Functions

```
// Javascript does not have passing by reference even though it looks like it's
passing by reference
// For objects you pass in a reference, a memory address of of the object, but
that reference itself is still a value
// pass IN a reference not BY a reference


const flight = 'LH234'; // Primitive type - On the stack
const kelsy = {         // Reference type - On the Heap
    name: 'Kelsy Watkins',
    passport: 8675309
}

// flightNum contains copy of flight value, not original value

// when passing primitive types to function the value is simply copied

// when passing reference type to function what is copied is the reference to the
object in the memory heap

// passenger takes copy of reference value to object

const checkIn = function (flightNum, passenger) {
    flightNum = 'LH999';
    passenger.name = 'Mrs. ' + passenger.name;

    if (passenger.passport === 8675309) {
        alert('Checked in')
    } else {
        alert('Wrong passport')
    }
}

checkIn(flight, kelsy);
console.log(flight); // still LH234 flightNum is on new memory address
console.log(kelsy); // name gets changed as memory address stay the same and
value does change on the heap
```

Flight = LH234

Passing in flight as parameter – points to memory address copies value

Sets new memory address to copied value

flightNum = LH999

---

kelsy = object

passing in passenger as parameter – points to memory address

CALL STACK

Line	Identifier	Address	Value
	flight	0003	LH234
	flightNum	0004	LH234
			~~LH234~~ LH999
	kelsy	0005	D30f
	passenger		

When we pass a reference type to a function what is copied the reference to the object in the memory HEAP

HEAP

Line	Address	Value
Origonal object	D30F	`const kelsy = { name: 'Kelsy Watkins', passport: 8675309 }`
passenger.name = 'Mrs. ' + passenger.name;		`const kelsy = { name: ' Mrs. Kelsy Watkins', passport: 8675309 }`

Passing by Value - what JavaScript does

Passing by Reference – JavaScript does not do this. You have a reference to a memory value but that memory address itself is still a value

First class functions

- JavaScript treats functions as **first-class citizens**
- This means that functions are **simply values**
- Functions are just another **type of object**

*Because functions are values you can:

```
const add = (a, b) => a + b; // store in variables, function expression
const counter = {
    value: 23,
    inc: function () {this.value++;} // store in object properties, object method
}
```

*Pass functions as arguments to OTHER functions

```
const greet = () => console.log('Hey Kelsy');
btnClose.addEventListener('click', greet) // pass greet function to
addEventListener function
```

*Return functions FROM functions

*Call methods on functions

```
counter.inc.bind(someOtherObject); // calling method on a function
```

Higher Order functions

- A function that **receives** another function as an argument, that **returns** a new function, or **both**
- This is only possible because of first class functions

*function that receives another function

```
const greet = () => console.log('Hey Kelsy');
btnClose.addEventListener('click', greet)
```

Higher order function

Callback function

*function that return new function

```
function count() {          <-- Higher order function
    let counter = 0;
    return function () {    <-- returned function
        counter++;
    };
}
```

Callback Functions

- So widely used because you can split up your code into more reusable and interconnected parts. They can allow you to create abstraction. You could place string manipulation functions inside this function but you want to abstract your code into other lower level functions (upperFirstWord) along with the higher order function(transformer).

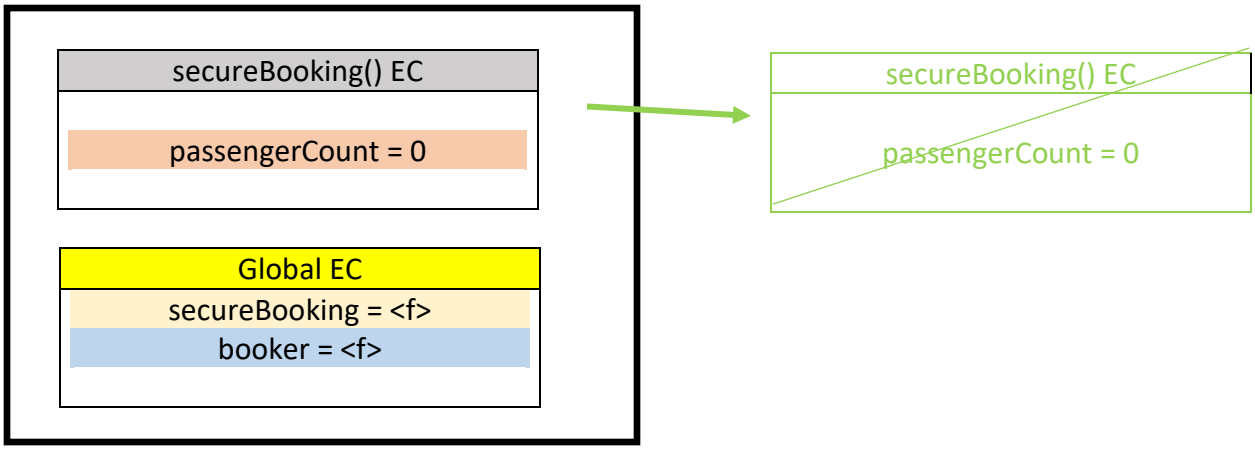
```
const transformer = function (str, fn = () => { }) {
    console.log(`Original string: ${str}`)
    console.log(`Transformed string: ${fn(str)}`)

    console.log(`Transformed by: ${fn.name}`)
};
// Pass CallBack functions
transformer('JavaScript is the best!', upperFirstWord);
```

Closures

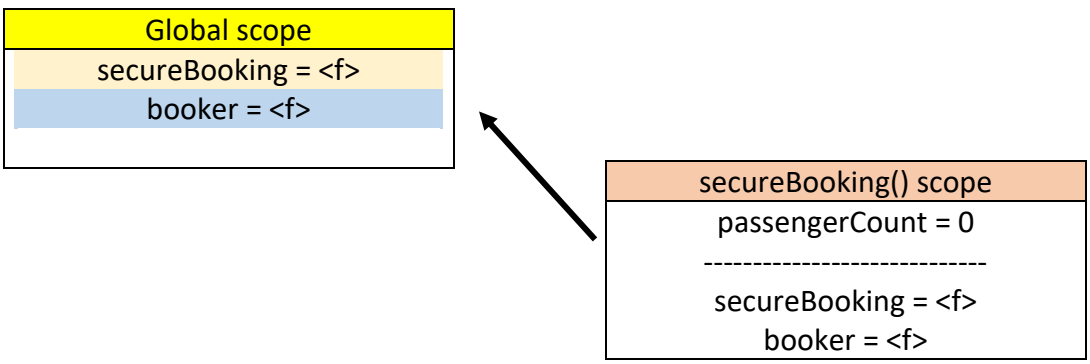
```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount} passengers`);
  };
};

const booker = secureBooking();
```

1. secureBooking function is placed in global execution context. Global scope contains secureBooking function
2. secureBooking executes and places new execution context on top of stack
3. Each EC contains a variable environment which contains all its local variables. This variable environment is also the scope of this function. Passenger count is in the local scope, this gets access to all variables of the parent scope (global scope in this case)
4. New function is returned and stored in the booker variable. Global EC contains this variable.
5. Also on return the secureBooking EC pops off the stack

CALL STACK – order in which functions were called – EC (Execution Context)

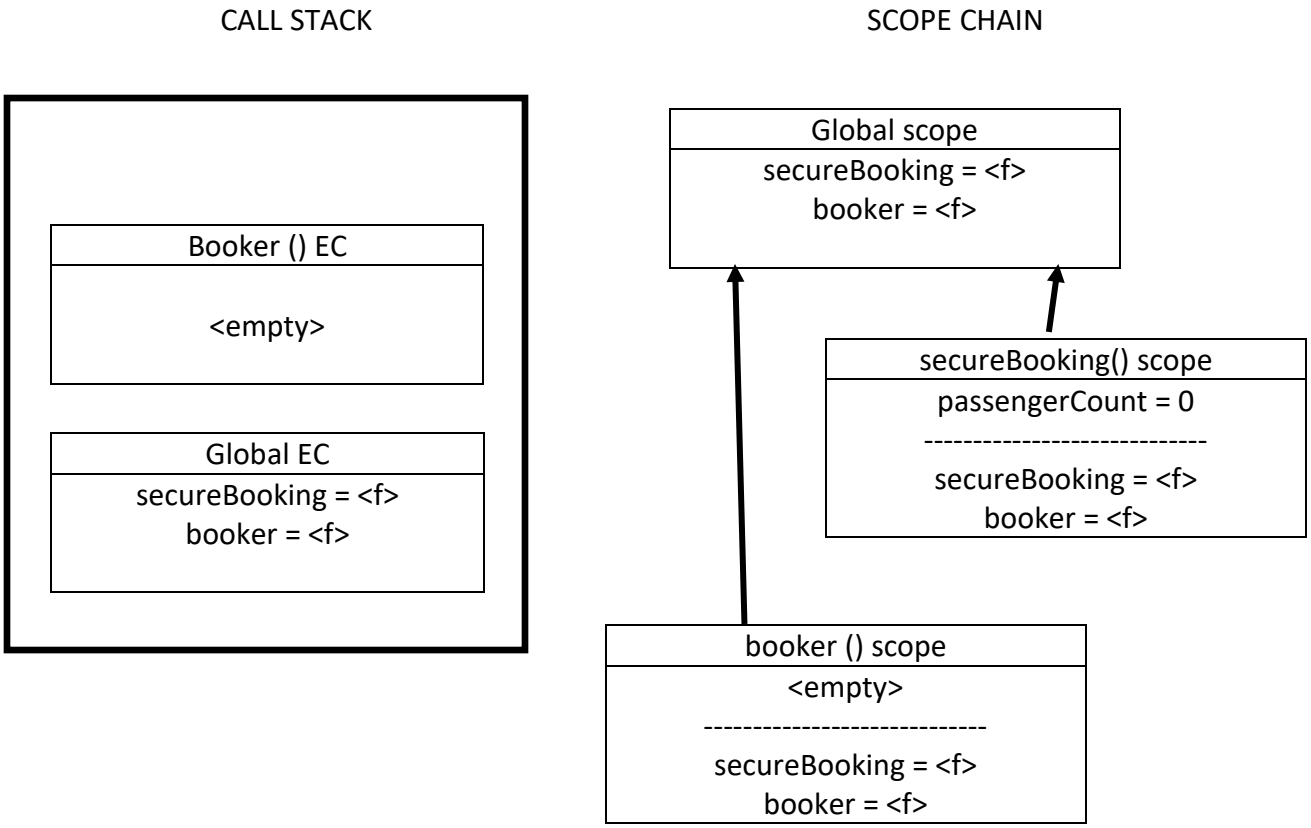


SCOPE CHAIN



Note: the booker function is a function that exists in the global scope or environment. And the environment in which the function was created, is no longer active, it's gone. But still the booker function somehow continues to have access to the variables that were present at the time that the function was created (passengerCount). A closure makes a function remember all the variables that existed at the functions birth place essentially.

```
const booker = secureBooking();

booker(); // 1 passengers
booker(); // 2 passengers
```

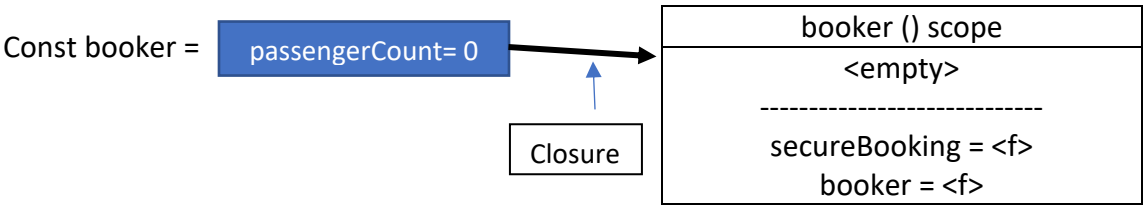


So how does booker function have access the passengerCount variable if its no where in the scope chain? Any function always has access to the variable environment of the execution context in which the function was created, even after that EC is done.

In other words, a closure gives a function access to all of the variables of its parent function, even after that parent function has returned. The function keeps a reference to its outer scope, which preserves the scope chain throughout time.

In the case of booker, this function was created in the EC of secureBooking (which was popped off the stack previously). Therefor the booker function will get access to that variable environment which contains the passengerCount.

Closure is the variable environment attached to the function




```
const secureBooking = function () {
  let passengerCount = 0;

  return function () {
    passengerCount++;
    console.log(`${passengerCount} passengers`);
  };
};

const booker = secureBooking();

booker(); // 1 passengers
```

You can say the booker function closed over its parent scope, now this closed over variable environment stays with the function forever!

So the booker function attempts to increase the passengerCount variable however this variable is not in the current scope, so JavaScript will immediately look into the closure and see if it can find the variable there. Closure actually takes priority over the scope chain, for example if you had a global passengerCount variable set to 10 JavaScript would still first use the one in the closure.

So the booker function is executed and popped off the stack and the next function call will increase the number further, so on and so forth.

NOTE: you do not have to manually create closures, this is a JavaScript feature that happens automatically. You can't explicitly access closed-over variables, closures are not a tangible thing its not like an object you can just access. A closure is just an internal property of a function.

Inspect scopes internal property in the console to see variable environment of booker function.

```
console.dir(booker);
```

1. [[Scopes]]: Scopes[3]
    1. 0: Closure (secureBooking) {passengerCount: 3}