Chapter 40: Unary Operators

Section 40.1: Overview

Unary operators are operators with only one operand. Unary operators are more efficient than standard JavaScript function calls. Additionally, unary operators can not be overridden and therefore their functionality is guaranteed.

The following unary operators are available:

Operator	Operation	Example
`delete`	The delete operator deletes a property from an object.	example
`void`	The void operator discards an expression's return value.	example
`typeof`	The typeof operator determines the type of a given object.	example
`+`	The unary plus operator converts its operand to Number type.	example
`-`	The unary negation operator converts its operand to Number, then negates it.	example
`~`	Bitwise NOT operator.	example
`!`	Logical NOT operator.	example

Section 40.2: The typeof operator

The `typeof` operator returns the data type of the unevaluated operand as a string.

Syntax:

```
typeof operand
```

Returns:

These are the possible return values from `typeof`:

Type	Return value
`Undefined`	`"undefined"`
`Null`	`"object"`
`Boolean`	`"boolean"`
`Number`	`"number"`
`String`	`"string"`
`Symbol` (ES6)	`"symbol"`
`Function` object	`"function"`
`document.all`	`"undefined"`
Host object (provided by the JS environment)	Implementation-dependent
Any other object	`"object"`

The unusual behavior of `document.all` with the `typeof` operator is from its former usage to detect legacy browsers. For more information, see Why is document.all defined but typeof document.all returns "undefined"?

Examples:

```
// returns 'number'
typeof 3.14;
typeof Infinity;
typeof NaN;               // "Not-a-Number" is a "number"

// returns 'string'
typeof "";
```

```
typeof "bla";
typeof (typeof 1);        // typeof always returns a string

// returns 'boolean'
typeof true;
typeof false;

// returns 'undefined'
typeof undefined;
typeof declaredButUndefinedVariable;
typeof undeclaredVariable;
typeof void 0;
typeof document.all       // see above

// returns 'function'
typeof function(){};
typeof class C {};
typeof Math.sin;

// returns 'object'
typeof { /*<...>*/ };
typeof null;
typeof /regex/;           // This is also considered an object
typeof [1, 2, 4];         // use Array.isArray or Object.prototype.toString.call.
typeof new Date();
typeof new RegExp();
typeof new Boolean(true); // Don't use!
typeof new Number(1);     // Don't use!
typeof new String("abc"); // Don't use!

// returns 'symbol'
typeof Symbol();
typeof Symbol.iterator;
```

Section 40.3: The delete operator

The `delete` operator deletes a property from an object.

Syntax:

```
delete object.property

delete object['property']
```

Returns:

If deletion is successful, or the property did not exist:

- `true`

If the property to be deleted is an own non-configurable property (can't be deleted):

- `false` in non-strict mode.
- Throws an error in strict mode

Description

The `delete` operator does not directly free memory. It can indirectly free memory if the operation means all references to the property are gone.

`delete` works on an object's properties. If a property with the same name exists on the object's prototype chain, the

property will be inherited from the prototype.
`delete` does not work on variables or function names.

Examples:

```
// Deleting a property
foo = 1;              // a global variable is a property of `window`: `window.foo`
delete foo;           // true
console.log(foo);     // Uncaught ReferenceError: foo is not defined

// Deleting a variable
var foo = 1;
delete foo;           // false
console.log(foo);     // 1 (Not deleted)

// Deleting a function
function foo(){ };
delete foo;           // false
console.log(foo);     // function foo(){ } (Not deleted)

// Deleting a property
var foo = { bar: "42" };
delete foo.bar;       // true
console.log(foo);     // Object { } (Deleted bar)

// Deleting a property that does not exist
var foo = { };
delete foo.bar;       // true
console.log(foo);     // Object { } (No errors, nothing deleted)

// Deleting a non-configurable property of a predefined object
delete Math.PI;       // false  ()
console.log(Math.PI); // 3.141592653589793 (Not deleted)
```

Section 40.4: The unary plus operator (+)

The unary plus (+) precedes its operand *and evaluates* to its operand. It attempts to convert the operand to a number, if it isn't already.

Syntax:

```
+expression
```

Returns:

- a `Number`.

Description

The unary plus (+) operator is the fastest (and preferred) method of converting something into a number.

It can convert:

- string representations of integers (decimal or hexadecimal) and floats.
- booleans: `true`, `false`.
- `null`

Values that can't be converted will evaluate to `NaN`.

Examples:

```
+42           // 42
```

```
+"42"          // 42
+true          // 1
+false         // 0
+null          // 0
+undefined     // NaN
+NaN           // NaN
+"foo"         // NaN
+{}            // NaN
+function(){} // NaN
```

Note that attempting to convert an array can result in unexpected return values.
In the background, arrays are first converted to their string representations:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

The operator then attempts to convert those strings to numbers:

```
+[]            // 0   ( === +'' )
+[1]           // 1   ( === +'1' )
+[1, 2]        // NaN ( === +'1,2' )
```

Section 40.5: The void operator

The `void` operator evaluates the given expression and then returns `undefined`.

Syntax:

```
void expression
```

Returns:

- `undefined`

Description

The `void` operator is often used to obtain the `undefined` primitive value, by means of writing `void 0` or `void(0)`.
Note that `void` is an operator, not a function, so `()` is not required.

Usually the result of a `void` expression and `undefined` can be used interchangeably.
However, in older versions of ECMAScript, `window.undefined` could be assigned any value, and it is still possible to use `undefined` as name for function parameters variables inside functions, thus disrupting other code that relies on the value of `undefined`.
`void` will always yield the *true* `undefined` value though.

`void 0` is also commonly used in code minification as a shorter way of writing `undefined`. In addition, it's probably safer as some other code could've tampered with `window.undefined`.

Examples:

Returning `undefined`:

```
function foo(){
    return void 0;
}
console.log(foo()); // undefined
```

Changing the value of `undefined` inside a certain scope:

```
(function(undefined){
    var str = 'foo';
    console.log(str === undefined); // true
})('foo');
```

Section 40.6: The unary negation operator (-)

The unary negation (-) precedes its operand and negates it, after trying to convert it to number.

Syntax:

```
-expression
```

Returns:

- a `Number`.

Description

The unary negation (-) can convert the same types / values as the unary plus (+) operator can.

Values that can't be converted will evaluate to `NaN` (there is no `-NaN`).

Examples:

```
-42           // -42
-"42"         // -42
-true         // -1
-false        // -0
-null         // -0
-undefined    // NaN
-NaN          // NaN
-"foo"        // NaN
-{}           // NaN
-function(){} // NaN
```

Note that attempting to convert an array can result in unexpected return values.
In the background, arrays are first converted to their string representations:

```
[].toString() === '';
[1].toString() === '1';
[1, 2].toString() === '1,2';
```

The operator then attempts to convert those strings to numbers:

```
-[]           // -0  ( === -'' )
-[1]          // -1  ( === -'1' )
-[1, 2]       // NaN ( === -'1,2' )
```

Section 40.7: The bitwise NOT operator (~)

The bitwise NOT (~) performs a NOT operation on each bit in a value.

Syntax:

```
~expression
```

Returns:

- a `Number`.

Description

The truth table for the NOT operation is:

a NOT a
0 1
1 0

```
1337  (base 10) = 0000010100111001 (base 2)
~1337 (base 10) = 1111101011000110 (base 2) = -1338 (base 10)
```

A bitwise not on a number results in: `-(x + 1)`.

Examples:

value (base 10)	value (base 2)	return (base 2)	return (base 10)
2	00000010	11111100	-3
1	00000001	11111110	-2
0	00000000	11111111	-1
-1	11111111	00000000	0
-2	11111110	00000001	1
-3	11111100	00000010	2

Section 40.8: The logical NOT operator (!)

The logical NOT (`!`) operator performs logical negation on an expression.

Syntax:

```
!expression
```

Returns:

- a `Boolean`.

Description

The logical NOT (`!`) operator performs logical negation on an expression.

Boolean values simply get inverted: `!true === false` and `!false === true`.
Non-boolean values get converted to boolean values first, then are negated.

This means that a double logical NOT (`!!`) can be used to cast any value to a boolean:

```
!!"FooBar" === true
!!1 === true
!!0 === false
```

These are all equal to `!true`:

```
!'true' === !new Boolean('true');
!'false' === !new Boolean('false');
!'FooBar' === !new Boolean('FooBar');
![] === !new Boolean([]);
!{} === !new Boolean({});
```

These are all equal to `!false`:

```
!0 === !new Boolean(0);
!'' === !new Boolean('');
!NaN === !new Boolean(NaN);
!null === !new Boolean(null);
!undefined === !new Boolean(undefined);
```

Examples:

```
!true          // false
!-1            // false
!"-1"          // false
!42            // false
!"42"          // false
!"foo"         // false
!"true"        // false
!"false"       // false
!{}            // false
![]            // false
!function(){}  // false

!false         // true
!null          // true
!undefined     // true
!NaN           // true
!0             // true
!""            // true
```