COMP519 Web Programming

Lecture 15: JavaScript (Part 6)

Handouts

Ullrich Hustadt

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

Contents

1. (User-defined) Objects
   - Object Literals
   - Object Constructors
   - Prototype Property
   - Public and Private Static Variables
   - for/in-loops
   - Inheritance
   - Classes as Syntactic Sugar
2. Pre-defined Objects
   - Regular Expressions
   - String
   - Date
3. Further Reading

Object Literals

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

```
{ memberName1: value1, memberName2: value2, ...}
```

memberName1, *memberName2*, . . . are member names
value1, *value2*, . . . are member values (expressions)

- Terminology:
  member value is function ⇝ method
  member value is some other value ⇝ property

```
{
  age:        (30 + 2),
  gender:     'male',
  nme:        { first : 'Ben', last : 'Budd' },
  interests: ['music', 'skiing'],
  hello: function() { return 'Hi! I\'m ' + this.nme.first }
};
```

Object Literals

- Member values can be accessed using dot notation or bracket notation

```
var person1 = {
  age:        (30 + 2),
  gender:     'male',
  nme:        { first : 'Ben', last : 'Budd' },
  interests: ['music', 'skiing'],
  hello: function() { return 'Hi! I\'m ' + this.nme.first }
};
```

```
person1.age              --> 32       // dot notation
person1['gender']        --> 'male'   // bracket notation
person1.nme.first        --> 'Ben'
person1['nme']['last'] --> 'Budd'
```

Object Literals

```
var person1 = {
  ...
  nme:   { first : 'Ben', last : 'Budd' },
  hello: function() { return 'Hi! I\'m ' + this.nme.first }
};
```

```
person1.hello()     --> "Hi! I'm Ben"
person1['hello']() --> "Hi! I'm Ben"
```

- 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`
  $\leadsto$ `this.nme.first` is `person1.nme.first`

Object Literals

```
var nme = { first: 'Adam', last: 'Alby' }

var person1 = {
  nme:   { first: 'Ben', last : 'Budd' },
  hello: function() { return 'I\'m ' + this.nme.first },

  full1: this.nme.first + " " + this.nme.last,

  full2:      nme.first + " " + nme.last,
  greet: function() { return 'I\'m ' + nme.first }
}
console.log('hello =',person1.hello())
```

```
hello = I'm Ben
```

```
console.log('greet =',person1.greet())
console.log('full1 =',person1.full1)
console.log('full1 =',person1.full2)
```

```
greet = I'm Adam
full1 = Adam Alby
full2 = Adam Alby
```

Object Literals

```
var nme = { first: 'Adam', last: 'Alby' }
var person1 = {
  nme:   { first: 'Ben', last : 'Budd' },
  hello: function() { return 'I\'m ' + this.nme.first },
  full1: this.nme.first + " " + this.nme.last,
  full2:      nme.first + " " + nme.last,
  greet: function() { return 'I\'m ' + nme.first }
}
```

- In the construction of the object literal itself, `this` does not refer to `person1` but its execution context (the `window` object)
  $\leadsto$ none of `nme.first`, `nme.last`, `this.nme.first`, and `this.nme.last` refers to properties of this object literal
- In `person1.greet()` the execution context of `greet()` is `person1`
  $\leadsto$ but `nme.first` does not refer to `person1.nme.first`
- Do not think of an object literal as a block of statements (and of property/value pairs as assignments within that block)

Object 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 properties of the object
    ⤳ each object will have its own copy of these variables
  - it is possible to make such properties private or public
  - inner functions will be methods of the object
  - it is possible to make such functions/methods private or public
  - private properties/methods can only be accessed by the object itself
  - public properties/methods can be accessed from outside the object
- 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() {
  this.prop1 = 'A'          // public property
  var  prop2 = 'B'          // private property

  // public method
  this.method1 = function() {
    // (use of a) public  variable must     be preceded by `this'
    // (use of a) private variable must NOT be preceded by `this'
    return 'm1[prop1=' + this.prop1 + ' prop2=' + prop2 + ']' }

  // private method
  var method2 = function() {
    return 'm2[]' }

  // public method
  this.method3 = function() {
    // (call of a) public  method must     be preceded by `this'
    // (call of a) private method must NOT be preceded by `this'
    return 'm3[' + this.method1() + ' ' + method2() + ']' }
}
obj1 = new SomeObj()                     // creates a new object
```

```
obj1.prop1     = 'A'
obj1.prop2     = undefined
obj1.method1() = 'm1[prop1=A prop2=B]'
obj1.method2()   --> TypeError: obj.method2 is not a function
obj1.method3() = 'm3[m1[prop1=A prop2=B] m2[]]'
```

Objects: Definition and Use

```
function SomeObj() {
  this.prop1 = 'A'          // public property
  var  prop2 = 'B'          // private property
  var  that  = this

  // private method
  var method4 = function() {
    // (use of a) public  variable must     be preceded by 'that'
    // (use of a) private variable must NOT be preceded by 'that'
    return 'm4[prop1=' + that.prop1 + ' prop2=' + prop2 + ']' }

  // private method
  var method6 = function() {
    // (call of a) public  method must     be preceded by 'that'
    // (call of a) private method must NOT be preceded by 'that'
    return 'm6[' + that.method1() + ' ' + method4() + ']' }

  this.method5 { return method4() }
  this.method7 { return method6() }
}
obj1 = new SomeObj()                     // creates a new object
```

```
obj1.method5() = m4[prop1=A prop2=B]
obj1.method7() = m6[m1[prop1=A prop2=B] m4[prop1=A prop2=B]]
```

Objects: Definition and Use

```
function SomeObj() {
  this.prop1 = 'A'                      // public property
  var  prop2 = 'B'                      // private property
  this.method1 = function() {           // public method
    return 'm1[prop1=' + this.prop1 + ' prop2=' + prop2 + ']' }
  var  method2 = function() { ... }     // private method
}
obj1 = new SomeObj()
obj2 = new SomeObj()
console.log('obj1.method3() =',obj1.method3())
```

```
obj1.method3() = m3[m1[prop1=A prop2=B] m2[]]
```

```
obj1.method1 = function() { return 'modified' }
obj2.prop1   = 'C'
console.log('obj1.method3() =',obj1.method3())
console.log('obj2.method3() =',obj2.method3())
```

```
obj1.method3() = m3[modified m2[]]
obj2.method3() = m3[m1[prop1=C prop2=B] m2[]]
```

- prop1, prop2, method1 to method2 are all members (instance variables) of SomeObj
- The only difference is that prop1 and prop2 store strings while method1 and method2 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.prop1 = 'A'        // public property
  var  prop2 = 'B'        // private property

  SomeObj.prototype.method1 = function() { ... }  // public
  SomeObj.prototype.method3 = function() { ... }  // public

  var method2 = 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()
console.log(obj1.prop3)                                undefined
console.log(obj2.prop3)                                undefined

SomeObj.prototype.prop3 = 'A'
console.log('obj1.prop3 =',obj1.prop3)                 'A'
console.log('obj2.prop3 =',obj2.prop3)                 'A'

SomeObj.prototype.prop3 = 'B'
console.log('obj1.prop3 =',obj1.prop3)                 'B'
console.log('obj2.prop3 =',obj2.prop3)                 'B'

obj1.prop3 = 'C' // creates a new property for obj1
SomeObj.prototype.prop3 = 'D'
console.log('obj1.prop3 = ',obj1.prop3)                'C'
console.log('obj2.prop3 = ',obj2.prop3)                '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.prop4 = 'E'

SomeObj.prototype.setProp4 = function(arg) {
  this.prop4 = arg
}

obj1.setProp4('E')
obj2.setProp4('F')
console.log('obj1.prop4 =',obj1.prop4)
console.log('obj2.prop4 =',obj2.prop4)
```

```
E
F
```

‘Class’ Variables and ‘Class’ Methods

Function properties can be used to emulate Java’s class variables (static variables shared among objects) and class methods

```
function Circle(radius) { this.r = radius }

// `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 object of the Circle class
c1.r    = 2.2;                // set the r property
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 object
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 property
    this.setName = function (value) { name = value }
    this.getName = function () { return name }
    this.getPop  = function () { return population }
  }
}())

person1  = new Person('Peter')
person2  = new Person('James')
```

```
console.log( person1.getName() )
console.log( person2.getName() )
console.log( person1.name )
console.log( Person.population || person1.population)
console.log( person1.getPop() )
person1.setName('David')
console.log( person1.getName() )
```

```
Peter
James
undefined
undefined
2

David
```

for/in-loop

- The for/in-loop allows us to go through the properties of an object

```
for (var in object) { statements }
```

- Within the loop we can use *object*[*var*] to access the value of the property *var*

```
var person1 = {  age:       32,
                 gender:    'male',
                 name:      'Bob Smith'
}
for (prop in person1) {
  console.log('person1[' + prop + '] has value '
              + person1[prop]);
}
```

```
person1[gender] has value male
person1[name] has value Bob Smith
person1[age] has value 32
```

Inheritance

- The prototype property can also be used to establish an inheritance relationship between objects

```
function Rectangle(width, height) {
  this.width  = width
  this.height = height
  this.type   = 'Rectangle'
  this.area   = function() { return this.width * this.height }
}

function Square(length) {
  this.width = this.height = length;
  this.type  = 'Square'
}

// Square inherits from Rectangle
Square.prototype = new Rectangle();

var sq1 = new Square(5);

console.log("The area of sq1 is " + sq1.area() );
```

```
The area of sq1 is 25
```

Classes as Syntactic Sugar

- ECMAScript 2015 introduced classes as syntactic sugar for prototype-based objects

```
class Rectangle {
  constructor(width, height) {
    this.width  = width
    this.height = height
    this.type   = 'Rectangle'
  }
  get area() { return this.width * this.height }
}

class Square extends Rectangle {
  constructor(length) {
    super(length,length)
    this.type  = 'Square'
  }
}

var sq1 = new Square(5)
console.log("The area of sq1 is " + sq1.area ) // not sq1.area()!
```

Pre-defined Objects: RegExp

- JavaScript has a collection of pre-defined objects, including Array, Date, RegExp, String
- RegExp objects are called regular expressions
- Regular expressions are patterns that are matched against strings
- Regular expressions are created via

```
/regexp/                // regular expression literal
new RegExp('regexp')  // converting a string into a reg exp
```

- There are two methods provided by RegExp:

test(*str*)	Tests for a match in a string, returns true or false
exec(*str*)	Executes a search for a match in the string *str*, returns an array with a match or null otherwise

```
/^\w+$/.test('abc_d0')        true
/^\w+$/.test('ab-def')        false
/\w+/.exec('ac0$adef')        ['ac0']
```

Pre-defined Objects: RegExp

- The simplest regular expressions consist of a sequence of
  - alphanumeric characters and
  - non-alphanumeric characters escaped by a backslash:

  that matches exactly this sequence of characters

```
/100\$/ matches "100$" in "This 100$ bill"
```

- There is a range of special characters that match characters other than themselves or have special meaning

.	Matches any character except the newline character \n
\n	Matches the newline character \n
\w	Matches a 'word' character (alphanumeric plus '_')
\s	Matches a whitespace character
\d	Matches a decimal digit character

```
/\w\d/ matches "P5", "51", and "19" in "COMP519"
```

Pre-defined Objects: RegExp

- There is a range of special characters that match characters other than themselves or have special meaning

^	Matches beginning of input/line
$	Matches end of input/line
+	Matches the preceding expression 1 or more times
*	Matches the preceding expression 0 or more times
[*set*]	Matches any character in *set* which consists of characters, special characters and ranges of characters
[^*set*]	Matches any character not in *set*

```
/^[a-z]+$/       matches "abc",  "x"
                 but not "0abc", "abc1", " ab", "AB", ""

/^\s*[a-z]+\s*$/ matches "abc",  "x",    " ab"
                 but not "0abc", "abc1", "AB",  ""

/^[^a-z]+$/      matches "AB",   "0123"
                 but not "abc",  "x",    "0abc"
```

Pre-defined Objects: RegExp

- It is possible to remember and reuse parts of a match via capture groups

(*rex*)	Matches regular expression *rex* and remembers the match; this construct is called a capture group or capturing parentheses
\N	Matches the same substring as the *N*th capture group in the regular expression (couting left parentheses)
$N	In a replacement operation, the substring matched by the *N*th capture group

```
/^(\w)(\w)\2\1$/  matches "abba",  "1991"
                  but not "abbc", "abca", "19912", "19 91"

/(\w+)(\d).\2\1/  matches "ab1Z1ab", "abc0 0c", "_1|1_9"
                  but not "ab1Z1ca", "1Z1"

/((\d+)\$)\1\2$/  matches "10$10$10", "A9$9$9"
                  but not "9$8$9", "9$9$9A"
```

Pre-defined Objects: RegExp

- Regular expressions are created via

```
/regexp/                 // regular expression literal
new RegExp('regexp')     // converting a string into a reg exp
```

- Remember that in a string, the escape character `\`
  has a special meaning, e.g.,

  `\n` stands for a newline character   `\'` stands for an apostrophe

  `\w` stands for `w`   `\s` stands for `s`

  `\\` stands for `\`

  ⇝ additional escape characters are required
  in `RegExp('regexp')` versus `/regexp/`

```
/\w\d/            becomes   new RegExp('\\w\\d')
/(\w+)\s(\w+)/    becomes   new RegExp('(\\w+)\\s(\\w+)')
/\((\d+)\)/       becomes   new RegExp('\\((\\d+)\\)')
```

Pre-defined Objects: String

- JavaScript has a collection of pre-defined objects, including Array, Date, Regular Expression, String
- 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

```
'abcd'.search(/^\w+$/)                        0
'ab$d'.search(/^\w+$/)                        -1
'ac0$adef'.match(/\w+/)                       ['ac0']
'Ada Duff'.replace(/(\w+)\s(\w+)/, "$2, $1")  'Duff, Ada'
```

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 January 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 and Further Reading

- Read
  - Chapter 15: Objects

  of R. Nixon: Learning PHP, MySQL & JavaScript:
  with jQuery, CSS & HTML5. O'Reilly, 2018.
- Read
  - Chapter 5: Reference Types: The Object Type
  - Chapter 6: Object-Oriented Programming
  - Chapter 7: Anonymous Functions

  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`

Revision and Further Reading

- Read
  - Mozilla and individual contributors: JavaScript Reference: Classes. *MDN Web Docs*, 5 October 2019. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes` (accessed 24 October 2019)
  - Mozilla and individual contributors: JavaScript Guide: Regular Expressions. *MDN Web Docs*, 19 September 2019. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions` (accessed 24 October 2019)