
JS Closures

The Meetup logo is written in a red, cursive script font.The Jambit logo features the word "jambit" in a bold, sans-serif font, with "jam" in black and "bit" in orange. Below it, the tagline "WHERE INNOVATION WORKS" is written in a smaller, black, sans-serif font. To the right of the text is a circular icon containing a stylized black and white coffee bean.

jambit
WHERE INNOVATION WORKS

Hamed Fatehi 22.03.2023

Hiker-Analogie

- Ein Hiker benötigt verschiedene Vorräte wie Essen, Wasser und Kleidung, um seine Wanderung durchzuführen.
- Der Rucksack enthält alle notwendigen Vorräte.
- Die Vorräte sind jederzeit leicht zugänglich



Hiker-Analogie

- Man kann sich der Closure wie einen Rucksack vorstellen, den ein Wanderer auf seiner Wandertour trägt.
- Der Rucksack enthält alle wichtigen Dinge, die der Wanderer braucht, um seine Tour erfolgreich zu beenden.



Hiker-Analogie

- Man kann sich der Closure wie einen Rucksack vorstellen, den ein Wanderer auf seiner Wandertour trägt.
- Der Rucksack enthält alle wichtigen Dinge, die der Wanderer braucht, um seine Tour erfolgreich zu beenden.

Behalte diese Analogie im Hinterkopf,
wir werden später darauf zurückkommen.



Warum sollte ich etwas über "Closure" lernen?

- Closure ist ein wichtiges Konzept in JavaScript, mit dem private Variablen, Funktionen erstellt werden können, um die Modularität, Wartbarkeit und Sicherheit des Codes zu verbessern.
- Closure wird häufig in JavaScript-Frameworks und -Bibliotheken wie React, Angular und jQuery verwendet.
- Closure wird auch in funktionalen Programmierparadigmen verwendet, die in der modernen Webentwicklung immer beliebter werden.
- Durch die Verwendung von Closure lässt sich mehr wiederverwendbarer Code erstellen.
- Closure ist ein leistungsfähiges Tool für das Management von asynchronen Tasks und den Umgang mit event-driven Programmierung.

Scope

- Der Scope verwaltet die Zugänglichkeit von Variablen.
- Innerhalb eines Scope ist der Zugriff auf die definierte Variable frei, außerhalb des Scopes ist die Variable nicht mehr zugänglich.

```
function outerFunction() {  
  var outerVar = "I'm defined in outerFunction";  
  
  function innerFunction() {  
    var innerVar = "I'm defined in innerFunction";  
  
    console.log(outerVar); // outputs "I'm defined in outerFunction"  
    console.log(innerVar); // outputs "I'm defined in innerFunction"  
  }  
  
  innerFunction();  
  console.log(outerVar); // outputs "I'm defined in outerFunction"  
  console.log(innerVar); // throws a ReferenceError, as innerVar is not defined in this scope  
}  
  
outerFunction();
```

Definition

“A closure is the combination of a function bundled together (enclosed) with references to its surrounding state.” [\[MDN\]](#)

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
  
myCounter(); // Output: 1  
myCounter(); // Output: 2
```

Definition

Global Memory

Der Global Memory ist ein Speicherbereich, in dem Variablen und Funktionen gespeichert werden.

Definition

Global Memory

Der Global Memory ist ein Speicherbereich, in dem Variablen und Funktionen gespeichert werden.

Execution Context

Wenn eine Funktion aufgerufen wird, wird ein Ausführungskontext erstellt, der Informationen darüber enthält, welche Variablen und Argumente an die Funktion übergeben wurden.

Definition

Global Memory

Der Global Memory ist ein Speicherbereich, in dem Variablen und Funktionen gespeichert werden.

Execution Context

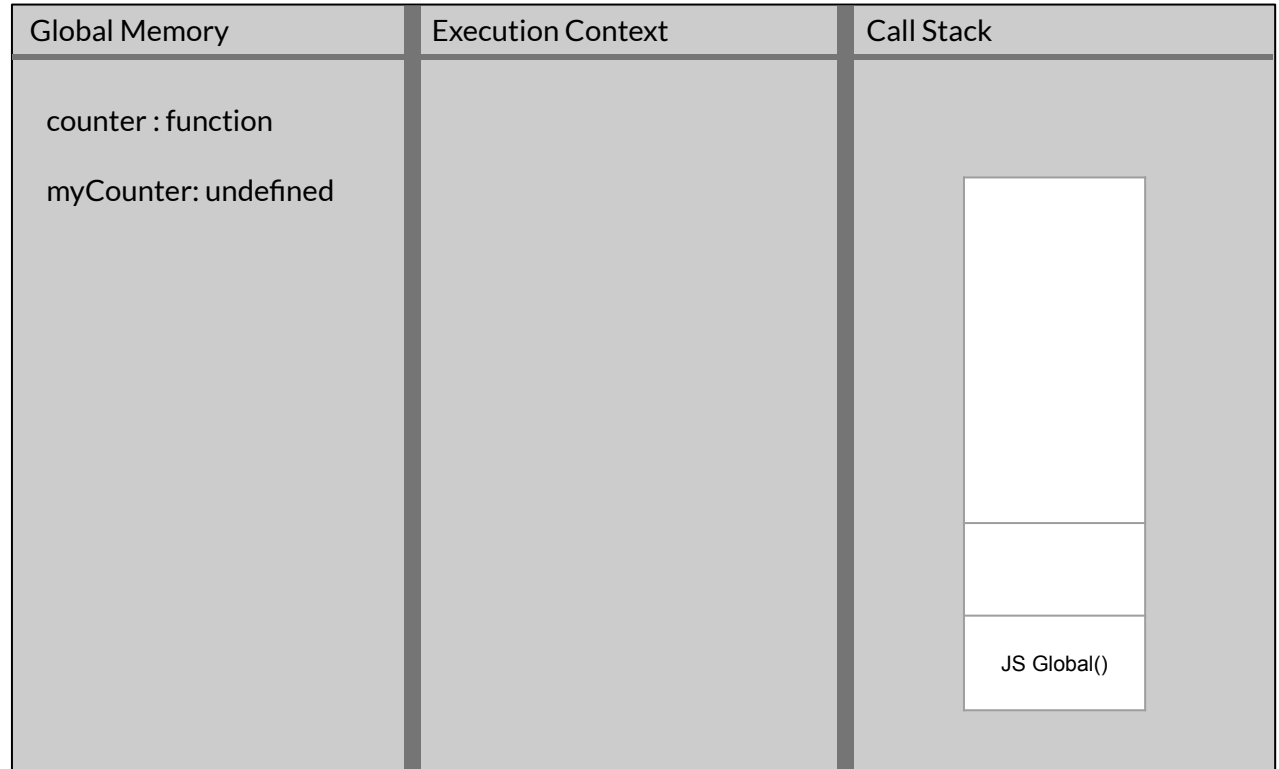
Wenn eine Funktion aufgerufen wird, wird ein Ausführungskontext erstellt, der Informationen darüber enthält, welche Variablen und Argumente an die Funktion übergeben wurden.

Call Stack

Call Stack ist ein Stapel, der alle Ausführungskontexte speichert, die derzeit aktiv sind. Wenn eine Funktion aufgerufen wird, wird ein neuer Ausführungskontext erstellt und auf den Call Stack gelegt. Wenn die Funktion beendet ist, wird der Ausführungskontext vom Stack entfernt.

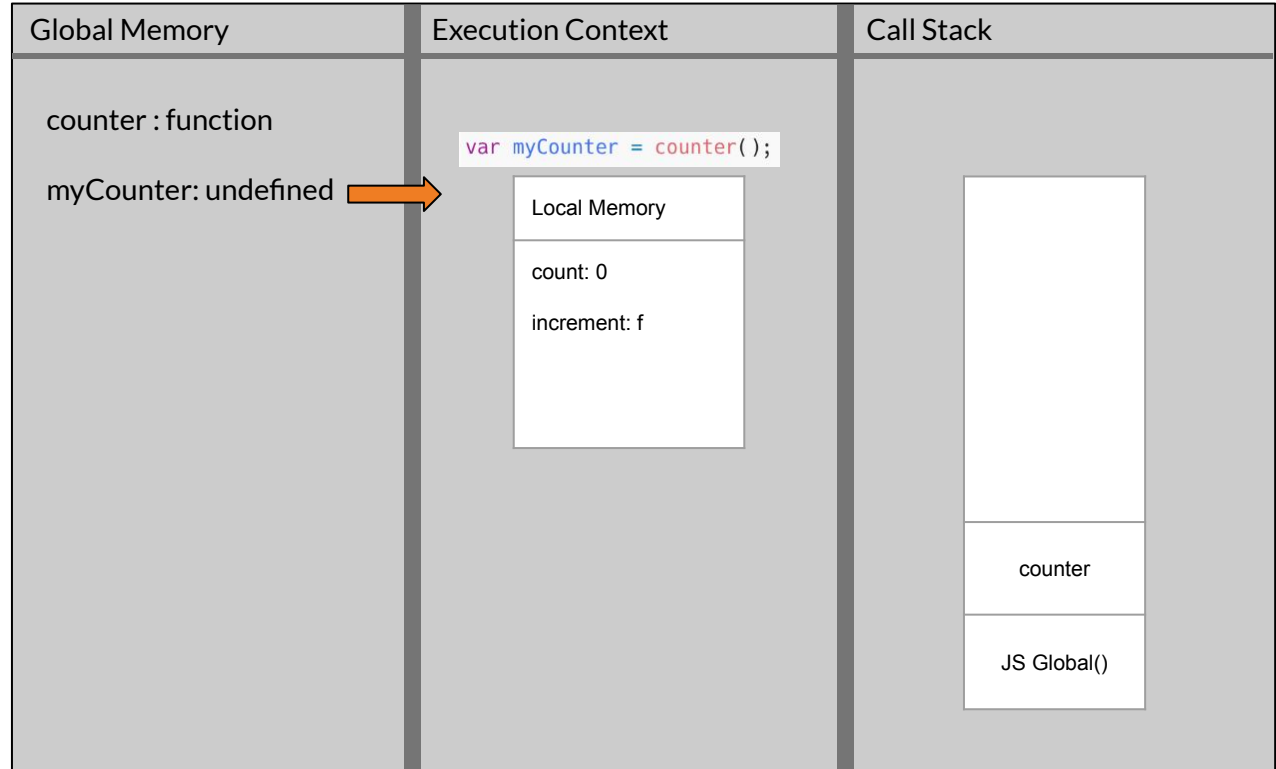
A peak under the hood

```
➡ function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
➡ var myCounter = counter();  
myCounter(); // Output: 1
```



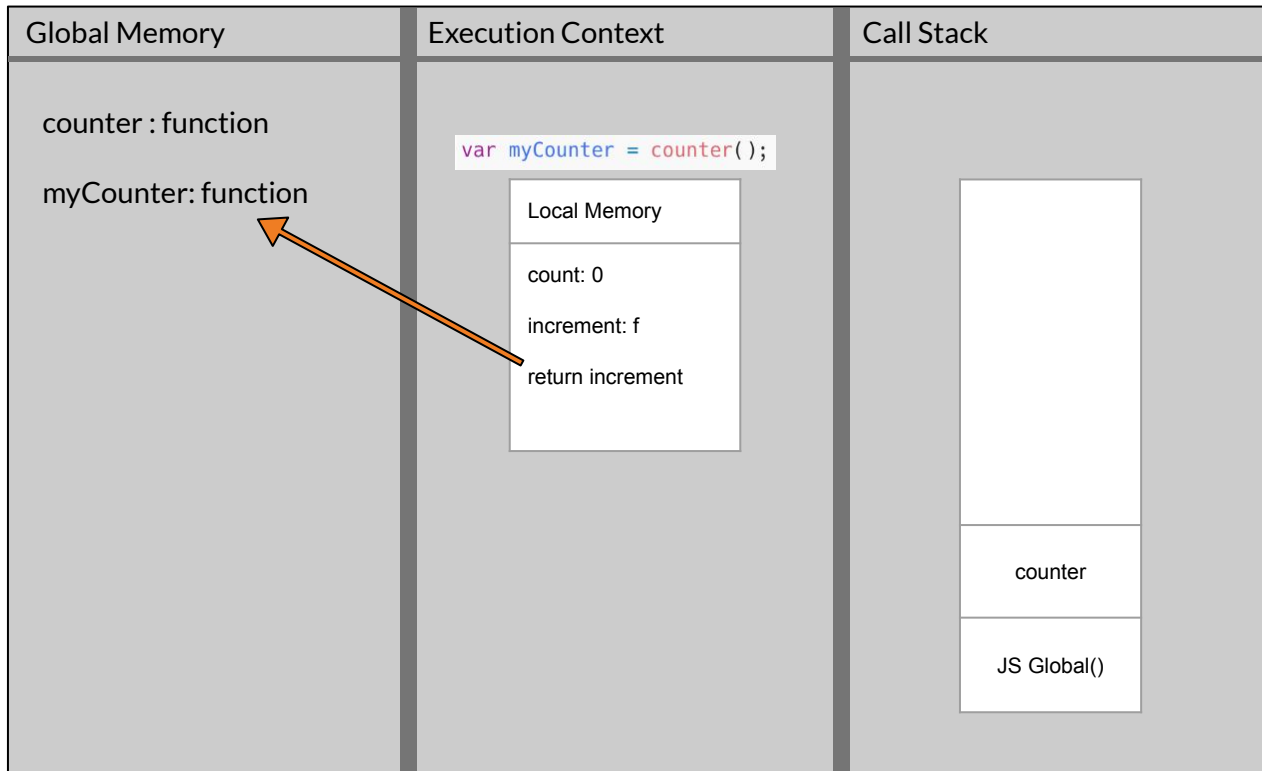
A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
myCounter(); // Output: 1
```

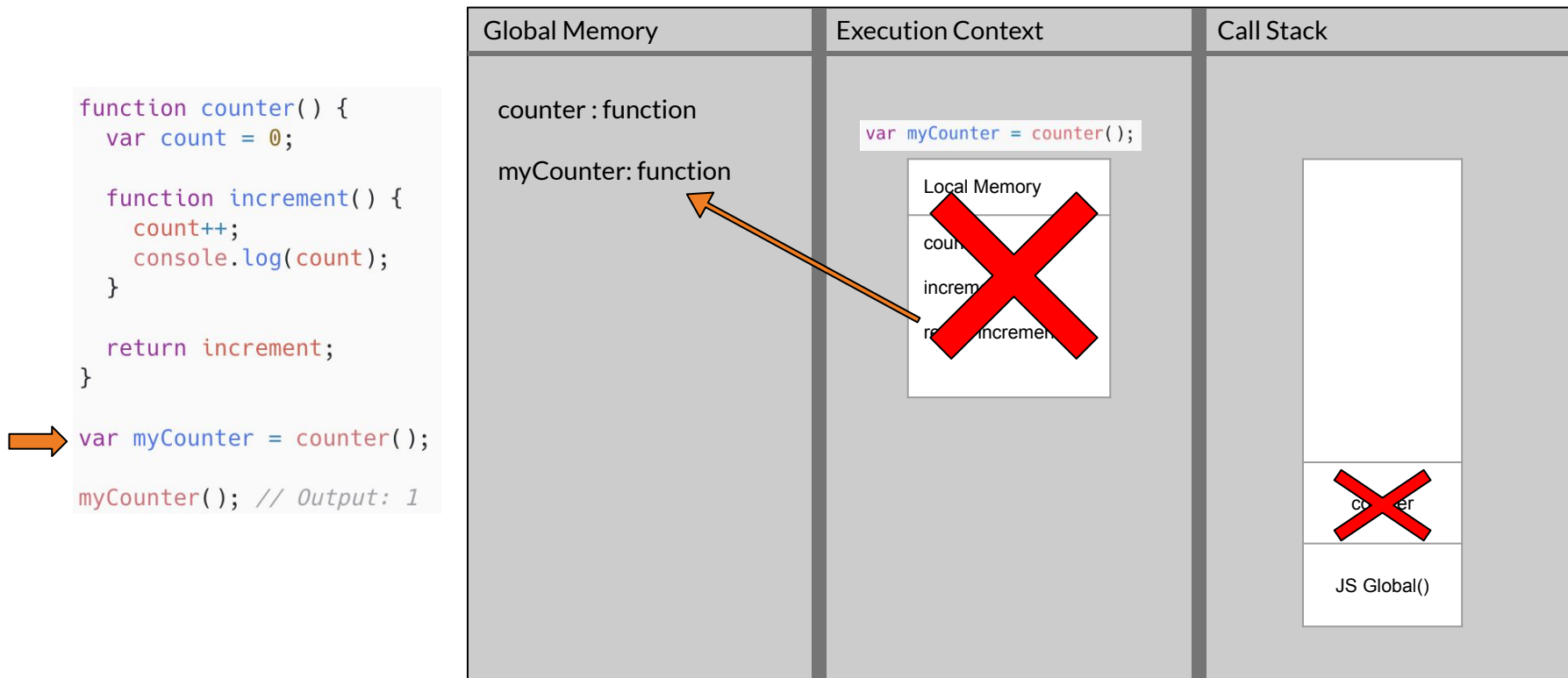


A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
myCounter(); // Output: 1
```

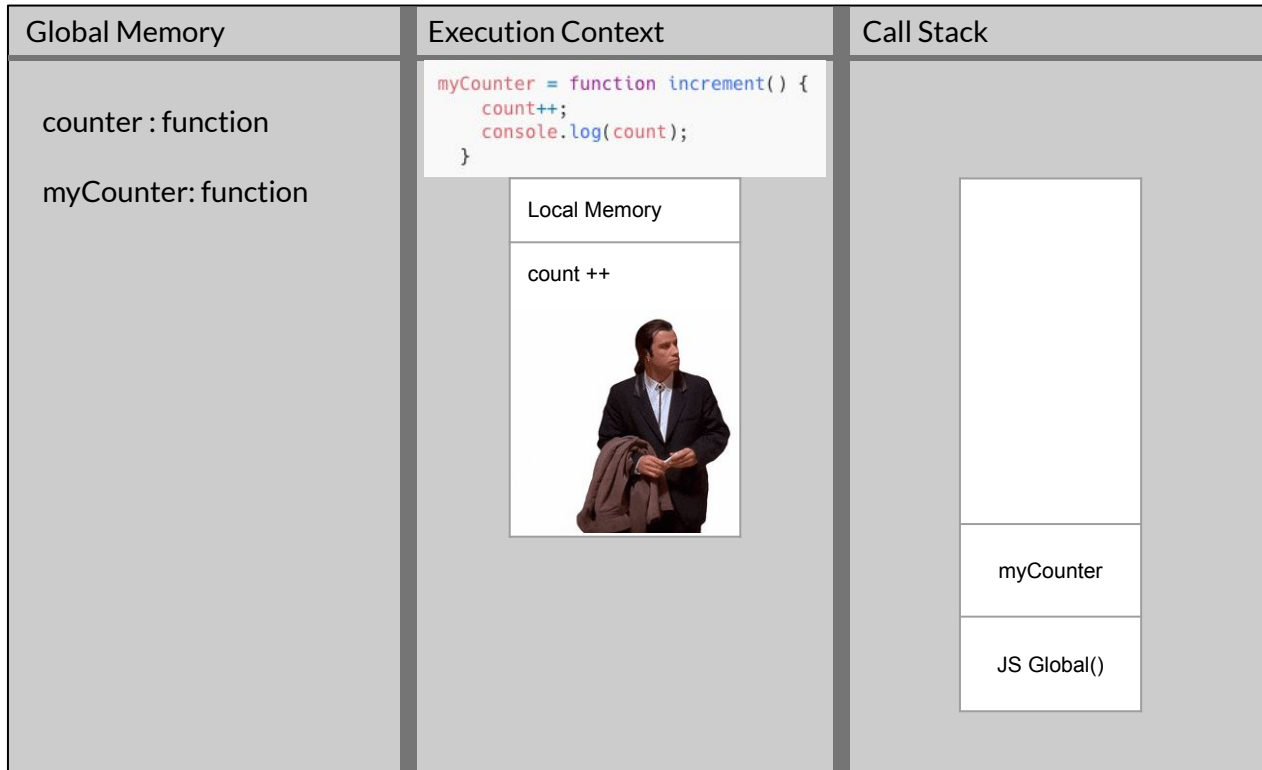


A peak under the hood



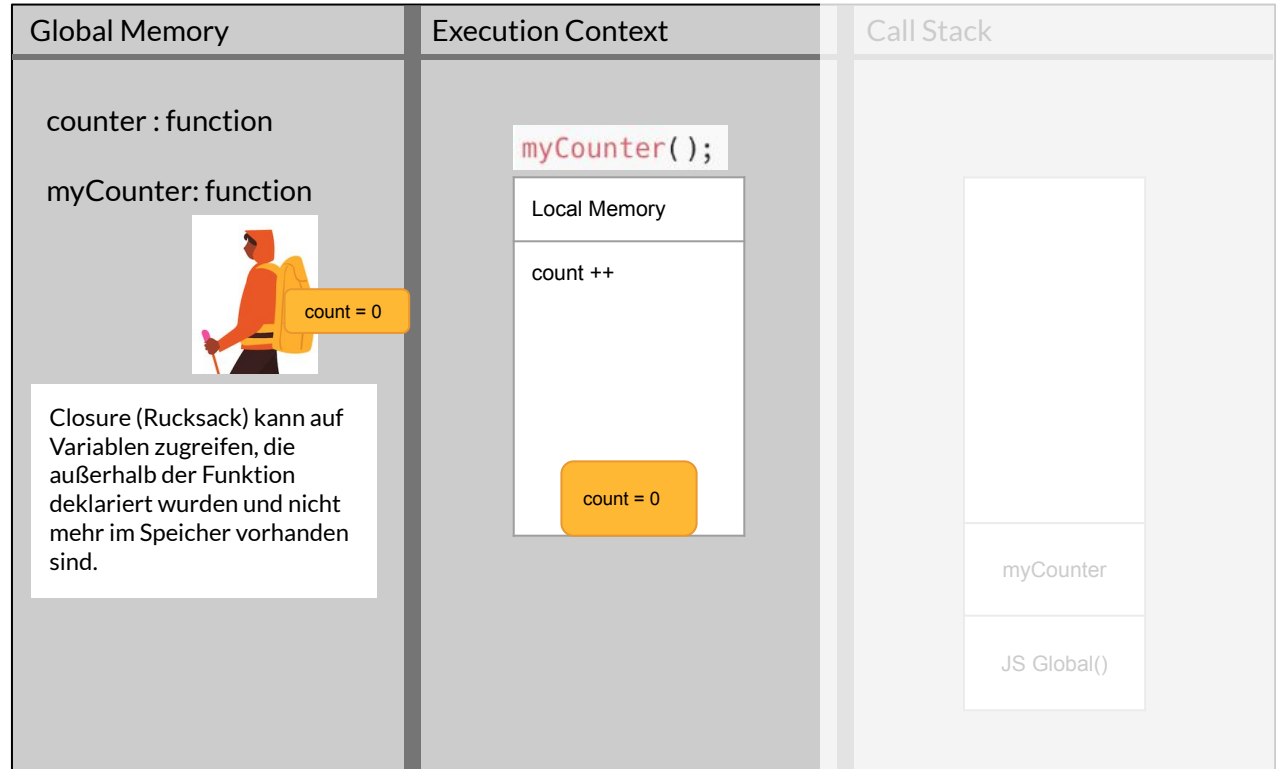
A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
→ myCounter(); // Output: 1
```



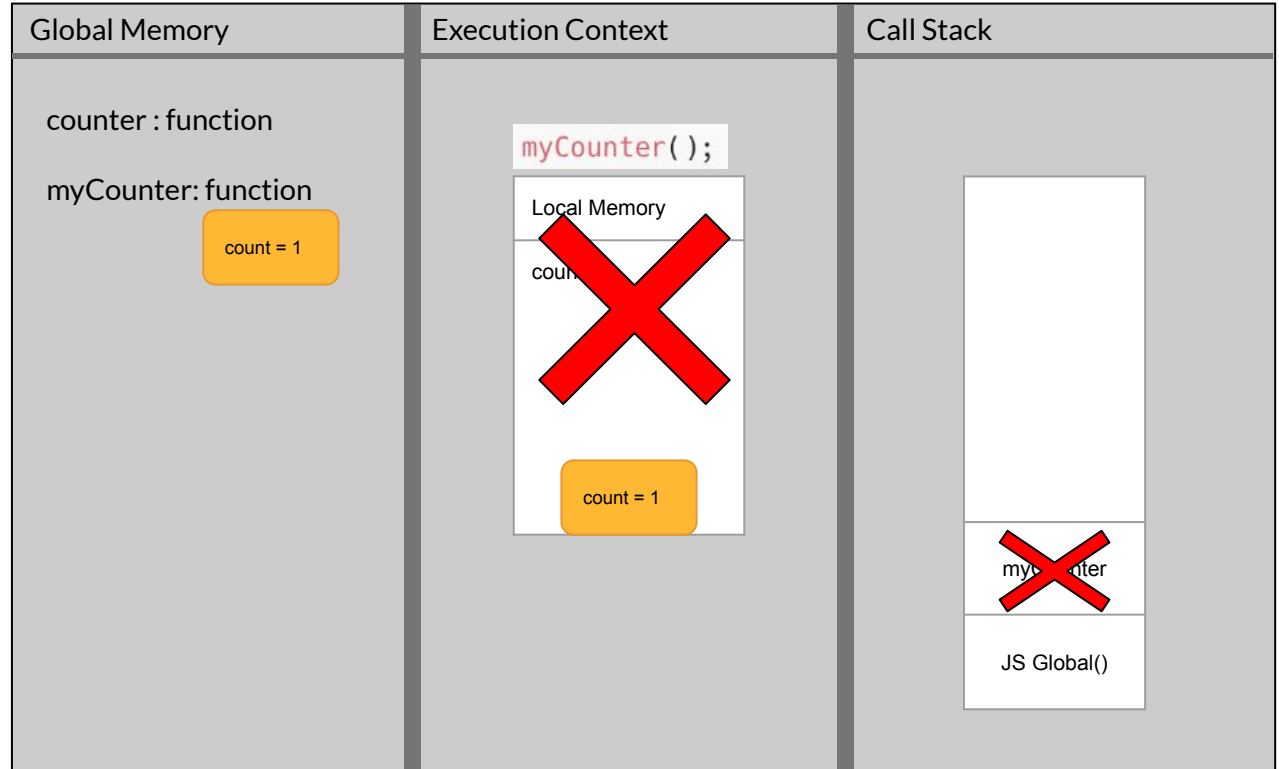
A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
→ myCounter(); // Output: 1
```



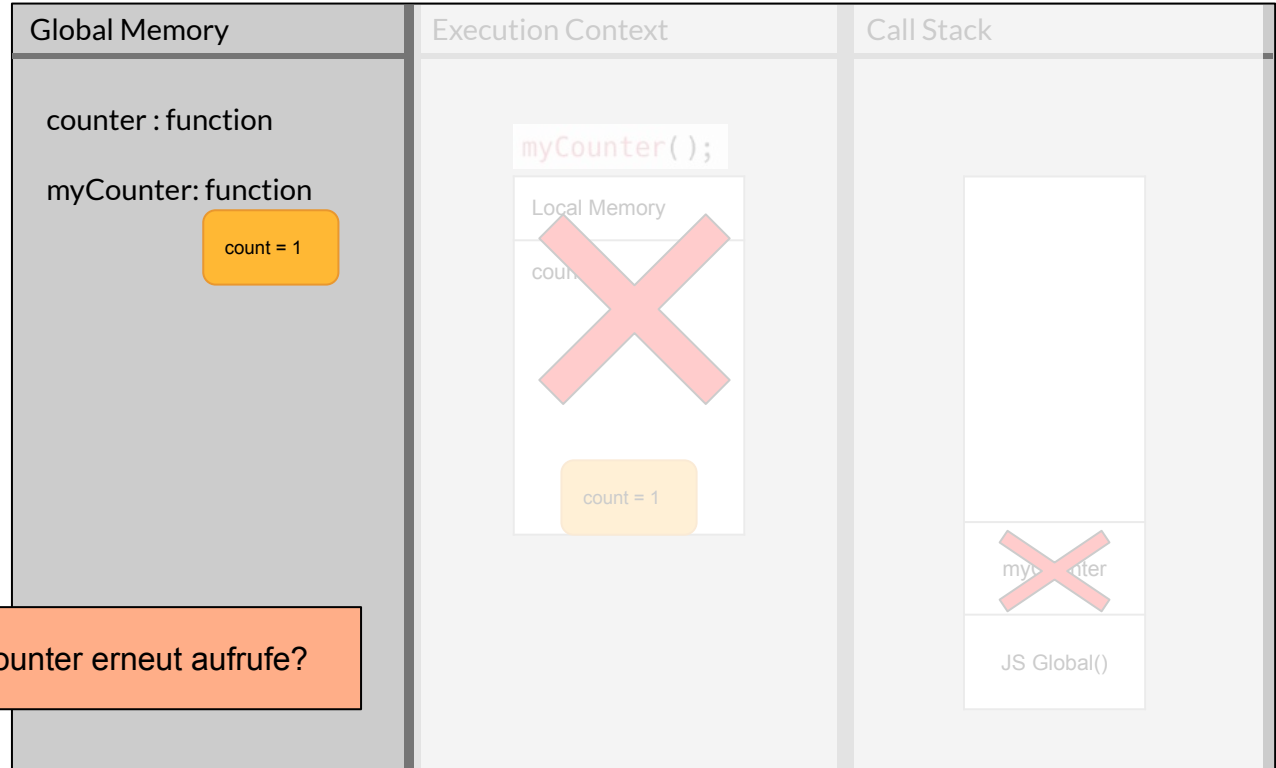
A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
  
myCounter(); // Output: 1
```



A peak under the hood

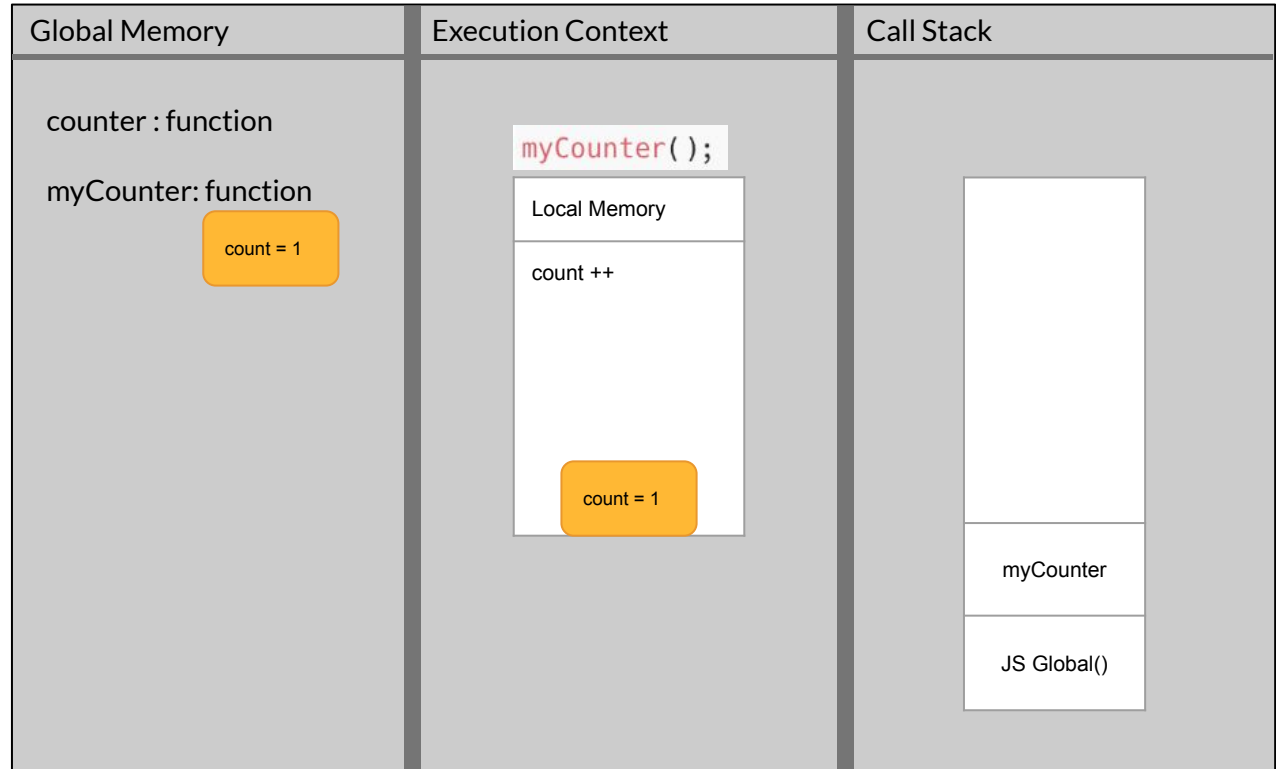
```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
  
myCounter(); // Output: 1
```



Was passiert, wenn ich myCounter erneut aufrufe?

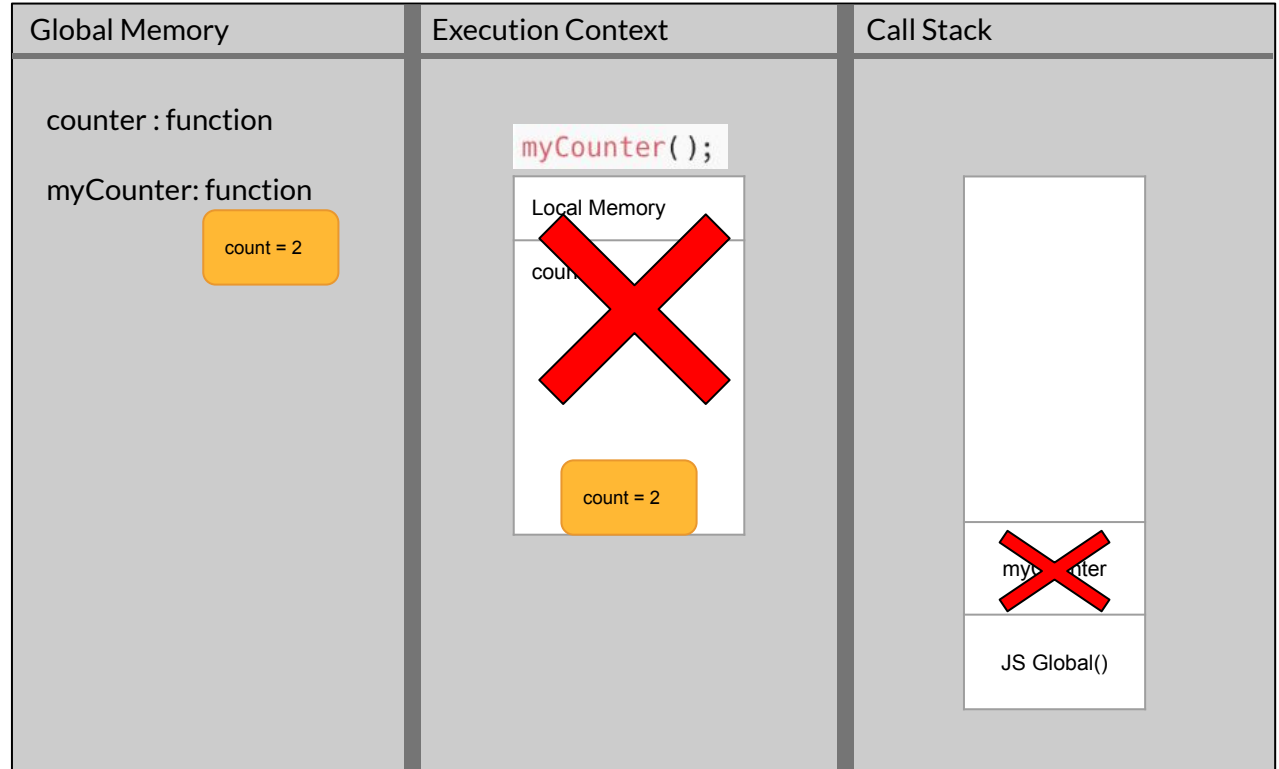
A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
  
myCounter(); // Output: 1  
myCounter(); // Output: 2
```




A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
  
myCounter(); // Output: 1  
myCounter(); // Output: 2
```



A peak under the hood

```
function counter() {  
  var count = 0;  
  
  function increment() {  
    count++;  
    console.log(count);  
  }  
  
  return increment;  
}  
  
var myCounter = counter();  
  
myCounter(); // Output: 1  
myCounter(); // Output: 2
```



console.dir(myCounter)

```
▼ f increment() ⓘ  
  arguments: null  
  caller: null  
  length: 0  
  name: "increment"  
  ► prototype: {constructor: f}  
    [[FunctionLocation]]: voqoqov  
  ► [[Prototype]]: f ()  
  ▼ [[Scopes]]: Scopes[2]  
    ▼ 0: Closure (counter)  
      count: 2  
      ► [[Prototype]]: Object  
    ► 1: Global {window: Window,
```

Closures in JavaScript enable functions to maintain state that is effectively immutable from external code.

- Erstellen von privaten Variablen und Funktionen
- Reduzierung von Redundanz
- Vermeidung von globalen Variablen
- Verwendung in asynchronem Code
- Implementierung von Callback-Funktionen
- Verwendung in Higher-Order-Funktionen

Example: Private Variables



```
export function checkUserAccess(userId) {  
  let hasAccess = false;  
  
  hasAccess = fetchAccess();  
  
  // Return a function that can be used to check access without  
exposing implementation details  
  return function () {  
    return hasAccess;  
  };  
}
```

Cache mit Closures



```
export function createExpensiveCalculation() {  
  const cache = new Map();  
  
  return function (n: number) {  
    if (cache.has(n)) {  
      console.log("hit", n);  
      return cache.get(n);  
    }  
    console.log("miss", n);  
    const result = n * n;  
    cache.set(n, result);  
    return result;  
  };  
}
```

```
const expensiveCalculation1 = createExpensiveCalculation();
```

```
expensiveCalculation1(2); // miss 2  
expensiveCalculation1(2); // hit 2
```


Example: Cache with Closures with 2 Instances

```
export function createExpensiveCalculation() {  
  const cache = new Map();  
  
  return function (n: number) {  
    if (cache.has(n)) {  
      console.log("hit", n);  
      return cache.get(n);  
    }  
    console.log("miss", n);  
    const result = n * n;  
    cache.set(n, result);  
    return result;  
  };  
}  
  
// Create two instances of the expensive calculation function  
const expensiveCalculation1 = createExpensiveCalculation();  
const expensiveCalculation2 = createExpensiveCalculation();  
  
expensiveCalculation1(2); // miss 2  
expensiveCalculation1(2); // hit 2  
  
expensiveCalculation2(2); // miss 2 -> This will miss! compare this  
with cache-without-closure  
expensiveCalculation2(2); // hit 2
```

Cache ohne Closure

```
const cache = new Map();
export function createExpensiveCalculation(n: number) {
  // Without closure, we would need to define a global cache and
  // keep track of each
  // cache manually, which can be error-prone and difficult to
  // maintain

  if (cache.has(n)) {
    console.log("hit", n);
    return cache.get(n);
  }
  console.log("hit", n);
  const result = n * n;
  cache.set(n, result);
  return result;
}

// Create two instances of the expensive calculation function
const expensiveCalculation1 = createExpensiveCalculation(2);
const expensiveCalculation2 = createExpensiveCalculation(2);

expensiveCalculation1(); // miss 2
expensiveCalculation2(); // hit 2
```

Cache ohne Closure ---> Redundanz

```
const cache1 = new Map<number, number>();
const cache2 = new Map<number, number>();

function expensiveCalculation1(n: number) {
  if (cache1.has(n)) {
    return cache1.get(n);
  }

  const result = n * n;
  cache1.set(n, result);
  return result;
}

function expensiveCalculation2(n: number) {
  if (cache2.has(n)) {
    return cache2.get(n);
  }

  const result = n * n;
  cache2.set(n, result);
  return result;
}
```

Functional Programming (Higher order functions)

```
export function addErrorLogging(fn) {  
  // Returns a new function that closes over the original function  
  (fn) and handles errors  
  return function (...args) {  
    try {  
      return fn.apply(this, args);  
    } catch (error) {  
      console.error(`Error occurred: ${error.message}`);  
      // Re-throws the error to propagate it  
      throw error;  
    }  
  };  
}  
  
export function divide(a, b) {  
  if (b === 0) {  
    throw new Error("Cannot divide by zero.");  
  }  
  return a / b;  
}  
  
export const loggedDivide = addErrorLogging(divide);  
  
console.log(loggedDivide(10, 2)); // Output: 5  
console.log(loggedDivide(10, 0)); // Output: Error occurred: Cannot  
divide by zero. (error thrown)
```

Example: Functional Programming (Higher order functions) ohne Closure

```
export function addErrorLogging(fn) {
  const args = arguments
  try {
    return fn.apply(this, args);
  } catch (error) {
    console.error(`Error occurred: ${error.message}`);
    throw error;
  }
}

export function divide(a, b) {
  if (b === 0) {
    throw new Error("Cannot divide by zero.");
  }
  console.log('result: ' + a / b)
  return a / b;
}

/**
 * loggedDivide is tightly coupled to divide. If the implementation
 * | details of divide changes,
 * then usage of loggedDivide has to be changed.
 * In addition, loggedDivide is not reuseable with other parameters.
 */
const loggedDivide = addErrorLogging(divide.bind(null, 10, 2));
```

React

- <https://www.swyx.io/hooks> ✓
- <https://epicreact.dev/how-react-uses-closures-to-avoid-bugs/>
-

Nachteile

- **Memory Leak:** Closures können zu Speicherproblemen führen, da sie Variablen und Funktionen im Speicher halten, auch wenn sie nicht mehr benötigt werden. Es ist wichtig sicherzustellen, dass es eine Möglichkeit gibt, die restlichen Daten in Closure zu löschen. (Falls die Gefahr besteht, dass der Garbage Collector den verwendeten Speicher aufgrund bestehender Referenzen nicht löschen kann.)
- **Komplexität:** Closures können die Komplexität des Codes erhöhen, insbesondere wenn sie verschachtelt werden und mehrere Ebenen von Funktionalität umfassen.
- **Sicherheitsrisiko:** Closures können ein Sicherheitsrisiko darstellen, da sie auf Variablen außerhalb ihrer eigenen Funktion zugreifen können. Wenn diese Variablen sensibel sind, kann dies zu Sicherheitsproblemen führen. Die Entwickler sollten darauf achten, wie viel von der internen Logik über das Closure offengelegt wird.

Memory Leak

```
function dataApi() {
  const data = [];

  function fetchData() {
    data.push(createLargeObject());
  }

  function unsubscribe() {
    data.length = 0;
  }

  return {
    fetchData,
    unsubscribe,
  };
}

const dataFromApi = dataApi();

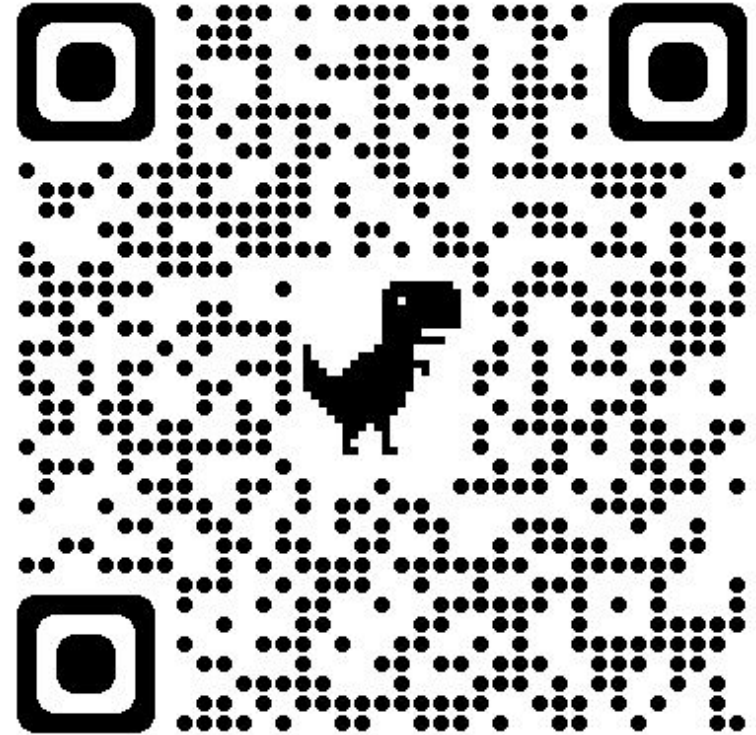
function createLargeObject() {
  let obj = {};
  for (let i = 0; i < 10000; i++) {
    obj[i] = Math.random().toString(36).substring(2, 15);
  }
  return obj;
}
```


Codes: github.com/Confrontend/JS-Closure

master 1 branch 0 tags Go to file Code

Confrontend Js Closures ccaa588 5 hours ago 1 commit

cache	Js Closures	5 hours ago
counter	Js Closures	5 hours ago
debounce	Js Closures	5 hours ago
higher-order	Js Closures	5 hours ago
memory-leak	Js Closures	5 hours ago
private	Js Closures	5 hours ago
react-closure	Js Closures	5 hours ago
react-stale-closure-useeffect	Js Closures	5 hours ago
react-stale-closure	Js Closures	5 hours ago
scope	Js Closures	5 hours ago
.gitignore	Js Closures	5 hours ago
higherOrder.ts	Js Closures	5 hours ago
index.html	Js Closures	5 hours ago
package.json	Js Closures	5 hours ago
tsconfig.json	Js Closures	5 hours ago
utils.js	Js Closures	5 hours ago
yarn.lock	Js Closures	5 hours ago



Referenzen & Weiterlernen

- [Master the JavaScript Interview: What is a Closure? | by Eric Elliott | JavaScript Scene | Medium](#)
- [JavaScript Closure: A Simple Explanation](#)
- [Practical Closures - iRi](#)
- [Be Aware of Stale Closures when Using React Hooks](#)
- [JavaScript the Hard Parts: Closure, Scope & Execution Context](#)
- [Closures - JavaScript | MDN](#)
- [The Ultimate Guide to Hoisting, Scopes, and Closures in JavaScript](#)
- code snippets: [Carbon](#)



SOFTWARE & SYSTEM DEVELOPER
INNOVATION PARTNER
COFFEE LOVER

Geschäftsführer:

Peter F. Fellingner, Markus Hartinger

Friedenheimer Brücke 20 | 80639 Munich | +49.89.45 23 47 - 0

Friedrichstraße 45 | 70174 Stuttgart | +49.711.21 95 28 - 0

Klostergasse 3 | 04109 Leipzig | +49.341.22 178 - 0

Zeil 109 | 60313 Frankfurt am Main | +49.89.45 23 47 - 0

office@jambit.com

www.jambit.com

Dieses Dokument ist vertraulich. Eine Weitergabe an Dritte ist nur mit Zustimmung von jambit möglich.

THANK YOU!

Hamed Fatehi
Software architect

Hamed.Fatehi@jambit.com