BigInt: arbitrary-precision integers in JavaScript

By Mathias Bynens
V8 JavaScript whisperer

`BigInt`s are a new numeric primitive in JavaScript that can represent integers with arbitrary precision. With **`BigInt`**s, you can safely store and operate on large integers even beyond the safe integer limit for **`Number`**s. This article walks through some use cases and explains the new functionality in Chrome 67 by comparing **`BigInt`**s to **`Number`**s in JavaScript.

Use cases

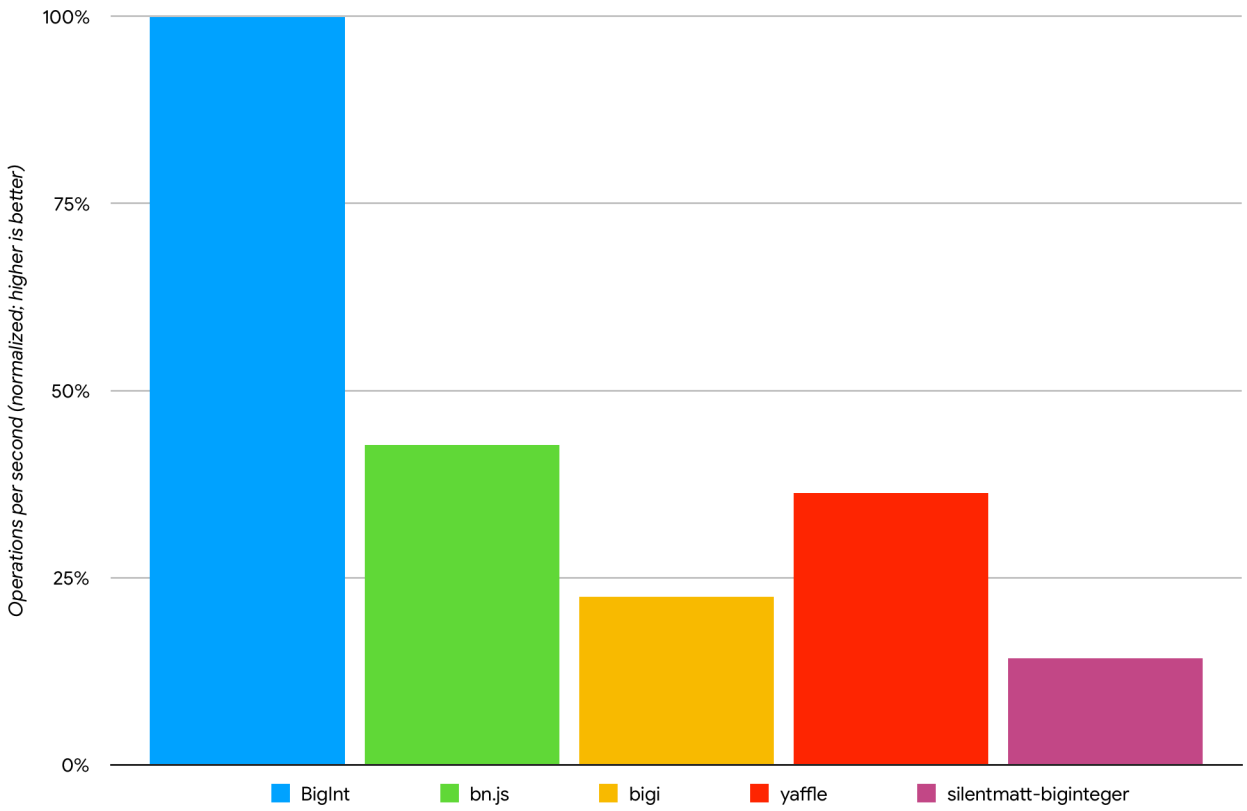
Arbitrary-precision integers unlock lots of new use cases for JavaScript.

`BigInt`s make it possible to correctly perform integer arithmetic without overflowing. That by itself enables countless new possibilities. Mathematical operations on large numbers are commonly used in financial technology, for example.

Large integer IDs and high-accuracy timestamps cannot safely be represented as **`Number`**s in JavaScript. This often leads to real-world bugs, and causes JavaScript developers to represent them as strings instead. With **`BigInt`**, this data can now be represented as numeric values.

`BigInt` could form the basis of an eventual **`BigDecimal`** implementation. This would be useful to represent sums of money with decimal precision, and to accurately operate on them (a.k.a. the `0.10 + 0.20 !== 0.30` problem).

Previously, JavaScript applications with any of these use cases had to resort to userland libraries that emulate **`BigInt`**-like functionality. When **`BigInt`** becomes widely available, such applications can drop these run-time dependencies in favor of native **`BigInt`**s. This helps reduce load time, parse time, and compile time, and on top of all that offers significant run-time performance improvements.



The native **`BigInt`** implementation in Chrome performs better than popular userland libraries.

"Polyfilling" **`BigInt`**s requires a run-time library that implements similar functionality, as well as a transpilation step to turn the new syntax into a call to the library's API. Babel currently supports parsing **`BigInt`** literals through a plugin, but doesn't transpile them. As such, we don't expect **`BigInt`**s to be used in production sites that require broad cross-browser compatibility just yet. It's still early days, but now that the functionality is starting to ship in browsers, you can start to experiment with **`BigInt`**s. Expect wider **`BigInt`** support soon.

The status quo: **`Number`**

`Number`s in JavaScript are represented as double-precision floats. This means they have limited precision. The **`Number.MAX_SAFE_INTEGER`** constant gives the greatest possible integer that can safely be incremented. Its value is `2**53-1`.

```
const max = Number.MAX_SAFE_INTEGER;
// → 9_007_199_254_740_991
```

Note: For readability, I'm grouping the digits in this large number per thousand, using underscores as separators. The numeric literal separators proposal enables exactly that for common JavaScript numeric

literals.

Incrementing it once gives the expected result:

```
max + 1;
// → 9_007_199_254_740_992 ✅
```

But if we increment it a second time, the result is no longer exactly representable as a JavaScript `Number`:

```
max + 2;
// → 9_007_199_254_740_992 ❌
```

Note how `max + 1` produces the same result as `max + 2`. Whenever we get this particular value in JavaScript, there is no way to tell whether it's accurate or not. Any calculation on integers outside the safe integer range (i.e. from `Number.MIN_SAFE_INTEGER` to `Number.MAX_SAFE_INTEGER`) potentially loses precision. For this reason, we can only rely on numeric integer values within the safe range.

The new hotness: `BigInt`

`BigInt`s are a new numeric primitive in JavaScript that can represent integers with arbitrary precision. With `BigInt`s, you can safely store and operate on large integers even beyond the safe integer limit for `Number`s.

To create a `BigInt`, add the `n` suffix to any integer literal. For example, `123` becomes `123n`. The global `BigInt(number)` function can be used to convert a `Number` into a `BigInt`. In other words, `BigInt(123) === 123n`. Let's use these two techniques to solve the problem we were having earlier:

```
BigInt(Number.MAX_SAFE_INTEGER) + 2n;
// → 9_007_199_254_740_993n ✅
```

Here's another example, where we're multiplying two `Number`s:

```
1234567890123456789 * 123;
// → 151851850485185200000 ❌
```

Looking at the least significant digits, `9` and `3`, we know that the result of the multiplication should end in `7` (because `9 * 3 === 27`). However, the result ends in a bunch of zeroes. That can't be right! Let's try again with `BigInt`s instead:

```
1234567890123456789n * 123n;
// → 151851850485185185047n ✅
```

This time we get the correct result.

The safe integer limits for **`Number`**s don't apply to **`BigInt`**s. Therefore, with **`BigInt`** we can perform correct integer arithmetic without having to worry about losing precision.

A new primitive

`BigInt`s are a new primitive in the JavaScript language. As such, they get their own type that can be detected using the **`typeof`** operator:

```
typeof 123;
// → 'number'
typeof 123n;
// → 'bigint'
```

Because **`BigInt`**s are a separate type, a **`BigInt`** is never strictly equal to a **`Number`**, e.g. **`42n !== 42`**. To compare a **`BigInt`** to a **`Number`**, convert one of them into the other's type before doing the comparison or use abstract equality (**`==`**):

```
42n === BigInt(42);
// → true
42n == 42;
// → true
```

When coerced into a boolean (which happens when using **`if`**, **`&&`**, **`||`**, or **`Boolean(int)`**, for example), **`BigInt`**s follow the same logic as **`Number`**s.

```
if (0n) {
  console.log('if');
} else {
  console.log('else');
}
// → logs 'else', because `0n` is falsy.
```

Operators

`BigInt`s support the most common operators. Binary **`+`**, **`-`**, **`*`**, and **`**`** all work as expected. **`/`** and **`%`** work, and round towards zero as needed. Bitwise operations **`|`**, **`&`**, **`<<`**, **`>>`**, and **`^`** perform

bitwise arithmetic assuming a two's complement representation for negative values, just like they do for **Numbers**.

```
(7 + 6 - 5) * 4 ** 3 / 2 % 3;
// → 1
(7n + 6n - 5n) * 4n ** 3n / 2n % 3n;
// → 1n
```

Unary `-` can be used to denote a negative **BigInt** value, e.g. **-42n**. Unary **+** is *not* supported because it would break asm.js code which expects **+x** to always produce either a **Number** or an exception.

One gotcha is that it's not allowed to mix operations between **BigInts** and **Numbers**. This is a good thing, because any implicit coercion could lose information. Consider this example:

```
BigInt(Number.MAX_SAFE_INTEGER) + 2.5;
// → ?? 🤔
```

What should the result be? There is no good answer here. **BigInts** can't represent fractions, and **Numbers** can't represent **BigInts** beyond the safe integer limit. For that reason, mixing operations between **BigInts** and **Numbers** results in a **TypeError** exception.

The only exception to this rule are comparison operators such as `===` (as discussed earlier), `<`, and `>=` – because they return booleans, there is no risk of precision loss.

```
1 + 1n;
// → TypeError
123 < 124n;
// → true
```

Note: Because **BigInts** and **Numbers** generally don't mix, please avoid overloading or magically "upgrading" your existing code to use **BigInts** instead of **Numbers**. Decide which of these two domains to operate in, and then stick to it. For *new* APIs that operate on potentially large integers, **BigInt** is the best choice. **Numbers** still make sense for integer values that are known to be in the safe integer range.

Another thing to note is that the `>>>` operator, which performs an unsigned right shift, does not make sense for **BigInts** since they're always signed. For this reason, `>>>` does not work for **BigInts**.

API

Several new **BigInt**-specific APIs are available.

The global **`BigInt`** constructor is similar to the **`Number`** constructor: it converts its argument into a **`BigInt`** (as mentioned earlier). If the conversion fails, it throws a **`SyntaxError`** or **`RangeError`** exception.

```
BigInt(123);
// → 123n
BigInt(1.5);
// → RangeError
BigInt('1.5');
// → SyntaxError
```

Two library functions enable wrapping **`BigInt`** values as either signed or unsigned integers, limited to a specific number of bits. **`BigInt.asIntN(width, value)`** wraps a **`BigInt`** value to a **`width`**-digit binary signed integer, and **`BigInt.asUintN(width, value)`** wraps a **`BigInt`** value to a **`width`**-digit binary unsigned integer. If you're doing 64-bit arithmetic for example, you can use these APIs to stay within the appropriate range:

```
// Highest possible BigInt value that can be represented as a
// signed 64-bit integer.
const max = 2n ** (64n - 1n) - 1n;
BigInt.asIntN(64, max);
→ 9223372036854775807n
BigInt.asIntN(64, max + 1n);
// → -9223372036854775808n
//   ^ negative because of overflow
```

Note how overflow occurs as soon as we pass a **`BigInt`** value exceeding the 64-bit integer range (i.e. 63 bits for the absolute numeric value + 1 bit for the sign).

`BigInt`s make it possible to accurately represent 64-bit signed and unsigned integers, which are commonly used in other programming languages. Two new typed array flavors, **`BigInt64Array`** and **`BigUint64Array`**, make it easier to efficiently represent and operate on lists of such values:

```
const view = new BigInt64Array(4);
// → [0n, 0n, 0n, 0n]
view.length;
// → 4
view[0];
// → 0n
view[0] = 42n;
view[0];
// → 42n
```

The **`BigInt64Array`** flavor ensures that its values remain within the signed 64-bit limit.

```
// Highest possible BigInt value that can be represented as a
// signed 64-bit integer.
const max = 2n ** (64n - 1n) - 1n;
view[0] = max;
view[0];
// → 9_223_372_036_854_775_807n
view[0] = max + 1n;
view[0];
// → -9_223_372_036_854_775_808n
//   ^ negative because of overflow
```

The **`BigUint64Array`** flavor does the same using the unsigned 64-bit limit instead.

If you're interested in how **`BigInt`**s work behind the scenes (e.g. how they are represented in memory, and how operations on them are performed), read our V8 blog post with implementation details.

Have fun with **`BigInt`**s!

Note: Thanks to Daniel Ehrenberg, the **`BigInt`** proposal champion, for reviewing this article.

Last updated July 12, 2018.