Chapter 3: Built-in Constants

Section 3.1: null

`null` is used for representing the intentional absence of an object value and is a primitive value. Unlike `undefined`, it is not a property of the global object.

It is equal to `undefined` but not identical to it.

```
null == undefined; // true
null === undefined; // false
```

CAREFUL: The `typeof null` is `'object'`.

```
typeof null; // 'object';
```

To properly check if a value is `null`, compare it with the strict equality operator

```
var a = null;

a === null; // true
```

Section 3.2: Testing for NaN using isNaN()

`window.isNaN()`

The global function `isNaN()` can be used to check if a certain value or expression evaluates to `NaN`. This function (in short) first checks if the value is a number, if not tries to convert it (*), and then checks if the resulting value is `NaN`. For this reason, **this testing method may cause confusion**.

(*) The "conversion" method is not that simple, see ECMA-262 18.2.3 for a detailed explanation of the algorithm.

These examples will help you better understand the `isNaN()` behavior:

```
isNaN(NaN);          // true
isNaN(1);            // false: 1 is a number
isNaN(-2e-4);        // false: -2e-4 is a number (-0.0002) in scientific notation
isNaN(Infinity);     // false: Infinity is a number
isNaN(true);         // false: converted to 1, which is a number
isNaN(false);        // false: converted to 0, which is a number
isNaN(null);         // false: converted to 0, which is a number
isNaN("");           // false: converted to 0, which is a number
isNaN(" ");          // false: converted to 0, which is a number
isNaN("45.3");       // false: string representing a number, converted to 45.3
isNaN("1.2e3");      // false: string representing a number, converted to 1.2e3
isNaN("Infinity");   // false: string representing a number, converted to Infinity
isNaN(new Date);     // false: Date object, converted to milliseconds since epoch
isNaN("10$");        // true : conversion fails, the dollar sign is not a digit
isNaN("hello");      // true : conversion fails, no digits at all
isNaN(undefined);    // true : converted to NaN
isNaN();             // true : converted to NaN (implicitly undefined)
isNaN(function(){}); // true : conversion fails
isNaN({});           // true : conversion fails
isNaN([1, 2]);       // true : converted to "1, 2", which can't be converted to a number
```

This last one is a bit tricky: checking if an `Array` is `NaN`. To do this, the `Number()` constructor first converts the array

to a string, then to a number; this is the reason why `isNaN([])` and `isNaN([34])` both return **`false`**, but `isNaN([1, 2])` and `isNaN([true])` both return **`true`**: because they get converted to `""`, `"34"`, `"1,2"` and `"true"` respectively. In general, **an array is considered `NaN` by `isNaN()` unless it only holds one element whose string representation can be converted to a valid number**.

Version ≥ 6

`Number.isNaN()`

In ECMAScript 6, the `Number.isNaN()` function has been implemented primarily to avoid the problem of `window.isNaN()` of forcefully converting the parameter to a number. `Number.isNaN()`, indeed, **doesn't try to convert** the value to a number before testing. This also means that **only values of the type number, that are also `NaN`, return `true`** (which basically means only `Number.isNaN(NaN)`).

From ECMA-262 20.1.2.4:

> When the `Number.isNaN` is called with one argument `number`, the following steps are taken:
>
> 1. If Type(number) is not Number, return **`false`**.
> 2. If number is **`NaN`**, return **`true`**.
> 3. Otherwise, return **`false`**.

Some examples:

```
// The one and only
Number.isNaN(NaN);          // true

// Numbers
Number.isNaN(1);            // false
Number.isNaN(-2e-4);        // false
Number.isNaN(Infinity);     // false

// Values not of type number
Number.isNaN(true);         // false
Number.isNaN(false);        // false
Number.isNaN(null);         // false
Number.isNaN("");           // false
Number.isNaN(" ");          // false
Number.isNaN("45.3");       // false
Number.isNaN("1.2e3");      // false
Number.isNaN("Infinity");   // false
Number.isNaN(new Date);     // false
Number.isNaN("10$");        // false
Number.isNaN("hello");      // false
Number.isNaN(undefined);    // false
Number.isNaN();             // false
Number.isNaN(function(){}); // false
Number.isNaN({});           // false
Number.isNaN([]);           // false
Number.isNaN([1]);          // false
Number.isNaN([1, 2]);       // false
Number.isNaN([true]);       // false
```

Section 3.3: NaN

`NaN` stands for "Not a Number." When a mathematical function or operation in JavaScript cannot return a specific number, it returns the value **`NaN`** instead.

It is a property of the global object, and a reference to `Number.NaN`

```
window.hasOwnProperty('NaN'); // true
NaN; // NaN
```

Perhaps confusingly, `NaN` is still considered a number.

```
typeof NaN; // 'number'
```

Don't check for `NaN` using the equality operator. See `isNaN` instead.

```
NaN == NaN  // false
NaN === NaN // false
```

Section 3.4: undefined and null

At first glance it may appear that `null` and `undefined` are basically the same, however there are subtle but important differences.

`undefined` is the absence of a value in the compiler, because where it should be a value, there hasn't been put one, like the case of an unassigned variable.

- `undefined` is a global value that represents the absence of an assigned value.
    - `typeof undefined === 'undefined'`
- `null` is an object that indicates that a variable has been explicitly assigned "no value".
    - `typeof null === 'object'`

Setting a variable to `undefined` means the variable effectively does not exist. Some processes, such as JSON serialization, may strip `undefined` properties from objects. In contrast, `null` properties indicate will be preserved so you can explicitly convey the concept of an "empty" property.

The following evaluate to `undefined`:

- A variable when it is declared but not assigned a value (i.e. defined)
    ```
    let foo;
    console.log('is undefined?', foo === undefined);
    // is undefined? true
    ```
- Accessing the value of a property that doesn't exist
    ```
    let foo = { a: 'a' };
    console.log('is undefined?', foo.b === undefined);
    // is undefined? true
    ```
- The return value of a function that doesn't return a value
    ```
    function foo() { return; }
    console.log('is undefined?', foo() === undefined);
    // is undefined? true
    ```
- The value of a function argument that is declared but has been omitted from the function call
    ```
    function foo(param) {
      console.log('is undefined?', param === undefined);
    }
    foo('a');
    foo();
    // is undefined? false
    // is undefined? true
    ```

`undefined` is also a property of the global `window` object.

```
// Only in browsers
console.log(window.undefined); // undefined
window.hasOwnProperty('undefined'); // true
```

Version < 5

Before ECMAScript 5 you could actually change the value of the `window.undefined` property to any other value potentially breaking everything.

Section 3.5: Infinity and -Infinity

```
1 / 0; // Infinity
// Wait! WHAAAT?
```

`Infinity` is a property of the global object (therefore a global variable) that represents mathematical infinity. It is a reference to `Number.POSITIVE_INFINITY`

It is greater than any other value, and you can get it by dividing by 0 or by evaluating the expression of a number that's so big that overflows. This actually means there is no division by 0 errors in JavaScript, there is Infinity!

There is also `-Infinity` which is mathematical negative infinity, and it's lower than any other value.

To get `-Infinity` you negate `Infinity`, or get a reference to it in `Number.NEGATIVE_INFINITY`.

```
- (Infinity); // -Infinity
```

Now let's have some fun with examples:

```
Infinity > 123192310293; // true
-Infinity < -123192310293; // true
1 / 0; // Infinity
Math.pow(123123123, 9123192391023); // Infinity
Number.MAX_VALUE * 2; // Infinity
23 / Infinity; // 0
-Infinity; // -Infinity
-Infinity === Number.NEGATIVE_INFINITY; // true
-0; // -0 , yes there is a negative 0 in the language
0 === -0; // true
1 / -0; // -Infinity
1 / 0 === 1 / -0; // false
Infinity + Infinity; // Infinity

var a = 0, b = -0;

a === b; // true
1 / a === 1 / b; // false

// Try your own!
```

Section 3.6: Number constants

The `Number` constructor has some built in constants that can be useful

```
Number.MAX_VALUE;          // 1.7976931348623157e+308
Number.MAX_SAFE_INTEGER;   // 9007199254740991
```

```
Number.MIN_VALUE;           // 5e-324
Number.MIN_SAFE_INTEGER;    // -9007199254740991

Number.EPSILON;             // 0.00000000000000022204460492503130

Number.POSITIVE_INFINITY;   // Infinity
Number.NEGATIVE_INFINITY;   // -Infinity

Number.NaN;                 // NaN
```

In many cases the various operators in JavaScript will break with values outside the range of (`Number.MIN_SAFE_INTEGER`, `Number.MAX_SAFE_INTEGER`)

Note that `Number.EPSILON` represents the different between one and the smallest `Number` greater than one, and thus the smallest possible difference between two different `Number` values. One reason to use this is due to the nature of how numbers are stored by JavaScript see Check the equality of two numbers

Section 3.7: Operations that return NaN

Mathematical operations on values other than numbers return NaN.

```
"b" * 3
"cde" - "e"
[1, 2, 3] * 2
```

An exception: Single-number arrays.

```
[2] * [3]  // Returns 6
```

Also, remember that the `+` operator concatenates strings.

```
"a" + "b"  // Returns "ab"
```

Dividing zero by zero returns **`NaN`**.

```
0 / 0         // NaN
```

Note: In mathematics generally (unlike in JavaScript programming), dividing by zero is not possible.

Section 3.8: Math library functions that return NaN

Generally, `Math` functions that are given non-numeric arguments will return NaN.

```
Math.floor("a")
```

The square root of a negative number returns NaN, because `Math.sqrt` does not support imaginary or complex numbers.

```
Math.sqrt(-1)
```