"CREATING" A CLOSURE

secureBooking() EC

passengerCount = 0

Global scope

secureBooking = <f>
booker = <f>

secureBooking() scope

passengerCount = 0

secureBooking = <f>
booker = <f>

Global EC

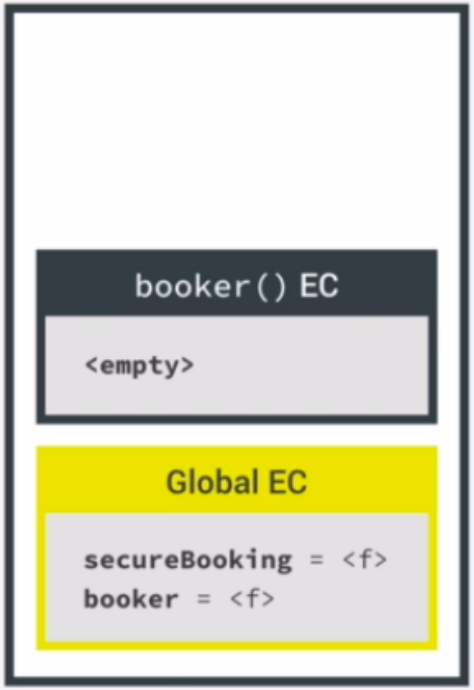
secureBooking = <f>
booker = <f>

```
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

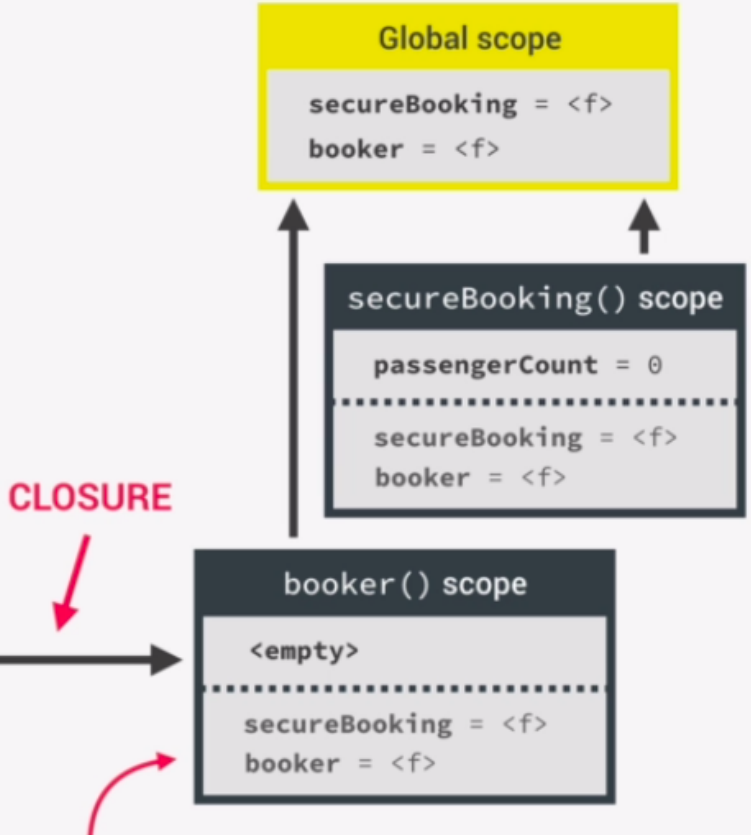



```
const secureBooking = function () {
  let passengerCount = 0;

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

const booker = secureBooking();

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

This is the function

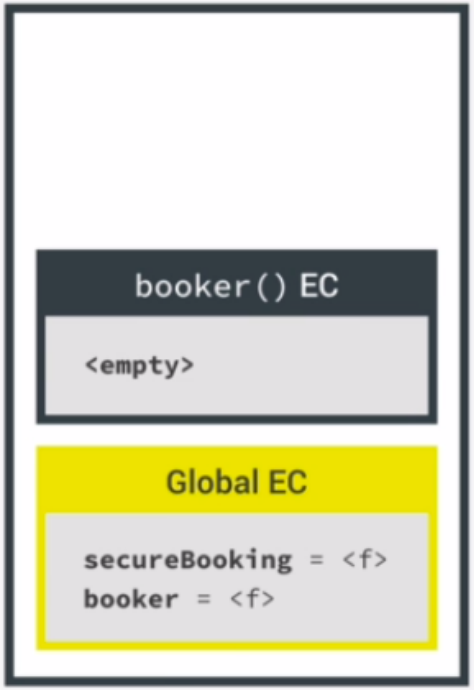
passengerCount = 0

CLOSURE



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



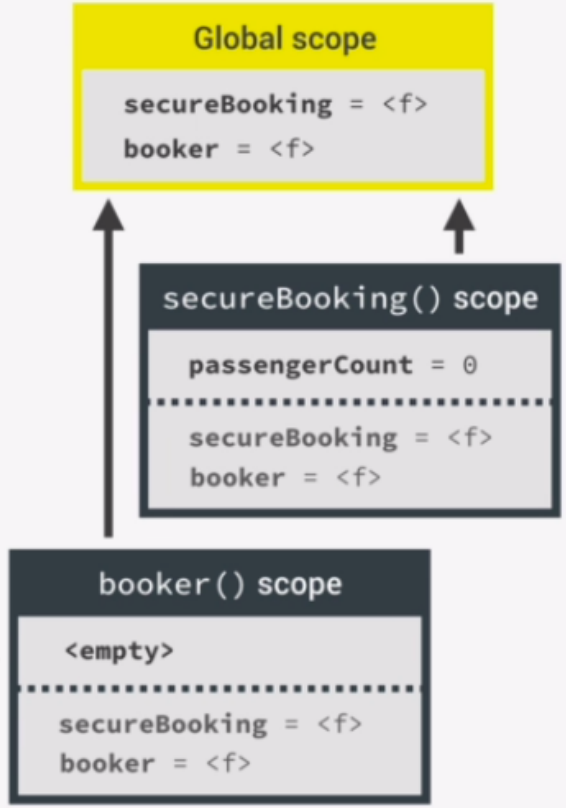
CALL STACK

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

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

const booker = secureBooking();

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

This is the function

passengerCount = 0

CLOSURE

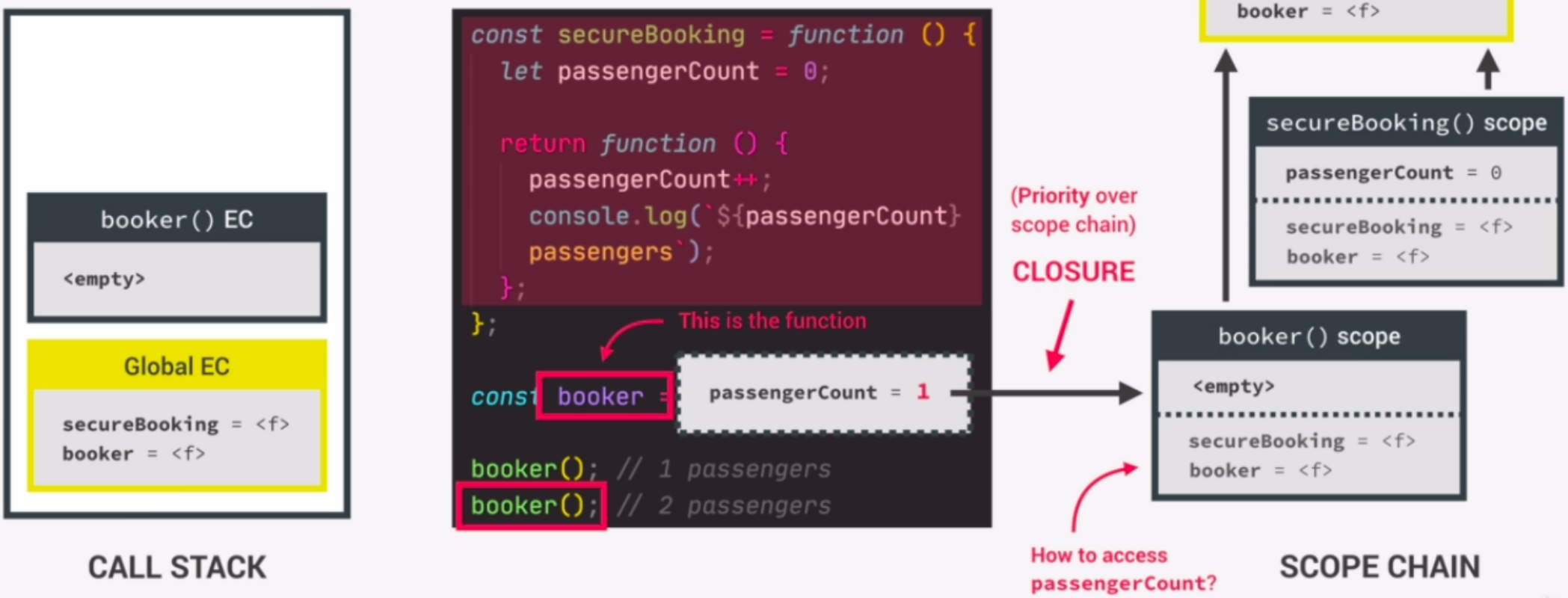


How to access passengerCount?

SCOPE CHAIN

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



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;

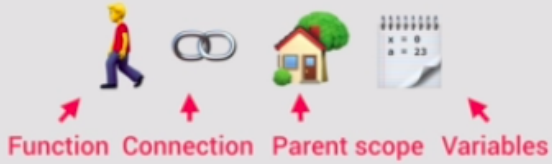
↓ Less formal

👉 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.

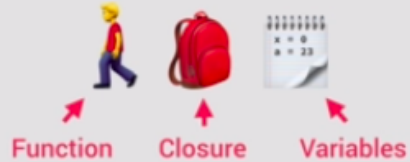
↓ Less formal

👉 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.

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