**Closures**

A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives you access to an outer function’s scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

[**Lexical scoping**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures#lexical_scoping)

Consider the following example code:

function init() {

var name = 'Mozilla'; // name is a local variable created by init

function displayName() { // displayName() is the inner function, a closure

alert(name); // use variable declared in the parent function

}

displayName();

}

init();

Copy to Clipboard

init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is available only within the body of the init() function. Note that the displayName() function has no local variables of its own. However, since inner functions have access to the variables of outer functions, displayName() can access the variable name declared in the parent function, init().

Run the code using [this JSFiddle link](https://jsfiddle.net/xAFs9/3/) and notice that the alert() statement within the displayName() function successfully displays the value of the name variable, which is declared in its parent function. This is an example of *lexical* *scoping*, which describes how a parser resolves variable names when functions are nested. The word *lexical* refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

[**Closure**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures#closure)

Consider the following code example:

function makeFunc() {

var name = 'Mozilla';

function displayName() {

alert(name);

}

return displayName;

}

var myFunc = makeFunc();

myFunc();

Copy to Clipboard

Running this code has exactly the same effect as the previous example of the init() function above. What's different (and interesting) is that the displayName() inner function is returned from the outer function *before being executed*.

At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once makeFunc() finishes executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A *closure* is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created. In this case, myFunc is a reference to the instance of the function displayName that is created when makeFunc is run. The instance of displayName maintains a reference to its lexical environment, within which the variable name exists. For this reason, when myFunc is invoked, the variable name remains available for use, and "Mozilla" is passed to alert.

Here's a slightly more interesting example—a makeAdder function:

function makeAdder(x) {

return function(y) {

return x + y;

};

}

var add5 = makeAdder(5);

var add10 = makeAdder(10);

console.log(add5(2)); // 7

console.log(add10(2)); // 12