## Stackoverflow question about closure concept

function getObj()

{

var objAddress =

{

address: "Client Address",

getAddress: function() {

return this.address;

},

setAddress: function(newAddress)

{

this.address = newAddress;

}

};

var objClient =

{

name: "Client name",

getAddress: function()

{

return objAddress.getAddress();

},

setAddress: function(newAddress) {

objAddress.setAddress(newAddress);

}

};

return objClient;

}

gObj = getObj();

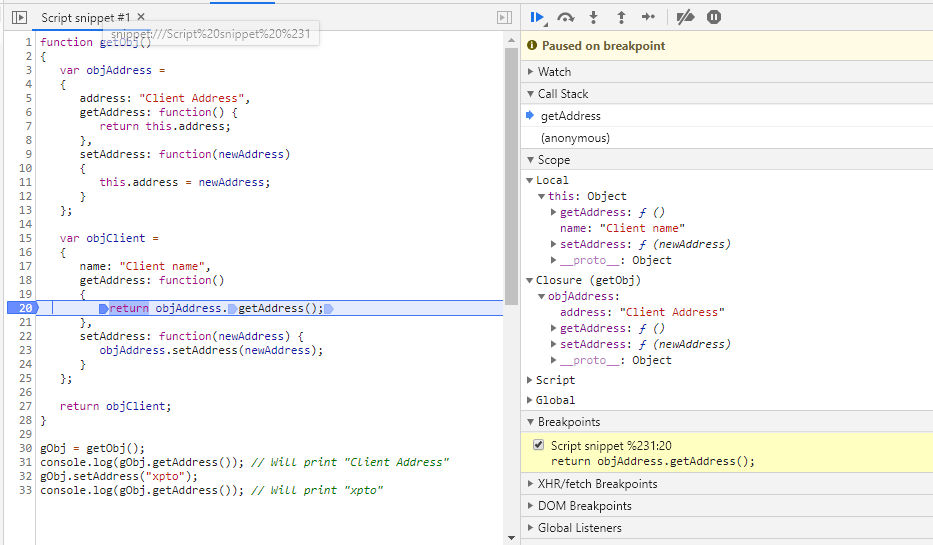
console.log(gObj.getAddress()); // Will print "Client Address"

gObj.setAddress("xpto");

console.log(gObj.getAddress()); // Will print "xpto"

 A programmer thought it would not work since getAddress() calls another method of an object that should not exist after leaving the function. But, as this is working,

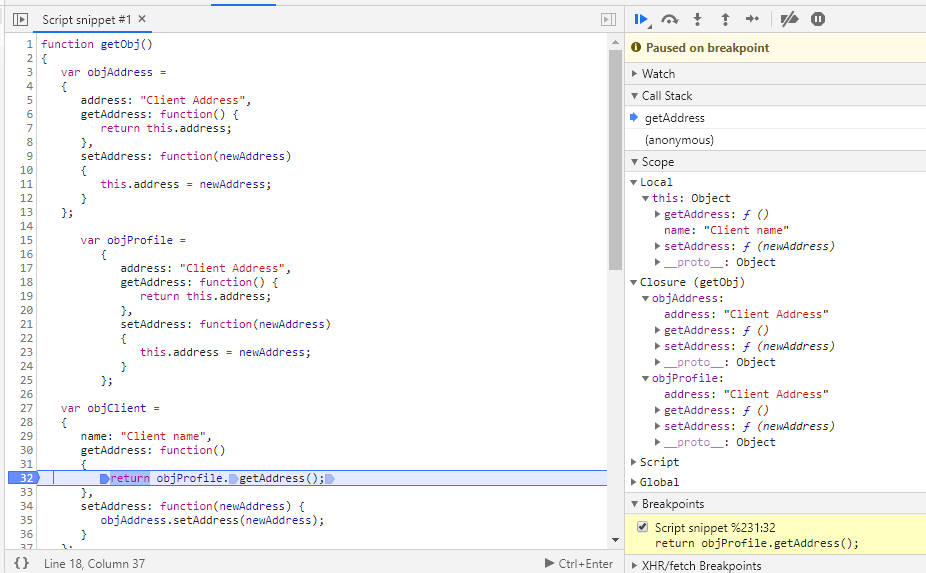
Chrome – Developer Tools



Set the breakpoint at line number 20 and run the code snippet.

On the right side, you will be able to see the local variables and closures coming within the scope of line in execution.

If a new object names objProfile is added to getObj function then that object will also become part of closure.



Comment from stackoverflow:

When you create function, which uses local variable, function "remembers" this local variable. All needed local variables are stored in special object called closure. You cannot directly access it, but function can. Chrome Developer Tools in javascript debug shows closure object:

## 2. Under the hood: variables lifecycle

When the engine works with variables, their lifecycle consists of the following phases:

1. **Declaration phase** is registering a variable in the scope.
2. **Initialization phase** is allocating memory and creating a binding for the variable in the scope. At this step the variable is automatically initialized with undefined.
3. **Assignment phase** is assigning a value to the initialized variable.

## var variables lifecycle

The variable passes the declaration phase and right away the initialization phase at the beginning of the scope, before any statements are executed

Strictly hoisting consists in the idea that a variable is declared and initialized at the beginning of the function scope.

## Function declaration lifecycle

The declaration, initialization and assignment phases happen at once at the beginning of the enclosing function scope (only one step).

## let variables lifecycle

let lifecycle however decouples declaration and initialization phases.

Now let’s study a scenario when the interpreter enters a block scope that contains a let variable statement. Immediately the variable passes the *declaration phase*, registering its name in the scope (step 1).  
Then interpreter continues parsing the block statements line by line.

If you try to access variable at this stage, JavaScript will throw ReferenceError: variable is not defined. It happens because the variable state is *uninitialized*.  
variable is in the *temporal dead zone*.