JavaScript

Functions

A function can be declared by using the function keyword and then providing a name (also known as an identifier), the optional list of parameters enclosed in parentheses, and a set of curly braces with the grouping of statements, as follows:

```
function Add(x, y) {
    return x + y;
}
```

This is an example of a function declaration, in which the function is called Add and has two parameters, x and y. The function has a grouping of statements, denoted by the curly braces (also known as a code block). This function has only one statement, but it could have many statements.

A function that does not return a value has no **return** statement.

Functions can be assigned to variables. For instance:

```
var myFunction = Add;
```

will assign to the variable myFunction the function Add. Note that to refer to the function (and not to the value of the function) the name of the function must not be followed by (). After the assignment above, `myFunction(3,5)` will return `8.`

It is also possible to define *anonymous* functions. for instance:

```
function(x, y) {
    return x + y;
}
```

To use an anonymous function, you must immediately call it or assign it to a variable. For instance:

```
(function(x, y) {
    return x + y;
})(3,5);
```

which will produce 8. Or,

```
var myFunction = function(x, y) {
                     return x + y;
                 }
```

Remark: So,

```
function Add(x, y) {
    return x + y;
}
```

is essentially equivalent to:

```
var Add = function(x, y) {
              return x + y;
          }
```

There's a third notation for functions, which looks very different from the others. Instead of the function keyword, it uses an arrow (=>) made up of an equal sign and a greater-than character (not to be confused with the greater than- or-equal operator, which is written >=).

```
(x, y) => {
    return x + y;
}
```

and it can be used, for instance, as:

```
((x, y) => {
    return x + y;
})(3,5);
```

to produce 8. Or

```
var Add = (x, y) => {
              return x + y;
          }
```

Remark: A function may return as its result a function. For instance:

```
function multiplier(factor) {
         return number => number * factor;
}

var twice = multiplier(2);
```

and then the result of `twice(5)` will be `10`.

Thinking about programs like this takes some practice. A good mental model is to think of function values as containing both the code in their body and the environment in which they are created. When called, the function body sees the environment in which it was created, not the environment in which it is called.

In the example, `multiplier` is called and creates an environment in which its factor parameter is bound to 2. The function value it returns, which is stored in `twice`, remembers this environment. So when that is called, it multiplies its argument by 2.

Remark: Variables declared without using `let`, `var` or `const`, are GLOBAL variables, even if they are declared inside a function.

JavaScript Function Parameters

A JavaScript function does not perform any checking on parameter values (arguments).

Parameter Rules:

JavaScript function definitions do not specify data types for parameters.

JavaScript functions do not perform type checking on the passed arguments.

JavaScript functions do not check the number of arguments received.

Parameter Defaults

If a function is called with missing arguments (less than declared), the missing values are set to: undefined

Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter:

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 0;
  }
}
```

ECMAScript 2015 allows default parameter values in the function declaration:

```
function (a=1, b=1) {
  // function code
}
```

The `arguments` Object

JavaScript functions have a built-in object called the `arguments` object.

The `arguments` object contains an array of the arguments used when the function was called (invoked).

This way you can simply use a function to find (for instance) the highest value in a list of numbers:

```
x = findMax(1, 123, 500, 115, 44, 88);

function findMax() {
  var i;
  var max = -Infinity;
  for (i = 0; i < arguments.length; i++) {
    if (arguments[i] > max) {
      max = arguments[i];
    }
  }
  return max;
}
```

Or create a function to sum all input values:

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
  var i;
  var sum = 0;
  for (i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
}
```

If a function is called with too many arguments (more than declared), these arguments can be reached using the `arguments` object.

Arguments are Passed by Value

JavaScript arguments are passed by value: The function only gets to know the values, not the argument's locations.

If a function changes an argument's value, it does not change the parameter's original value.

Changes to arguments are not visible (reflected) outside the function.

Objects are Passed by "Reference" (actually, passed by value of pointers)

In JavaScript, objects are passed by value too, but object references are values.

Because of this, objects will behave (in certain situations) **like** they are passed by reference:

If a function changes a property of the object received as argument, it changes the original value outside the function. Changes to object properties are visible (reflected) outside the function. So in this case, it behaves as "call by reference".

However, if a function assigns a new object to a parameter that has received an object as argument, it will not change the original value outside the function. So in this case, it does **not** behave as "call by reference".

Nested Functions

In JavaScript, you can nest function declarations inside function declarations. JavaScript allows multiple levels of function declaration nesting. Remember that a function produces a local scope, so you can get additional scopes by using this function-nesting trick.

Nested functions are private to the function in which they are defined. Nested functions also start their own local context, whereas defined variables are accessible within this function only. In addition, a nested function can access variables defined in the parent function's local context, the grandparent function's local context, and so on. Here is an example of using a nested function to get a nested local scope:

```
function areaOfPizzaSlice(diameter, slicesPerPizza) {
  return areaOfPizza(diameter) / slicesPerPizza;
  function areaOfPizza(diameter) {
       var radius = diameter / 2;
       return 3.141592 * radius * radius;
  }
}
```

In this example, the areaOfPizza function is nested in the areaOfPizzaSlice function, which means that the areaOfPizza function is declared inside the areaOfPizzaSlice function's local scope. The areaOfPizza function is not accessible outside the areaOfPizzaSlice function. The radius variable is declared inside the nested local scope and is accessible only from within the areaOfPizza function. There are two diameter variables, one in the local scope of areaOfPizzaSlice and one in the areaOfPizza. When in the areaOfPizza function, the diameter variable that is defined in that local context is accessible. When in the areaOfPizzaSlice function, the diameter variable that is defined in that local scope is accessible. The slicesPerPizza variable is accessible in both functions because parent variables are accessible to the children as long as they are not hidden by local variables with the same name.

References:

Johnson, Glenn - *Programmingin HTML5 with JavaScript and CSS3 – Training Guide*, Microsoft Press, 2013

Haverbeke, Marijn - *Eloquent JavaScript* – 3rd Edition, No Starch Press, 2018

Patel, Meher Krishna - *HTML, CSS, Bootstrap, Javascript and jQuery* – PythonDSP, October 2018

http://eloquentjavascript.net/

https://www.w3schools.com/js