Introduction to Web programming (1DV525)

Introduction to JavaScript, pt 1

Licence for this work

This work is produced by Johan Leitet for the course Introduction to web programming (1DV525) at Linnaeus University.

You are free to

- ✓ copy and redistribute the material in any medium or format
- ✓ spread the whole or parts of the content
- ✓ show the whole or parts of the content publicly and digital
- ✓ convert the content to another format
- ✓ change the content

If you change the content do not use the photographs, icons, picture of the course literature or Linnaeus University logotype and symbol in your new work!

At all times you must give credit to: ”Linnaeus university – Introduction to Web Programming (1DV525)” with the link https://coursepress.lnu.se/kurs/introduction-to-web-programming/ and to the Creative Common-

JavaScript

History

- ✓ Timeline
  - ☐ 1995 - JavaScript 1.0 med Netscape Navigator 2.
  - ☐ 1997 - JavaScript 1.1 standardiseras till ECMAScript 1.
  - ☐ 1998 - ECMAScript 2.
  - ☐ 1999 - ECMAScript 3.
  - ☐ 2009 - ECMAScript 5.
  - ☐ 2011 - ECMAScript 5.1.
  - ☐ 2015 - ECMAScript 2015 (ES6).
  - ☐ 2016 - ECMAScript 2016 (ES7).
  - ☐ 2017 - ECMAScript 2017 (ES8).
  - ☐ 2018 - ES9

Datatypes

✓ Javascript is loosly typed.

- ☐ Five (primitive) datatypes: `Undefined`, `Null`, `Boolean`, `Number` och `String`.
- ☐ One more complex type, `Object`, unordered list with name/value couples.
- ☐ All values must be of one of the types above.

✓

The operator `typeof` gets the type of a value as a String.

```
typeof 42                                  // number
typeof 3.14                                // number
typeof 'Hi hope!'                          // string
typeof true                                // boolean
typeof function foo() { console.log('bar'); } // function
typeof (() => {})                          // function
typeof {age: 42, name: 'Ellen Nu'}         // object
typeof [1, 4, 1, 4, 2]                     // object
```

(**Note!** Functions are of the type `Object`, even if the opertor `typeof` returns the string `function`.)

The datatype Number

✓ All Numbers are stored as 64 bit numbers;

✓ Numbers can be expressed in different ways.

```
42      // integer
3.14    // float
4.712e7 // exponential number (4,712 · 10⁷)
```

✓ Arethmetic **operators** works as usual.

```
23 + 19      // addition (→ 42)
115 - 73     // subtraction (→ 42)
7 * 6        // multiplication (→ 42)
210 / 5      // division (→ 42)
18 % 5       // rest (→ 3, "remainder operator")
4 + 2 * 7    // multiplication have higher priority than addition (→ 18)
(4 + 2) * 7  // parentheses have the highest priority (→ 42)
```

✓ Special Numbers: `infinity`, `-infinity`, `NaN` (Not-A-Number)

The datatype `String`

✓ Strings represent text.

```
'Can be written inside apostrophes.'
"Can be written inside quotes aswell." // NOT in this course!
```

☐ The code standard in the course will prohibit the use of quotes!

✓ Strings can be concatenated (latin: *catena*, chain, "chaining").

```
'You can ' + 'add strings ' + 'togheter.'
```

✓ Newline character \n, an escape sequence, is used to create Strings spanning over several lines.

```
'A String can be divided\nover several lines\nwithout a problem.'
```

The datatype Boolean

✓ Only two values possible, `true` or `false`.

✓ Boolean values can be created with **comparisons**, <, >, <=, >=, ==, !=, === (compare type as well), !== (compare type as well).

```
5 > 3 // true
5 < 3 // false
'Johan' < 'A good guy' // false, works with strings as well
'Johan' < 'a good guy' // true, lowercase characters are ordered before UPPERCASE
```

✓ **Logical operators** are used to reason with boolean values; AND (&&), OR (||), NOT (!).

```
true && false // false
true || false // true
!false        // true
```

Automatic type convertion

✓ If possible, automatic type conversion is used:

```
42;      // 42
'4' + 3 + 2 // "432"
4 + 3 + '2' // "72"
```

Variables

✓ Variables are defined using `let`, `const` or `var`, followed by an identifier.

```
let count                   // undefined
count = 21
var result = count * 2
```

✓ Different types can be stored in the same variable

```
let result = 42              // Number
result =  'Meaning of life'  // String
result = 12.3                // Number
```

`var`, *function scope*

`let`, *block scope*

`const`, *identifier can not be reassigned*

Variable names

✓ The following words are reserved

```
break       do          in          typeof
case        else        instanceof  throw
catch       export      new         try
class       extends     let         var
const       finally     return      void
continue    for         static      while
debugger    function    super       with
default     if          switch      yield
delete      import      this
```

```
await       implements  package     protected
enum        interface   private     public
```

Control flow

- ✓ Sequences
- ✓ Conditional statements
  - ☐ `if`-statement,
  - ☐ `switch`-statement,
  - ☐ The conditional operator (`?:`)
- ✓ Iterations
  - ☐ `while`-statement,
  - ☐ `do...while`-statement,
  - ☐ `for`-statement.
- ✓ Recursion
  - ☐ Self-invoking functions

Conditional statements

✓ `if`-statement

```
let number = 42
if (number === 0) {
    console.log('The Number is 0.')
}
```

✓ `if...else`-statement

```
let number = 42
if (number < 0) {
  console.log("Less than 0.")
} else if (number === 0) {
  console.log("Exactly 0")
} else {
  console.log("More than 0.")
}
```

Conditional statements

`switch`-statement

```
let number = 4
switch (number) {
 case 1:
 case 2:
 case 3:
 case 4:
 case 5:
 case 6:
   console.log(To small!")
   break
 case 7:
   console.log("That is right!")
   break
 case 8:
 case 9:
 case 10:
   console.log("To large!")
   break

 default:
   console.log(number + " is not part of the interval 1-10")
   break
}
```

Conditional statements

Conditional operator (ternery operator)

```
let number = 42
let output = 'The number is '
output += number % 2 === 0 ? 'even.' : 'odd.'
console.log(output) // → The number is even.
```

Iterations

while-statement

```
let sum = 0
while (sum < 500) {
  sum += sum + 42
}
console.log(sum) // → 630
```

do...while-statement

```
let i = 10
do {
  console.log(i + " ")
  i += 1
} while (i < 10)
// → 10
```

Iterations

for-statement

```
let output = ''

for (let i = 0; i < 10; i += 1) {
  output += i + ' '
}

console.log(output) // → 0 1 2 3 4 5 6 7 8 9
```

Iterations

✓ `break`-statement vill cancel the loop.

```
let output = ''

for (let i = 0; i < 10; i += 1) {
  if (i === 4) {
    break
  }
  output += i + ' '
}

console.log(output) // → 0 1 2 3
```

✓ `continue`-statements cancels the active iteration and starts a new one.

```
let output = ''

for (let i = 0; i < 10; i += 1) {
  if (i % 3 === 0) {
    continue
  }
  output += i + ' '
}

console.log(output) // → 1 2 4 5 7 8
```

(`continue` is by some marked as "bad practice". Code can often be written in such a way that continue is not needed.)

Comments

```
let product = 1

// Row comments
for (let i = 3; i < 100; i += 2) {
  product *= i
}

/* This is a block comment spanning
   over many rows
*/
console.log(product)

/**
 * (Documentation comment (JSDOC).)
 * Returns a string where all...
 *
 * @param {string} str  The string being...
 * @returns {string} A new string with...
 */
let foo = function(data) {

  // do something with data to create some new data...

  return newData
}
```

Functions

✓ You defines functions in the same way as variables. The value of the variable is the function.

```
const sayYourNameAndAge = function(name, age) {
    let greeting = 'Hello '+ name + '. You are ' + age + ' years old.'

    return greeting
}

let message = sayYourNameAndAge('Ellen', 30, 'ignored argument')
console.log(message) // → Hello Ellen. You are 30 years old.
```

All functions returns

```
const ageUndefined = function(name, age) {
    return age
};

const returnUndefined = function() {

}

console.log(ageUndefined("Ellen")) // → undefined

console.log(returnUndefined()) // → undefined
```

Default parameters

Old way

```
const sayYourNameAndAge = function(name, age) {

    name = name ? name : 'Noname'
    age = age ? age : '18' // 0 will evaluate to false!

    let greeting = 'Hello '+ name + '. You are ' + age + ' years old.'

    return greeting
}

console.log(sayYourNameAndAge()); // ⟶ Hello Ellen. You are 18 years old.
```

ECMAScript 6 (ES2015)

```
const sayYourNameAndAge = function(name = 'Noname', age = 18) {

    let greeting = 'Hello '+ name + '. You are ' + age + ' years old.'

    return greeting
}

console.log(sayYourNameAndAge()) // ⟶ Hello Ellen. You are 18 years old.
```

Hoisting

Variables defined by `var` are "hoisted" to the top of the scope.

What you see:

```
const theScope = function() {
    console.log(variable) // → undefined

    var variable = 10

    console.log(variable) // → 10
}
theScope();
```

What is executed:

```
const theScope = function() {
    var variable

    console.log(variable) // → undefined

    variable = 10

    console.log(variable) // → 10
}
theScope()
```

Variables declared with `let` will hoist but referencing the variable before the declaration will result in a ReferenceError.

Expression vs. declaration

Function Expression:

```
theScope() // → ReferenceError: theScope is not defined

const theScope = function(a = 0, b = 0) {
    return a + b
}
```

Function Declaration:

```
theScope()

function theScope(a = 0, b = 0) {
    return a + b
}
```

Function declarations are hoisted in JavaScript

Anonymous functions

Functions that are not named are called "Anonymous functions"

```
const creatorFunction = function() {
  return function (){
    return 'We Are Anonymous'
  }
}
console.log(creatorFunction()())
```

The returned, anonomys, function will still have a reference to the outer functions parameter "factor"

Arrow functions

Old way:

```
const creatorFunction = function() {
  return function (){
    return 'I am Anonymous' // ⟶ I am Anonymous
  }
}
console.log(creatorFunction()())
```

ES2015 using Arrow functions:

```
let creatorFunction = function() {
  return () => 'I am Anonymous'
}
console.log(creatorFunction()()) // ⟶ I am Anonymous
```

Nested scopes

```
function outerScope(runFunction = 1) {
  let result

  const firstInnerScope = function(){
      let a = 10
      return a
  };

  const secondInnerScope = function(){
      let a = 20
      return a
  };

  if(runFunction === 1){
    result = firstInnerScope()
  } else {
    result = secondInnerScope()
  }

  return result
};

console.log(outerScope(2)) // → 20
```

Nested scopes

Inner functions can access outer functions scopes

```
let outerResult = 0

function outerScope() {
  let result = 0

  const firstInnerScope = function(){
      result += 10
      outerResult += 1
  };

  const secondInnerScope = function(){
      result += 20
      outerResult += 1
  };

  firstInnerScope()
  secondInnerScope()

  return result
};

console.log(outerScope()) // → 30
console.log(outerResult)  // → 2
```

Closure

```
function multiplier(factor){
  return function(number) {
    return number * factor
  }
}
let twice = multiplier(2)
console.log(twice(5)) // → 10
```

or with arrow functions:

```
function multiplier(factor){
  return (number) => number * factor
}
let twice = multiplier(2)
console.log(twice(5)) // → 10
```

The returned, anonymous, function will still have a reference to the outer functions parameter "factor".

`multiplier` is called a "Higher order function".

Regular Expressions

```
let regexp = /(\w+)\s(\w+)/
let str = "John Smith"
let newstr = str.replace(regexp, "$2, $1")
console.log(newstr)
```

MDN Regular Expressions

Error handeling

```
try {
   throw new Error('My error 123')
} catch (error) {
   console.log("Error:", error.message)
}
finally {
   console.log("Always")
}
```

To finish off...

```
let nbr1    = 42
let str     = ''
let bool    = true
let func    = () => {}
let regexp  = / /
let arr     = []
let obj     = {}
```