COMP284 Scripting Languages
Lecture 15: JavaScript (Part 2)
Handouts (8 on 1)

Ullrich Hustadt

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

Contents

Integers and Floating-point numbers

- The JavaScript datatype number covers both
  - integer numbers 0 2012 -40 1263978
  - floating-point numbers 1.25 256.0 -12e19 2.4e-10
- The Math object provides a wide range of mathematical functions

`Math.abs(number)`	absolute value
`Math.ceil(number)`	round fractions up
`Math.floor(number)`	round fractions down
`Math.round(number)`	round fractions
`Math.log(number)`	natural logarithm
`Math.random()`	random number between 0 and 1
`Math.sqrt(number)`	square root

- There are also some pre-defined number constants including

`Math.PI`	(case sensitive)	3.14159265358979323846
`NaN`	(case sensitive)	'not a number'
`Infinity`	(case sensitive)	'infinity'

Numbers: NaN and Infinity

- The constants `NaN` and `Infinity` are used as return values for applications of mathematical functions that do not return a number
  - `Math.log(0)` returns `-Infinity` (negative 'infinity')
  - `Math.sqrt(-1)` returns `NaN` ('not a number')
  - `1/0` returns `Infinity` (positive 'infinity')
  - `0/0` returns `NaN` ('not a number')
- Equality and comparison operators produce the following results for `NaN` and `Infinity`:

```
NaN == NaN            ⇝ false    NaN === NaN           ⇝ false
Infinity == Infinity  ⇝ true     Infinity === Infinity ⇝ true
NaN == 1              ⇝ false    Infinity == 1         ⇝ false
NaN < NaN             ⇝ false    Infinity < Infinity   ⇝ false
1 < Infinity          ⇝ true     1 < NaN               ⇝ false
Infinity < 1          ⇝ false    NaN < 1               ⇝ false
NaN < Infinity        ⇝ false    Infinity < NaN        ⇝ false
```

Integers and Floating-point numbers: NaN and Infinity

- JavaScript provides two functions to test whether a value is or is not `NaN`, `Infinity` or `-Infinity`:
  - `bool isNaN(value)`
    returns TRUE iff *value* is `NaN`
  - `bool isFinite(value)`
    returns TRUE iff *value* is neither `NaN` nor `Infinity`/`-Infinity`

  There is no `isInfinite` function
- In conversion to a boolean value,
  - `NaN` converts to `false`
  - `Infinity` converts to `true`
- In conversion to a string,
  - `NaN` converts to `'NaN'`
  - `Infinity` converts to `'Infinity'`

Booleans

- JavaScript has a boolean datatype
  with constants `true` and `false` (case sensitive)
- JavaScript offers the same short-circuit boolean operators as Java, Perl and PHP:

  `&&` (conjunction) `||` (disjunction) `!` (negation)

  But `and` and `or` cannot be used instead of `&&` and `||`, respectively
- The truth tables for these operators are the same as for Perl and PHP, taking into account that the conversion of non-boolean values to boolean values differs
- Remember that `&&` and `||` are not commutative, that is,
  `(A && B)` is not the same as `(B && A)`
  `(A || B)` is not the same as `(B || A)`

Type conversion to boolean

When converting to boolean, the following values are considered `false`:
- the boolean `false` itself
- the number 0 (zero)
- the empty string, but not the string '0'
- `undefined`
- `null`
- `NaN`

Every other value is converted to `true` including
- `Infinity`
- `'0'`
- functions
- objects, in particular, arrays with zero elements

Strings

- JavaScript supports both single-quoted and double-quoted strings
- JavaScript uses + for string concatenation
- Within double-quoted strings JavaScript supports the following escape characters

`\b` (backspace)	`\f` (form feed)	`\n` (newline)
`\r` (carriage return)	`\t` (tab)	`\` (backslash)
`\'` (single quote)	`\"` (double quote)	

- JavaScript does not support variable interpolation
- JavaScript also does not support heredocs,
  but multi-line strings are possible

```
document.writeln("Your\
          name is " + name + "and\
          you are studying " + degree + "\
          at " + university);
```

Arrays

- An array is created by assigning an array value to a variable

```
var arrayVar = []
var arrayVar = [elem0, elem1, ... ]
```

- JavaScript uses

```
arrayVar[index]
```

  to denote the element stored at position *index* in *arrayVar*
  The first array element has index 0
- Arrays have no fixed length and it is always possible to add more elements to an array
- Accessing an element of an array that has not been assigned a value yet returns `undefined`
- For an array *arrayVar*, *arrayVar*`.length` returns the maximal index *index* such that *arrayVar*[*index*] has been assigned a value (including the value `undefined`) plus one

Arrays

- It is possible to assign a value to *arrayVar*`.length`
  - if the assigned value is greater than the previous value of *arrayVar*`.length`, then the array is 'extended' by additional `undefined` elements
  - if the assigned value is smaller than the previous value of *arrayVar*`.length`, then array elements with greater or equal index will be deleted
- Assigning an array to a new variable creates a reference to the original array
  ⇝ changes to the new variable affect the original array
- Arrays are also passed to functions by reference
- The `slice` function can be used to create a proper copy of an array:
  `object` *arrayVar*`.slice(`*start*`,`*end*`)`
  returns a copy of those elements of array *variable* that have indices between *start* and *end*

Arrays: Example

```
var array1 = ['hello', [1, 2], function() {return 5}, 43]
document.writeln("1:␣array1.length␣=␣"+array1.length+"<br>")
1: array1.length = 4<br>
document.writeln("2:␣array1[3]␣="+array1[3]+"<br>")
2: array1[3] = 43<br>
array1[5] = 'world'
document.writeln("3:␣array1.length␣=␣"+array1.length+"<br>")
3: array1.length = 6<br>
document.writeln("4:␣array1[4]␣="+array1[4]+"<br>")
4: array1[4] = undefined<br>
document.writeln("5:␣array1[5]␣="+array1[5]+"<br>")
5: array1[5] = world<br>
array1.length = 4
document.writeln("6:␣array1[5]␣="+array1[5]+"<br>")
6: array1[5] = undefined<br>
var array2 = array1
array2[3] = 7
document.writeln("7:␣array1[3]␣="+array1[3]+"<br>")
7: array1[3] = 7<br>
```

forEach-method

- The recommended way to iterate over all elements of an array is a for-loop

```
for (index = 0; index < arrayVar.length; index++) {
    ... arrayVar[index] ...
}
```

- An alternative is the use of the forEach method:

```
var callback = function (elem, index, arrayArg) {
    statements
}
array.forEach(callback);
```

  - The forEach method takes a function as an argument
  - It iterates over all indices/elements of an array
  - It passes the current array element (*elem*), the current index (*index*) and a pointer to the array (*arrayArg*) to the function
  - Return values of that function are ignored,
    but the function may have side effecs

forEach-method: Example

```
var myArray = ['Michele␣Zito','Ullrich␣Hustadt'];

var rewriteNames = function (elem, index, arr) {
  arr[index] = elem.replace(/(\w+)\s(\w+)/, "$2,␣$1");
}

myArray.forEach(rewriteNames);

for (i=0; i<myArray.length; i++) {
  document.write('['+i+']␣=␣'+myArray[i]+"␣");
}
document.writeln("<br>");
```

```
[0] = Zito, Michele [1] = Hustadt, Ullrich <br>
```

Array operators

JavaScript has no stack or queue data structures,
but has stack and queue functions for arrays:

- `number` *array*`.push(`*value1*`,` *value2*`,...)`
  appends one or more elements at the end of an array;
  returns the number of elements in the resulting array
- `mixed` *array*`.pop()`
  extracts the last element from an array and returns it
- `mixed` *array*`.shift()`
  shift extracts the first element of an array and returns it
- `number` *array*`.unshift(`*value1*`,` *value2*`,...)`
  inserts one or more elements at the start of an array variable;
  returns the number of elements in the resulting array

Note: In contrast to PHP and Perl, *array* does not need to be a variable

Array operators: push, pop, shift, unshift

```
planets = ["earth"]
planets.unshift("mercury","venus")
planets.push("mars","jupiter","saturn");
document.writeln("planets\@1:␣"+planets.join("␣")+"␣<br>")
planets@1: mercury venus earth mars jupiter saturn <br>
last = planets.pop()
document.writeln("planets\@2:␣"+planets.join("␣")+"␣<br>")
planets@2: mercury venus earth mars jupiter <br>
first = planets.shift()
document.writeln("planets\@3:␣"+planets.join("␣")+"␣<br>")
planets@3: venus earth mars jupiter <br>
document.writeln("␣␣␣␣␣␣␣\@4:␣"+first+"␣"+last+"␣<br>")
       @4: mercury saturn <br>
home = ["mercury","venus","earth"].pop()
document.writeln("␣␣␣␣␣␣␣\@5:␣"+ home + "␣<br>")
       @5: earth <br>
number = ["earth"].push("mars");
document.writeln("␣␣␣␣␣␣␣\@6:␣"+ number + "␣<br>")
       @6: 2 <br>
```

Control structures

JavaScript control structures

- conditional statements
- switch statements
- while- and do while-loops
- for-loops
- break and continue

are identical to those of PHP except for conditional statements

Control structures: conditional statements

JavaScript conditional statements do not allow for `elsif`- or `elseif`-clauses, but conditional statements can be nested:

```
if (condition) {
    statements
} else if (condition) {
    statements
} else {
    statements
}
```

- The else-clause is optional but there can be at most one
- Curly brackets can be omitted if there is only a single statement in a clause

JavaScript also supports conditional expressions

```
condition ? if_true_expr : if_false_expr
```

Control structures: switch statement

Switch statements in JavaScript take the same form as in PHP:

```
switch (expr) {
  case expr1:
       statements
       break;
  case expr2:
       statements
       break;
  default:
       statements
       break;
}
```

- there can be arbitrarily many case-clauses
- the default-clause is optional but there can be at most one
- *expr* is evaluated only once and then compared to *expr1*, *expr2*, etc using (loose) equality ==
- once two expressions are found to be equal the corresponing clause is executed
- if none of *expr1*, *expr2*, etc are equal to *expr*, then the default-clause will be executed
- `break` 'breaks out' of the switch statement
- if a clause does not contain a `break` command, then execution moves to the next clause

Control structures: switch statement

Not every case-clause needs to have associated statements

Example:

```
switch (month) {
  case 1:    case 3:    case 5:    case 7:
  case 8:    case 10:   case 12:
     days = 31;
     break;
  case 4:    case 6:    case 9:    case 11:
     days = 30;
     break;
  case 2:
     days = 28;
     break;
  default:
     days = 0;
     break;
}
```

Control structures: while- and do while-loops

- JavaScript offers while-loops and do while-loops

```
while (condition) {
   statements
}

do {
   statements
} while (condition);
```

- As usual, curly brackets can be omitted if the loop onsists of only one statement

Example:

```
// Compute the factorial of a given number
factorial = 1;
do {
    factorial *= number--;
} while (number > 0);
```

Control structures: for-loops

- for-loops in JavaScript take the form

```
for (initialisation; test; increment) {
    statements
}
```

  Again, the curly brackets are not required if the body of the loop only consists of a single statement
- In JavaScript, as in PHP, *initialisation* and *increment* can consist of more than one statement, separated by commas instead of semicolons
  Example:

```
for (i = 3, j = 3; j >= 0; i++, j--)
   document.writeln(i + "␣-␣" + j + "␣-␣" + i*j)
   // Indentation has no 'meaning' in JavaScript,
   // the next line is not part of the loop
   document.writeln("After␣loop:␣" + i + "␣-␣" + j)
```

- Note: Variables introduced in a for-loop are still global even if declared using `var`

Control structures: break and continue

- The break command can also be used in while-, do while-, and for-loops and discontinues the execution of the loop

```
while (value < 100) {
  if (value == 0) break;
  value++
}
```

- The continue command stops the execution of the current iteration of a loop and moves the execution to the next iteration

```
for (x = -2; x <= 2; x++) {
   if (x == 0) continue;
   document.writeln("10␣/␣" + x + "␣=␣" + (10/x));
}
```

```
10 / -2 = -5
10 / -1 = -10
10 / 1 = 10
10 / 2 = 5
```

Revision

Read

- Chapter 15: Expressions and Control Flow in JavaScript
- Chapter 16: JavaScript Functions, Objects, and Arrays

of

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