v9.0.0 [API](#gjdgxs) CONSTRUCTOR

[BigNumber](#30j0zll)

[Methods](#1fob9te)

[clone](#3znysh7)

[config](#2et92p0)set

[DECIMAL\_PLACES](#tyjcwt)

[ROUNDING\_MODE](#3dy6vkm)

[EXPONENTIAL\_AT](#1t3h5sf)

[RANGE](#4d34og8)

[CRYPTO](#2s8eyo1)

[MODULO\_MODE](#17dp8vu)

[POW\_PRECISION](#3rdcrjn)

[FORMAT](#26in1rg)

[ALPHABET](#lnxbz9)

[isBigNumber](#35nkun2)

[maximum](#1ksv4uv)max

[minimum](#44sinio)min

[random](#2jxsxqh)

[sum](#z337ya)

[Properties](#3j2qqm3)

[ROUND\_UP](#1y810tw)

[ROUND\_DOWN](#4i7ojhp)

[ROUND\_CEIL](#2xcytpi)

[ROUND\_FLOOR](#1ci93xb)

[ROUND\_HALF\_UP](#3whwml4)

[ROUND\_HALF\_DOWN](#2bn6wsx)

[ROUND\_HALF\_EVEN](#qsh70q)

[ROUND\_HALF\_CEIL](#3as4poj)

[ROUND\_HALF\_FLOOR](#1pxezwc)

[DEBUG](#49x2ik5)

INSTANCE [Methods](#2p2csry)

[absoluteValue](#147n2zr) abs

[comparedTo](#3o7alnk)

[decimalPlaces](#23ckvvd) dp

[dividedBy](#ihv636) div

[dividedToIntegerBy](#32hioqz) idiv

[exponentiatedBy](#1hmsyys) pow

[integerValue](#41mghml)

[isEqualTo](#2grqrue) eq

[isFinite](#vx1227)

[isGreaterThan](#3fwokq0) gt

[isGreaterThanOrEqualTo](#1v1yuxt)gte

[isInteger](#4f1mdlm)

[isLessThan](#2u6wntf) lt

[isLessThanOrEqualTo](#19c6y18) lte

[isNaN](#3tbugp1)

[isNegative](#28h4qwu)

[isPositive](#nmf14n)

[isZero](#37m2jsg)

[minus](#1mrcu09)

[modulo](#46r0co2) mod

[multipliedBy](#2lwamvv) times

[negated](#111kx3o)

[plus](#3l18frh)

[precision](#206ipza) sd

[shiftedBy](#4k668n3)

[squareRoot](#2zbgiuw) sqrt

[toExponential](#1egqt2p)

[toFixed](#3ygebqi)

[toFormat](#2dlolyb)

[toFraction](#sqyw64)

[toJSON](#3cqmetx)

[toNumber](#1rvwp1q)

[toPrecision](#4bvk7pj)

[toString](#2r0uhxc)

[valueOf](#1664s55)

[Properties](#3q5sasy)

[c: coefficient](#25b2l0r)

[e: exponent](#kgcv8k)

[s: sign](#34g0dwd)

[Zero, NaN & Infinity](#1jlao46) [Errors](#43ky6rz) [Type coercion](#2iq8gzs) [FAQ](#xvir7l)

bignumber.js

A JavaScript library for arbitrary-precision arithmetic.

[Hosted on GitHub](https://github.com/MikeMcl/bignumber.js).

## API

See the [README](https://github.com/MikeMcl/bignumber.js) on GitHub for a quick-start introduction.

In all examples below, var and semicolons are not shown, and if a commented-out value is in quotes it means toString has been called on the preceding expression.

### CONSTRUCTOR

##### BigNumberBigNumber(n [, base]) ***⇒ BigNumber***

n: *number|string|BigNumber*

base: *number*: integer, 2 to 36 inclusive. (See [ALPHABET](#lnxbz9) to extend this range).

Returns a new instance of a BigNumber object with value n, where n is a numeric value in the specified base, or base 10 if base is omitted or is null or undefined.

x = new BigNumber(123.4567) // '123.4567'  
// 'new' is optional  
y = BigNumber(x) // '123.4567'

If n is a base 10 value it can be in normal (fixed-point) or exponential notation. Values in other bases must be in normal notation. Values in any base can have fraction digits, i.e. digits after the decimal point.

new BigNumber(43210) // '43210'  
new BigNumber('4.321e+4') // '43210'  
new BigNumber('-735.0918e-430') // '-7.350918e-428'  
new BigNumber('123412421.234324', 5) // '607236.557696'

Signed 0, signed Infinity and NaN are supported.

new BigNumber('-Infinity') // '-Infinity'  
new BigNumber(NaN) // 'NaN'  
new BigNumber(-0) // '0'  
new BigNumber('.5') // '0.5'  
new BigNumber('+2') // '2'

String values in hexadecimal literal form, e.g. '0xff', are valid, as are string values with the octal and binary prefixs '0o' and '0b'. String values in octal literal form without the prefix will be interpreted as decimals, e.g. '011' is interpreted as 11, not 9.

new BigNumber(-10110100.1, 2) // '-180.5'  
new BigNumber('-0b10110100.1') // '-180.5'  
new BigNumber('ff.8', 16) // '255.5'  
new BigNumber('0xff.8') // '255.5'

If a base is specified, n is rounded according to the current [DECIMAL\_PLACES](#tyjcwt) and [ROUNDING\_MODE](#3dy6vkm) settings. *This includes base 10 so don't include a base parameter for decimal values unless this behaviour is wanted.*

BigNumber.config({ DECIMAL\_PLACES: 5 })  
new BigNumber(1.23456789) // '1.23456789'  
new BigNumber(1.23456789, 10) // '1.23457'

An error is thrown if base is invalid. See [Errors](#43ky6rz).

There is no limit to the number of digits of a value of type *string* (other than that of JavaScript's maximum array size). See [RANGE](#4d34og8) to set the maximum and minimum possible exponent value of a BigNumber.

new BigNumber('5032485723458348569331745.33434346346912144534543')  
new BigNumber('4.321e10000000')

BigNumber NaN is returned if n is invalid (unless BigNumber.DEBUG is true, see below).

new BigNumber('.1\*') // 'NaN'  
new BigNumber('blurgh') // 'NaN'  
new BigNumber(9, 2) // 'NaN'

To aid in debugging, if BigNumber.DEBUG is true then an error will be thrown on an invalid n. An error will also be thrown if n is of type *number* with more than 15 significant digits, as calling [toString](#2r0uhxc) or [valueOf](#1664s55) on these numbers may not result in the intended value.

console.log(823456789123456.3) // 823456789123456.2  
new BigNumber(823456789123456.3) // '823456789123456.2'  
BigNumber.DEBUG = true  
// '[BigNumber Error] Number primitive has more than 15 significant digits'  
new BigNumber(823456789123456.3)  
// '[BigNumber Error] Not a base 2 number'  
new BigNumber(9, 2)

A BigNumber can also be created from an object literal. Use [isBigNumber](#35nkun2) to check that it is well-formed.

new BigNumber({ s: 1, e: 2, c: [ 777, 12300000000000 ], \_isBigNumber: true }) // '777.123'

#### Methods

The static methods of a BigNumber constructor.

##### clone .clone([object]) ***⇒ BigNumber constructor***

object: *object*

Returns a new independent BigNumber constructor with configuration as described by object (see [config](#2et92p0)), or with the default configuration if object is null or undefined.

Throws if object is not an object. See [Errors](#43ky6rz).

BigNumber.config({ DECIMAL\_PLACES: 5 })  
BN = BigNumber.clone({ DECIMAL\_PLACES: 9 })  
  
x = new BigNumber(1)  
y = new BN(1)  
  
x.div(3) // 0.33333  
y.div(3) // 0.333333333  
  
// BN = BigNumber.clone({ DECIMAL\_PLACES: 9 }) is equivalent to:  
BN = BigNumber.clone()  
BN.config({ DECIMAL\_PLACES: 9 })

##### configset([object]) ***⇒ object***

object: *object*: an object that contains some or all of the following properties.

Configures the settings for this particular BigNumber constructor.

DECIMAL\_PLACES *number*: integer, 0 to 1e+9 inclusive

Default value: 20 The maximum number of decimal places of the results of operations involving division, i.e. division, square root and base conversion operations, and power operations with negative exponents.

BigNumber.config({ DECIMAL\_PLACES: 5 })  
BigNumber.set({ DECIMAL\_PLACES: 5 }) // equivalent

ROUNDING\_MODE *number*: integer, 0 to 8 inclusive

Default value: 4 [(ROUND\_HALF\_UP)](#3whwml4) The rounding mode used in the above operations and the default rounding mode of [decimalPlaces](#23ckvvd), [precision](#206ipza), [toExponential](#1egqt2p), [toFixed](#3ygebqi), [toFormat](#2dlolyb) and [toPrecision](#4bvk7pj). The modes are available as enumerated properties of the BigNumber constructor.

BigNumber.config({ ROUNDING\_MODE: 0 })  
BigNumber.set({ ROUNDING\_MODE: BigNumber.ROUND\_UP }) // equivalent

EXPONENTIAL\_AT *number*: integer, magnitude 0 to 1e+9 inclusive, or

*number*[]: [ integer -1e+9 to 0 inclusive, integer 0 to 1e+9 inclusive ]

Default value: [-7, 20] The exponent value(s) at which toString returns exponential notation. If a single number is assigned, the value is the exponent magnitude.

If an array of two numbers is assigned then the first number is the negative exponent value at and beneath which exponential notation is used, and the second number is the positive exponent value at and above which the same. For example, to emulate JavaScript numbers in terms of the exponent values at which they begin to use exponential notation, use [-7, 20].

BigNumber.config({ EXPONENTIAL\_AT: 2 })  
new BigNumber(12.3) // '12.3' e is only 1  
new BigNumber(123) // '1.23e+2'  
new BigNumber(0.123) // '0.123' e is only -1  
new BigNumber(0.0123) // '1.23e-2'  
  
BigNumber.config({ EXPONENTIAL\_AT: [-7, 20] })  
new BigNumber(123456789) // '123456789' e is only 8  
new BigNumber(0.000000123) // '1.23e-7'  
  
// Almost never return exponential notation:  
BigNumber.config({ EXPONENTIAL\_AT: 1e+9 })  
  
// Always return exponential notation:  
BigNumber.config({ EXPONENTIAL\_AT: 0 })

Regardless of the value of EXPONENTIAL\_AT, the toFixed method will always return a value in normal notation and the toExponential method will always return a value in exponential form. Calling toString with a base argument, e.g. toString(10), will also always return normal notation. RANGE *number*: integer, magnitude 1 to 1e+9 inclusive, or

*number*[]: [ integer -1e+9 to -1 inclusive, integer 1 to 1e+9 inclusive ]

Default value: [-1e+9, 1e+9] The exponent value(s) beyond which overflow to Infinity and underflow to zero occurs. If a single number is assigned, it is the maximum exponent magnitude: values wth a positive exponent of greater magnitude become Infinity and those with a negative exponent of greater magnitude become zero. If an array of two numbers is assigned then the first number is the negative exponent limit and the second number is the positive exponent limit. For example, to emulate JavaScript numbers in terms of the exponent values at which they become zero and Infinity, use [-324, 308].

BigNumber.config({ RANGE: 500 })  
BigNumber.config().RANGE // [ -500, 500 ]  
new BigNumber('9.999e499') // '9.999e+499'  
new BigNumber('1e500') // 'Infinity'  
new BigNumber('1e-499') // '1e-499'  
new BigNumber('1e-500') // '0'  
  
BigNumber.config({ RANGE: [-3, 4] })  
new BigNumber(99999) // '99999' e is only 4  
new BigNumber(100000) // 'Infinity' e is 5  
new BigNumber(0.001) // '0.01' e is only -3  
new BigNumber(0.0001) // '0' e is -4

The largest possible magnitude of a finite BigNumber is 9.999...e+1000000000.

The smallest possible magnitude of a non-zero BigNumber is 1e-1000000000. CRYPTO *boolean*: true or false.

Default value: false The value that determines whether cryptographically-secure pseudo-random number generation is used. If CRYPTO is set to true then the [random](#2jxsxqh) method will generate random digits using crypto.getRandomValues in browsers that support it, or crypto.randomBytes if using Node.js. If neither function is supported by the host environment then attempting to set CRYPTO to true will fail and an exception will be thrown. If CRYPTO is false then the source of randomness used will be Math.random (which is assumed to generate at least 30 bits of randomness). See [random](#2jxsxqh).

// Node.js  
global.crypto = require('crypto')  
  
BigNumber.config({ CRYPTO: true })  
BigNumber.config().CRYPTO // true  
BigNumber.random() // 0.54340758610486147524

MODULO\_MODE *number*: integer, 0 to 9 inclusive

Default value: 1 ([ROUND\_DOWN](#4i7ojhp)) The modulo mode used when calculating the modulus: a mod n. The quotient, q = a / n, is calculated according to the [ROUNDING\_MODE](#3dy6vkm) that corresponds to the chosen MODULO\_MODE. The remainder, r, is calculated as: r = a - n \* q. The modes that are most commonly used for the modulus/remainder operation are shown in the following table. Although the other rounding modes can be used, they may not give useful results.

|  |  |  |
| --- | --- | --- |
| Property | Value | Description |
| ROUND\_UP | 0 | The remainder is positive if the dividend is negative, otherwise it is negative. |
| ROUND\_DOWN | 1 | The remainder has the same sign as the dividend.  This uses 'truncating division' and matches the behaviour of JavaScript's remainder operator %. |
| ROUND\_FLOOR | 3 | The remainder has the same sign as the divisor.  This matches Python's % operator. |
| ROUND\_HALF\_EVEN | 6 | The *IEEE 754* remainder function. |
| EUCLID | 9 | The remainder is always positive. Euclidian division:  q = sign(n) \* floor(a / abs(n)) |

The rounding/modulo modes are available as enumerated properties of the BigNumber constructor. See [modulo](#46r0co2).

BigNumber.config({ MODULO\_MODE: BigNumber.EUCLID })  
BigNumber.config({ MODULO\_MODE: 9 }) // equivalent

POW\_PRECISION *number*: integer, 0 to 1e+9 inclusive.

Default value: 0 The *maximum* precision, i.e. number of significant digits, of the result of the power operation (unless a modulus is specified). If set to 0, the number of significant digits will not be limited. See [exponentiatedBy](#1hmsyys).

BigNumber.config({ POW\_PRECISION: 100 })

FORMAT *object* The FORMAT object configures the format of the string returned by the [toFormat](#2dlolyb) method. The example below shows the properties of the FORMAT object that are recognised, and their default values. Unlike the other configuration properties, the values of the properties of the FORMAT object will not be checked for validity. The existing FORMAT object will simply be replaced by the object that is passed in. The object can include any number of the properties shown below. See [toFormat](#2dlolyb) for examples of usage.

BigNumber.config({  
 FORMAT: {  
 // string to prepend  
 prefix: '',  
 // decimal separator  
 decimalSeparator: '.',  
 // grouping separator of the integer part  
 groupSeparator: ',',  
 // primary grouping size of the integer part  
 groupSize: 3,  
 // secondary grouping size of the integer part  
 secondaryGroupSize: 0,  
 // grouping separator of the fraction part  
 fractionGroupSeparator: ' ',  
 // grouping size of the fraction part  
 fractionGroupSize: 0,  
 // string to append  
 suffix: ''  
 }  
});

ALPHABET *string*

Default value: '0123456789abcdefghijklmnopqrstuvwxyz' The alphabet used for base conversion. The length of the alphabet corresponds to the maximum value of the base argument that can be passed to the [BigNumber](#30j0zll) constructor or [toString](#2r0uhxc). There is no maximum length for the alphabet, but it must be at least 2 characters long, and it must not contain whitespace or a repeated character, or the sign indicators '+' and '-', or the decimal separator '.'.

// duodecimal (base 12)  
BigNumber.config({ ALPHABET: '0123456789TE' })  
x = new BigNumber('T', 12)  
x.toString() // '10'  
x.toString(12) // 'T'

Returns an object with the above properties and their current values.

Throws if object is not an object, or if an invalid value is assigned to one or more of the above properties. See [Errors](#43ky6rz).

BigNumber.config({  
 DECIMAL\_PLACES: 40,  
 ROUNDING\_MODE: BigNumber.ROUND\_HALF\_CEIL,  
 EXPONENTIAL\_AT: [-10, 20],  
 RANGE: [-500, 500],  
 CRYPTO: true,  
 MODULO\_MODE: BigNumber.ROUND\_FLOOR,  
 POW\_PRECISION: 80,  
 FORMAT: {  
 groupSize: 3,  
 groupSeparator: ' ',  
 decimalSeparator: ','  
 },  
 ALPHABET: '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ$\_'  
});  
  
obj = BigNumber.config();  
obj.DECIMAL\_PLACES // 40  
obj.RANGE // [-500, 500]

##### isBigNumber.isBigNumber(value) ***⇒ boolean***

value: *any*

Returns true if value is a BigNumber instance, otherwise returns false.

x = 42  
y = new BigNumber(x)  
  
BigNumber.isBigNumber(x) // false  
y instanceof BigNumber // true  
BigNumber.isBigNumber(y) // true  
  
BN = BigNumber.clone();  
z = new BN(x)  
z instanceof BigNumber // false  
BigNumber.isBigNumber(z) // true

If value is a BigNumber instance and BigNumber.DEBUG is true, then this method will also check if value is well-formed, and throw if it is not. See [Errors](#43ky6rz).

The check can be useful if creating a BigNumber from an object literal. See [BigNumber](#30j0zll).

x = new BigNumber(10)  
  
// Change x.c to an illegitimate value.  
x.c = NaN  
  
BigNumber.DEBUG = false  
  
// No error.  
BigNumber.isBigNumber(x) // true  
  
BigNumber.DEBUG = true  
  
// Error.  
BigNumber.isBigNumber(x) // '[BigNumber Error] Invalid BigNumber'

##### maximum.max(n...) ***⇒ BigNumber***

n: *number|string|BigNumber*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the maximum of the arguments.

The return value is always exact and unrounded.

x = new BigNumber('3257869345.0378653')  
BigNumber.maximum(4e9, x, '123456789.9') // '4000000000'  
  
arr = [12, '13', new BigNumber(14)]  
BigNumber.max.apply(null, arr) // '14'

##### minimum.min(n...) ***⇒ BigNumber***

n: *number|string|BigNumber*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the minimum of the arguments.

The return value is always exact and unrounded.

x = new BigNumber('3257869345.0378653')  
BigNumber.minimum(4e9, x, '123456789.9') // '123456789.9'  
  
arr = [2, new BigNumber(-14), '-15.9999', -12]  
BigNumber.min.apply(null, arr) // '-15.9999'

##### random.random([dp]) ***⇒ BigNumber***

dp: *number*: integer, 0 to 1e+9 inclusive

Returns a new BigNumber with a pseudo-random value equal to or greater than 0 and less than 1.

The return value will have dp decimal places (or less if trailing zeros are produced).

If dp is omitted then the number of decimal places will default to the current [DECIMAL\_PLACES](#tyjcwt) setting.

Depending on the value of this BigNumber constructor's [CRYPTO](#2s8eyo1) setting and the support for the crypto object in the host environment, the random digits of the return value are generated by either Math.random (fastest), crypto.getRandomValues (Web Cryptography API in recent browsers) or crypto.randomBytes (Node.js).

To be able to set [CRYPTO](#2s8eyo1) to true when using Node.js, the crypto object must be available globally:

global.crypto = require('crypto')

If [CRYPTO](#2s8eyo1) is true, i.e. one of the crypto methods is to be used, the value of a returned BigNumber should be cryptographically-secure and statistically indistinguishable from a random value.

Throws if dp is invalid. See [Errors](#43ky6rz).

BigNumber.config({ DECIMAL\_PLACES: 10 })  
BigNumber.random() // '0.4117936847'  
BigNumber.random(20) // '0.78193327636914089009'

##### sum.sum(n...) ***⇒ BigNumber***

n: *number|string|BigNumber*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the sum of the arguments.

The return value is always exact and unrounded.

x = new BigNumber('3257869345.0378653')  
BigNumber.sum(4e9, x, '123456789.9') // '7381326134.9378653'  
  
arr = [2, new BigNumber(14), '15.9999', 12]  
BigNumber.sum.apply(null, arr) // '43.9999'

#### Properties

The library's enumerated rounding modes are stored as properties of the constructor.

(They are not referenced internally by the library itself.)

Rounding modes 0 to 6 (inclusive) are the same as those of Java's BigDecimal class.

|  |  |  |
| --- | --- | --- |
| Property | Value | Description |
| ROUND\_UP | 0 | Rounds away from zero |
| ROUND\_DOWN | 1 | Rounds towards zero |
| ROUND\_CEIL | 2 | Rounds towards Infinity |
| ROUND\_FLOOR | 3 | Rounds towards -Infinity |
| ROUND\_HALF\_UP | 4 | Rounds towards nearest neighbour.  If equidistant, rounds away from zero |
| ROUND\_HALF\_DOWN | 5 | Rounds towards nearest neighbour.  If equidistant, rounds towards zero |
| ROUND\_HALF\_EVEN | 6 | Rounds towards nearest neighbour.  If equidistant, rounds towards even neighbour |
| ROUND\_HALF\_CEIL | 7 | Rounds towards nearest neighbour.  If equidistant, rounds towards Infinity |
| ROUND\_HALF\_FLOOR | 8 | Rounds towards nearest neighbour.  If equidistant, rounds towards -Infinity |

BigNumber.config({ ROUNDING\_MODE: BigNumber.ROUND\_CEIL })  
BigNumber.config({ ROUNDING\_MODE: 2 }) // equivalent

##### DEBUG

*undefined|false|true*

If BigNumber.DEBUG is set true then an error will be thrown if this [BigNumber](#30j0zll) constructor receives an invalid value, such as a value of type *number* with more than 15 significant digits. See [BigNumber](#30j0zll).

An error will also be thrown if the [isBigNumber](#35nkun2) method receives a BigNumber that is not well-formed. See [isBigNumber](#35nkun2).

BigNumber.DEBUG = true

### INSTANCE

#### Methods

The methods inherited by a BigNumber instance from its constructor's prototype object.

A BigNumber is immutable in the sense that it is not changed by its methods.

The treatment of ±0, ±Infinity and NaN is consistent with how JavaScript treats these values.

Many method names have a shorter alias.

##### absoluteValue.abs() ***⇒ BigNumber***

Returns a BigNumber whose value is the absolute value, i.e. the magnitude, of the value of this BigNumber.

The return value is always exact and unrounded.

x = new BigNumber(-0.8)  
y = x.absoluteValue() // '0.8'  
z = y.abs() // '0.8'

##### comparedTo.comparedTo(n [, base]) ***⇒ number***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

|  |  |
| --- | --- |
| Returns |  |
| 1 | If the value of this BigNumber is greater than the value of n |
| -1 | If the value of this BigNumber is less than the value of n |
| 0 | If this BigNumber and n have the same value |
| null | If the value of either this BigNumber or n is NaN |

x = new BigNumber(Infinity)  
y = new BigNumber(5)  
x.comparedTo(y) // 1  
x.comparedTo(x.minus(1)) // 0  
y.comparedTo(NaN) // null  
y.comparedTo('110', 2) // -1

##### decimalPlaces.dp([dp [, rm]]) ***⇒ BigNumber|number***

dp: *number*: integer, 0 to 1e+9 inclusive

rm: *number*: integer, 0 to 8 inclusive

If dp is a number, returns a BigNumber whose value is the value of this BigNumber rounded by rounding mode rm to a maximum of dp decimal places.

If dp is omitted, or is null or undefined, the return value is the number of decimal places of the value of this BigNumber, or null if the value of this BigNumber is ±Infinity or NaN.

If rm is omitted, or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) is used.

Throws if dp or rm is invalid. See [Errors](#43ky6rz).

x = new BigNumber(1234.56)  
x.decimalPlaces(1) // '1234.6'  
x.dp() // 2  
x.decimalPlaces(2) // '1234.56'  
x.dp(10) // '1234.56'  
x.decimalPlaces(0, 1) // '1234'  
x.dp(0, 6) // '1235'  
x.decimalPlaces(1, 1) // '1234.5'  
x.dp(1, BigNumber.ROUND\_HALF\_EVEN) // '1234.6'  
x // '1234.56'  
y = new BigNumber('9.9e-101')  
y.dp() // 102

##### dividedBy.div(n [, base]) ***⇒ BigNumber***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the value of this BigNumber divided by n, rounded according to the current [DECIMAL\_PLACES](#tyjcwt) and [ROUNDING\_MODE](#3dy6vkm) settings.

x = new BigNumber(355)  
y = new BigNumber(113)  
x.dividedBy(y) // '3.14159292035398230088'  
x.div(5) // '71'  
x.div(47, 16) // '5'

##### dividedToIntegerBy.idiv(n [, base]) ⇒ ***BigNumber***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the integer part of dividing the value of this BigNumber by n.

x = new BigNumber(5)  
y = new BigNumber(3)  
x.dividedToIntegerBy(y) // '1'  
x.idiv(0.7) // '7'  
x.idiv('0.f', 16) // '5'

##### exponentiatedBy.pow(n [, m]) ***⇒ BigNumber***

n: *number|string|BigNumber*: integer

m: *number|string|BigNumber*

Returns a BigNumber whose value is the value of this BigNumber exponentiated by n, i.e. raised to the power n, and optionally modulo a modulus m.

Throws if n is not an integer. See [Errors](#43ky6rz).

If n is negative the result is rounded according to the current [DECIMAL\_PLACES](#tyjcwt) and [ROUNDING\_MODE](#3dy6vkm) settings.

As the number of digits of the result of the power operation can grow so large so quickly, e.g. 123.45610000 has over 50000 digits, the number of significant digits calculated is limited to the value of the [POW\_PRECISION](#3rdcrjn) setting (unless a modulus m is specified).

By default [POW\_PRECISION](#3rdcrjn) is set to 0. This means that an unlimited number of significant digits will be calculated, and that the method's performance will decrease dramatically for larger exponents.

If m is specified and the value of m, n and this BigNumber are integers, and n is positive, then a fast modular exponentiation algorithm is used, otherwise the operation will be performed as x.exponentiatedBy(n).modulo(m) with a [POW\_PRECISION](#3rdcrjn) of 0.

Math.pow(0.7, 2) // 0.48999999999999994  
x = new BigNumber(0.7)  
x.exponentiatedBy(2) // '0.49'  
BigNumber(3).pow(-2) // '0.11111111111111111111'

##### integerValue.integerValue([rm]) ***⇒ BigNumber***

rm: *number*: integer, 0 to 8 inclusive

Returns a BigNumber whose value is the value of this BigNumber rounded to an integer using rounding mode rm.

If rm is omitted, or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) is used.

Throws if rm is invalid. See [Errors](#43ky6rz).

x = new BigNumber(123.456)  
x.integerValue() // '123'  
x.integerValue(BigNumber.ROUND\_CEIL) // '124'  
y = new BigNumber(-12.7)  
y.integerValue() // '-13'  
y.integerValue(BigNumber.ROUND\_DOWN) // '-12'

The following is an example of how to add a prototype method that emulates JavaScript's Math.round function. Math.ceil, Math.floor and Math.trunc can be emulated in the same way with BigNumber.ROUND\_CEIL, BigNumber.ROUND\_FLOOR and BigNumber.ROUND\_DOWN respectively.

BigNumber.prototype.round = function (n) {  
 return n.integerValue(BigNumber.ROUND\_HALF\_CEIL);  
};  
x.round() // '123'

##### isEqualTo.eq(n [, base]) ***⇒ boolean***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns true if the value of this BigNumber is equal to the value of n, otherwise returns false.

As with JavaScript, NaN does not equal NaN.

Note: This method uses the [comparedTo](#3o7alnk) method internally.

0 === 1e-324 // true  
x = new BigNumber(0)  
x.isEqualTo('1e-324') // false  
BigNumber(-0).eq(x) // true ( -0 === 0 )  
BigNumber(255).eq('ff', 16) // true  
  
y = new BigNumber(NaN)  
y.isEqualTo(NaN) // false

##### isFinite.isFinite() ***⇒ boolean***

Returns true if the value of this BigNumber is a finite number, otherwise returns false.

The only possible non-finite values of a BigNumber are NaN, Infinity and -Infinity.

x = new BigNumber(1)  
x.isFinite() // true  
y = new BigNumber(Infinity)  
y.isFinite() // false

Note: The native method isFinite() can be used if n <= Number.MAX\_VALUE.

##### isGreaterThan.gt(n [, base]) ***⇒ boolean***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns true if the value of this BigNumber is greater than the value of n, otherwise returns false.

Note: This method uses the [comparedTo](#3o7alnk) method internally.

0.1 > (0.3 - 0.2) // true  
x = new BigNumber(0.1)  
x.isGreaterThan(BigNumber(0.3).minus(0.2)) // false  
BigNumber(0).gt(x) // false  
BigNumber(11, 3).gt(11.1, 2) // true

##### isGreaterThanOrEqualTo.gte(n [, base]) ***⇒ boolean***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns true if the value of this BigNumber is greater than or equal to the value of n, otherwise returns false.

Note: This method uses the [comparedTo](#3o7alnk) method internally.

(0.3 - 0.2) >= 0.1 // false  
x = new BigNumber(0.3).minus(0.2)  
x.isGreaterThanOrEqualTo(0.1) // true  
BigNumber(1).gte(x) // true  
BigNumber(10, 18).gte('i', 36) // true

##### isInteger.isInteger() ***⇒ boolean***

Returns true if the value of this BigNumber is an integer, otherwise returns false.

x = new BigNumber(1)  
x.isInteger() // true  
y = new BigNumber(123.456)  
y.isInteger() // false

##### isLessThan.lt(n [, base]) ***⇒ boolean***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns true if the value of this BigNumber is less than the value of n, otherwise returns false.

Note: This method uses the [comparedTo](#3o7alnk) method internally.

(0.3 - 0.2) < 0.1 // true  
x = new BigNumber(0.3).minus(0.2)  
x.isLessThan(0.1) // false  
BigNumber(0).lt(x) // true  
BigNumber(11.1, 2).lt(11, 3) // true

##### isLessThanOrEqualTo.lte(n [, base]) ***⇒ boolean***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns true if the value of this BigNumber is less than or equal to the value of n, otherwise returns false.

Note: This method uses the [comparedTo](#3o7alnk) method internally.

0.1 <= (0.3 - 0.2) // false  
x = new BigNumber(0.1)  
x.isLessThanOrEqualTo(BigNumber(0.3).minus(0.2)) // true  
BigNumber(-1).lte(x) // true  
BigNumber(10, 18).lte('i', 36) // true

##### isNaN.isNaN() ***⇒ boolean***

Returns true if the value of this BigNumber is NaN, otherwise returns false.

x = new BigNumber(NaN)  
x.isNaN() // true  
y = new BigNumber('Infinity')  
y.isNaN() // false

Note: The native method isNaN() can also be used.

##### isNegative.isNegative() ***⇒ boolean***

Returns true if the sign of this BigNumber is negative, otherwise returns false.

x = new BigNumber(-0)  
x.isNegative() // true  
y = new BigNumber(2)  
y.isNegative() // false

Note: n < 0 can be used if n <= -Number.MIN\_VALUE.

##### isPositive.isPositive() ***⇒ boolean***

Returns true if the sign of this BigNumber is positive, otherwise returns false.

x = new BigNumber(-0)  
x.isPositive() // false  
y = new BigNumber(2)  
y.isPositive() // true

##### isZero.isZero() ***⇒ boolean***

Returns true if the value of this BigNumber is zero or minus zero, otherwise returns false.

x = new BigNumber(-0)  
x.isZero() && x.isNegative() // true  
y = new BigNumber(Infinity)  
y.isZero() // false

Note: n == 0 can be used if n >= Number.MIN\_VALUE.

##### minus.minus(n [, base]) ***⇒ BigNumber***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the value of this BigNumber minus n.

The return value is always exact and unrounded.

0.3 - 0.1 // 0.19999999999999998  
x = new BigNumber(0.3)  
x.minus(0.1) // '0.2'  
x.minus(0.6, 20) // '0'

##### modulo.mod(n [, base]) ***⇒ BigNumber***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the value of this BigNumber modulo n, i.e. the integer remainder of dividing this BigNumber by n.

The value returned, and in particular its sign, is dependent on the value of the [MODULO\_MODE](#17dp8vu) setting of this BigNumber constructor. If it is 1 (default value), the result will have the same sign as this BigNumber, and it will match that of Javascript's % operator (within the limits of double precision) and BigDecimal's remainder method.

The return value is always exact and unrounded.

See [MODULO\_MODE](#17dp8vu) for a description of the other modulo modes.

1 % 0.9 // 0.09999999999999998  
x = new BigNumber(1)  
x.modulo(0.9) // '0.1'  
y = new BigNumber(33)  
y.mod('a', 33) // '3'

##### multipliedBy.times(n [, base]) ***⇒ BigNumber***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the value of this BigNumber multiplied by n.

The return value is always exact and unrounded.

0.6 \* 3 // 1.7999999999999998  
x = new BigNumber(0.6)  
y = x.multipliedBy(3) // '1.8'  
BigNumber('7e+500').times(y) // '1.26e+501'  
x.multipliedBy('-a', 16) // '-6'

##### negated.negated() ***⇒ BigNumber***

Returns a BigNumber whose value is the value of this BigNumber negated, i.e. multiplied by -1.

x = new BigNumber(1.8)  
x.negated() // '-1.8'  
y = new BigNumber(-1.3)  
y.negated() // '1.3'

##### plus.plus(n [, base]) ***⇒ BigNumber***

n: *number|string|BigNumber*

base: *number*

*See* [*BigNumber*](#30j0zll) *for further parameter details.*

Returns a BigNumber whose value is the value of this BigNumber plus n.

The return value is always exact and unrounded.

0.1 + 0.2 // 0.30000000000000004  
x = new BigNumber(0.1)  
y = x.plus(0.2) // '0.3'  
BigNumber(0.7).plus(x).plus(y) // '1.1'  
x.plus('0.1', 8) // '0.225'

##### precision.sd([d [, rm]]) ***⇒ BigNumber|number***

d: *number|boolean*: integer, 1 to 1e+9 inclusive, or true or false

rm: *number*: integer, 0 to 8 inclusive.

If d is a number, returns a BigNumber whose value is the value of this BigNumber rounded to a precision of d significant digits using rounding mode rm.

If d is omitted or is null or undefined, the return value is the number of significant digits of the value of this BigNumber, or null if the value of this BigNumber is ±Infinity or NaN.

If d is true then any trailing zeros of the integer part of a number are counted as significant digits, otherwise they are not.

If rm is omitted or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) will be used.

Throws if d or rm is invalid. See [Errors](#43ky6rz).

x = new BigNumber(9876.54321)  
x.precision(6) // '9876.54'  
x.sd() // 9  
x.precision(6, BigNumber.ROUND\_UP) // '9876.55'  
x.sd(2) // '9900'  
x.precision(2, 1) // '9800'  
x // '9876.54321'  
y = new BigNumber(987000)  
y.precision() // 3  
y.sd(true) // 6

##### shiftedBy.shiftedBy(n) ***⇒ BigNumber***

n: *number*: integer, -9007199254740991 to 9007199254740991 inclusive

Returns a BigNumber whose value is the value of this BigNumber shifted by n places.

The shift is of the decimal point, i.e. of powers of ten, and is to the left if n is negative or to the right if n is positive.

The return value is always exact and unrounded.

Throws if n is invalid. See [Errors](#43ky6rz).

x = new BigNumber(1.23)  
x.shiftedBy(3) // '1230'  
x.shiftedBy(-3) // '0.00123'

##### squareRoot.sqrt() ***⇒ BigNumber***

Returns a BigNumber whose value is the square root of the value of this BigNumber, rounded according to the current [DECIMAL\_PLACES](#tyjcwt) and [ROUNDING\_MODE](#3dy6vkm) settings.

The return value will be correctly rounded, i.e. rounded as if the result was first calculated to an infinite number of correct digits before rounding.

x = new BigNumber(16)  
x.squareRoot() // '4'  
y = new BigNumber(3)  
y.sqrt() // '1.73205080756887729353'

##### toExponential.toExponential([dp [, rm]]) ***⇒ string***

dp: *number*: integer, 0 to 1e+9 inclusive

rm: *number*: integer, 0 to 8 inclusive

Returns a string representing the value of this BigNumber in exponential notation rounded using rounding mode rm to dp decimal places, i.e with one digit before the decimal point and dp digits after it.

If the value of this BigNumber in exponential notation has fewer than dp fraction digits, the return value will be appended with zeros accordingly.

If dp is omitted, or is null or undefined, the number of digits after the decimal point defaults to the minimum number of digits necessary to represent the value exactly.

If rm is omitted or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) is used.

Throws if dp or rm is invalid. See [Errors](#43ky6rz).

x = 45.6  
y = new BigNumber(x)  
x.toExponential() // '4.56e+1'  
y.toExponential() // '4.56e+1'  
x.toExponential(0) // '5e+1'  
y.toExponential(0) // '5e+1'  
x.toExponential(1) // '4.6e+1'  
y.toExponential(1) // '4.6e+1'  
y.toExponential(1, 1) // '4.5e+1' (ROUND\_DOWN)  
x.toExponential(3) // '4.560e+1'  
y.toExponential(3) // '4.560e+1'

##### toFixed.toFixed([dp [, rm]]) ***⇒ string***

dp: *number*: integer, 0 to 1e+9 inclusive

rm: *number*: integer, 0 to 8 inclusive

Returns a string representing the value of this BigNumber in normal (fixed-point) notation rounded to dp decimal places using rounding mode rm.

If the value of this BigNumber in normal notation has fewer than dp fraction digits, the return value will be appended with zeros accordingly.

Unlike Number.prototype.toFixed, which returns exponential notation if a number is greater or equal to 1021, this method will always return normal notation.

If dp is omitted or is null or undefined, the return value will be unrounded and in normal notation. This is also unlike Number.prototype.toFixed, which returns the value to zero decimal places.

It is useful when fixed-point notation is required and the current [EXPONENTIAL\_AT](#1t3h5sf) setting causes [toString](#2r0uhxc) to return exponential notation.

If rm is omitted or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) is used.

Throws if dp or rm is invalid. See [Errors](#43ky6rz).

x = 3.456  
y = new BigNumber(x)  
x.toFixed() // '3'  
y.toFixed() // '3.456'  
y.toFixed(0) // '3'  
x.toFixed(2) // '3.46'  
y.toFixed(2) // '3.46'  
y.toFixed(2, 1) // '3.45' (ROUND\_DOWN)  
x.toFixed(5) // '3.45600'  
y.toFixed(5) // '3.45600'

##### toFormat.toFormat([dp [, rm[, format]]]) ***⇒ string***

dp: *number*: integer, 0 to 1e+9 inclusive

rm: *number*: integer, 0 to 8 inclusive

format: *object*: see [FORMAT](#26in1rg)

Returns a string representing the value of this BigNumber in normal (fixed-point) notation rounded to dp decimal places using rounding mode rm, and formatted according to the properties of the format object.

See [FORMAT](#26in1rg) and the examples below for the properties of the format object, their types, and their usage. A formatting object may contain some or all of the recognised properties.

If dp is omitted or is null or undefined, then the return value is not rounded to a fixed number of decimal places.

If rm is omitted or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) is used.

If format is omitted or is null or undefined, the [FORMAT](#26in1rg) object is used.

Throws if dp, rm or format is invalid. See [Errors](#43ky6rz).

fmt = {  
 prefix: '',  
 decimalSeparator: '.',  
 groupSeparator: ',',  
 groupSize: 3,  
 secondaryGroupSize: 0,  
 fractionGroupSeparator: ' ',  
 fractionGroupSize: 0,  
 suffix: ''  
}  
  
x = new BigNumber('123456789.123456789')  
  
// Set the global formatting options  
BigNumber.config({ FORMAT: fmt })  
  
x.toFormat() // '123,456,789.123456789'  
x.toFormat(3) // '123,456,789.123'  
  
// If a reference to the object assigned to FORMAT has been retained,  
// the format properties can be changed directly  
fmt.groupSeparator = ' '  
fmt.fractionGroupSize = 5  
x.toFormat() // '123 456 789.12345 6789'  
  
// Alternatively, pass the formatting options as an argument  
fmt = {  
 prefix: '=> ',  
 decimalSeparator: ',',  
 groupSeparator: '.',  
 groupSize: 3,  
 secondaryGroupSize: 2  
}  
  
x.toFormat() // '123 456 789.12345 6789'  
x.toFormat(fmt) // '=> 12.34.56.789,123456789'  
x.toFormat(2, fmt) // '=> 12.34.56.789,12'  
x.toFormat(3, BigNumber.ROUND\_UP, fmt) // '=> 12.34.56.789,124'

##### toFraction.toFraction([maximum\_denominator]) ***⇒ [BigNumber, BigNumber]***

maximum\_denominator: *number|string|BigNumber*: integer >= 1 and <= Infinity

Returns an array of two BigNumbers representing the value of this BigNumber as a simple fraction with an integer numerator and an integer denominator. The denominator will be a positive non-zero value less than or equal to maximum\_denominator.

If a maximum\_denominator is not specified, or is null or undefined, the denominator will be the lowest value necessary to represent the number exactly.

Throws if maximum\_denominator is invalid. See [Errors](#43ky6rz).

x = new BigNumber(1.75)  
x.toFraction() // '7, 4'  
  
pi = new BigNumber('3.14159265358')  
pi.toFraction() // '157079632679,50000000000'  
pi.toFraction(100000) // '312689, 99532'  
pi.toFraction(10000) // '355, 113'  
pi.toFraction(100) // '311, 99'  
pi.toFraction(10) // '22, 7'  
pi.toFraction(1) // '3, 1'

##### toJSON.toJSON() ***⇒ string***

As [valueOf](#1664s55).

x = new BigNumber('177.7e+457')  
y = new BigNumber(235.4325)  
z = new BigNumber('0.0098074')  
  
// Serialize an array of three BigNumbers  
str = JSON.stringify( [x, y, z] )  
// "["1.777e+459","235.4325","0.0098074"]"  
  
// Return an array of three BigNumbers  
JSON.parse(str, function (key, val) {  
 return key === '' ? val : new BigNumber(val)  
})

##### toNumber.toNumber() ***⇒ number***

Returns the value of this BigNumber as a JavaScript number primitive.

This method is identical to using type coercion with the unary plus operator.

x = new BigNumber(456.789)  
x.toNumber() // 456.789  
+x // 456.789  
  
y = new BigNumber('45987349857634085409857349856430985')  
y.toNumber() // 4.598734985763409e+34  
  
z = new BigNumber(-0)  
1 / z.toNumber() // -Infinity  
1 / +z // -Infinity

##### toPrecision.toPrecision([sd [, rm]]) ***⇒ string***

sd: *number*: integer, 1 to 1e+9 inclusive

rm: *number*: integer, 0 to 8 inclusive

Returns a string representing the value of this BigNumber rounded to sd significant digits using rounding mode rm.

If sd is less than the number of digits necessary to represent the integer part of the value in normal (fixed-point) notation, then exponential notation is used.

If sd is omitted, or is null or undefined, then the return value is the same as n.toString().

If rm is omitted or is null or undefined, [ROUNDING\_MODE](#3dy6vkm) is used.

Throws if sd or rm is invalid. See [Errors](#43ky6rz).

x = 45.6  
y = new BigNumber(x)  
x.toPrecision() // '45.6'  
y.toPrecision() // '45.6'  
x.toPrecision(1) // '5e+1'  
y.toPrecision(1) // '5e+1'  
y.toPrecision(2, 0) // '4.6e+1' (ROUND\_UP)  
y.toPrecision(2, 1) // '4.5e+1' (ROUND\_DOWN)  
x.toPrecision(5) // '45.600'  
y.toPrecision(5) // '45.600'

##### toString.toString([base]) ***⇒ string***

base: *number*: integer, 2 to ALPHABET.length inclusive (see [ALPHABET](#lnxbz9)).

Returns a string representing the value of this BigNumber in the specified base, or base 10 if base is omitted or is null or undefined.

For bases above 10, and using the default base conversion alphabet (see [ALPHABET](#lnxbz9)), values from 10 to 35 are represented by a-z (as with Number.prototype.toString).

If a base is specified the value is rounded according to the current [DECIMAL\_PLACES](#tyjcwt) and [ROUNDING\_MODE](#3dy6vkm) settings.

If a base is not specified, and this BigNumber has a positive exponent that is equal to or greater than the positive component of the current [EXPONENTIAL\_AT](#1t3h5sf) setting, or a negative exponent equal to or less than the negative component of the setting, then exponential notation is returned.

If base is null or undefined it is ignored.

Throws if base is invalid. See [Errors](#43ky6rz).

x = new BigNumber(750000)  
x.toString() // '750000'  
BigNumber.config({ EXPONENTIAL\_AT: 5 })  
x.toString() // '7.5e+5'  
  
y = new BigNumber(362.875)  
y.toString(2) // '101101010.111'  
y.toString(9) // '442.77777777777777777778'  
y.toString(32) // 'ba.s'  
  
BigNumber.config({ DECIMAL\_PLACES: 4 });  
z = new BigNumber('1.23456789')  
z.toString() // '1.23456789'  
z.toString(10) // '1.2346'

##### valueOf.valueOf() ***⇒ string***

As [toString](#2r0uhxc), but does not accept a base argument and includes the minus sign for negative zero.

x = new BigNumber('-0')  
x.toString() // '0'  
x.valueOf() // '-0'  
y = new BigNumber('1.777e+457')  
y.valueOf() // '1.777e+457'

#### Properties

The properties of a BigNumber instance:

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Description | Type | Value |
| c | coefficient\* | *number*[] | Array of base 1e14 numbers |
| e | exponent | *number* | Integer, -1000000000 to 1000000000 inclusive |
| s | sign | *number* | -1 or 1 |

\*significand

The value of any of the c, e and s properties may also be null.

The above properties are best considered to be read-only. In early versions of this library it was okay to change the exponent of a BigNumber by writing to its exponent property directly, but this is no longer reliable as the value of the first element of the coefficient array is now dependent on the exponent.

Note that, as with JavaScript numbers, the original exponent and fractional trailing zeros are not necessarily preserved.

x = new BigNumber(0.123) // '0.123'  
x.toExponential() // '1.23e-1'  
x.c // '1,2,3'  
x.e // -1  
x.s // 1  
  
y = new Number(-123.4567000e+2) // '-12345.67'  
y.toExponential() // '-1.234567e+4'  
z = new BigNumber('-123.4567000e+2') // '-12345.67'  
z.toExponential() // '-1.234567e+4'  
z.c // '1,2,3,4,5,6,7'  
z.e // 4  
z.s // -1

#### Zero, NaN and Infinity

The table below shows how ±0, NaN and ±Infinity are stored.

|  |  |  |  |
| --- | --- | --- | --- |
|  | c | e | s |
| ±0 | [0] | 0 | ±1 |
| NaN | null | null | null |
| ±Infinity | null | null | ±1 |

x = new Number(-0) // 0  
1 / x == -Infinity // true  
  
y = new BigNumber(-0) // '0'  
y.c // '0' ( [0].toString() )  
y.e // 0  
y.s // -1

#### Errors

The table below shows the errors that are thrown.

The errors are generic Error objects whose message begins '[BigNumber Error]'.

|  |  |
| --- | --- |
| Method | Throws |
| BigNumber  comparedTo  dividedBy  dividedToIntegerBy  isEqualTo  isGreaterThan  isGreaterThanOrEqualTo  isLessThan  isLessThanOrEqualTo  minus  modulo  plus  multipliedBy | Base not a primitive number |
| Base not an integer |
| Base out of range |
| Number primitive has more than 15 significant digits\* |
| Not a base... number\* |
| Not a number\* |
| clone | Object expected |
| config | Object expected |
| DECIMAL\_PLACES not a primitive number |
| DECIMAL\_PLACES not an integer |
| DECIMAL\_PLACES out of range |
| ROUNDING\_MODE not a primitive number |
| ROUNDING\_MODE not an integer |
| ROUNDING\_MODE out of range |
| EXPONENTIAL\_AT not a primitive number |
| EXPONENTIAL\_AT not an integer |
| EXPONENTIAL\_AT out of range |
| RANGE not a primitive number |
| RANGE not an integer |
| RANGE cannot be zero |
| RANGE cannot be zero |
| CRYPTO not true or false |
| crypto unavailable |
| MODULO\_MODE not a primitive number |
| MODULO\_MODE not an integer |
| MODULO\_MODE out of range |
| POW\_PRECISION not a primitive number |
| POW\_PRECISION not an integer |
| POW\_PRECISION out of range |
| FORMAT not an object |
| ALPHABET invalid |
| decimalPlaces  precision  random  shiftedBy  toExponential  toFixed  toFormat  toPrecision | Argument not a primitive number |
| Argument not an integer |
| Argument out of range |
| decimalPlaces  precision | Argument not true or false |
| exponentiatedBy | Argument not an integer |
| isBigNumber | Invalid BigNumber\* |
| minimum  maximum | Not a number\* |
| random | crypto unavailable |
| toFormat | Argument not an object |
| toFraction | Argument not an integer |
| Argument out of range |
| toString | Base not a primitive number |
| Base not an integer |
| Base out of range |

\*Only thrown if BigNumber.DEBUG is true.

To determine if an exception is a BigNumber Error:

try {  
 // ...  
} catch (e) {  
 if (e instanceof Error && e.message.indexOf('[BigNumber Error]') === 0) {  
 // ...  
 }  
}

#### Type coercion

To prevent the accidental use of a BigNumber in primitive number operations, or the accidental addition of a BigNumber to a string, the valueOf method can be safely overwritten as shown below.

The [valueOf](#1664s55) method is the same as the [toJSON](#3cqmetx) method, and both are the same as the [toString](#2r0uhxc) method except they do not take a base argument and they include the minus sign for negative zero.

BigNumber.prototype.valueOf = function () {  
 throw Error('valueOf called!')  
}  
  
x = new BigNumber(1)  
x / 2 // '[BigNumber Error] valueOf called!'  
x + 'abc' // '[BigNumber Error] valueOf called!'

#### FAQ

###### Why are trailing fractional zeros removed from BigNumbers?

Some arbitrary-precision libraries retain trailing fractional zeros as they can indicate the precision of a value. This can be useful but the results of arithmetic operations can be misleading.

x = new BigDecimal("1.0")  
y = new BigDecimal("1.1000")  
z = x.add(y) // 2.1000  
  
x = new BigDecimal("1.20")  
y = new BigDecimal("3.45000")  
z = x.multiply(y) // 4.1400000

To specify the precision of a value is to specify that the value lies within a certain range.

In the first example, x has a value of 1.0. The trailing zero shows the precision of the value, implying that it is in the range 0.95 to 1.05. Similarly, the precision indicated by the trailing zeros of y indicates that the value is in the range 1.09995 to 1.10005.

If we add the two lowest values in the ranges we have, 0.95 + 1.09995 = 2.04995, and if we add the two highest values we have, 1.05 + 1.10005 = 2.15005, so the range of the result of the addition implied by the precision of its operands is 2.04995 to 2.15005.

The result given by BigDecimal of 2.1000 however, indicates that the value is in the range 2.09995 to 2.10005 and therefore the precision implied by its trailing zeros may be misleading.

In the second example, the true range is 4.122744 to 4.157256 yet the BigDecimal answer of 4.1400000 indicates a range of 4.13999995 to 4.14000005. Again, the precision implied by the trailing zeros may be misleading.

This library, like binary floating point and most calculators, does not retain trailing fractional zeros. Instead, the toExponential, toFixed and toPrecision methods enable trailing zeros to be added if and when required.