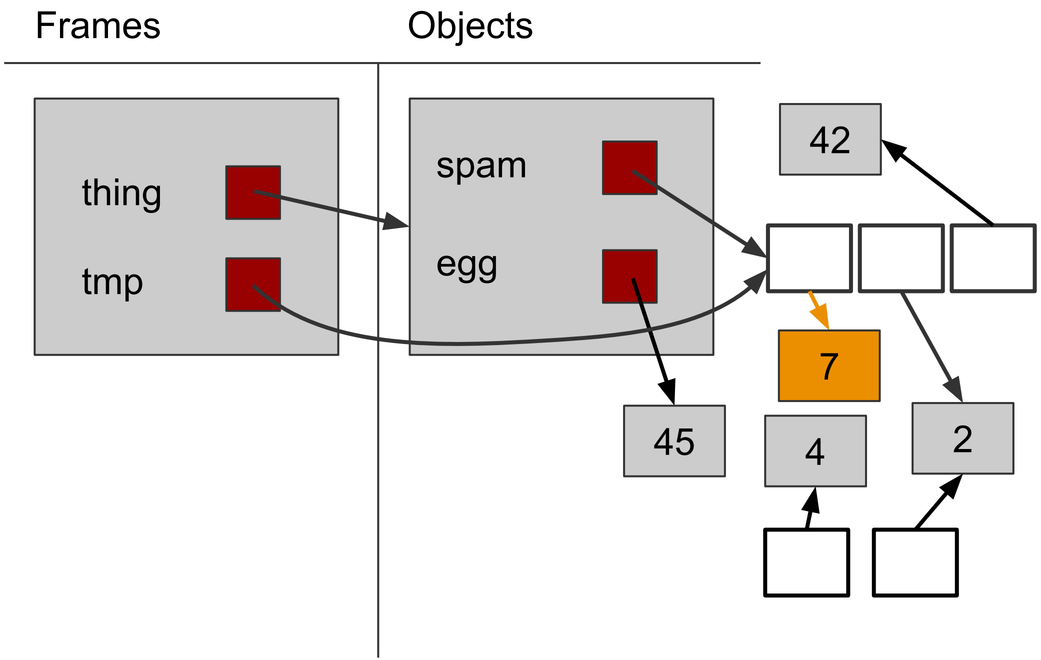
Exercise 2: Solution



Line 11: we increment thing.egg by 3, meaning thing.egg now points to a different number (45). This does not affect thing.spam[2]. The 42 that thing.egg originally pointed to is not modified to become 45, as numbers are immutable.

```
11 thing.egg += 3;
12 thing.spam[0] += 3;
13 tmp[1] += 3;
```

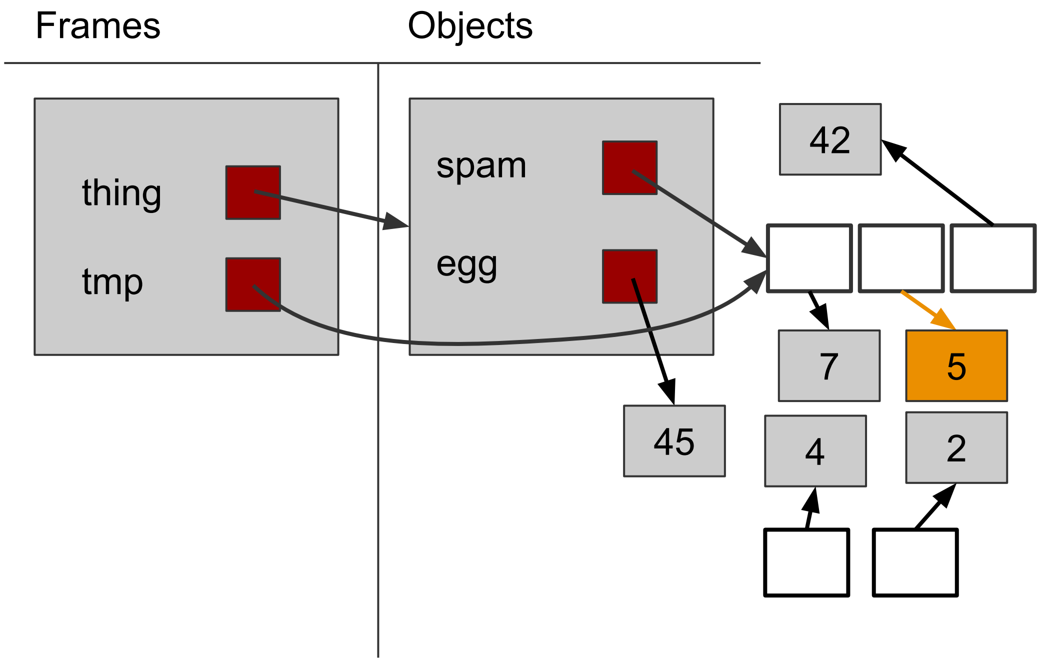
Exercise 2: Solution



Line 12: we increment the first element of the array referenced by thing.spam by 3. Now, thing.spam[0] points to 7. Similar reasoning applies from the previous slide for why the original 4 is not modified.

```
11 thing.egg += 3;
12 thing.spam[0] += 3;
13 tmp[1] += 3;
```

Exercise 2: Solution



Line 13: we increment the second element of the array referenced by tmp by 3. Now, tmp[1] points to 5. Similar reasoning applies from the previous slide for why the original 2 referenced by tmp[1] is not modified.

```
11 thing.egg += 3;
12 thing.spam[0] += 3;
13 tmp[1] += 3;
```

Exercise 3: More closures

- Write a function with no arguments that returns a closure with no arguments.
  When called the nth time (n >= 1), the closure should return
  the nth approximation for the number e: 1 + 1/1! + 1/2! + … + 1/n!
- Avoid needless recomputation in the factorial and in the sum.
- Example outputs: 2, 2.5, 2.666…, 2.70833…, 2.7166…, 2.718055…, etc.

Exercise 3: Solution

- Write a function with no arguments that returns a closure with no arguments. When called the nth time (n >= 1), the closure should return the nth approximation for the number e: 1 + 1/1! + 1/2! + ... + 1/n!
- Avoid needless recomputation in the factorial and in the sum.
- Example outputs: 2, 2.5, 2.666..., 2.70833..., 2.7166..., 2.718055..., etc.

```
function approxE(): () => number {
    let n = 1, factorial = 1, res = 1;
    // can you shorten this even further?
    return () => {
        factorial *= n++; // why n++ and not ++n?
        return (res += 1/factorial);
    }
}
```