|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| +100 | **What is a closure?** A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.  The inner function has access not only to the outer function’s variables, but also to the outer function’s parameters. Note that the inner function cannot call the outer function’s *arguments* object, however, even though it can call the outer function’s parameters directly.  You create a closure by adding a function inside another function.  **A Basic Example of Closures in JavaScript:**   |  |  | | --- | --- | |  | function showName (firstName, lastName) { | |  | ​var nameIntro = "Your name is "; | |  | // this inner function has access to the outer function's variables, including the parameter​ | |  | ​function makeFullName () { | |  | ​return nameIntro + firstName + " " + lastName; | |  | } | |  | ​ | |  | ​return makeFullName (); | |  | } | |  | ​ | |  | showName ("Michael", "Jackson"); // Your name is Michael Jackson |   function sayHello(name) {  var text = 'Hello ' + name;  var say = function() { console.log(text); }  say();  }  sayHello('Joe');   Run code snippetHide results  Full page  An example of a closure  Two one sentence summaries:   * A closure is one way of supporting [first-class functions](https://en.wikipedia.org/wiki/First-class_function); it is an expression that can reference variables within its scope (when it was first declared), be assigned to a variable, be passed as an argument to a function, or be returned as a function result. * Or, a closure is a stack frame which is allocated when a function starts its execution, and *not freed* after the function returns (as if a 'stack frame' were allocated on the heap rather than the stack!).   The following code returns a reference to a function:  function sayHello2(name) {  var text = 'Hello ' + name; // Local variable  var say = function() { console.log(text); }  return say;  }  var say2 = sayHello2('Bob');  say2(); // logs "Hello Bob"   Run code snippet  Expand snippet  Most JavaScript programmers will understand how a reference to a function is returned to a variable (say2) in the above code. If you don't, then you need to look at that before you can learn closures. A programmer using C would think of the function as returning a pointer to a function, and that the variables say and say2 were each a pointer to a function.  There is a critical difference between a C pointer to a function and a JavaScript reference to a function. In JavaScript, you can think of a function reference variable as having both a pointer to a function *as well* as a hidden pointer to a closure.  The above code has a closure because the anonymous function function() { console.log(text); } is declared *inside* another function, sayHello2() in this example. In JavaScript, if you use the function keyword inside another function, you are creating a closure.  In C and most other common languages, *after* a function returns, all the local variables are no longer accessible because the stack-frame is destroyed.  In JavaScript, if you declare a function within another function, then the local variables can remain accessible after returning from the function you called. This is demonstrated above, because we call the function say2() after we have returned from sayHello2(). Notice that the code that we call references the variable text, which was a *local variable* of the function sayHello2().  function() { console.log(text); } // Output of say2.toString();  Looking at the output of say2.toString(), we can see that the code refers to the variable text. The anonymous function can reference text which holds the value 'Hello Bob' because the local variables of sayHello2() are kept in a closure.  The magic is that in JavaScript a function reference also has a secret reference to the closure it was created in — similar to how delegates are a method pointer plus a secret reference to an object.  More examples  For some reason, closures seem really hard to understand when you read about them, but when you see some examples it becomes clear how they work (it took me a while). I recommend working through the examples carefully until you understand how they work. If you start using closures without fully understanding how they work, you would soon create some very weird bugs!  Example 3  This example shows that the local variables are not copied — they are kept by reference. It is kind of like keeping a stack-frame in memory when the outer function exits!  function say667() {  // Local variable that ends up within closure  var num = 42;  var say = function() { console.log(num); }  num++;  return say;  }  var sayNumber = say667();  sayNumber(); // logs 43   Run code snippet  Expand snippet  Example 4  All three global functions have a common reference to the *same* closure because they are all declared within a single call to setupSomeGlobals().  var gLogNumber, gIncreaseNumber, gSetNumber;  function setupSomeGlobals() {  // Local variable that ends up within closure  var num = 42;  // Store some references to functions as global variables  gLogNumber = function() { console.log(num); }  gIncreaseNumber = function() { num++; }  gSetNumber = function(x) { num = x; }  }  setupSomeGlobals();  gIncreaseNumber();  gLogNumber(); // 43  gSetNumber(5);  gLogNumber(); // 5  var oldLog = gLogNumber;  setupSomeGlobals();  gLogNumber(); // 42  oldLog() // 5   Run code snippet  Expand snippet  The three functions have shared access to the same closure — the local variables of setupSomeGlobals() when the three functions were defined.  Note that in the above example, if you call setupSomeGlobals() again, then a new closure (stack-frame!) is created. The old gLogNumber, gIncreaseNumber, gSetNumber variables are overwritten with *new* functions that have the new closure. (In JavaScript, whenever you declare a function inside another function, the inside function(s) is/are recreated again *each* time the outside function is called.)  Example 5  This one is a real gotcha for many people, so you need to understand it. Be very careful if you are defining a function within a loop: the local variables from the closure do not act as you might first think.  function buildList(list) {  var result = [];  for (var i = 0; i < list.length; i++) {  var item = 'item' + i;  result.push( function() {console.log(item + ' ' + list[i])} );  }  return result;  }  function testList() {  var fnlist = buildList([1,2,3]);  // Using j only to help prevent confusion -- could use i.  for (var j = 0; j < fnlist.length; j++) {  fnlist[j]();  }  }  testList() //logs "item2 undefined" 3 times   Run code snippet  Expand snippet  The line result.push( function() {console.log(item + ' ' + list[i])} adds a reference to an anonymous function three times to the result array. If you are not so familiar with anonymous functions think of it like:  pointer = function() {console.log(item + ' ' + list[i])};  result.push(pointer);  Note that when you run the example, "item2 undefined" is alerted three times! This is because just like previous examples, there is only one closure for the local variables for buildList. When the anonymous functions are called on the line fnlist[j](); they all use the same single closure, and they use the current value for i and item within that one closure (where i has a value of 3 because the loop had completed, and item has a value of 'item2'). Note we are indexing from 0 hence item has a value of item2. And the i++ will increment i to the value 3.  Example 6  This example shows that the closure contains any local variables that were declared inside the outer function before it exited. Note that the variable alice is actually declared after the anonymous function. The anonymous function is declared first; and when that function is called it can access the alice variable because alice is in the same scope (JavaScript does [variable hoisting](https://stackoverflow.com/a/3725763/1269037)). Also sayAlice()() just directly calls the function reference returned from sayAlice() — it is exactly the same as what was done previously but without the temporary variable.  function sayAlice() {  var say = function() { console.log(alice); }  // Local variable that ends up within closure  var alice = 'Hello Alice';  return say;  }  sayAlice()();// logs "Hello Alice"   Run code snippet  Expand snippet  Tricky: note also that the say variable is also inside the closure, and could be accessed by any other function that might be declared within sayAlice(), or it could be accessed recursively within the inside function.  Example 7  This final example shows that each call creates a separate closure for the local variables. There is *not* a single closure per function declaration. There is a closure for *each call* to a function.  function newClosure(someNum, someRef) {  // Local variables that end up within closure  var num = someNum;  var anArray = [1,2,3];  var ref = someRef;  return function(x) {  num += x;  anArray.push(num);  console.log('num: ' + num +  '; anArray: ' + anArray.toString() +  '; ref.someVar: ' + ref.someVar + ';');  }  }  obj = {someVar: 4};  fn1 = newClosure(4, obj);  fn2 = newClosure(5, obj);  fn1(1); // num: 5; anArray: 1,2,3,5; ref.someVar: 4;  fn2(1); // num: 6; anArray: 1,2,3,6; ref.someVar: 4;  obj.someVar++;  fn1(2); // num: 7; anArray: 1,2,3,5,7; ref.someVar: 5;  fn2(2); // num: 8; anArray: 1,2,3,6,8; ref.someVar: 5;   Run code snippet  Expand snippet  Summary  If everything seems completely unclear then the best thing to do is to play with the examples. Reading an explanation is much harder than understanding examples. My explanations of closures and stack-frames, etc. are not technically correct — they are gross simplifications intended to help understanding. Once the basic idea is grokked, you can pick up the details later.  Final points:   * Whenever you use function inside another function, a closure is used. * Whenever you use eval() inside a function, a closure is used. The text you eval can reference local variables of the function, and within eval you can even create new local variables by using eval('var foo = …') * When you use new Function(…) (the [Function constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function)) inside a function, it does not create a closure. (The new function cannot reference the local variables of the outer function.) * A closure in JavaScript is like keeping a copy of all the local variables, just as they were when a function exited. * It is probably best to think that a closure is always created just an entry to a function, and the local variables are added to that closure. * A new set of local variables is kept every time a function with a closure is called (given that the function contains a function declaration inside it, and a reference to that inside function is either returned or an external reference is kept for it in some way). * Two functions might look like they have the same source text, but have completely different behaviour because of their 'hidden' closure. I don't think JavaScript code can actually find out if a function reference has a closure or not. * If you are trying to do any dynamic source code modifications (for example: myFunction = Function(myFunction.toString().replace(/Hello/,'Hola'));), it won't work if myFunction is a closure (of course, you would never even think of doing source code string substitution at runtime, but...). * It is possible to get function declarations within function declarations within functions — and you can get closures at more than one level. |