# COMP519 Web Programming

## Lecture 14: JavaScript (Part 5)

## Handouts

Ullrich Hustadt

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

# Array Literals

- An array literal is a comma-separated list of values enclosed in square brackets

```
[2,3,5,7,11]
```

Each element of that list has an index position
given by the number of other elements preceding it
⇝ the first index position is 0

```
[2,3,5,7,11]
 |  |  |  |  |
 0  1  2  3  4
```

- The values in an array literal do not need to be of the same type

```
[2,'two',3,"three",1.1e1,true]
```

- The values in an array literal can include other array literals (nested array literal)

```
[2,[3,5],[7,[11]]]
```

# Operators on Array Literals

- It is possible to access an element of an array literal via its index position: *arrayLiteral*[*index*]

```
[2,3,5,7,11][1]                    // returns 3
```

- For a nested array literal, it is possible to iterate this access operation

```
[2,[3,5],[7,[11]]][1]              // returns [3,5]
 | |      |
 0 1      2

[2,[3,5],[7,[11]]][1][0]           // returns 3
```

- The `length` operation returns the number of elements in an array literal: *arrayLiteral*`.length`

```
[2,3,5,7,11].length                // returns 5
```

- We will discuss more operators in the context of array variables

# Arrays

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

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

- All operations that can be performed on array literals can be performed on arrays:

| | |
|---|---|
| `arrayVar[index]` | returns the element at position `index` |
| `arrayVar.length` | returns the length of the array (highest index position plus one) |

- 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`

# 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
  $\rightsquigarrow$ 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]
console.log("1: array1.length = "+array1.length)
console.log("2: array1[3] ="+array1[3])
```

```
1: array1.length = 4
2: array1[3] = 43
```

```
array1[5] = 'world'
console.log("3: array1.length = "+array1.length)
console.log("4: array1[4] ="+array1[4])
console.log("5: array1[5] ="+array1[5])
```

```
3: array1.length = 6
4: array1[4] = undefined
5: array1[5] = world
```

```
array1.length = 4
console.log("6: array1[5] ="+array1[5])
```

```
6: array1[5] = undefined
```

```
var array2 = array1
array2[3] = 7
console.log("7: array1[3] ="+array1[3])
```

```
7: array1[3] = 7
```

# 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

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

var myArray = ['Dave Jackson','Ullrich Hustadt'];

myArray.forEach(rewriteNames);

for (i=0; i<myArray.length; i++) {
  console.log('myArray['+i+'] = '+myArray[i]);
}
```

```
myArray[0] = Jackson, Dave
myArray[1] = Hustadt, Ullrich
```

# Arrays and Functions: Example

```
function bubble_sort(array) {
  if (!(array && array.constructor == Array))
     throw("Argument not an array")
  for (let i=0; i<array.length; i++) {
    for (let 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) {
  let tmp   = array[i]
  array[i] = array[j]
  array[j] = tmp
}
```

# Arrays and Functions: Example

- Inner functions have access to the variables of outer functions

```
function bubble_sort(array) {
  function swap(i, j) {
    // swap can change array because array is
    // a local variable of the outer function bubble_sort
    let 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 }
```

# Arrays and Functions: Example

```
function bubble_sort(array) { ... }

array = [20,4,3,9,6,8,5,10]
console.log("array  before sorting            " +
            array.join(", "))
sorted = bubble_sort(array.slice(0)) // slice creates copy
console.log("array  after  sorting of copy    " +
            array.join(", "))
sorted = bubble_sort(array)
console.log("array  after  sorting of itself " +
             array.join(", "))
console.log("sorted array                     " +
            sorted.join(", "))
```

```
array  before sorting            20, 4, 3, 9, 6, 8, 5, 10
array  after  sorting of copy    20, 4, 3, 9, 6, 8, 5, 10
array  after  sorting of itself  3, 4, 5, 6, 8, 9, 10, 20
sorted array                     3, 4, 5, 6, 8, 9, 10, 20
```

# Stacks and Queues

## Stack

A collection of items that are inserted and removed according to the last-in first-out (LIFO) principle

- push adds an item to the top of the stack
- pop removes the top item from the stack

## Queue

A collection of items that are inserted and removed according to the first-in first-out (FIFO) principle

- enqueue adds an item to the back of the queue
- dequeue removes the item at the front of the queue

# Array functions

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 (enqueue);
  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 (dequeue) 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

# Array operators: push, pop, shift, unshift

```
planets = ["earth"]
planets.unshift("mercury","venus")
planets.push("mars","jupiter","saturn");
console.log("planets[]: " + planets.join(" "))
```

```
planets[]: mercury venus earth mars jupiter saturn
```

```
last = planets.pop()
console.log("last     : " + planets.join(" "))
console.log("planets[]: " + planets.join(" "))
```

```
last     : saturn
planets[]: mercury venus earth mars jupiter
```

```
first = planets.shift()
console.log("first    : " + first)
console.log("planets[]: " + planets.join(" "))
```

```
first    : mercury
planets[]: venus earth mars jupiter
```

```
number = ["earth"].push("mars");
console.log("number   : " + number)
```

```
number   : 2
```

# Equality and Program Behaviour

Why do we care whether `5 == true` is true or false?

⇝ it influences how our programs behave

⇝ it influences whether more complex objects are equal or not

JavaScript:

```
if (5) document.writeln("5 is true");
  else document.writeln("5 is not true")
document.writeln(" and ")
if (5 == true) document.writeln("5 is equal to true")
          else document.writeln("5 is not equal to true")
```

Output: `5 is true and 5 is not equal to true`

PHP:

```
if (5) print("5 is true");
else   print("5 is not true");
print(" and ");
if (5 == true) print("5 is equal to true");
          else print("5 is not equal to true");
```

Output: `5 is true and 5 is equal to true`

# Equality and Program Behaviour

Why do we care whether `5 == true` is true or false?

⇝ it influences how our scripts behave

⇝ it influences whether more complex objects are equal or not

JavaScript:

```
$array3 = ["1.23e2",5]
$array4 = ["12.3e1",true]
if (($array3[1] == $array4[1]) && ($array3[2] == $array4[2]))
     console.log("The two arrays are equal")
else console.log("The two arrays are not equal")
```

Output: `The two arrays are not equal`

PHP:

```
$array3 = array("1.23e2",5);
$array4 = array("12.3e1",true);
if (($array3[1] == $array4[1]) && ($array3[2] == $array4[2]))
     print("The two arrays are equal");
else print("The two arrays are not equal");
```

Output: `The two arrays are equal`

# Equality and Program Behaviour

Note: The way in which more complex data structures are compared also differs between PHP and JavaScript

JavaScript:

```
$array3 = ["1.23e2",5]
$array4 = ["12.3e1",true]
if ($array3 == $array4)
     console.log("The two arrays are equal")
else console.log("The two arrays are not equal")
```

Output: `The two arrays are not equal`

PHP:

```
$array3 = array("1.23e2",5);
$array4 = array("12.3e1",true);
if ($array3 == $array4)
     print("The two arrays are equal");
else print("The two arrays are not equal");
```

Output: `The two arrays are equal`

# Equality and Program Behaviour

Note: The way in which more complex data structures are compared also differs between PHP and JavaScript

JavaScript:

```
$array3 = ["1.23e2",5]
$array4 = ["1.23e2",5]
if ($array3 == $array4)
     console.log("The two arrays are equal")
else console.log("The two arrays are not equal")
```

Output: `The two arrays are not equal`

PHP:

```
$array3 = array("1.23e2",5);
$array4 = array("12.3e1",5);
if ($array3 == $array4)
     print("The two arrays are equal");
else print("The two arrays are not equal");
```

Output: `The two arrays are equal`

# 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 src="url"></script>
```

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

```
<script 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 (see next slide)
- Web browers typically cache libraries

# JavaScript Libraries: Import Statements

- JavaScript code may change the HTML markup of an HTML document
- Consequently, whenever a browers encounters a `script` element, by default, it stops parsing the remaining HTML markup of the page until that `script` element has been processed

  $\rightsquigarrow$ poor user experience
- 'Safe solution': Put `script` elements at the end of the `body` element
- 'Better solution': Use the `async` or `defer` attribute of the `script` element to change the default behaviour

```
<script src="jsLib1.js" async></script>
<script src="jsLib2.js" async></script>
```

Do not wait for the processing of the `script` elements, fetch and execute `jsLib1.js` and `jsLib2.js` (in parallel) in any order

```
<script src="jsLib1.js" defer></script>
<script src="jsLib2.js" defer></script>
```

Do not wait, fetch `jsLib1.js` and `jsLib2.js` (in parallel), execute in order after parsing is finished

# JavaScript Libraries: Example

~ullrich/public_html/sort.js

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

example.html

```
<html lang="en-GB">
  <head><title>Sorting example</title>
    <script src="http://cgi.csc.liv.ac.uk/~ullrich/sort.js">
    </script>
  </head>
  <body>
    <script>
    array  = [20,4,3,9,6,8,5,10];
    sorted = bubble_sort(array.slice(0))
    </script>
  </body>
</html>
```

# Revision and Further Reading

- Read
  - Chapter 15: Arrays

  of R. Nixon: Learning PHP, MySQL & JavaScript: with jQuery, CSS & HTML5. O'Reilly, 2018.
- Read
  - Chapter 5: Reference Types: The Array 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`

# Revision and Further Reading

- Read
  - Flavio Copes: Efficiently load JavaScript with defer and async. 24 March 2018. `https://flaviocopes.com/javascript-async-defer/` (accessed 16 October 2019)
  - Ravi Roshan: Async Defer: JavaScript Loading Strategies. *Medium*, 3 January 2019. `https://medium.com/@raviroshan.talk/async-defer-javascript-loading-strategies-da489a0ba47e`. (accessed 16 October 2019)
  - Mozilla and individual contributors: Window: load event. *MDN Web DOcs*, 4 October 2019. `https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event` (accessed 16 October 2019)