8.6 Closures

Execution context and closures

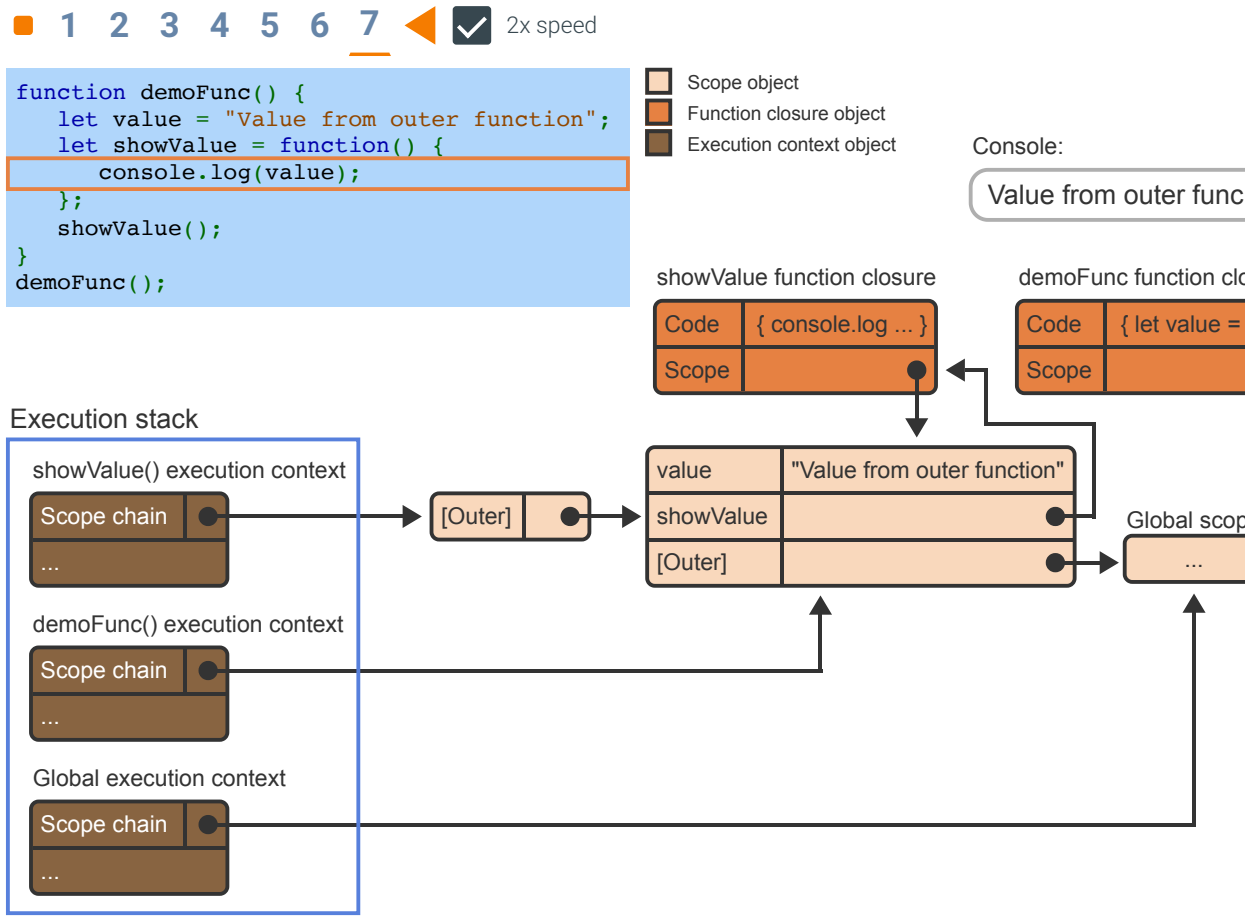
An ***execution context*** is an object that stores information needed to execute JavaScript code, and includes, but is not limited to:

- information about code execution state, such as the line of code being executed and the line to return to when a function completes, and
- a reference to a scope chain.

Execution contexts are stored on an execution stack. The ***current execution context*** (***running execution context***) is the execution context at the top of the execution stack. The ***current scope chain*** is the scope chain of the current execution context.

A ***closure*** is a combination of a function's code and a reference to a scope chain. When a JavaScript function is declared, a closure is created that includes the function's code and a reference to the current scope chain. Closures are what allow an inner function to access the outer function's variables.

PARTICIPATION ACTIVITY 8.6.1: A closure allows an inner function to access the outer function's scope.



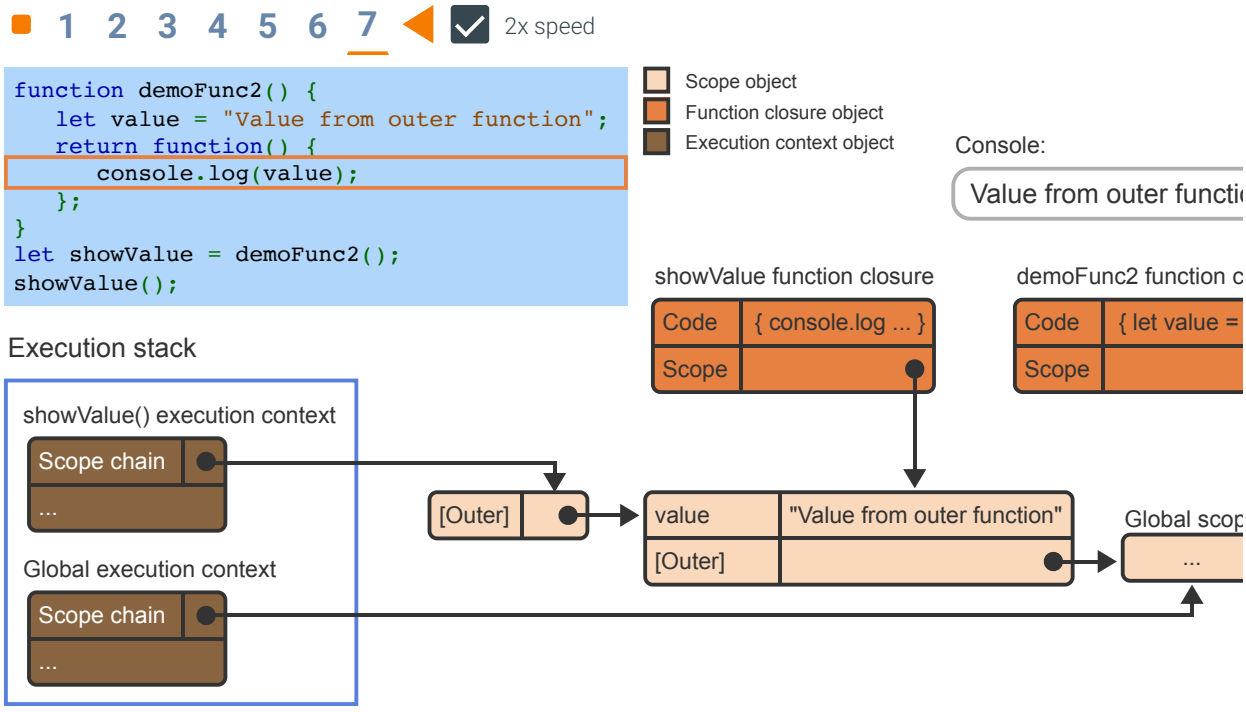
showValue() finds the value variable in the scope chain's second object.

Captions

1. Initially, code is executing in the global context. So the execution stack has 1 execution context object. The referenced scope chain consists of only the global scope object.
2. When the demoFunc() function is declared, a closure is created. The closure's scope equals the execution context's scope chain, which consists of only the global scope object.
3. Calling demoFunc() does two things: First, a new execution context is pushed onto the execution stack, with the scope chain set to the demoFunc() function closure.
4. Second, a new scope object for demoFunc's variables is prepended to the current scope chain. The value string is set inside the scope object.
5. Declaring showValue as an inner function creates a closure that references the scope chain of the current execution context.
6. Calling showValue() pushes a new execution context onto the stack, and prepends to the current scope chain. showValue() has no local variables, so the front of the scope chain has only an outer reference.
7. showValue() finds the value variable in the scope chain's second object.

Feedback?

PARTICIPATION ACTIVITY 8.6.2: Closures can access local variables from functions that have completed execution.



The value string can be found in the current scope chain and logged to the console.

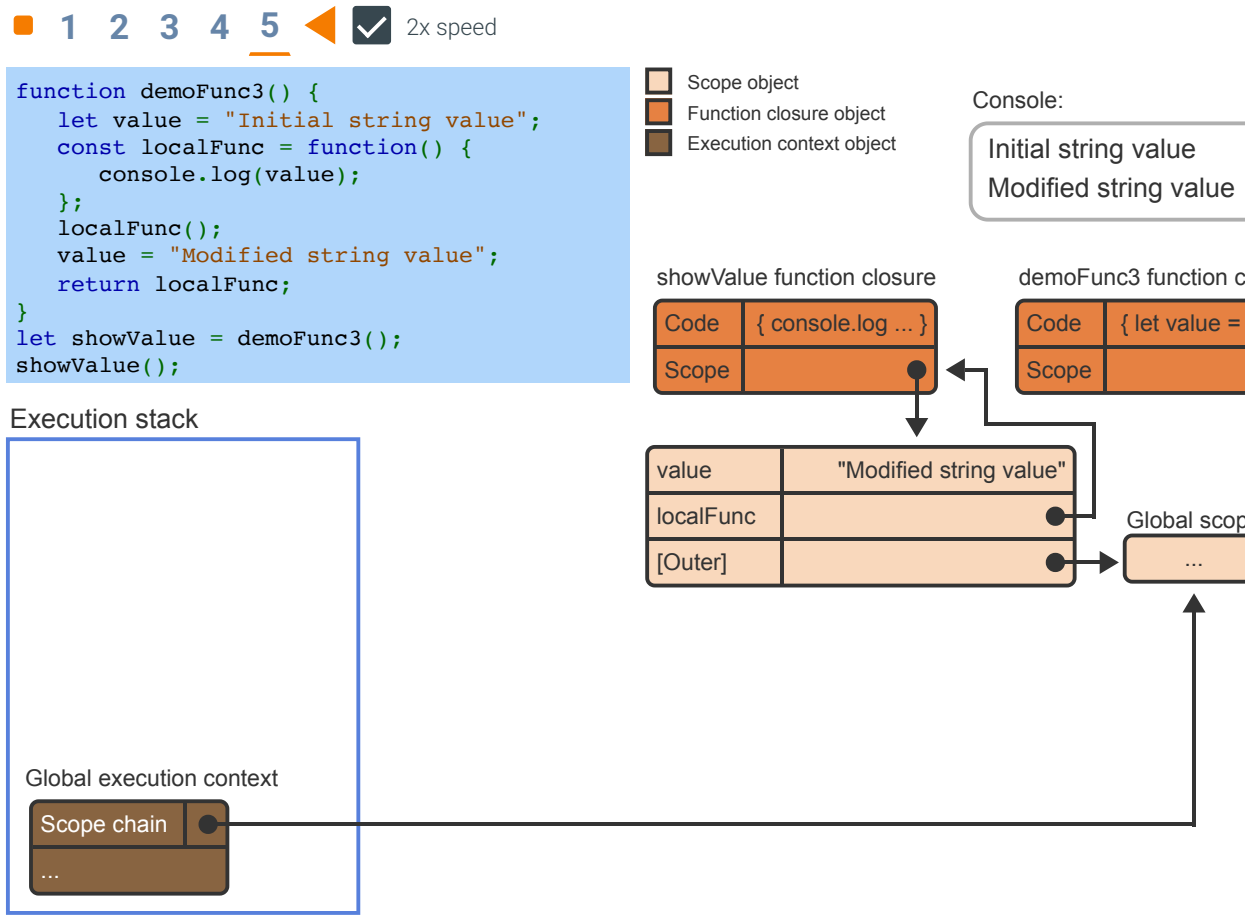
Captions

1. A closure is created for the demoFunc2() function, referencing the execution context's scope chain, which consists of only the global scope object.

2. demoFunc2() is called, creating a new execution context and prepending a scope object to that context's scope chain.
3. An anonymous function is created inside demoFunc2(), referencing the current scope chain.
4. The closure is returned and assigned to the global showValue variable. The current execution context is now the global execution context.
5. demoFunc2() has completed, and the associated execution context has been popped off the stack. But the scope object from demoFunc2() stays in memory, due to being referenced by the showValue closure.
6. Calling showValue() first pushes a new execution context with a scope chain equal to that of the showValue() closure. Then a new scope object is prepended.
7. The value string can be found in the current scope chain and logged to the console.

Feedback?

PARTICIPATION ACTIVITY 8.6.3: Changing a local variable after a closure's creation can affect functionality.



Calling showValue() logs the modified string value.

Captions

1. A closure is created for demoFunc3(). demoFunc3() is then called, prepending a scope object to the current scope chain and setting the string's initial value.
2. localFunc is created with a reference to the current scope chain. Calling localFunc() immediately after logs the initial string value.

3. After calling localFunc(), demoFunc3() resumes execution and changes the string's value. The function closure object itself hasn't changed, but the referenced scope object has.
4. localFunc is returned and assigned to showValue.
5. Calling showValue() logs the modified string value.

Feedback?

PARTICIPATION ACTIVITY 8.6.4: Execution context and closures.

Assume the following code is executed before each question:

```
let inc = null;
let dec = null;
let log;
function createIncDecLog() {
   let number = 0;
   inc = function() { number++; };
   dec = function() { number--; };
   log = function() { console.log(number); };
}
createIncDecLog();
```

1) What does the following code log?

```
inc();
inc();
log();
```

- 0
- 2 (selected)

Correct

Calling `inc()` increments the `number` value stored in a scope object. The same scope object is referenced by the `log()` function closure, so the logged value will be 2.

2) What does the following code log?

```
inc();
dec();
dec();
inc();
inc();
log();
```

- -2
- 1 (selected)
- 3

Correct

All 3 closures share the same scope, and thus the same `number`. Two decrements and three increments occur, setting `number` to 1 before the `log()` call.

3) What does the following code log?

```
inc();
inc();
inc();
createIncDecLog();
log();
```

- (●) 0
- ( ) 3

Correct

Calling `createIncDecLog()` creates a new scope object and recreates the 3 closures, each referencing the scope object with `number` set to 0.

Feedback?

Closures and loops

Each time a loop iteration begins, a new block scope object is prepended to the current scope chain. Block-scoped variables declared with `let` or `const` are stored in the block scope object. When a loop iteration ends, the block scope object is removed from the front of the current scope chain. Therefore, a loop that executes N iterations will have caused N distinct block scope objects to have been prepended and then removed from the current scope chain.

PARTICIPATION ACTIVITY 8.6.5: A new block scope is created each loop iteration, storing block-scoped variables.

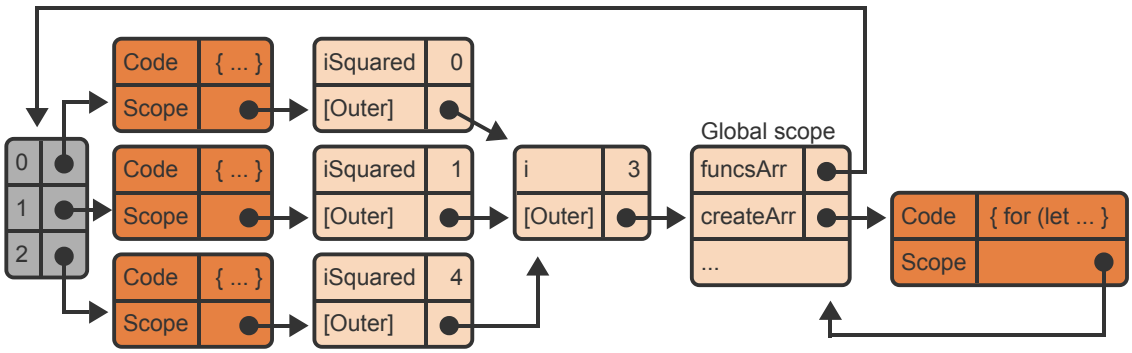
1 2 3 4 5 6 7 2x speed

```
let funcsArr = [null,null,null];

function createArr() {
   for (let i = 0; i < 3; i++) {
      const iSquared = i * i;
      funcsArr[i] = function() {
         console.log(iSquared);
      };
   }
}

createArr();
for (let func of funcsArr) {
   func();
}
```

Scope object
Function closure object
JavaScript array object

Console:

```
0
1
4
```



The three functions are called, logging 0, 1, and 4.

Captions

1. funcsArr and createArr() are created in the global scope.
2. Calling createArr() prepends to the current scope chain. Entering the first loop iteration prepends again, adding a block scope object that stores iSquared.
3. The first closure is created, referencing the current scope chain.
4. Ending the first loop iteration removes the block scope from the current scope chain.
5. Starting the next iteration prepends a new block scope, and the next closure is created.
6. The third closure is created similarly.
7. The three functions are called, logging 0, 1, and 4.

Feedback?

PARTICIPATION ACTIVITY 8.6.6: Closures and loops.

1) At the start of each loop iteration, a new block scope object is prepended to the current scope chain.

- (●) True
- ( ) False

Correct

A loop's body is a block scope, so each new iteration prepends a new block scope object.

2) At the end of each loop iteration, a block scope object is removed from the front of the current scope chain.

- (●) True
- ( ) False

Correct

The end of a loop iteration implies the end of that iteration's scope, so the front scope object is removed.

3) For a loop that executes N times, any variable declared inside a loop's body is stored in N distinct block scope objects.

- ( ) True
- (●) False

Correct

Only variables declared with `let` or `const` are stored in N distinct block scope objects. Variables declared with `var` are in the function's scope object, not the loop's block scope object.

Feedback?

Closures and loops: var declarations

Variables declared with `var` always have function scope and are never stored in a loop's block scope object. *A common error is to declare a variable inside a loop with `var`, in an attempt to capture a variable's value at that point in time.* However, each iteration reassigns the same

variable in the function's scope object. Using `const` or `let` inside a loop can often solve this problem.

PARTICIPATION ACTIVITY 8.6.7: Variables declared with var have function scope, even if declared inside a loop.

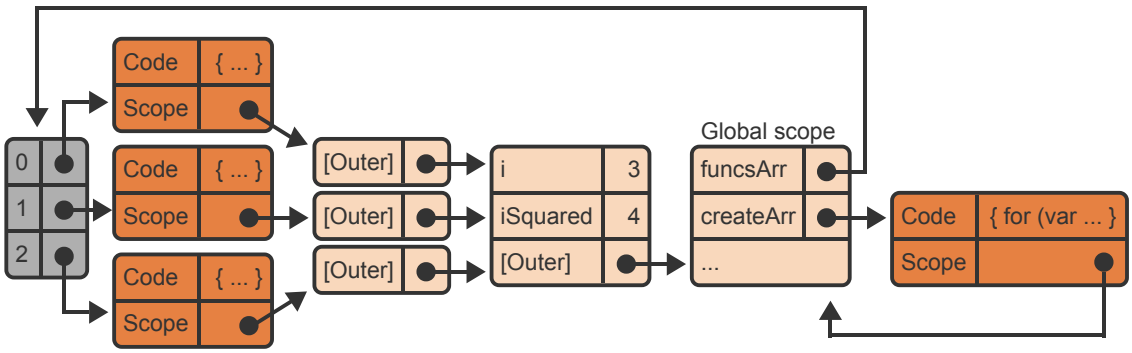
1 2 3 4 5 2x speed

```
var funcsArr = [null,null,null];

function createArr() {
   for (var i = 0; i < 3; i++) {
      var iSquared = i * i;
      funcsArr[i] = function() {
         console.log(iSquared);
      };
   }
}

createArr();
for (var func of funcsArr) {
   func();
}
```

Scope object
Function closure object
JavaScript array object

Console:

```
4
4
4
```



Calling the functions logs 4 three times.

Captions

1. The funcsArr array and the createArr closure are initialized the same as in the previous example. The only change is that iSquared is declared with var instead of const.
2. Calling createArr() prepends to the current scope chain. Entering the first loop iteration then prepends a block scope.
3. Only const and let create block-scoped variables. iSquared is declared with var and is therefore in the function scope object along with i.
4. After createArr() completes, 3 distinct closures reference 3 distinct, empty block scopes. Each scope references the outer scope, where iSquared is stored once with a value of 4.
5. Calling the functions logs 4 three times.

Feedback?

PARTICIPATION ACTIVITY 8.6.8: Closures and loops.

1) What does the following code log to the console?

```
function
getFunction() {
   var
functionToReturn =
null;
   var i = 0;
   while (i < 5) {
      if (i === 0) {

functionToReturn =
function() {
console.log(i); };
      }
      i++;
   }
   return
functionToReturn;
}
const theFunction =
getFunction();
theFunction();
```

- 0
- 4
- (•) 5

Correct

The closure is created when `i` is 0, and the closure references a function scope object with `i`'s value. `i`'s value is changed to 5 within the scope object before `getFunction()` returns. Then the returned function is called, logging `i`'s latest value of 5.

2) What does the following code log to the console?

```
function
getFunction() {
   var
functionToReturn =
null;
   var i = 0;
   while (i < 5) {
      var saved_i =
i;
      if (i === 0) {

functionToReturn =
function() {
console.log(saved_i);
};
      }
      i++;
   }
   return
functionToReturn;
}
const theFunction =
getFunction();
theFunction();
```

- 0
- (•) 4
- 5

Correct

`saved_i` is declared with `var`, giving the variable function scope, just like the variable `i`. So the most recently assigned value of 4 is logged when `theFunction()` is called.

3) What does the following code log to the console?

```
function
getFunction() {
   var
functionToReturn =
null;
   var i = 0;
   while (i < 5) {
      const saved_i =
i;
      if (i === 0) {

functionToReturn =
function() {
console.log(saved_i);
};
      }
      i++;
   }
   return
functionToReturn;
}
const theFunction =
getFunction();
theFunction();
```

- (●) 0
- ( ) 4

Correct

The block-scoped `saved_i` is stored in a new block scope object that is prepended to the current scope chain at the start of each loop iteration. `saved_i` captures `i`'s value at the moment the closure is created, which is when `i` is 0.

Feedback?

JavaScript runtimes may optimize scope chains

Some JavaScript debugging tools show a closure's scopes. When constructing a closure, the JavaScript runtime may optimize the scope chain by removing scope objects without any variables referenced by the closure's code. Therefore, debugging tools may show scope chains slightly different than those in this section's animations.

Exploring further:

- Closures (MDN)

How was this section? Provide section feedback