**Number data type**

Numbers is a primitive data type; these numbers may beinteger or real (floating) numbers. Where;

Integer numbers**:** means whole numbers. (i.e. 32)

Real numbers or floating numbers: means numbers containing floating point (decimal point). (i.e. 23.3).

Number data type has three special **values** as shown in this table.

|  |  |  |  |
| --- | --- | --- | --- |
| Data type | Data type **value** | Description | Result when |
| Numbers | Infinity | Infinity represents the mathematical Infinity ∞, which is greater than any number. |  |
| -Infinity | -Infinity represents the mathematical Infinity -∞, which is smaller than any number. -Infinity is the result of dividing |  |
| NaN | NaN represents a special *Not-a-Number* value. It is a result of an invalid or an undefined mathematical operation, | or “string”\* number | |

**NAN represents a non-defined value like.**

* **0 /0**
* **String \* number**
* **String – number**
* **String / number**

As NAN may be all these values so when I say NAN == NAN returns false, as JS engine doesn’t know what exactly value you mean by saying NAN.

**It may be: == or == ….** Then return **false.**

****

**How can I write numbers in JS?**

There is **4** ways to write numbers in JS:

1. **Traditional way:** writing numbers directly.
2. **Exponential (scientific) way:**

**We can represent any number by using exponential way by following these steps:**

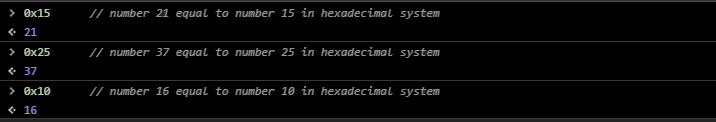
1. Move the floating point left & right its first position until you get a number between (1:10).
2. Multiply the new number \* 10n or (\* e±n). Where (n) is number of steps that floating point moves to get the new number.

**Example:**

****

1. **Hexadecimal number system: 0xn where n is a number.**

**Example:**

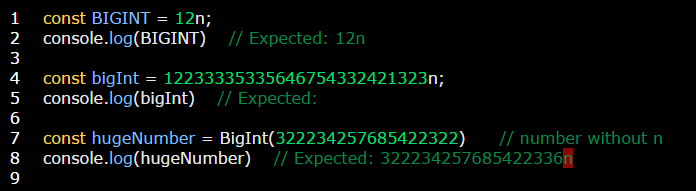
****

1. **bigInt way:**

In JavaScript, the “number” type cannot represent integer values larger than (253-1) (that’s 9007199254740991), or less than -(253-1) for negatives. It’s a technical limitation caused by their internal representation. For most purposes that’s quite enough, but sometimes we need really big numbers, e.g. for cryptography or microsecond-precision timestamps.

A BigInt value is created by appending **n** to the end of an integer or by using **BigInt** constructor.

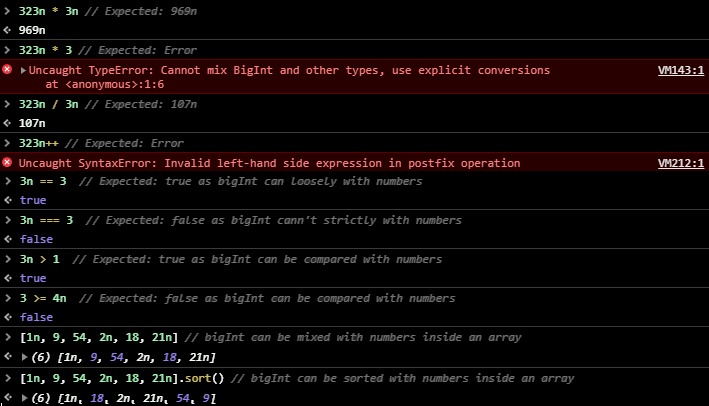
**Example:**



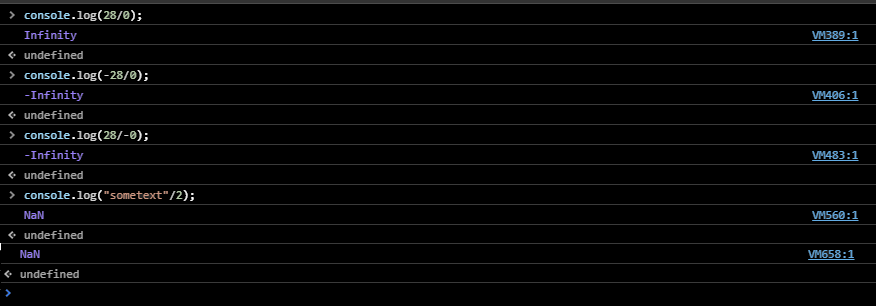
Some important notes:

1. +, -, /, \*, %, \*\* and bitwise operators are supported while unary (+) operator isn’t supported in bigInt numbers since these are BigInts and not BigDecimals, division operation will round towards 0.
2. BigInt aren’t strictly (===) equal to numbers but loosely (==).
3. BigInt and a number can be compared as usual.
4. BigInts can mix with numbers when using in arrays.
5. BigInts can sort with numbers when using in arrays.

**Examples:**



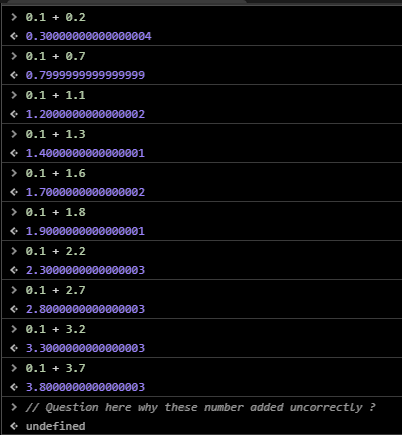
**Example:** on special numbers



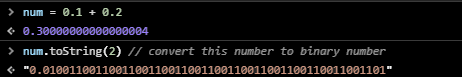
**Note:**

* You can write single numbers with or without parenthesis. [i.e.(33) or 33]. But several numbers must be written inside parenthesis [i.e. (33 / 3)].
* There are some methods related to numbers called “[number methods](Number%20methods.docx)”.

**Interview question**: **why the next numbers added wrongly? And How we can solve this problem?**

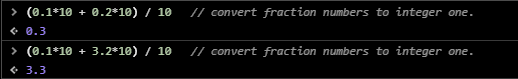


**Answer: we all know that the computer doesn’t understand any system number except the binary one, so the given number “0.1 + 0.2” will converted to binary system like in this picture:**

****

**"0.0100110011001100110011001100110011001100110011001101" will be return. So this numbers “0.1 + 0.2” need a huge memory to be stored, so what occur that computer stored the previous binary number until its memory became filling Then it doesn’t store remain numbers! Because of this “not all binary number of these fraction numbers stored in computer memory” a slightly error happen during adding process.**

**And we can solve this problem by convert the previous fraction number to integer numbers like in this picture:**

****