DATA STRUCTURES, MODERN OPERATORS AND STRINGS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

DATA STRUCTURES, MODERN OPERATORS AND STRINGS

LECTURE

SUMMARY: WHICH DATA STRUCTURE TO USE?

JS

DATA STRUCTURES OVERVIEW

SOURCES OF DATA

1. From the program itself: Data written directly in source code (e.g. status messages)
2. From the UI: Data input from the user or data written in DOM (e.g tasks in todo app)
3. From external sources: Data fetched for example from web API (e.g. recipe objects)

Application Programming Interface

Collection of data

Data structure

SIMPLE LIST?

KEY/VALUE PAIRS?

Arrays or Sets

Objects or Maps

Keys allow us to *describe* values

OTHER BUILT-IN:

- WeakMap
- WeakSet

NON-BUILT IN:

- Stacks
- Queues
- Linked lists
- Trees
- Hash tables

"Object"

Array

"Object"


```
{
  "count": 3,
  "recipes": [
    {
      "publisher": "101 Cookbooks",
      "title": "Best Pizza Dough Ever",
      "source_url": "http://www.101cookbooks.com/archiv
      "recipe_id": "47746",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 100,
      "publisher_url": "http://www.101cookbooks.com"
    },
    {
      "publisher": "The Pioneer Woman",
      "title": "Deep Dish Fruit Pizza",
      "source_url": "http://thepioneerwoman.com/cooking
      "recipe_id": "46956",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 100,
      "publisher_url": "http://thepioneerwoman.com"
    },
    {
      "publisher": "Closet Cooking",
      "title": "Pizza Dip",
      "source_url": "http://www.closetcooking.com/2011/
      "recipe_id": "35477",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 99.99999999999994,
      "publisher_url": "http://closetcooking.com"
    }
  ]
}
```


JSON data format example

ARRAYS VS. SETS AND OBJECTS VS. MAPS

ARRAYS VS. SETS

ARRAYS

```
tasks = ['Code', 'Eat', 'Code'];
// ["Code", "Eat", "Code"]
```

- Use when you need ordered list of values (might contain duplicates)
- Use when you need to manipulate data

SETS

```
tasks = new Set(['Code', 'Eat', 'Code']);
// {"Code", "Eat"}
```

- Use when you need to work with unique values
- Use when high-performance is *really* important
- Use to remove duplicates from arrays

OBJECTS VS. MAPS

OBJECTS

```
task = {
  task: 'Code',
  date: 'today',
  repeat: true
};
```

- More “traditional” key/value store (“abused” objects)
- Easier to write and access values with `.` and `[]`
- Use when you need to include functions (methods)
- Use when working with JSON (can convert to map)

MAPS

```
task = new Map([
  ['task', 'Code'],
  ['date', 'today'],
  [false, 'Start coding!']
]);
```

- Better performance
- Keys can have any data type
- Easy to iterate
- Easy to compute size
- Use when you simply need to map key to values
- Use when you need keys that are not strings

A CLOSER LOOK AT FUNCTIONS

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

A CLOSER LOOK AT FUNCTIONS

LECTURE

FIRST-CLASS AND HIGHER-ORDER FUNCTIONS

JS

FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

FIRST-CLASS FUNCTIONS

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

- Store functions in variables or properties:

```
const add = (a, b) => a + b;
```

```
const counter = {
  value: 23,
  inc: function() { this.value++; }
```

- Pass functions as arguments to OTHER functions:

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

- Return functions FROM functions
- Call methods on functions:

```
counter.inc.bind(someOtherObject);
```

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

1 Function that receives another function

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

Higher-order function

Callback function

✅📞💬

2 Function that returns new function

```
function count() {
  let counter = 0;
  return function() {
    counter++;
  };
}
```

Higher-order function

Returned function

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

SECTION

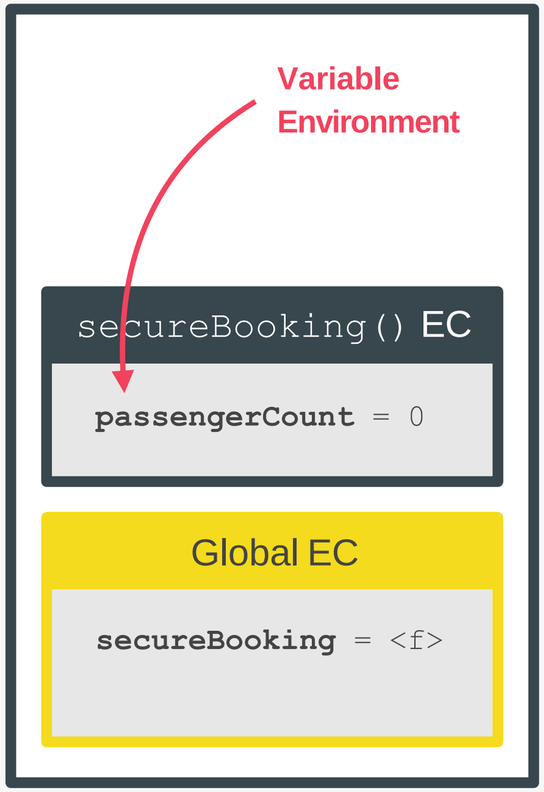
A CLOSER LOOK AT FUNCTIONS

LECTURE

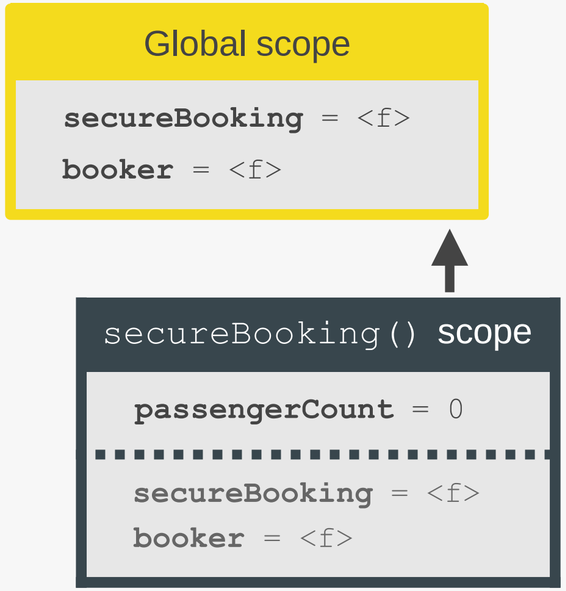
CLOSURES

JS

"CREATING" A CLOSURE



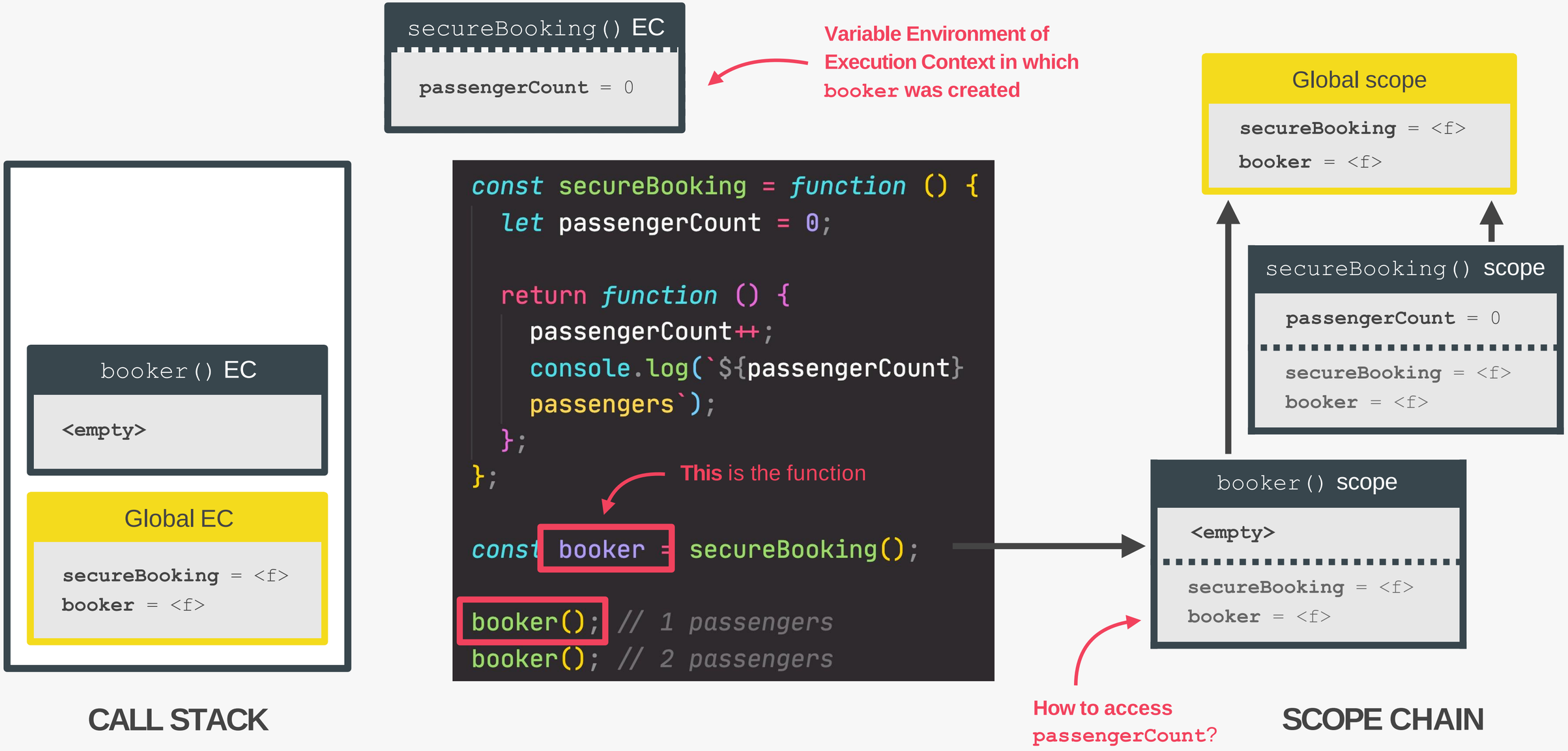

```
const secureBooking = function () {
  let passengerCount = 0;

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

const booker = secureBooking();
```



CALL STACK ← Order in which functions were ***called***

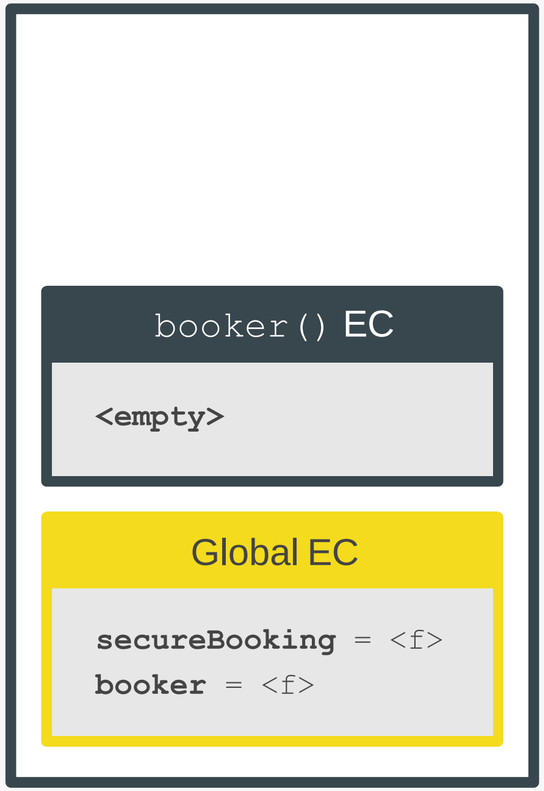
Order in which functions are ***written in the code*** → **SCOPE CHAIN**

UNDERSTANDING CLOSURES



UNDERSTANDING CLOSURES

- A function has access to the variable environment (VE) of the execution context in which it was created
- **Closure:** VE attached to the function, exactly as it was at the time and place the function was created



CALL STACK

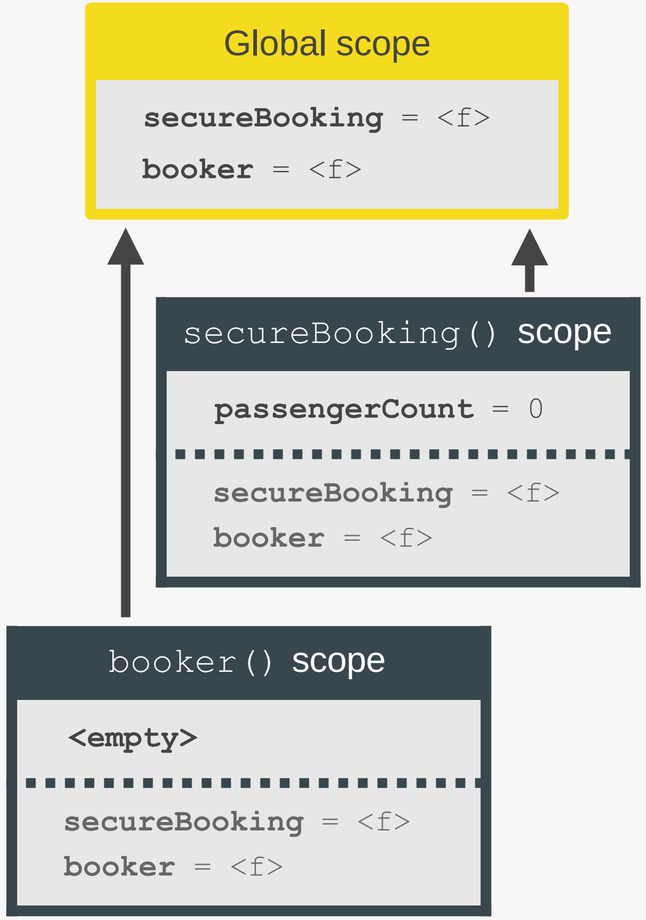
```
const secureBooking = function () {
  let passengerCount = 0;

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

const booker = passengerCount = 2

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

This is the function

(**Priority** over scope chain)

CLOSURE

How to access `passengerCount`?



SCOPE CHAIN

CLOSURES SUMMARY 🥳

👉 A closure is the closed-over **variable environment** of the execution context **in which a function was created**, even ***after*** that execution context is gone;

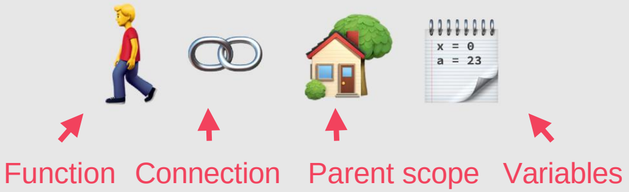
↓ Less formal

👉 A closure gives a function access to all 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.

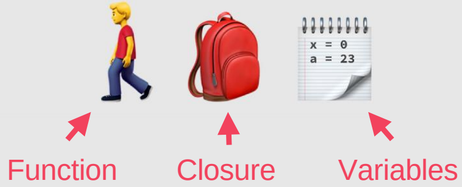
↓ Less formal

👉 A closure makes sure that a function doesn't loose connection to **variables that existed at the function's birth place**;

↓ Less formal



👉 A closure is like a **backpack** that a function carries around wherever it goes. This backpack has all the **variables that were present in the environment where the function was created**.



☝️ We do **NOT** have to manually create closures, this is a JavaScript feature that happens automatically. We can't even access closed-over variables explicitly. A closure is **NOT** a tangible JavaScript object.