Scope Chain

 (https://github.com/learn-co-curriculum/phase-1-scope-chain-readme)  (https://github.com/learn-co-curriculum/phase-1-scope-chain-readme/issues/new)

Learning Goals

- Create nested functions.
- Explain that the environment in which a function is created gets added to its scope chain.
- Describe how the scope chain makes variables and functions declared in the outer environment available within a nested function.
- Explain how the JavaScript engine makes a first pass over your code before executing it.
- Describe what happens during the *execution phase* of the JavaScript runtime.

Introduction

Every function in JavaScript has access to a *scope chain*, which includes references to the function's outer scope (the scope in which the function was declared), the outer scope's outer scope, and so on. In this lesson, we'll discuss how the scope chain allows us to access variables and functions declared in outer scopes within an inner function. We'll also talk about what's happening under the hood when we run JavaScript code and how that impacts *identifier resolution* and the *scope chain*.

Nested scopes and the scope chain

In addition to the `#engineering` channel, every software engineer is a member of Flatbook's `#general` channel. Engineers can see all of the messages sent in both channels. If a message in `#general` piques our interest, we can refer to the message in `#engineering` despite the fact that it was posted in `#general`. To bend the analogy back towards functions and scope, everything *declared* in `#general` is accessible in `#engineering`. `#general`, our global scope, is effectively the *outer scope* for `#engineering`.

For a function declared at the top level of our JavaScript file (that is, not declared inside of another function), its outer scope is the *global scope*. When that new function is invoked, it can access all of the variables and functions declared in the global scope. Upon invocation, the function creates a new scope and **retains a reference to the *outer scope* in which it was declared**. Inside the new function's body, in addition to

variables and functions declared in that function, **we also have access to variables and functions declared in the outer scope**. Let's see that in action:

```
const globalVar = 1;

function firstFunc() {
  const firstVar = 2;

  return firstVar + globalVar;
}

firstFunc();
// => 3
```

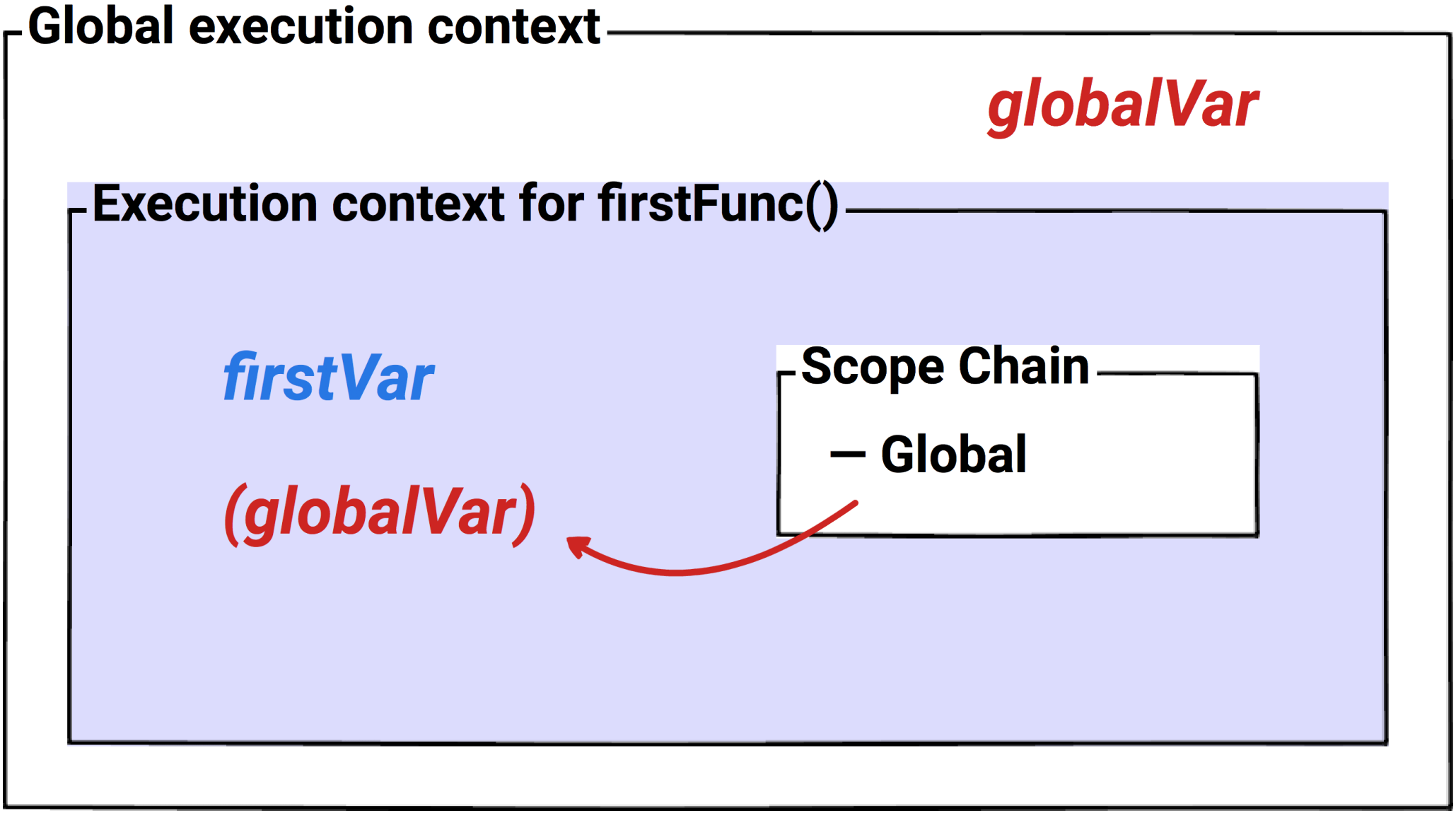
`firstVar` is declared inside the function, and `globalVar` is declared in the outer scope, but we have access to **both** inside `firstFunc()`.

When we invoke `firstFunc()`, the first line of code inside the function, `const firstVar = 2;`, runs first, creating a new local variable. When the JavaScript engine reaches the function's second line, it sees the reference to `firstVar` and says, "Great, I know what that means: it's a local variable!" Then, the engine encounters the reference to `globalVar` and says, "What the heck is this?! That's not declared locally!"

When the engine can't find a local match, it then goes looking in the outer scope and — voilà! — finds a match there. Because of the way functions can look up variables declared in outer scopes, we say they have access to a *scope chain*. Through the scope chain, a function has access to all variables and functions declared in its outer scope.

Top Tip: What matters for the scope chain is where the function is declared — not where it is invoked.

We can think of JavaScript scopes as a nested system:

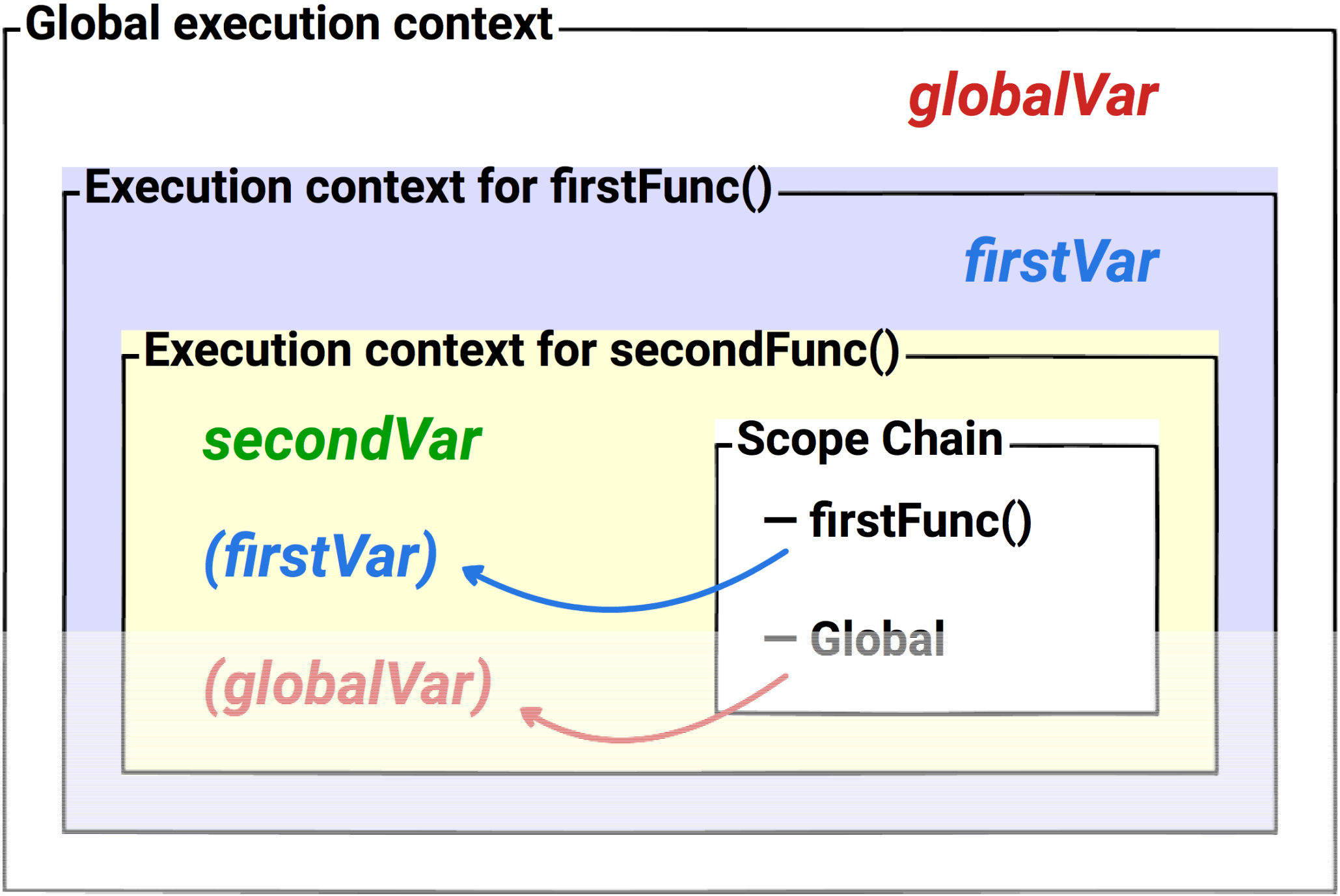


All variables and functions declared in outer scopes are available in inner scopes via the scope chain. This can go on ad infinitum, with functions nested in functions nested in functions, each new level creating a new scope that can reference functions and variables declared in its outer scopes through the scope chain:

```
const globalVar = 1;
```

```
function firstFunc() {
  const firstVar = 2;

  function secondFunc() {
    const secondVar = 3;

    return secondVar + firstVar + globalVar;
  }

  const resultFromSecondFunc = secondFunc();

  return resultFromSecondFunc;
}

firstFunc();
// => 6
```

Inside `firstFunc()`, we've defined a second function, `secondFunc()`. That second function creates yet another scope, and in it we can reference `firstVar` **and** `globalVar` via the scope chain:


Global execution context
globalVar
Execution context for firstFunc()
firstVar
Execution context for secondFunc()
secondVar
(firstVar)
(globalVar)
Scope Chain
— firstFunc()
— Global

Inside `secondFunc()`, `firstVar` is accessible via the outer scope, and `globalVar` is accessible via the outer scope's outer scope. Head spinning? Just remember that the scope chain is **scopes all the way down** (https://en.wikipedia.org/wiki/Turtles_all_the_way_down). If `a()` is declared inside `b()` and `b()` is declared inside `c()`, `a()` has access to functions and variables declared in its own scope, `b()`'s scope, and `c()`'s scope. That's the scope chain in action!

NOTE: The scope chain only goes in one direction. An outer scope **does not have access to things declared in an inner scope**. In the previous code snippet, `firstFunc()` **cannot access** `secondVar`. In addition, two functions declared in the same scope do not have access to anything declared in the other's scope:

```
const fruit = "Apple";

function first() {
  const vegetable = "Broccoli";

  console.log("fruit:", fruit);
  console.log("vegetable:", vegetable);
  console.log("legume:", legume);
}

function second() {
  const legume = "Peanut";

  console.log("fruit:", fruit);
  console.log("legume:", legume);
  console.log("vegetable:", vegetable);
}
```

Both `first()` and `second()` have access to `fruit`, but `first()` cannot access `legume` and `second()` cannot access `vegetable`:

```
first();
// LOG: fruit: Apple
// LOG: vegetable: Broccoli
```

```
// ERROR: Uncaught ReferenceError: legume is not defined

second();
// LOG: fruit: Apple
// LOG: legume: Peanut
// ERROR: Uncaught ReferenceError: vegetable is not defined
```

Okay, we have an idea of what the scope chain is, but how does it actually work under the hood?

The JavaScript engine and identifier resolution

Identifiers

As a brief refresher, when we declare a variable or a function, we provide a name that allows us to refer back to it:

```
const myVar = "myVar refers to the variable that contains this string";
// => undefined

function myFunc() {
  return "myFunc refers to this function that returns this string";
}
// => undefined
```

We call those names *identifiers* because they allow us to **identify** the variable or function we're referring to.

The JavaScript engine

When our JavaScript code is run in the browser, the JavaScript engine actually makes two separate passes over our code:

Compilation phase

The first pass is the *compilation phase*, in which the engine steps through our code line-by-line:

1. When it reaches a variable declaration, the engine allocates memory and sets up a reference to the variable's identifier, e.g., `myVar`.
2. When the engine encounters a function declaration, it does three things:
    - Allocates memory and sets up a reference to the function's identifier, e.g., `myFunc`.
    - Creates a new execution context with a new scope.
    - Adds a reference to the parent scope (the outer environment) to the scope chain, making variables and functions declared in the outer environment available in the new function's scope.

Execution phase

The second pass is the *execution phase*. The JavaScript engine again steps through our code line-by-line, but this time it actually runs our code, assigning values to variables and invoking functions.

One of the engine's tasks is the process of matching identifiers to the corresponding values stored in memory. Let's walk through the following code:

```
const myVar = 42;

myVar;
// => 42
```

During the compilation phase, a reference to the identifier `myVar` is stored in memory. The variable isn't yet assigned a value, and the second line (`myVar;`) is skipped over entirely because it isn't a declaration.

During the execution phase, the value `42` is assigned to `myVar`. When the engine reaches the second line, it sees the identifier `myVar` and resolves it to a value through a process known as *identifier resolution*. The engine first checks the current scope to see if `myVar` has been declared in it. If it finds no declaration for `myVar` in the current scope, the engine then starts moving up the scope chain, checking the parent scope and then the parent scope's parent scope and so on until it finds a matching declared identifier or reaches the global scope. If the engine traverses all the way up to the global scope and still can't find a match, it will throw a `ReferenceError` and inform you that the identifier is not declared anywhere in the scope chain.

Let's look at an example. Remember, the engine will continue to move up the scope chain **only** if it can't find a matching identifier in the current scope. Because of this, we can actually use the same identifier to declare variables or functions in multiple scopes:

```
const myVar = 42;

function myFunc() {
  const myVar = 9001;

  return myVar;
}

myFunc();
// => 9001
```

During the compilation phase, a reference to `myVar` is created in the global scope, and a reference to a **different** `myVar` is created in `myFunc()` 's scope. The global `myVar` exists in the scope chain for `myFunc()` , but the engine never makes it that far. The engine finds a matching reference within `myFunc()` , and it resolves the `myVar` identifier to `9001` without having to traverse up the scope chain.

Conclusion

This topic might feel a bit esoteric, but it's critical to understanding how identifier lookups happen in JavaScript. That is, when the JavaScript engine encounters a variable or function, how it knows what value or function to retrieve from memory. If the engine finds the identifier declared locally, it uses that value. However, if it doesn't find a local match, it then looks up (or down, depending on your perspective) the scope chain until it either finds a match in an outer scope or throws an `Uncaught ReferenceError` .

Resources

- **MDN: Functions — Name conflicts (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Name_conflicts)**