Functions

Scope

Function Scopes

When you create a function, the variables and other things defined inside the function are **inside their own separate scope**, meaning that they are locked away in their own separate compartments, **unreachable from inside other functions or from code outside the functions**.

The top level outside all your functions is called the **global scope**. Values defined in the global scope are **accessible from everywhere** in the code.

Function Scopes - example

```
<script>

var x = 1;

function a() {
    var y = 2;
}

function b() {
    var z = 3;
}

</script>
```

Nested Scopes

When you declare a variable, it is available anywhere in that scope, as well as any lower/inner scopes.

Nested Scopes Example

(1) encompasses the global scope, and has just one identifier in it: `foo`.

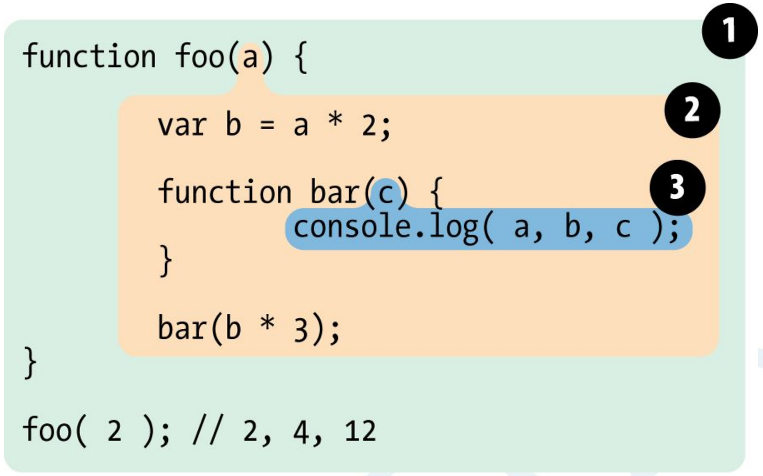
(2) encompasses the scope of `foo`,

which includes the three identifiers:

`a`, `bar` and `b`.

(3) encompasses the scope of `bar`,

and it includes just one identifier: `c`.



```
function foo(a) {
        var b = a * 2;
        function bar(c) {
                console.log( a, b, c );
        }
        bar(b * 3);
}
foo( 2 ); // 2, 4, 12
```

Global Scope

Global variables are also automatically properties of the global object (`window` in browsers, etc.), so it is possible to reference a global variable not directly by its lexical name, but instead indirectly as a property reference of the global object.

```
window.a
```

Look-ups

Scope look-up stops once it finds the first match. The same identifier name can be specified at multiple layers of nested scope, which is called "shadowing" (the inner identifier "shadows" the outer identifier). Regardless of shadowing, scope look-up always starts at the innermost scope being executed at the time, and works its way outward/upward until the first match, and stops.

Examples

```
// code here can not use carName

function myFunction() {
    var carName = "Volvo";

    // code here can use carName

}
```

Examples

```
var carName = " Volvo";

// code here can use carName

function myFunction() {

    // code here can use carName

}
```

Function Scopes - exercise

Visit this link:
https://github.com/mdn/learning-area/blob/master/javascript/building-blocks/functions/function-scope.html

Create a function-scope.html file on your computer and copy the content from the link into your file.

Open the file in a browser and in your text editor.

Function Scopes - exercise

Open the JavaScript console in your browser developer tools. In the JavaScript console, enter the following command:

```
output(x);
```

You should see the value of variable x output to the screen.

Function Scopes - exercise

Now try entering the following in your console:

```
output(y);
```

```
output(z);
```

Both of these should return an error. Why is that?

Function Scopes - exercise

Try editing a() and b() so they look like this:

```
function a() {
  var y = 2;
  output(y);
}

function b() {
  var z = 3;
  output(z);
}
```

Function Scopes - exercise

Save the code and reload it in your browser, then try calling the a() and b() functions from the JavaScript console:

```
a();
```

```
b();
```

Is it working? Why?

Function Scopes - exercise

Now try updating your code like this:

```
function a() {
  var y = 2;
  output(x);
}

function b() {
  var z = 3;
  output(x);
}
```

Function Scopes - exercise

Save the code and reload it in your browser, then try calling the a() and b() functions from the JavaScript console:

```
a();
```

```
b();
```

Is it working? Why?

JavaScript Functions

Functions As Values

You recall typical function declaration syntax as follows:

```
function foo() {
    // ..
}
```

Though it may not seem obvious from that syntax, `foo` is basically just a variable in the outer enclosing scope that's given a reference to the `function` being declared. That is, the `function` itself is a value, just like `42` or `[1,2,3]` would be.

Functions As Values

This may sound like a strange concept at first, so take a moment to ponder it. Not only can you pass a value (argument) to a function, but a function itself can be a value that's assigned to variables, or passed to or returned from other functions.

As such, a function value should be thought of as an expression, much like any other value or expression.

Functions As Values

```
var foo = function() {
    // ..
};

var x = function bar(){
    // ..
};
```

The first function expression assigned to the foo variable is called anonymous because it has no name.

The second function expression is named (bar), even as a reference to it is also assigned to the x variable. Named function expressions are generally more preferable, though anonymous function expressions are still extremely common.

Anonymous vs. Named

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

This is called an "anonymous function expression", because `function()...` has no name identifier on it. Function expressions can be anonymous, but function declarations cannot omit the name -- that would be illegal JS grammar.

Anonymous vs. Named

```
function myFunction() {
  alert('hello');
}
```

```
function() {
  alert('hello');
}
```

Anonymous functions usage

```
var arr = ['a', 'b', 'c'];

arr.forEach(function(element) {
    console.log(element);
});

// a
// b
// c
```

Exercise

1. Create a function `find(array, callback)` which as a first argument accepts array, and as a second one callback. Function returns first element from array for which callback returns true.