Lecture 2C: Closures

Warning: this topic is tricky for most students

you probably need to see it more than once

Background: Lexical Scope

```
let x=1;
let y=1;

function f() {
    let x=3;
    y=3;
}

x=2;
y=2;
f();
console.log(`${x} and ${y}`);
```

2 3

Variable declarations

JavaScript is not always lexically scoped...

- `var` - old style, "functionally scoped" (hoisted)
    - confusing behavior
- `let` - new style, lexically scoped
    - does what you expect from other languages
- `const` - like `let`, but specifies it won't change
    - I should use this more often

Functions inside Functions

```
function outer() {
    let a="outer";
    let b="outer";

    function inner() {
        a = "inner";
        let b = "inner";
        console.log(a,b);
    }
    console.log(a,b);
    inner();
    console.log(a,b);
}
outer();
```

inner

outer, outer

inner, inner

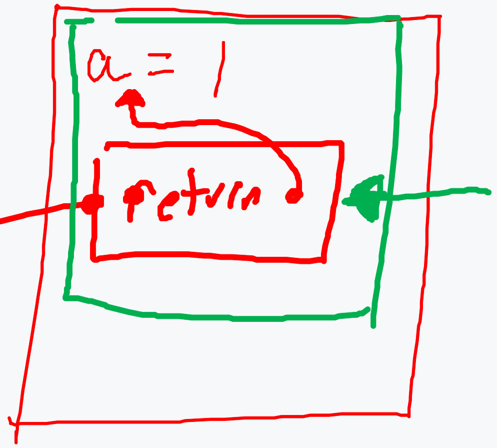
inner, outer

Functions can make functions

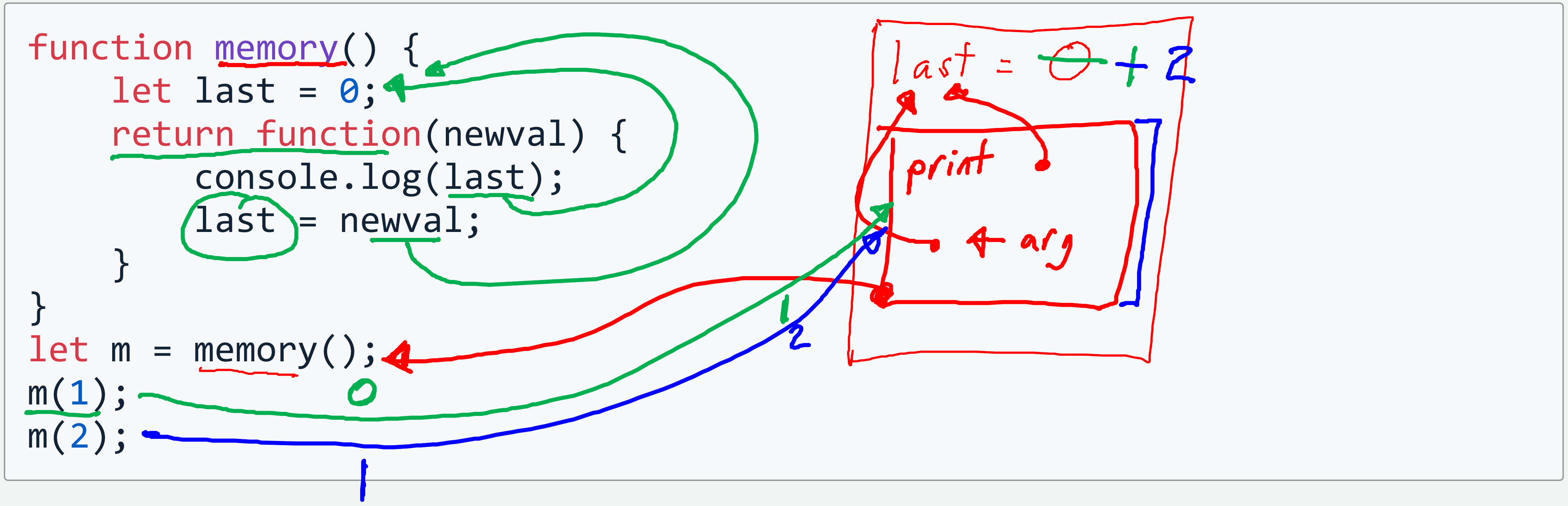
```
function makeFunction() {
    return function(x) {
        return x+1;
    }
}

makeFunction();   x ↦ x+1

makeFunction()(5);  ⇒  6
```

Closure

```
function makeFunction() {
  let a = 1;

  return function () {
    return a;
  }
}

let g = makeFunction();
g();
```



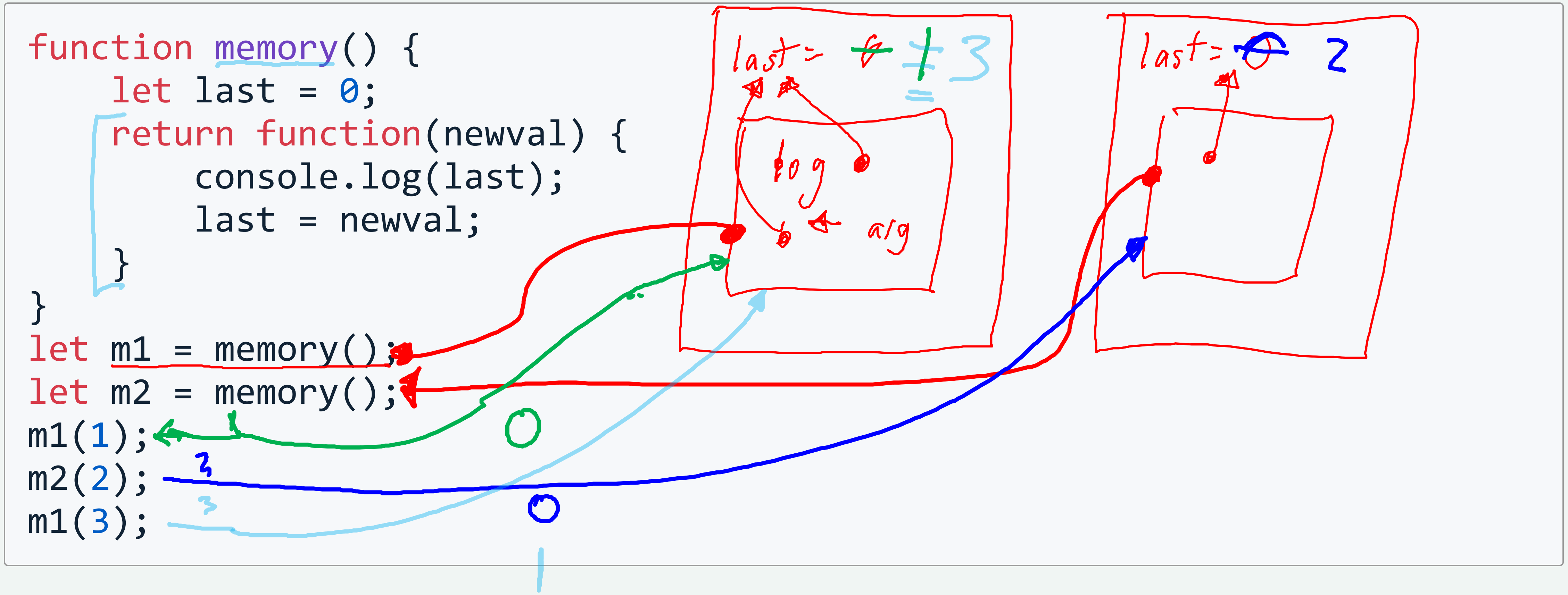
Closure



```
function memory() {
    let last = 0;
    return function(newval) {
        console.log(last);
        last = newval;
    }
}
let m = memory();
m(1);
m(2);
```

Closures (plural)

```
function memory() {
    let last = 0;
    return function(newval) {
        console.log(last);
        last = newval;
    }
}
let m1 = memory();
let m2 = memory();
m1(1);
m2(2);
m1(3);
```

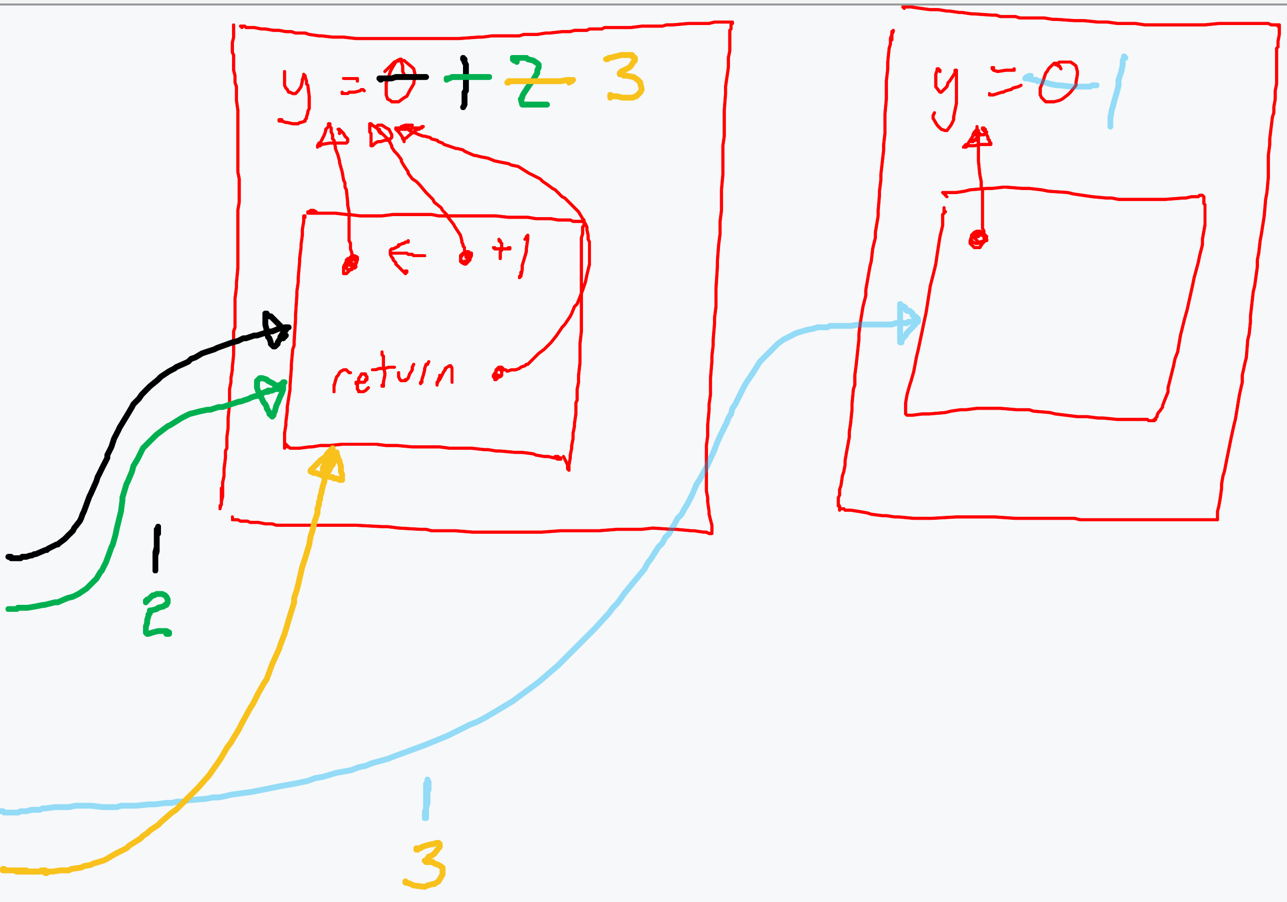


Close variables not values

```
function ex3() {
    let a = ~~"before"~~; after
    function getA() { console.log(a); }
    a = "after";
    return getA;
}
let f = ex3();
f(); ←        after
```

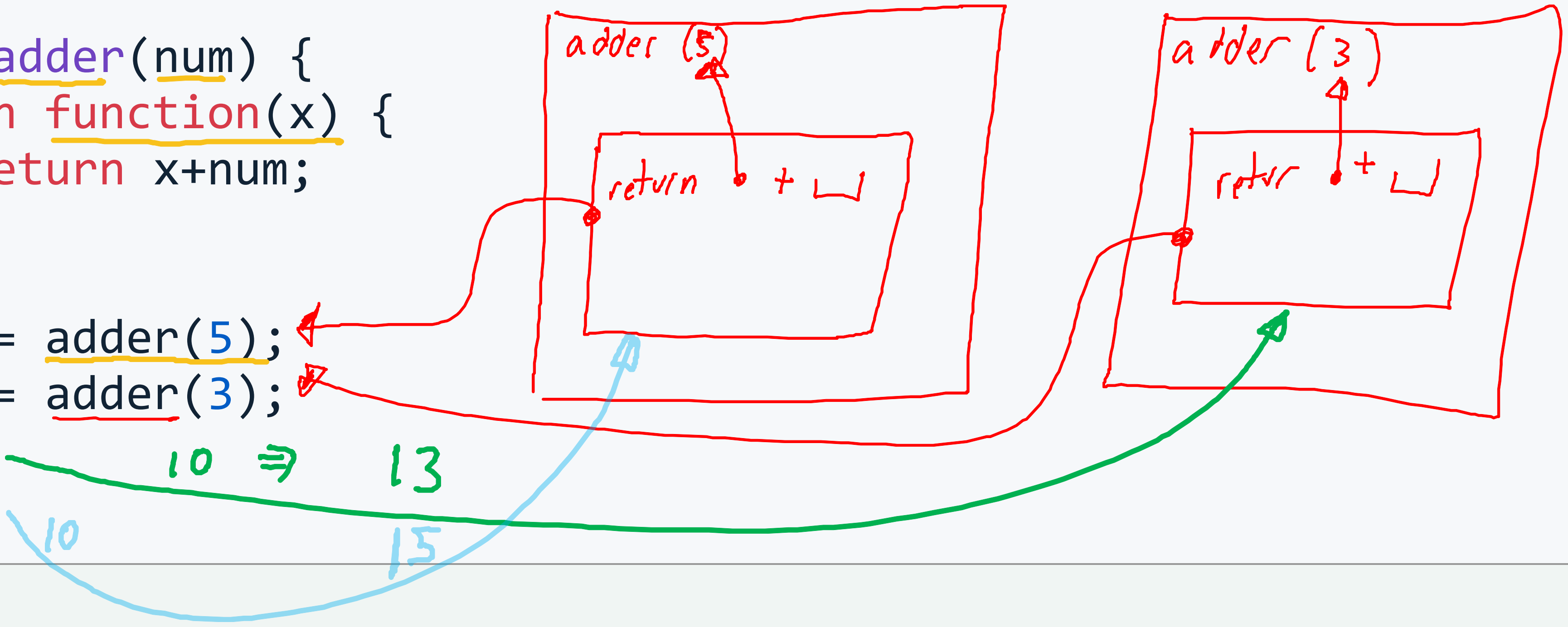
Closures

```
function closureTest() {
    let y=0;
    return function() {
        y = y+1;
        return y;
    }
}
let ct = closureTest();
ct();          // returns 1
ct();          // returns 2

let ct2 = closureTest();
ct2();         // returns 1
ct();          // returns 3
```



Closure over an Argument

```
function adder(num) {
    return function(x) {
        return x+num;
    }
}
let add5 = adder(5);
let add3 = adder(3);
add3(10);
add5(10);
```



A closure Example (in the Workbook)

How do we run multiple functions at `window.onload` ?

```
function mainA () {
    console.log("A: Something to write to console");
}
window.onload = mainA;
```

someplace else...

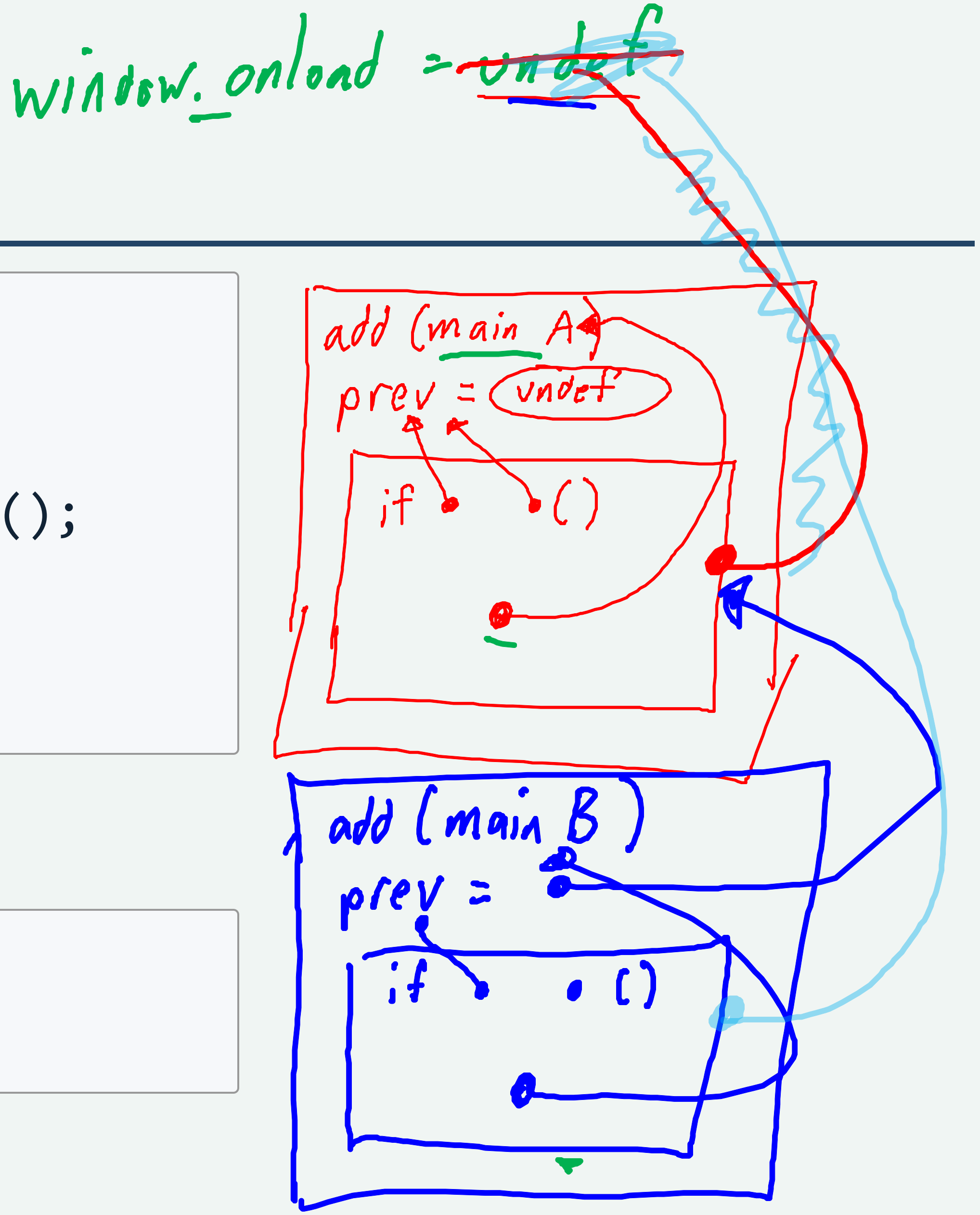
```
function mainB () {
    console.log("B: Something else to write to console");
}
window.onload = mainB;
```

AddStart

```
function addStart(func) {
    let previousStart = window.onload;
    window.onload = function() {
        if (previousStart) previousStart();
        func();
    }
}
```

What happens?

```
addStart(mainA);
addStart(mainB);
```



Could we do this without a closure?

Keep a list of the functions to call

1. We need a global variable (issues with module boundaries)
2. We need to make sure the list is initialized first

```
listOfStarts = [];  // really need to define this appropriately
function addStart(func) {
    global listOfStarts;    // not really JavaScript syntax!
    listOfStarts.push(func);
}
window.onload = function() {
    global listOfStarts;
    listOfStarts.forEach(function(f) { f() });
}
```

Summary

1. **Lexical scope** - functions access code before them
2. **Closure** - environment of a function's definition is kept

Closures take some getting used to - but are useful