COMP519 Web Programming

Lecture 12: JavaScript (Part 3)

Handouts

Ullrich Hustadt

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

Contents

1. Type Conversion
   - Type Coercion
   - Type Casting
2. Comparison Operators
   - Equality
   - Less and Greater Than
   - Numbers Revisited: NaN and Infinity
3. Control Structures
   - Blocks
   - Conditional statements
   - Switch statements
   - While- and Do While-loops
   - For-loops
   - Try, throw, catch, finally statements
4. Further Reading

Type Coercion

- JavaScript automatically converts a value to the appropriate type as required by the operation applied to the value (type coercion)

```
5 *  "3"   // result is 15
5 +  "3"   // result is "53"
5 && "3"   // result is "3"
```

- The value `undefined` is converted as follows:

Type	Default	Type	Default	Type	Default
bool	false	string	'undefined'	number	NaN

```
undefined ||  true   // result is true
undefined +   "-!"   // result is "undefined-!"
undefined +   1      // result is NaN
```

Type Coercion

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

Type Casting

JavaScript provides several ways to explicitly type cast a value

- Apply an identity function of the target type to the value

`"12" * 1`	$\leadsto$	`12`	`!!"1"`	$\leadsto$	`true`
`12 + ""`	$\leadsto$	`"12"`	`!!"0"`	$\leadsto$	`true`
`false + ""`	$\leadsto$	`"false"`	`!!""`	$\leadsto$	`false`
`[12,[3,4]] + ""`	$\leadsto$	`"12,3,4"`	`!!1`	$\leadsto$	`true`
			`[12,13] * 1`	$\leadsto$	`NaN`
			`[12] * 1`	$\leadsto$	`12`

Type Casting

JavaScript provides several ways to explicitly type cast a value

- Wrap a value of a primitive type into an object
  - ⤳ JavaScript has objects `Number`, `String`, and `Boolean` with unary constructors/wrappers for values of primitive types (JavaScript does not have classes but prototypical objects)

`Number("12")`	⤳	`12`	`Boolean("0")`	⤳	`true`
`String(12)`	⤳	`"12"`	`Boolean(1)`	⤳	`true`
`String(false)`	⤳	`"false"`	`Number(true)`	⤳	`1`

- Use parser functions `parseInt` or `parseFloat`

`parseInt("12")`	⤳	`12`	`parseFloat("2.5")`	⤳	`2.5`
`parseInt("2.5")`	⤳	`2`	`parseFloat("2.5e1")`	⤳	`25`
`parseInt("E52")`	⤳	`NaN`	`parseFloat("E5.2")`	⤳	`NaN`
`parseInt(" 42")`	⤳	`42`	`parseFloat(" 4.2")`	⤳	`4.2`
`parseInt("2014Mar")`	⤳	`2014`	`parseFloat("4.2end")`	⤳	`4.2`

Comparison Operators

JavaScript distinguishes between (loose) equality `==`
and strict equality `===`:

expr1 `==` *expr2*	Equal	TRUE iff *expr1* is equal to *expr2* after type coercion
expr1 `!=` *expr2*	Not equal	TRUE iff *expr1* is not equal to *expr2* after type coercion

- When comparing a number and a string, the string is converted to a number
- When comparing with a boolean,
  the boolean is converted to 1 if `true` and to 0 if `false`
- If an object is compared with a number or string, JavaScript uses the `valueOf` and `toString` methods of the objects to produce a primitive value for the object
- If two objects are compared, then the equality test is true only if both refer to the same object

Comparison Operators

JavaScript distinguishes between (loose) equality ==
and strict equality ===:

expr1 === *expr2*	Strictly equal	TRUE iff *expr1* is equal to *expr2*, and they are of the same type
expr1 !== *expr2*	Strictly not equal	TRUE iff *expr1* is not equal to *expr2*, or they are not of the same type

```
"123" == 123              ⇝  true
"123" != 123              ⇝  false
"1.23e2" == 123           ⇝  true
"1.23e2" == "12.3e1"      ⇝  false
5 == true                 ⇝  false
```

```
"123" === 123             ⇝  false
"123" !== 123             ⇝  true
1.23e2 === 123            ⇝  false
"1.23e2" === "12.3e1"     ⇝  false
5 === true                ⇝  false
```

Comparison Operators

JavaScript's comparison operators also apply type coercion to their operands and do so following the same rules as equality ==:

expr1 < *expr2*	Less than	true iff *expr1* is strictly less than *expr2* after type coercion
expr1 > *expr2*	Greater than	true iff *expr1* is strictly greater than *expr2* after type coercion
expr1 <= *expr2*	Less than or equal to	true iff *expr1* is less than or equal to *expr2* after type coercion
expr1 >= *expr2*	Greater than or equal to	true iff *expr1* is greater than or equal to *expr2* after type coercion

```
'35.5' > 35              ⇝  true
'ABD' > 'ABC'            ⇝  true
'1.23e2' > '12.3e1'      ⇝  false
"F1" < "G0"              ⇝  true
true > false             ⇝  true
5 > true                 ⇝  true
```

```
'35.5' >= 35             ⇝  true
'ABD' >= 'ABC'           ⇝  true
'1.23e2' >= '12.3e1'     ⇝  false
"F1" <= "G0"             ⇝  true
true >= false            ⇝  true
5 >= true                ⇝  true
```

Numbers Revisited: NaN and Infinity

- JavaScript's number type includes constants

`NaN`	(case sensitive)	'not a number'
`Infinity`	(case sensitive)	'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')

Numbers Revisited: NaN and Infinity

- Equality and comparison operators need to be extended to cover `NaN` and `Infinity`:

`NaN == NaN`	$\leadsto$ `false`	`NaN === NaN`	$\leadsto$ `false`
`Infinity == Infinity`	$\leadsto$ `true`	`Infinity === Infinity`	$\leadsto$ `true`
`NaN == 1`	$\leadsto$ `false`	`Infinity == 1`	$\leadsto$ `false`
`NaN < NaN`	$\leadsto$ `false`	`Infinity < Infinity`	$\leadsto$ `false`
`1 < Infinity`	$\leadsto$ `true`	`1 < NaN`	$\leadsto$ `false`
`Infinity < 1`	$\leadsto$ `false`	`NaN < 1`	$\leadsto$ `false`
`NaN < Infinity`	$\leadsto$ `false`	`Infinity < NaN`	$\leadsto$ `false`

$\leadsto$ A lot of standard mathematical properties for numbers do not apply to the `number` type, e.g.

$$\forall x, y \in \mathbb{R} : (x < y) \vee (x = y) \vee (x > y)$$

$\leadsto$ equality cannot be used to check for `NaN`

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'`

Control Structures

JavaScript control structures

- block statements
- conditional statements
- switch statements
- while- and do while-loops
- for-loops
- break and continue
- try, throw, catch finally statements

are syntactically identical to those of Java

Control structures: block statements

A block statement is a sequence of zero or more statements delimited by a pair of curly brackets

```
{
  statements
}
```

- It allows to use multiple statements where JavaScript expects only one statement

```
{
  var x = 1
  var y = x++
}
```

Control structures: conditional statements

Conditional statements take the following form in JavaScript:

```
if (condition)
  statement
else if (condition)
  statement
else
  statement
```

- There are no `elsif`- or `elseif`-clauses in JavaScript
- The else-clause is optional but there can be at most one
- Each statement can be a block statement

Control structures: conditional statements

```
if (age < 18) {
  price = 10;
  categ = 'child;
} else if (age >= 65) {
  price = 20;
  categ = 'pensioner';
} else {
  price = 100;
  categ = 'adult';
}
```

JavaScript also supports conditional expressions

condition ? *if_true_expr* : *if_false_expr*

```
y = (x < 0) ? -1/x : 1/x

price = (age < 18) ? 10 : (age >= 65) ? 20 : 100
categ = (age < 18) ? 'child' : (age >= 65) ? 'pensioner' :
                                               'adult'
```

Control structures: switch statement

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

```
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)
  statement

do
  statement
while (condition)
```

Example:

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

Control structures: for-loops

- for-loops in JavaScript take the form

```
for (initialisation; test; increment)
  statement
```

```
var factorial = 1
for (var i = 1; i <= number; i++)
  factorial *= i
```

- A for-loop is equivalent to the following while-loop:

```
initialisation
while (test) {
    statement
    increment
}
```

```
for (var factorial = 1; number; number--)
  factorial *= number
```

Control structures: for-loops

- In JavaScript, *initialisation* and *increment* can consist of more than one statement, separated by commas instead of semicolons

```
for (i = 1, j = 1; j >= 0; i++, j--)
  console.log(i + " * " + j + " = " + i*j)
  // Indentation has no ‘meaning’ in JavaScript,
  // the next line is not part of the loop !!BAD STYLE!!
  console.log("Outside loop: i = " + i + "j = " + j)
```

```
1 * 1 = 1
2 * 0 = 0
Outside loop: i = 3 | j = -1
```

- Note: Variables declared with `var` inside a for-loop are still visible outside

```
for (var i = 0; i < 1; i++)
  console.log("Inside  loop: i = " + i)
console.log("Outside loop: i = " + i)
```

```
Inside  loop: i = 0
Outside loop: i = 1
```

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

```
// Looking for a value x, 0<x<100, for which f(x) equals 0
x = 1
while (x < 100) {
  if (f(x) == 0) break;
  x++
}
```

- 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;
  console.log("10 / " + x + " = " + (10/x));
}
```

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

Error handling

- When a JavaScript statement generates an error, an exception is thrown
- Exceptions can also explicitly be thrown via a `throw` statement
- A `try ... catch ...` statement allows for error / exception handling

```
try { statements }
catch (error) { statements }
finally { statements }
```

- Statements in the `try` block are executed first
- If any statement within the `try` block throws an exception, control immediately shifts to the `catch` block
- *error* is a variable used to store the error / exception
- Statements in the `finally` block are executed after `try` and `catch` statements, regardless of whether an exception was thrown or caught
- Curly brackets are necessary even where a block consists of only one statement

Error handling

- When a JavaScript statement generates an error, an exception is thrown
- Exceptions can also explicitly be thrown via a `throw` statement
- A `try ... catch ...` statement allows for error / exception handling

```
throw expression
```

- A throw statement throws (generates) an exception and interrupts the normal flow of control
- The exception *expression* can be a string, a number, a boolean or an object

- `try ... catch ...` statements can be nested

Error handling: Example

```
x = "A"
try {
  if(isNaN(x)) throw "x is NaN"
  y = x.toFixed(2)
} catch (e) {
  console.log('Caught: ' + e)
  y = 0
} finally {
  console.log('y = ',y)
}
```

```
Caught TypeError: x.toFixed is not a function
y = 0
```

Coding Styles

EAFP (easier to ask for forgiveness than permission)

- Makes the default assumption that code works without error
- Any problems are caught as exceptions
- Code contains a lot of `try ... catch ...` statements

```
try { y = x.toFixed(2) } catch (e) { y = 0 }
```

LBYL (look before you leap)

- Before executing a statement that might fail, first check whether the condition for its success are true, only then proceed
- Code contains a lot of conditional statements

```
if ((typeof(x) == 'number') && isFinite(x))
  y = x.toFixed(2)
else
  y = 0
```

Revision and Further Reading

- Read
  - Chapter 14: Exploring JavaScript
  - Chapter 15: Expressions and Control Flow in JavaScript

  of R. Nixon: Learning PHP, MySQL & JavaScript: with jQuery, CSS & HTML5. O'Reilly, 2018.
- Read
  - Chapter 3: Language Basics: Data Types
  - Chapter 3: Language Basics: Operators
  - Chapter 3: Language Basics: Statements

  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`