Faraz Kelhini / JUL 22, 2019 / 12 comments

# The Essential Guide To JavaScript's Newest Data Type: BigInt

8 min read JavaScript, Techniques Share on Twitter, LinkedIn

QUICK SUMMARY *In JavaScript, the `Number` type cannot safely represent integer values larger than $2^{53}$. This limitation has forced developers to use inefficient workarounds and third-party libraries. `BigInt` is a new data type intended to fix that.*

The `BigInt` data type aims to enable JavaScript programmers to represent integer values larger than the range supported by the `Number` data type. The ability to represent integers with arbitrary precision is particularly important when performing mathematical operations on large integers. With `BigInt`, integer overflow will no longer be an issue.

Additionally, you can safely work with high-resolution timestamps, large integer IDs, and more without having to use a workaround. `BigInt` is currently a stage 3 proposal. Once added to the specification, it will become the second numeric data type in JavaScript, which will bring the total number of supported data types to eight:

- Boolean
- Null
- Undefined
- Number
- BigInt
- String
- Symbol
- Object

In this article, we will take a good look at `BigInt` and see how it can help overcome the limitations of the `Number` type in JavaScript.

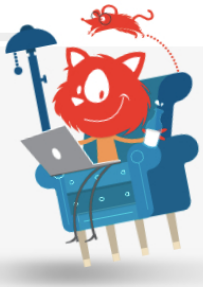
## The Problem #

The lack of an explicit integer type in JavaScript is often baffling to programmers coming from other languages. Many programming languages support multiple numeric types such as float, double, integer, and bignum, but that's not the case with JavaScript. In JavaScript, all numbers are represented in double-precision 64-bit floating-point format as defined by the IEEE 754-2008 standard.

ABOUT THE AUTHOR

Faraz is a professional JavaScript developer who is passionate about moving the web forward and promoting patterns and ideas that will make development more ... More about Faraz








Under this standard, very large integers that cannot be exactly represented are automatically rounded. To be precise, the `Number` type in JavaScript can only safely represent integers between -9007199254740991 ($-(2^{53}-1)$) and 9007199254740991 ($2^{53}-1$). Any integer value that falls out of this range may lose precision.

This can be easily examined by executing the following code:

```
console.log(9999999999999999);    // → 10000000000000000
```

This integer is larger than the largest number JavaScript can reliably represent with the `Number` primitive. Therefore, it's rounded. Unexpected rounding can compromise a program's reliability and security. Here's another example:

```
// notice the last digits
9007199254740992 === 9007199254740993;    // → true
```

JavaScript provides the `Number.MAX_SAFE_INTEGER` constant that allows you to quickly obtain the maximum safe integer in JavaScript. Similarly, you can obtain the minimum safe integer by using the `Number.MIN_SAFE_INTEGER` constant:

```
const minInt = Number.MIN_SAFE_INTEGER;

console.log(minInt);         // → -9007199254740991

console.log(minInt - 5);     // → -9007199254740996

// notice how this outputs the same value as above
console.log(minInt - 4);     // → -9007199254740996
```




## The Solution #

As a workaround to these limitations, some JavaScript developers represent large integers using the `String` type. The Twitter API, for example, adds a string version of IDs to objects when responding with JSON. Additionally, a number of libraries such as bignumber.js have been developed to make working with large integers easier.

With `BigInt`, applications no longer need a workaround or library to safely represent integers beyond `Number.MAX_SAFE_INTEGER` and

`Number.MIN_SAFE_INTEGER`. Arithmetic operations on large integers can now be performed in standard JavaScript without risking loss of precision. The added benefit of using a native data type over a third-party library is better run-time performance.

To create a `BigInt`, simply append `n` to the end of an integer. Compare:

```
console.log(9007199254740995n);    // → 9007199254740995n
console.log(9007199254740995);     // → 9007199254740996
```

Alternatively, you can call the `BigInt()` constructor:

```
BigInt("9007199254740995");    // → 9007199254740995n
```

Additionally, you can safely work with high-resolution timestamps, large integer IDs, and more without having to use a workaround. `BigInt` is currently a stage 3 proposal. Once added to the specification, it will become the second numeric data type in JavaScript, which will bring the total number of supported data types to eight:

- Boolean
- Null
- Undefined
- Number
- BigInt
- String

the minimum safe integer by using the `Number.MIN_SAFE_INTEGER` constant:

```
const minInt = Number.MIN_SAFE_INTEGER;

console.log(minInt);         // → -9007199254740991

console.log(minInt - 5);     // → -9007199254740996

// notice how this outputs the same value as above
console.log(minInt - 4);     // → -9007199254740996
```

## The Solution #

As a workaround to these limitations, some JavaScript developers represent large [illegible] important when performing mathematical operations on large integers. With `BigInt`, integer overflow will no longer be an issue.

Additionally, you can safely work with high-resolution timestamps, large integer IDs, and more without having to use a workaround. `BigInt` is currently a stage 3 proposal. Once added to the specification, it will become the second numeric data type in JavaScript, which will bring the total number of supported data types to eight:

- Boolean

- Null
- Undefined
- Number
- BigInt

```
// notice how this outputs the same value as above
console.log(minInt - 4);     // → -9007199254740996
```

## The Solution #

As a workaround to these limitations, some JavaScript developers represent large integers using the `String` type. The Twitter API, for example, adds a string version of IDs to objects when responding with JSON. Additionally, a number of libraries such as bignumber.js have been developed to make working with large integers easier.

With `BigInt`, applications no longer need a workaround or library to safely represent integers beyond `Number.MAX_SAFE_INTEGER` and

8 min read  JavaScript, Techniques  Share on Twitter, LinkedIn

**QUICK SUMMARY** ↬ *In JavaScript, the `Number` type cannot safely represent integer values larger than $2^{53}$. This limitation has forced developers to use inefficient workarounds and third-party libraries. `BigInt` is a new data type intended to fix that.*

---

The `BigInt` data type aims to enable JavaScript programmers to represent integer values larger than the range supported by the `Number` data type. The ability to represent integers with arbitrary precision is particularly important when performing mathematical operations on large integers. With `BigInt`, integer overflow will no longer be an issue.

```
// binary
console.log(0b100000000000000000000000000000000000000000000000000011n);
// → 9007199254740995n

// hex
console.log(0x20000000000003n);
// → 9007199254740995n

// octal
console.log(0o400000000000000003n);
// → 9007199254740995n

// note that legacy octal syntax is not supported
console.log(040000000000000003n);
// → SyntaxError
```

represent integers beyond `Number.MAX_SAFE_INTEGER` and `Number.Min_SAFE_INTEGER`. Arithmetic operations on large integers can now be performed in standard JavaScript without risking loss of precision. The added benefit of using a native data type over a third-party library is better run-time performance.




ABOUT THE AUTHOR

Faraz is a professional JavaScript developer who is passionate about moving the web forward and promoting patterns and ideas that will make development more … More about Faraz ↬

To create a `BigInt`, simply append `n` to the end of an integer. Compare:

```
console.log(9007199254740995n);    // → 9007199254740995n
console.log(9007199254740995);     // → 9007199254740996
```

Alternatively, you can call the `BigInt()` constructor:



Instead, you can use the equality operator, which performs implicit type conversion before compering its operands:

```
console.log(10n == 10);    // → true
```

All arithmetic operators can be used on `BigInt`s except for the unary plus (`+`) operator:

```
const minInt = Number.MIN_SAFE_INTEGER;

console.log(minInt);         // → -9007199254740991

console.log(minInt - 5);     // → -9007199254740996

// notice how this outputs the same value as above
console.log(minInt - 4);     // → -9007199254740996
```

## The Solution #

As a workaround to these limitations, some JavaScript developers represent large integers using the `String` type. The Twitter API, for example, adds a string version a mix of `Number` and `BigInt` operands. You also cannot pass a `BigInt` to Web APIs and built-in JavaScript functions that expect a `Number`. Attempting to do so will cause a `TypeError`:

```
10 + 10n;    // → TypeError
Math.max(2n, 4n, 6n);    // → TypeError
```

Note that relational operators do not follow this rule, as shown in this example:

```
10n > 5;    // → true
```

If you want to perform arithmetic computations with `BigInt` and `Number`, you first need to determine the domain in which the operation should be done. To do that, simply convert either of the operands by calling `Number()` or `BigInt()`:

```
// octal
console.log(0o4000000000000000003n);
// → 9007199254740995n

// note that legacy octal syntax is not supported
console.log(040000000000000003n);
// → SyntaxError
```

Keep in mind that you can't use the strict equality operator to compare a `BigInt` to a regular number because they are not of the same type:

```
console.log(10n === 10);    // → false

console.log(typeof 10n);    // → bigint
console.log(typeof 10);     // → number
```

Conversion: Mixed operands are not allowed. Here are some examples:

```
90 | 115;      // → 123
90n | 115n;    // → 123n
90n | 115;     // → TypeError
```



## The BigInt Constructor #

As with other primitive types, a `BigInt` can be created using a constructor function. The argument passed to `BigInt()` is automatically converted to a `BigInt`, if possible:

```
console.log(10n == 10);    // → true
```

All arithmetic operators can be used on `BigInt`s except for the unary plus (`+`) operator:

```
10n + 20n;    // → 30n
10n - 20n;    // → -10n
+10n;         // → TypeError: Cannot convert a BigInt value to a number
-10n;         // → -10n
10n * 20n;    // → 200n
20n / 10n;    // → 2n
23n % 10n;    // → 3n
10n ** 3n;    // → 1000n

let x = 10n;
++x;          // → 11n
```

Bitwise operators such as `|`, `&`, `<<`, `>>`, and `~` operate on `BigInt`s in a similar way to `Number`s. Negative numbers are interpreted as infinite-length two's complement. Mixed operands are not allowed. Here are some examples:

```
90 | 115;      // → 123
90n | 115n;    // → 123n
90n | 115;     // → TypeError
```

## The BigInt Constructor #

As with other primitive types, a `BigInt` can be created using a constructor

If you want to perform arithmetic computations with `BigInt` and `Number`, you first need to determine the domain in which the operation should be done. To do that, simply convert either of the operands by calling `Number()` or `BigInt()`:

```
BigInt(10) + 10n;    // → 20n
// or
10 + Number(10n);    // → 20
```

When encountered in a `Boolean` context, `BigInt` is treated similar to `Number`. In other words, a `BigInt` is considered a truthy value as long as it's not `0n`:

```
if (5n) {
    // this code block will be executed
```

supported browsers is available on Can I use....

Unluckily, transpiling `BigInt` is an extremely complicated process, which incurs hefty run-time performance penalty. It's also impossible to directly polyfill `BigInt` because the proposal changes the behavior of several existing operators. For now, a better alternative is to use the JSBI library, which is a pure-JavaScript implementation of the `BigInt` proposal.

This library provides an API that behaves exactly the same as the native `BigInt`. Here's how you can use JSBI:

```
import JSBI from './jsbi.mjs';

const b1 = JSBI.BigInt(Number.MAX_SAFE_INTEGER);
const b2 = JSBI.BigInt('10');
```

No implicit type conversion between `BigInt` and `Number` types occurs when sorting an array:

```
const arr = [3n, 4, 2, 1n, 0, -1n];

arr.sort();    // → [-1n, 0, 1n, 2, 3n, 4]
```

Bitwise operators such as `|`, `&`, `<<`, `>>`, and `^` operate on `BigInt`s in a similar way to `Number`s. Negative numbers are interpreted as infinite-length two's complement. Mixed operands are not allowed. Here are some examples:

```
90 | 115;      // → 123
90n | 115n;    // → 123n
90n | 115;     // → TypeError
```

timestamps, use large integer IDs, and more without the need to use a library.

It's important to keep in mind that you cannot perform arithmetic operations with a mix of `Number` and `BigInt` operands. You'll need to determine the domain in which the operation should be done by explicitly converting either of the operands. Moreover, for compatibility reasons, you are not allowed to use the unary plus (`+`)

operator on a `BigInt`.

What do you think? Do you find `BigInt` useful? Let us know in the comments!

*(dm, yk, il)*

*Explore more on*

As with other primitive types, a `BigInt` can be created using a constructor function. The argument passed to `BigInt()` is automatically converted to a `BigInt`, if possible:

```
BigInt("10");    // → 10n
BigInt(10);      // → 10n
BigInt(true);    // → 1n
```

Data types and values that cannot be converted throw an exception:

```
BigInt(10.2);     // → RangeError
BigInt(null);     // → TypeError
BigInt("abc");    // → SyntaxError
```