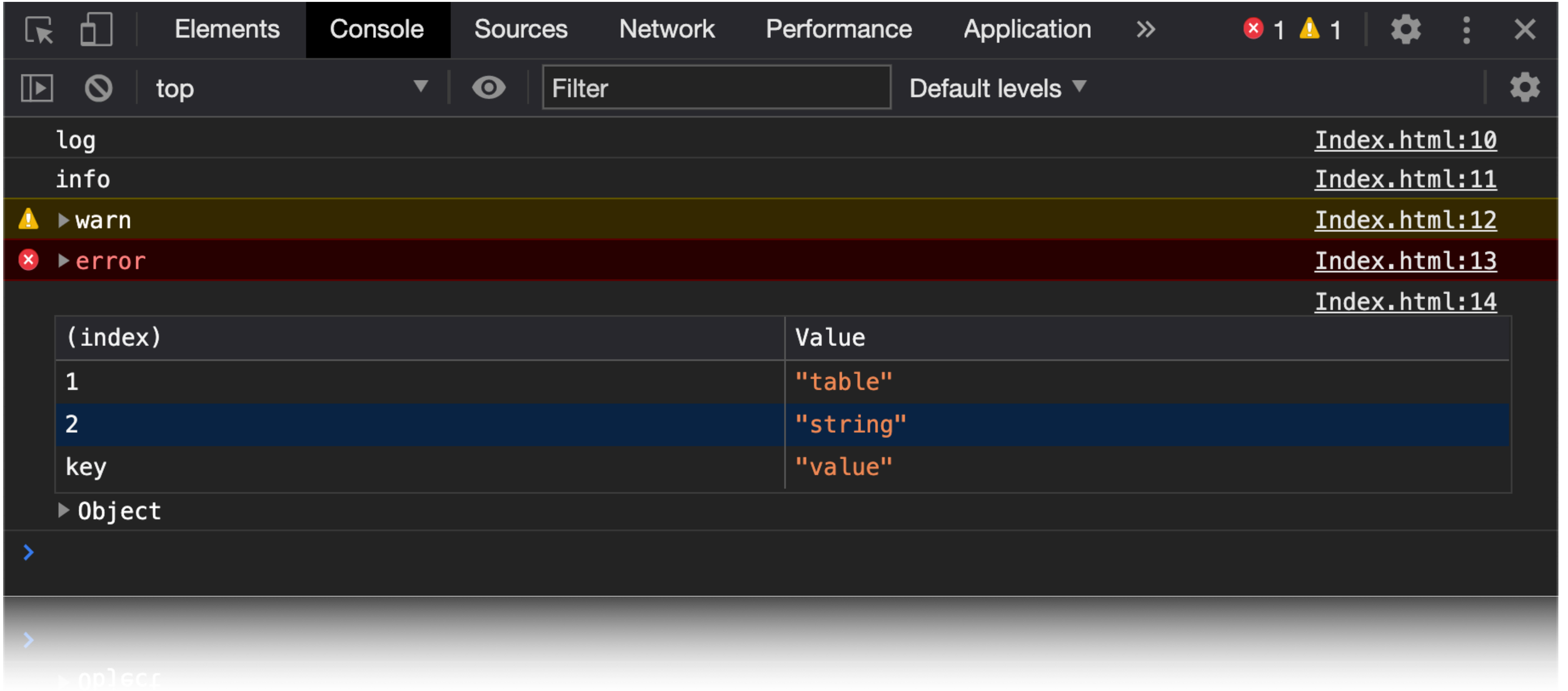
ALERTS

- alert([string message]);
- confirm([string message]);
- prompt([string message], [string defaultValue]);

CONSOLE

- console.log(obj1 [, obj2, ..., objN]);
- console.info(obj1 [, obj2, ..., objN]);
- console.warn(obj1 [, obj2, ..., objN]);
- console.error(obj1 [, obj2, ..., objN]);
- console.table(obj1 [, obj2, ..., objN]);
- console.clear();


Elements
Console
Sources
Network
Performance
Application
1
1
top
Filter
Default levels
log
Index.html:10
info
Index.html:11
warn
Index.html:12
error
Index.html:13
Index.html:14
(index)
Value
1
"table"
2
"string"
key
"value"
Object

VARIABLES & CONSTANTS

- var, let, const;
- To keep data;
- No digit allowed at the start;
- camelCase (js is case sensitive);
- Only latin letters; Only english words;
- Use 2 spaces for indentation (no tabs);
- End statements with a semicolon;

FORMATTING

- flatcase
- UPPERFLATCASE
- camelCase
- PascalCase
- snake_case
- SCREAMING_SNAKE_CASE, MACRO_CASE, CONSTANT_CASE
- kebab-case / dash-case / lisp-case
- TRAIN-CASE / COBOL-CASE / SCREAMING-KEBAB-CASE
- Train-Case / HTTP-Header-Case

VAR

- Allows the multiple declaration;
- The var statement declares a function-scoped or globally-scoped variable;
- Prone to hoisting;

Can be used:

- If project doesn’t support ES2015(ES6) e.g. ie6 support and no polyfills applied;
- If project was written on “vars”;

CONST

- declares a block-scoped local constant;

LET

- declares a block-scoped local variable;

<SCRIPT>

- JS can be executed in <script> tag
- src="path_to_file"
- async
- defer

“USE STRICT”

- Fixes some of downsides;
- Should be declared at the top of file/function;
- Turned off in a browser console;
- No way to cancel strict mode;

OPERATORS

- Arithmetic operators
- Assignment operators
- Comparison operators
- Conditional (ternary) operator
- Logical operators
- Bitwise operators

ARITHMETIC OPERATORS (BINARY)

Operator		Example	Result
Addition	+	3 + 6	9
Subtraction	-	7 - 2	5
Multiplication	*	5 * 5	25
Division	/	5 / 2	2.5
Remainder	%	31 / 6	1
Exponentiation	**	2 ** 3	8

GROUPING OPERATOR

- ()

- 2 * (4 + 3) // 14

ARITHMETIC OPERATORS (UNARY)

Operator		Example	Result
Plus	+	+"7"	7
Negation	-	let a = 2; -a	-2
Increment	++	let b = 3; ++b / b++	4
Decrement	--	let c = 4; --c / c--	3

ASSIGNMENT OPERATORS

Name	Shorthand operator	Meaning
Assignment	x = y	x = y
Addition assignment	x += y	x = x + y
Subtraction assignment	x -= y	x = x - y
Multiplication assignment	x *= y	x = x * y
Division assignment	x /= y	x = x / y
Remainder assignment	x %= y	x = x % y
Exponentiation assignment	x **= y	x = x ** y

COMPARISON OPERATORS

Operator	Description
Equal (==)	Returns true if the operands are equal.
Not equal (!=)	Returns `true` if the operands are not equal.
Strict equal (===)	Returns `true` if the operands are equal and of the same type. See also `Object.is` and sameness in JS.
Strict not equal (!==)	Returns `true` if the operands are of the same type but not equal, or are of different type.
Greater than (>)	Returns `true` if the left operand is greater than the right operand.
Greater than or equal (>=)	Returns `true` if the left operand is greater than or equal to the right operand.
Less than (<)	Returns `true` if the left operand is less than the right operand.
Less than or equal (<=)	Returns `true` if the left operand is less than or equal to the right operand.

CONDITIONAL (TERNARY) OPERATOR

- condition ? val1 : val2

- const title = isReady ? “Hello, Oleg!” : “loading...”;

LOGICAL OPERATORS

Operator	Usage	Description
Logical AND (`&&`)	expr1 && expr2	Returns `expr1` if it can be converted to `false`; otherwise, returns `expr2`. Thus, when used with Boolean values, `&&` returns `true` if both operands are true; otherwise, returns `false`.
Logical OR (`||`)	expr1 || expr2	Returns `expr1` if it can be converted to `true`; otherwise, returns `expr2`. Thus, when used with Boolean values, `||` returns `true` if either operand is true; if both are false, returns `false`.
Logical NOT (`!`)	!expr	Returns `false` if its single operand that can be converted to `true`; otherwise, returns `true`.

BITWISE OPERATORS

Operator	Usage	Description
Bitwise AND	a & b	Returns a one in each bit position for which the corresponding bits of both operands are ones.
Bitwise OR	a | b	Returns a zero in each bit position for which the corresponding bits of both operands are zeros.
Bitwise XOR	a ^ b	Returns a zero in each bit position for which the corresponding bits are the same.
Bitwise NOT	~ a	Inverts the bits of its operand.
Left shift	a << b	Shifts `a` in binary representation `b` bits to the left, shifting in zeros from the right.
Sign-propagating right shift	a >> b	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off.
Zero-fill right shift	a >>> b	Shifts `a` in binary representation `b` bits to the right, discarding bits shifted off, and shifting in zeros from the left.

DATA TYPES (PRIMITIVE)

- string
- number
- boolean
- undefined
- null
- bigInt
- symbol

DATA TYPES (NON-PRIMITIVE)

- object

TYPEOF

- typeof(“some string”) // string
- typeof true // boolean
- typeof 5 // number
- typeof NaN // number
- typeof null // object X
- typeof undefined // undefined
- typeof 4n // bigint
- typeof Symbol(“sym") // symbol
- typeof function() {} // function
- typeof object {} // object

TYPES CONVERSION

- Implicit
- Explicit

STRING

- "double quotes string";
- 'single quotes string';
- `backtick string`;
- String();

- Concatenation (.concat())
- Interpolation (Template strings)

STRING (CONVERSION)

Implicit:

- + converts to strings if any operator is string (as binary)

Explicit:

- String();

NUMBER

- **Number** type is a double-precision 64-bit binary format IEEE 754 value
- - (2 ** 53 - 1) … 2 ** 53 -1 (Number.MAX_SAFE_INTEGER, Number.MIN_SAFE_INTEGER)
- Infinity, - Infinity (Number.POSITIVE_INFINITY, Number.NEGATIVE_INFINITY)
- NaN (Number.NaN) NaN != NaN
- Number.EPSILON // 0.00000000000000022204
- -0

NUMBER (CONVERSION)

Implicit:

- - * / % converts everything to number
- ~ converts everything to false except -1
- + converts to number if unary

Explicit:

- Number();

NUMBER CONVERSION EXAMPLES

- Number(4) => 4
- Number(4n) => 4
- Number(false) => 0
- Number(“test”) => NaN
- Number(“123”) => 123
- Number(“2e2”) => 200
- Number(“”) => 0
- Number([]) => 0
- Number([1]) => 1
- Number([1, 2, 3]) => NaN
- Number({}) => NaN
- Number(Symbol(“1")) => Uncaught TypeError: Cannot convert a Symbol value to a number at Number

NUMBER (PARSE)

- parseInt()
- parseFloat()

Examples:

parseInt(“4dd3324n") => 4

parseInt(“d1") => NaN

parseFloat(“2.2.2hfsdh”) => 2.2

parseFloat(“a2.2.2") => NaN

BOOLEAN

- true/false or 1/0

FALSE

- false
- 0
- -0
- null
- undefined
- NaN
- “”

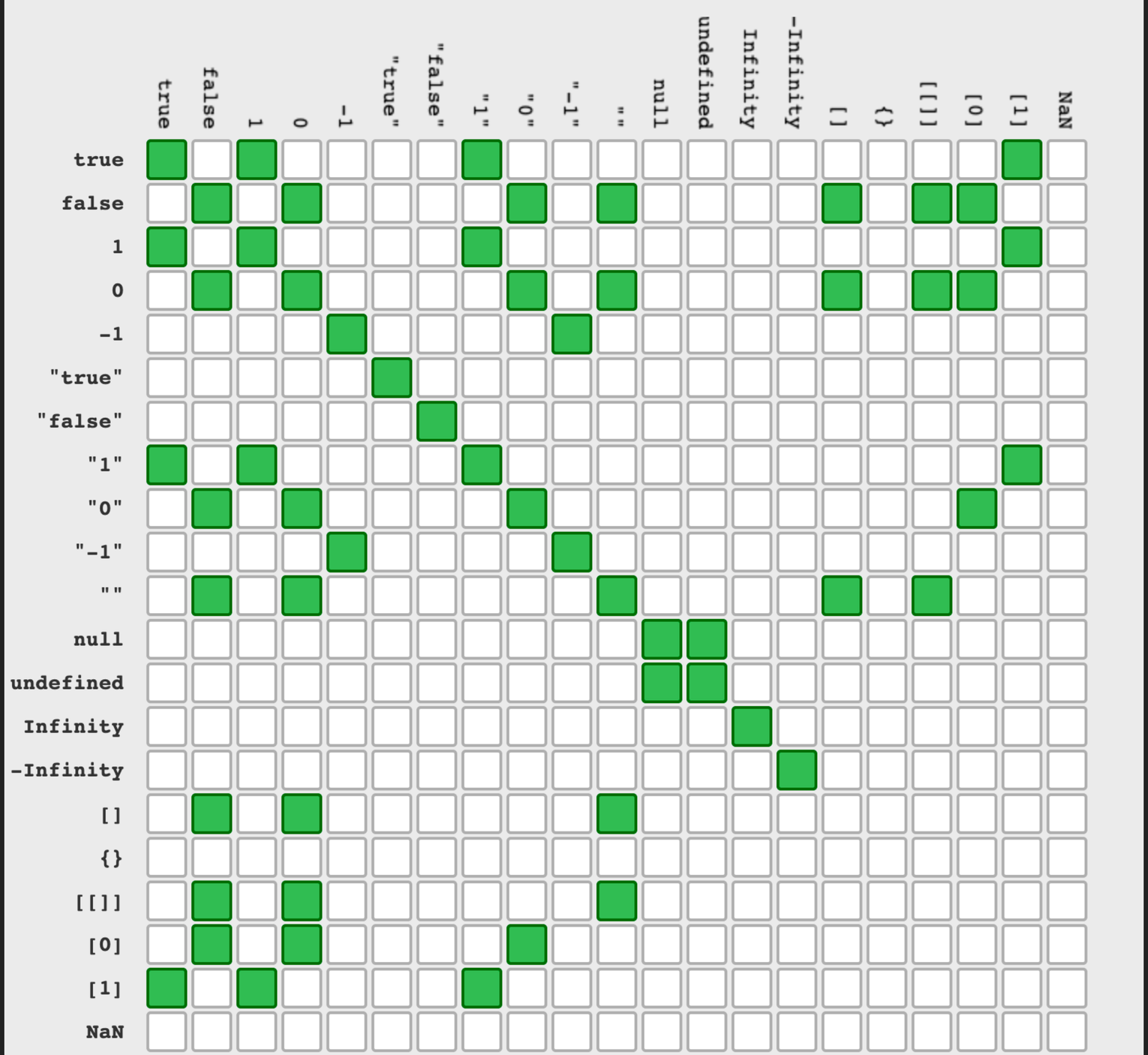
BOOLEAN (CONVERSION)

Implicit:

- if == === ! > < >= <=

Explicit:

- Boolean();


true
false
1
0
-1
"true"
"false"
"1"
"0"
"-1"
""
null
undefined
Infinity
-Infinity
[]
{}
[[]]
[0]
[1]
NaN

UNDEFINED

- **`undefined`** is a primitive value automatically assigned to variables that have just been declared, or to formal arguments for which there are no actual arguments.

- null == undefined // true
- null === undefined // false

NULL

- The value **`null`** represents the intentional absence of any object value

- typeof null === object

BIGINT

- number + n
- new BigInt(number);

- Similar to number
- Can be any length
- Can’t strict equal to regular number
- Can’t use Math

SYMBOL

- Symbol(string)

- always uniq

OBJECT

- Object
- Array
- Function
- Class
- Date
- Regex
- Map
- Set