COMP519 Web Programming

Lecture 13: JavaScript (Part 4)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Functions

Function definitions can take several different forms in JavaScript including:

```
function identifier(param1,param2, ...) {
  statements  }

identifier = function(param1,param2, ...) {
  statements  }
```

- Such function definitions are best placed in the head section of an HTML page or in a library that is then imported
- Function names are case-sensitive
- The function name must be followed by parentheses
- A function has zero, one, or more parameters that are variables
- Parameters are not typed
- `identifier.length` can be used inside the body of the function to determine the number of parameters

Functions

Function definitions can take several different forms in JavaScript including:

```
function identifier(param1,param2, ...) {
  statements  }

identifier = function(param1,param2, ...) {
  statements  }
```

- The return statement

  `return value`

  can be used to terminate the execution of a function and to make *value* the return value of the function
- The return value does not have to be of a primitive type
- A function can contain more than one return statement
- Different return statements can return values of different types
  $\leadsto$ there is no return type for a function

Calling a Function

A function is called by using the function name followed by a list of arguments in parentheses

```
function identifier(param1, param2, ...) {
  ...
}
... identifier(arg1, arg2,...) ... // Function call
```

- The list of arguments can be shorter as well as longer as the list of parameters
- If it is shorter, then any parameter without corresponding argument will have value `undefined`

```
function sum(num1,num2) { return num1 + num2 }

sum1 = sum(5,4)       // sum1 = 9
sum2 = sum(5,4,3)     // sum2 = 9
sum3 = sum(5)         // sum3 = NaN
```

‘Default Values’ for Parameters

- ECMAScript 2015 introduced default parameter values

```
function sum(num1 = 0,num2 = 0) { return num1 + num2 }

sum1 = sum(5,4)        // sum1 = 9
sum2 = sum(5,4,3)      // sum2 = 9
sum3 = sum(5)          // sum3 = 5
```

- In Internet Explorer or other older browsers, a function instead has to check whether an argument has the value `undefined` and take appropriate action

```
function sum(num1,num2) {
  if (num1 == undefined) num1 = 0
  if (num2 == undefined) num2 = 0
  return num1 + num2
}
```

Functions as Arguments

JavaScript functions are objects and can be passed as arguments to other functions

```
function apply(f,x,y) {
  return f(x,y)
}
function mult(x,y) {
  return x * y
}
console.log('2 * 3 =',apply(mult,2,3))
```

```
2 * 3 = 6
```

```
console.log('2 + 3 =',
            apply(function(a,b) { return a + b },
            2,3))
```

```
2 + 3 = 5
```

Variable-length Argument Lists

- Every JavaScript function has a property called `arguments`
- The `arguments` property consists of an array of all the arguments passed to a function
- As for any JavaScript array, `arguments.length` can be used to determine the number of arguments

```
// Function that returns the sum of all its arguments
function sumAll() {
  var sum = 0
  for (var i=0; i<arguments.length; i++)
    sum = sum + arguments[i]
  return sum
}

sum0 = sumAll()          // sum0 = 0
sum1 = sumAll(5)         // sum1 = 5
sum2 = sumAll(5,4)       // sum2 = 9
sum3 = sumAll(5,4,3)     // sum3 = 12
```

JavaScript Functions and Static Variables

- JavaScript does not have a `static` keyword to declare a variable to be static and preserve its value between different calls of a function
- A solution is to use a function property instead

```
function counter() {
  counter.count = counter.count || 0  // function property
  counter.count++
  return counter.count
}
document.writeln("1: static count = "+counter())
document.writeln("2: static count = "+counter())
document.writeln("3: global counter.count = "+counter.count)
```

```
1: static count = 1
2: static count = 2
3: global counter.count = 2
```

- As the example shows the function property is global/public
- Private static variables require more coding effort

Scope

Name Binding

An association of a name to an entity

Example: The association of a variable name to a 'container' for values

Scope of a Name Binding

The region of a program where the binding is valid, that is, where the name can be used to a refer to the entity

Typical regions are

- entire program (global)
- block
- function
- expression
- execution context

Name Resolution

Resolution of a name to the entity it is associated with

Scope

Static Scope/Scoping

Name resolution depends on the location in the source code and the lexical context, which is defined by where a named variable or function is defined/declared

Dynamic Scope/Scoping

Name resolution depends on the program state when a name is encountered which is determined by the execution context or calling context

Global / Local

- A name binding is global if its scope is the entire program and local otherwise
- A variable is global if the name binding of its name is global and local otherwise

Variable Declarations Revisited

- Variables can be declared (within a scope) using one of the following statements:

```
var variable1, variable2, ...
var variable1 = value1, variable2 = value2, ...
```

  - The second statement also initialises the variables
  - Used inside a function definition, creates a local variable, only accessible within the function
  - Used outside a function definition, creates a global variable

- A variable can be defined without an explicit declaration by assigning a value to it:

```
variable = value
```

  - Always creates a global variable

Variable Declarations Revisited

- Variables can be declared (within a block context) using one of the following statements:

```
let variable1, variable2, ...
let variable1 = value1, variable2 = value2, ...
```

  - The second statement also initialises the variables
  - Used inside a block, creates a local variable, only accessible within the block
  - Used outside any block, creates a global variable

```
for (var i=0; i<1; i++) {
  var j = i + 1
  console.log('I: i =',i,'j =',j)
}
console.log('O: i =',i,'j =',j)
```

```
I: i = 0 j = 1
O: i = 1 j = 1
```

```
for (let i=0; i<1; i++) {
  let j = i + 1
  console.log('I: i =',i,'j =',j)
}
console.log('O: i =',i,'j =',j)
```

```
I: i = 0 j = 1
ReferenceError: i is not defined
ReferenceError: j is not defined
```

Variable Declarations Revisited

- Variables can be declared (within a block context) using one of the following statements:

```
let variable1, variable2, ...
let variable1 = value1, variable2 = value2, ...
```

  - Variable declarations using `let` are not hoisted

```
var myVar1
console.log("myVar1 =",myVar1)
console.log("myVar2 =",myVar2)
var myVar2
```

```
myVar1 = undefined
myVar2 = undefined
```

```
let myVar1
console.log("myVar1 =",myVar1)
console.log("myVar2 =",myVar2)
let myVar2
```

```
myVar1 = undefined
ReferenceError: myVar2 is not defined
```

Functions and Scope (1)

```
x = "Hello"
function f1() {
  console.log("1: " + x)
}
function f2() {
  console.log("2: " + x)
  x = "Bye"
  console.log("3: " + x)
}
f1()
f2()
console.log("4: " + x)
```

```
1: Hello
2: Hello
3: Bye
4: Bye
```

- A variable defined or declared outside any function is global
- A global variable can be accessed from any part of the script, including inside a function

Functions and Scope (2)

```
x = "Hello"
function f1() {
  console.log("1: " + x)
}
function f2() {
  console.log("2: " + x)
  var x = "Bye"
  console.log("3: " + x)
}
f1()
f2()
console.log("4: " + x)
```

```
1: Hello
2: undefined
3: Bye
4: Hello
```

- A variable defined or declared outside any function is global
- A global variable can be accessed from any part of the script, including inside a function
- To create a local variable inside a function we need to declare it (and optionally initialise it)
- A global and a local variable can have the same name but are still different name bindings

Functions and Scope (3)

```
x = "Hello"
function f1() {
  console.log("1: " + x)
}
function f2() {
  // Name binding of x?
  console.log("2: " + x)
  let x = "Bye"
  console.log("3: " + x)
}
f1()
f2()
console.log("4: " + x)
```

```
1: Hello
console.log("2: " + x)
                    ^
ReferenceError: x is not
                defined
```

- A variable defined or declared outside any function is global
- A global variable can be accessed from any part of the script, including inside a function
- To create a local variable inside a function we need to declare it (and optionally initialise it)
- A global and a local variable can have the same name but are still different name bindings

Functions and Scope (4)

```
x = "Hello"


function f3(x) {
  x += '!'
  console.log("1: " + x)
}
f3('Bye')
console.log("2: " + x)
f3(x)
console.log("2: " + x)
```

```
1: Bye!
2: Hello
1: Hello!
2: Hello
```

- Parameters are local variables unrelated to any global variables of the same name

Static vs Dynamic Scoping

```
let s = 'static'

function f1() {
  console.log('scope =',s)
}

function f2() {
  let s = 'dynamic'
  f1()
}

f2()
// Trace:
// let s = 'static'
// f2()
//   let s = 'dynamic'
//   f1()
//     console.log('scope =',s)
```

```
scope = static
```

- JavaScript uses static scoping
- The example also works with `var` instead of `let`

Nested Function Definitions

```
function f1() {
  var x = 1
  f2()
  function f2() {
    var y = 2
    console.log('x =',x,'y =',y)
  }
}

f1()
f2()
```

```
x = 1 y = 2
scope2.js:11
f2()
^
ReferenceError: f2 is not defined
```

- Function definitions can be placed anywhere where a statement is allowed
  ⤳ but browser semantics may differ
- Function definitions can be nested
  ⤳ works across browsers
- Inner functions are local to outer functions
- Function definitions are hoisted: A function call can appear in the code before the function definition (but this is bad style)

Scope and Recursive Functions

```
function factorial(x) {
  console.log('1: y =',y)
  var y = x
  console.log('2: y =',y)
  if (y > 1) {
    y = x * factorial(x-1)
  }
  console.log('3: y =',y)
  return y
}
factorial(2)
```

```
// factorial(2)
1: y = undefined
2: y = 2
// factorial(1)
1: y = undefined
2: y = 1
3: y = 1
// y = 2 * 1
3: y = 2
```

- A function can call itself from within its code (recursion)
- Every function call creates a new scope

Revision and Further Reading

- Read
  - Chapter 20: The JavaScript Language: User-Defined Functions

  of S. Schafer: Web Standards Programmer's Reference.
  Wiley Publishing, 2005.
  Harold Cohen Library 518.532.S29 or
  E-book `http://library.liv.ac.uk/record=b2174141`
- Read
  - Chapter 5: Reference Types: The Function Type

  of N. C. Zakas: Professional JavaScript for Web developers.
  Wrox Press, 2009.
  Harold Cohen Library 518.59.Z21 or
  E-book `http://library.liv.ac.uk/record=b2238913`
- Data & Object Factory: JS Function Objects
  https://www.dofactory.com/tutorial/javascript-function-objects