COMP284 Scripting Languages

Lecture 16: JavaScript (Part 3)
Handouts (8 on 1)

Ullrich Hustadt

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

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
```


Contents

'Default values' for parameters

- JavaScript does not allow to specify default values for function parameters
- Instead a function has to check whether a parameter 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
}

sum3 = sum(5)         // sum3 = 5
sum4 = sum()          // sum4 = 0
```


Functions

Function definitions can take several different forms in JavaScript including:

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

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

- Such function definitions are best placed in the head section of a 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

Variable-length argument lists

- E ...ded `arguments`
  T ...rray of all the arguments p
- As for any JavaScript array, `arguments.length` can be used to determine the number of arguments

```
function sumAll() { // no minimum number of arguments
  if (arguments.length < 1) return null
  sum = 0
  for (var i=0; i<arguments.length; i++)
      sum = sum + arguments[i]
  return sum
}

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


Functions

Function definitions can take several different forms in JavaScript including:

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

var 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
  ⇝ there is no return type for a function

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
- The 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

JavaScript functions: Example

```
function bubble_sort(array) {
  if (!(array && array.constructor == Array))
     throw("Argument not an array")
  for (var i=0; i<array.length; i++) {
    for (var j=0; j<array.length-i; j++) {
      if (array[j+1] < array[j]) {
        // swap can change array because array is
        // passed by reference
        swap(array, j, j+1)
  }  }  }
  return array
}

function swap(array, i, j) {
  var tmp  = array[i]
  array[i] = array[j]
  array[j] = tmp
}
```


JavaScript functions: Example

```
function bubble_sort(array) { ... }
function swap(array, i, j)  { ... }

array = [2,4,3,9,6,8,5,1]
document.writeln("array  before sorting           "+
                 array.join(", ")+" <br>")
```

```
array  before sorting           2, 4, 3, 9, 6, 8, 5, 1 <br>
```

```
sorted = bubble_sort(array.slice(0)) // slice creates copy
document.writeln("array  after  sorting of copy   "+
                 array.join(", ")+"< br>")
```

```
array  after  sorting of copy   2, 4, 3, 9, 6, 8, 5, 1 <br>
```

```
sorted = bubble_sort(array)
document.writeln("array  after  sorting of itself "+
                 array.join(", ")+" <br>")
```

```
array  after  sorting of itself 1, 2, 3, 4, 5, 6, 8, 9 <br>
```

```
document.writeln("sorted array                    "+
                 sorted.join(", ")+" <br>"
```

```
sorted array                    1, 2, 3, 4, 5, 6, 8, 9 <br>
```


Nested function definitions

- Function definitions can be nested in JavaScript
- Inner functions have access to the variables of outer functions
- By default, inner functions can not be invoked from outside the function they are defined in

```
function bubble_sort(array) {
  function swap(i, j) {
    // swap can change array because array is
    // a local variable of the outer function bubble_sort
    var tmp = array[i]; array[i] = array[j]; array[j] = tmp;
  }
  if (!(array && array.constructor == Array))
     throw("Argument not an array")
  for (var i=0; i<array.length; i++) {
    for (var j=0; j<array.length-i; j++) {
      if (array[j+1] < array[j]) swap(j, j+1)
  } }
  return array }
```


JavaScript libraries

- Collections of JavaScript functions (and other code), libraries, can be stored in one or more files and then be reused
- By convention, files containing a JavaScript library are given the file name extension `.js`
- `<script>`-tags are not allowed to occur in the file
- A JavaScript library is imported using

```
<script type="text/javascript" src="url"></script>
```

where `url` is the (relative or absolute) URL for library

```
<script type="text/javascript"
 src="http://cgi.csc.liv.ac.uk/~ullrich/jsLib.js"></script>
```

- One such import statement is required for each library
- Import statements are typically placed in the head section of a page or at the end of the body section
- Web browers typically cache libraries

JavaScript libraries: Example

`~ullrich/public_html/sort.js`

```
function bubble_sort(array) {
  ... swap(array, j, j+1) ...
  return array
}

function swap(array, i, j) { ... }
```

`example.html`

```
<html><head><title>Sorting example</title>
<script type="text/javascript"
 src="http://cgi.csc.liv.ac.uk/~ullrich/sort.js">
</script></head>
<body>
<script type="text/javascript">
array  = [2,4,3,9,6,8,5,1];
sorted = bubble_sort(array.slice(0))
</script>
</body></html>
```


Object Literals

- JavaScript is an object-oriented language, but one without classes
- Instead of defining a class, we can simply state an object literal

  `{ property1: value1, property2: value2, ... }`

  where *property1*, *property2*, ... are variable names and *value1*, *value2*, ... are values (expressions)

```
var person1 = {
  age:       (30 + 2),
  gender:    'male',
  name:      { first : 'Bob', last : 'Smith' },
  interests: ['music', 'skiing'],
  hello: function() { return 'Hi! I\'m ' + this.name.first + '.' }
};
person1.age              --> 32        // dot notation
person1['gender']        --> 'male'    // bracket notation
```


Object Literals

```
var per
  n
  h
};
person1.hello()     --> "Hi!␣I'm␣Bob."
```

- Every part of a JavaScript program is executed in a particular execution context
- Every execution context offers a keyword `this` as a way of referring to itself
- In `person1.hello()` the execution context of `hello()` is `person1`
  ⇝ `this.name.first` is `person1.name.first`

Object Literals

```
var person1 = {
  name:  { first : 'Bob', last : 'Smith' },
  greet: function() { return 'Hi! I\'m ' + name.first + '.' },
  full1: this.name.first + " " +this.name.last,
  full2:      name.first + " " +     name.last
};
person1.greet()     --> "Hi!␣I'm␣undefined."
person1.full1       --> "undefined␣undefined"
person1.full2       --> "undefined␣undefined"
```

- In `person1.greet()` the execution context of `greet()` is `person1`
  ⇝ but `name.first` does not refer to `person1.name.first`
- In the (construction of the) object literal itself, `this` does not refer to `person1` but its execution context (the `window` object)
  ⇝ none of `name.first`, `name.last`, `this.name.first`, and `this.name.last` refers to properties of this object literal

Objects Constructors

- JavaScript is an object-oriented language, but one without classes
- Instead of defining a class,
  we can define a function that acts as object constructor
  - variables declared inside the function will be instance variables of the object
    ⇝ each object will have its own copy of these variables
  - it is possible to make such variables private or public
  - inner functions will be methods of the object
  - it is possible to make such functions/methods private or public
  - private variables/methods can only be accessed inside the function
  - public variables/methods can be accessed outside the function
- Whenever an object constructor is called,
  prefixed with the keyword `new`, then
  - a new object is created
  - the function is executed with the keyword `this` bound to that object

Objects: Definition and use

```
function SomeObj() {
  instVar2      = 'B'       // private variable
  var instVar3  = 'C'       // private variable

  this.instVar1 = 'A'       // public variable

  this.method1 = function() {  // public method
    // use of a public variable, e.g. 'instVar1', must be preceded by 'this'
    return 'm1[' + this.instVar1 + ']' + method3() }

  this.method2 = function() {   // public method
    // calls of a public method, e.g. 'method1', must be preceded by 'this'
    return ' m2[' + this.method1() + ']'          }

  method3 = function() {       // private method
    return ' m3[' + instVar2 + ']' + method4()     }

  var method4 = function() {   // private method
    return ' m4[' + instVar3 + ']'                 }
}
obj = new SomeObj()             // create a new object
obj.instVar1      --> "A"
obj.instVar2      --> undefined
obj.instVar3      --> undefined
obj.method1()     --> "m1[A] m3[B] m4[C]"
obj.method2()     --> "m2[m1[A] m3[B] m4[C]]"
obj.method3()     --> error
obj.method4()     --> error
```


Objects: Definition and use

```
function SomeObj() {
  this.instVar1 = 'A'       // public variable

  instVar2      = 'B'       // private variable
  var instVar3  = 'C'       // private variable

  this.method1 = function() { ... }  // public method
  this.method2 = function() { ... }  // public method

  method3 = function() { ... }       // private method
  var method4 = function() { ... }   // private method
}
```

- Note that all of `instVar1` to `instVar3`, `method1` to `method4` are instance variables (properties, members) of `someObj`
- The only difference is that `instVar1` to `instVar3` store strings while `method1` to `method4` store functions

⇝ every object stores its own copy of the methods

Objects: Prototype property

- All functions have a `prototype` property that can hold shared object properties and methods
  ⇝ objects do not store their own copies of these properties and methods but only store references to a single copy

```
function SomeObj() {
  this.instVar1 = 'A'       // public variable

  instVar2      = 'B'       // private variable
  var instVar3  = 'C'       // private variable

  SomeObj.prototype.method1 = function() { ... }  // public
  SomeObj.prototype.method2 = function() { ... }  // public

  method3 = function() { ... }       // private method
  var method4 = function() { ... }   // private method
}
```

Note: `prototype` properties and methods are always public!

Objects: Prototype property

- The `prototype` property can be modified 'on-the-fly'
  ⇝ all already existing objects gain new properties / methods
  ⇝ manipulation of properties / methods associated with the `prototype` property needs to be done with care

```
function SomeObj() { ... }
obj1 = new SomeObj()
obj2 = new SomeObj()
document.writeln(obj1.instVar4)   // undefined
document.writeln(obj2.instVar4)   // undefined

SomeObj.prototype.instVar4 = 'A'
document.writeln(obj1.instVar4)   // 'A'
document.writeln(obj2.instVar4)   // 'A'

SomeObj.prototype.instVar4 = 'B'
document.writeln(obj1.instVar4)   // 'B'
document.writeln(obj2.instVar4)   // 'B'

obj1.instVar4 = 'C' // creates a new instance variable for obj1
SomeObj.prototype.instVar4 = 'D'
document.writeln(obj1.instVar4)   // 'C' !!
document.writeln(obj2.instVar4)   // 'D' !!
```


Objects: Prototype property

- The `prototype` property can be modified 'on-the-fly'
  ⇝ all already existing objects gain new properties / methods
  ⇝ manipulation of properties / methods associated with the `prototype` property needs to be done with care

```
function SomeObj() { ... }
obj1 = new SomeObj()
obj2 = new SomeObj()

SomeObj.prototype.instVar5 = 'E'

SomeObj.prototype.setInstVar5 = function(arg) {
  this.instVar5 = arg
}
obj1.setInstVar5('E')
obj2.setInstVar5('F')

                                    // 'E' !!
                                    // 'F' !!
```


'Class' variables and 'Class' methods

Fun 's class variables
sta class methods

```
funct

// 'class variable' - property of the Circle constructor function
Circle.PI = 3.14159

// 'instance method'
Circle.prototype.area = function () {
   return Circle.PI * this.r * this.r; }

// 'class method'    - property of the Circle constructor function
Circle.max = function (cx,cy) {
   if (cx.r > cy.r) { return cx } else { return cy }
}

c1      = new Circle(1.0)      // create an instance of the Circle class
c1.r    = 2.2;                 // set the r instance variable
c1_area = c1.area();           // invoke the area() instance method
x       = Math.exp(Circle.PI) // use the PI class variable in a computation
c2      = new Circle(1.2)      // create another Circle instance
bigger  = Circle.max(c1,c2)    // use the max() class method
```


Private static variables

In order to create private static variables shared between objects we can use a self-executing anonymous function

```
var Person = (function () {
  var population = 0           // private static 'class' variable

  return function (value) {    // constructor
    population++
    var name     = value       // private instance variable
    this.setName = function (value) { name = value }
    this.getName = function () { return name }
    this.getPop  = function () { return population }
  }
}())

person1  = new Person('Peter')
person2  = new Person('James')
person1.getName()                          --> 'Peter'
person2.getName()                          --> 'James'
person1.name                               --> undefined
Person.population || person1.population    --> undefined
person1.getPop()                           --> 2
person1.setName('David')
person1.getName()                          --> 'David'
```

Pre-defined objects: String

- JavaScript has a collection of pre-defined objects, including Array, String, Date
- A String object encapsulates values of the primitive datatype `string`
- Properties of a String object include
  - `length` the number of characters in the string
- Methods of a String object include
  - `charAt(index)`
    the character at position *index* (counting from 0)
  - `substring(start,end)`
    returns the part of a string between positions *start* (inclusive) and *end* (exclusive)
  - `toUpperCase()`
    returns a copy of a string with all letters in uppercase
  - `toLowerCase()`
    returns a copy of a string with all letters in lowercase

Pre-defined objects: String and RegExp

- JavaScript supports (Perl-like) regular expressions and the String objects have methods that use regular expressions:
  - `search(regexp)`
    matches *regexp* with a string and returns the start position of the first match if found, -1 if not
  - `match(regexp)`
    – without *g* modifier returns the matching groups for the first match or if no match is found returns null
    – with *g* modifier returns an array containing all the matches for the whole expression
  - `replace(regexp,replacement)`
    replaces matches for *regexp* with *replacement*, and returns the resulting string

```
name1  = 'Dave Shield'.replace(/(\w+)\s(\w+)/, "$2, $1")
regexp =  new RegExp("(\\w+)\\s(\\w+)")
name2  = 'Ken Chan'.replace(regexp, "$2, $1")
```


Pre-defined objects: Date

- The Date object can be used to access the (local) date and time
- The Date object supports various constructors
  - `new Date()` current date and time
  - `new Date(milliseconds)` set date to milliseconds since 1 Januar 1970
  - `new Date(dateString)` set date according to *dateString*
  - `new Date(year, month, day, hours, min, sec, msec)`
- Methods provided by Date include
  - `toString()`
    returns a string representation of the Date object
  - `getFullYear()`
    returns a four digit string representation of the (current) year
  - `parse()`
    parses a date string and returns the number of milliseconds since midnight of 1 January 1970

Revision

Read

- Chapter 16: JavaScript Functions, Objects, and Arrays
- Chapter 17: JavaScript and PHP Validation and Error Handling (Regular Expressions)

of

R. Nixon:
Learning PHP, MySQL, and JavaScript.
O'Reilly, 2009.

- `http://coffeeonthekeyboard.com/private-variables-in-javascript-177/`
- `http://coffeeonthekeyboard.com/javascript-private-static-members-part-1-208/`
- `http://coffeeonthekeyboard.com/javascript-private-static-members-part-2-218/`