[API](#gjdgxs) CONSTRUCTOR

[BigNumber](#30j0zll)

[Methods](#1fob9te)

[another](#3znysh7)

[config](#2et92p0)

[DECIMAL\_PLACES](#tyjcwt)

[ROUNDING\_MODE](#3dy6vkm)

[EXPONENTIAL\_AT](#1t3h5sf)

[RANGE](#4d34og8)

[ERRORS](#2s8eyo1)

[CRYPTO](#17dp8vu)

[MODULO\_MODE](#3rdcrjn)

[POW\_PRECISION](#26in1rg)

[FORMAT](#lnxbz9)

[max](#35nkun2)

[min](#1ksv4uv)

[random](#44sinio)

[Properties](#2jxsxqh)

[ROUND\_UP](#z337ya)

[ROUND\_DOWN](#3j2qqm3)

[ROUND\_CEIL](#1y810tw)

[ROUND\_FLOOR](#4i7ojhp)

[ROUND\_HALF\_UP](#2xcytpi)

[ROUND\_HALF\_DOWN](#1ci93xb)

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

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

[ROUND\_HALF\_FLOOR](#qsh70q)

INSTANCE [Methods](#3as4poj)

[absoluteValue](#1pxezwc) abs

[ceil](#49x2ik5)

[comparedTo](#2p2csry) cmp

[decimalPlaces](#147n2zr) dp

[dividedBy](#3o7alnk) div

[dividedToIntegerBy](#23ckvvd) divToInt

[equals](#ihv636) eq

[floor](#32hioqz)

[greaterThan](#1hmsyys) gt

[greaterThanOrEqualTo](#41mghml)gte

[isFinite](#2grqrue)

[isInteger](#vx1227) isInt

[isNaN](#3fwokq0)

[isNegative](#1v1yuxt) isNeg

[isZero](#4f1mdlm)

[lessThan](#2u6wntf) lt

[lessThanOrEqualTo](#19c6y18) lte

[minus](#3tbugp1) sub

[modulo](#28h4qwu) mod

[negated](#nmf14n) neg

[plus](#37m2jsg) add

[precision](#1mrcu09) sd

[round](#46r0co2)

[shift](#2lwamvv)

[squareRoot](#111kx3o) sqrt

[times](#3l18frh) mul

[toDigits](#206ipza)

[toExponential](#4k668n3)

[toFixed](#2zbgiuw)

[toFormat](#1egqt2p)

[toFraction](#3ygebqi)

[toJSON](#2dlolyb)

[toNumber](#sqyw64)

[toPower](#3cqmetx) pow

[toPrecision](#1rvwp1q)

[toString](#4bvk7pj)

[truncated](#2r0uhxc) trunc

[valueOf](#1664s55)

[Properties](#3q5sasy)

[c: coefficient](#25b2l0r)

[e: exponent](#kgcv8k)

[s: sign](#34g0dwd)

[isBigNumber](#1jlao46)

[Zero, NaN & Infinity](#43ky6rz) [Errors](#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(value [, base]) ***⇒ BigNumber***

value *number|string|BigNumber*: see [RANGE](#4d34og8) for range. A numeric value. Legitimate values include ±0, ±Infinity and NaN. Values of type *number* with more than 15 significant digits are considered invalid (if [ERRORS](#2s8eyo1) is true) as calling [toString](#4bvk7pj) or [valueOf](#1664s55) on such numbers may not result in the intended value.

console.log( 823456789123456.3 ); // 823456789123456.2

There is no limit to the number of digits of a value of type *string* (other than that of JavaScript's maximum array size). Decimal string values may be in exponential, as well as normal (fixed-point) notation. Non-decimal values must be in normal notation. 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. Values in any base may have fraction digits. For bases from 10 to 36, lower and/or upper case letters can be used to represent values from 10 to 35. For bases above 36, a-z represents values from 10 to 35, A-Z from 36 to 61, and $ and \_ represent 62 and 63 respectively *(this can be changed by editing the ALPHABET variable near the top of the source file)*. base *number*: integer, 2 to 64 inclusive The base of value. If base is omitted, or is null or undefined, base 10 is assumed.

Returns a new instance of a BigNumber object.

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

See [Errors](#2iq8gzs) for the treatment of an invalid value or base.

x = new BigNumber(9) // '9'  
y = new BigNumber(x) // '9'  
  
// 'new' is optional if ERRORS is false  
BigNumber(435.345) // '435.345'  
  
new BigNumber('5032485723458348569331745.33434346346912144534543')  
new BigNumber('4.321e+4') // '43210'  
new BigNumber('-735.0918e-430') // '-7.350918e-428'  
new BigNumber(Infinity) // 'Infinity'  
new BigNumber(NaN) // 'NaN'  
new BigNumber('.5') // '0.5'  
new BigNumber('+2') // '2'  
new BigNumber(-10110100.1, 2) // '-180.5'  
new BigNumber(-0b10110100.1) // '-180.5'  
new BigNumber('123412421.234324', 5) // '607236.557696'  
new BigNumber('ff.8', 16) // '255.5'  
new BigNumber('0xff.8') // '255.5'

The following throws 'not a base 2 number' if [ERRORS](#2s8eyo1) is true, otherwise it returns a BigNumber with value NaN.

new BigNumber(9, 2)

The following throws 'number type has more than 15 significant digits' if [ERRORS](#2s8eyo1) is true, otherwise it returns a BigNumber with value 96517860459076820.

new BigNumber(96517860459076817.4395)

The following throws 'not a number' if [ERRORS](#2s8eyo1) is true, otherwise it returns a BigNumber with value NaN.

new BigNumber('blurgh')

A value is only rounded by the constructor if a base is specified.

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

#### Methods

The static methods of a BigNumber constructor.

##### another.another([obj]) ***⇒ BigNumber constructor***

obj: *object*

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

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

##### configset([obj]) ***⇒ object***

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

Configures the settings for this particular BigNumber constructor.

*Note: the configuration can also be supplied as an argument list, see below.*

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  
BigNumber.config(5) // equivalent

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

Default value: 4 [(ROUND\_HALF\_UP)](#2xcytpi) The rounding mode used in the above operations and the default rounding mode of [round](#46r0co2), [toExponential](#4k668n3), [toFixed](#2zbgiuw), [toFormat](#1egqt2p) and [toPrecision](#1rvwp1q). The modes are available as enumerated properties of the BigNumber constructor.

BigNumber.config({ ROUNDING\_MODE: 0 })  
BigNumber.config(null, 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. ERRORS *boolean|number*: true, false, 0 or 1.

Default value: true The value that determines whether BigNumber Errors are thrown.

If ERRORS is false, no errors will be thrown. See [Errors](#2iq8gzs).

BigNumber.config({ ERRORS: false })

CRYPTO *boolean|number*: true, false, 0 or 1.

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](#44sinio) method will generate random digits using crypto.getRandomValues in browsers that support it, or crypto.randomBytes if using a version of Node.js that supports it. If neither function is supported by the host environment then attempting to set CRYPTO to true will fail, and if ERRORS is true 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](#44sinio).

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](#3j2qqm3)) 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](#28h4qwu).

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* number of significant digits of the result of the power operation (unless a modulus is specified). If set to 0, the number of signifcant digits will not be limited. See [toPower](#3cqmetx).

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

FORMAT *object* The FORMAT object configures the format of the string returned by the [toFormat](#1egqt2p) 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. Note that all the properties shown below do not have to be included. See [toFormat](#1egqt2p) for examples of usage.

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

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

If the value to be assigned to any of the above properties is null or undefined it is ignored.

See [Errors](#2iq8gzs) for the treatment of invalid values.

BigNumber.config({  
 DECIMAL\_PLACES: 40,  
 ROUNDING\_MODE: BigNumber.ROUND\_HALF\_CEIL,  
 EXPONENTIAL\_AT: [-10, 20],  
 RANGE: [-500, 500],  
 ERRORS: true,  
 CRYPTO: true,  
 MODULO\_MODE: BigNumber.ROUND\_FLOOR,  
 POW\_PRECISION: 80,  
 FORMAT: {  
 groupSize: 3,  
 groupSeparator: ' ',  
 decimalSeparator: ','  
 }  
});  
  
// Alternatively but equivalently (excluding FORMAT):  
BigNumber.config( 40, 7, [-10, 20], 500, 1, 1, 3, 80 )  
  
obj = BigNumber.config();  
obj.ERRORS // true  
obj.RANGE // [-500, 500]

##### max.max([arg1 [, arg2, ...]]) ***⇒ BigNumber***

arg1, arg2, ...: *number|string|BigNumber*

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

Returns a BigNumber whose value is the maximum of arg1, arg2,... .

The argument to this method can also be an array of values.

The return value is always exact and unrounded.

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

##### min.min([arg1 [, arg2, ...]]) ***⇒ BigNumber***

arg1, arg2, ...: *number|string|BigNumber*

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

Returns a BigNumber whose value is the minimum of arg1, arg2,... .

The argument to this method can also be an array of values.

The return value is always exact and unrounded.

x = new BigNumber('3257869345.0378653')  
BigNumber.min(4e9, x, '123456789.9') // '123456789.9'  
  
arr = [2, new BigNumber(-14), '-15.9999', -12]  
BigNumber.min(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](#17dp8vu) 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).

If [CRYPTO](#17dp8vu) 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.

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

#### 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

### 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.

(Internally, the library always uses the shorter method names.)

##### 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'

##### ceil.ceil() ***⇒ BigNumber***

Returns a BigNumber whose value is the value of this BigNumber rounded to a whole number in the direction of positive Infinity.

x = new BigNumber(1.3)  
x.ceil() // '2'  
y = new BigNumber(-1.8)  
y.ceil() // '-1'

##### comparedTo.cmp(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.cmp(NaN) // null  
y.cmp('110', 2) // -1

##### decimalPlaces.dp() ***⇒ number***

Return the number of decimal places of the value of this BigNumber, or null if the value of this BigNumber is ±Infinity or NaN.

x = new BigNumber(123.45)  
x.decimalPlaces() // 2  
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) configuration.

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

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

n: *number|string|BigNumber*

base: *number*

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

Return 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.divToInt(0.7) // '7'  
x.divToInt('0.f', 16) // '5'

##### equals.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 equals the value of n, otherwise returns false.

As with JavaScript, NaN does not equal NaN.

Note: This method uses the [comparedTo](#2p2csry) method internally.

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

##### floor.floor() ***⇒ BigNumber***

Returns a BigNumber whose value is the value of this BigNumber rounded to a whole number in the direction of negative Infinity.

x = new BigNumber(1.8)  
x.floor() // '1'  
y = new BigNumber(-1.3)  
y.floor() // '-2'

##### greaterThan.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](#2p2csry) method internally.

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

##### greaterThanOrEqualTo.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](#2p2csry) method internally.

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

##### 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.

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

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

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

##### 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.isNeg() ***⇒ boolean***

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

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

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

##### 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.isNeg() // true  
y = new BigNumber(Infinity)  
y.isZero() // false

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

##### lessThan.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](#2p2csry) method internally.

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

##### lessThanOrEqualTo.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](#2p2csry) method internally.

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

##### 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](#3rdcrjn) 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](#3rdcrjn) 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'

##### negated.neg() ***⇒ 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.neg() // '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'  
x.plus('0.1', 8) // '0.225'

##### precision.sd([z]) ***⇒ number***

z: *boolean|number*: true, false, 0 or 1

Returns the number of significant digits of the value of this BigNumber.

If z is true or 1 then any trailing zeros of the integer part of a number are counted as significant digits, otherwise they are not.

x = new BigNumber(1.234)  
x.precision() // 4  
y = new BigNumber(987000)  
y.sd() // 3  
y.sd(true) // 6

##### round.round([dp [, rm]]) ***⇒ BigNumber***

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

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

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 n rounded to a whole number.

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

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range dp or rm values.

x = 1234.56  
Math.round(x) // 1235  
  
y = new BigNumber(x)  
y.round() // '1235'  
y.round(1) // '1234.6'  
y.round(2) // '1234.56'  
y.round(10) // '1234.56'  
y.round(0, 1) // '1234'  
y.round(0, 6) // '1235'  
y.round(1, 1) // '1234.5'  
y.round(1, BigNumber.ROUND\_HALF\_EVEN) // '1234.6'  
y // '1234.56'

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

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

Returns a BigNumber whose value is the value of this BigNumber shifted 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.

x = new BigNumber(1.23)  
x.shift(3) // '1230'  
x.shift(-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) configuration.

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'

##### times.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 times n.

The return value is always exact and unrounded.

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

##### toDigits.toDigits([sd [, rm]]) ***⇒ BigNumber***

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

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

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

If sd is omitted or is null or undefined, the return value will not be rounded.

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

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range sd or rm values.

BigNumber.config({ precision: 5, rounding: 4 })  
x = new BigNumber(9876.54321)  
  
x.toDigits() // '9876.5'  
x.toDigits(6) // '9876.54'  
x.toDigits(6, BigNumber.ROUND\_UP) // '9876.55'  
x.toDigits(2) // '9900'  
x.toDigits(2, 1) // '9800'  
x // '9876.54321'

##### 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.

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range dp or rm values.

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](#4bvk7pj) to return exponential notation.

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

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range dp or rm values.

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]]) ***⇒ 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, and formatted according to the properties of the [FORMAT](#lnxbz9) object.

See the examples below for the properties of the [FORMAT](#lnxbz9) object, their types and their usage.

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.

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range dp or rm values.

format = {  
 decimalSeparator: '.',  
 groupSeparator: ',',  
 groupSize: 3,  
 secondaryGroupSize: 0,  
 fractionGroupSeparator: ' ',  
 fractionGroupSize: 0  
}  
BigNumber.config({ FORMAT: format })  
  
x = new BigNumber('123456789.123456789')  
x.toFormat() // '123,456,789.123456789'  
x.toFormat(1) // '123,456,789.1'  
  
// If a reference to the object assigned to FORMAT has been retained,  
// the format properties can be changed directly  
format.groupSeparator = ' '  
format.fractionGroupSize = 5  
x.toFormat() // '123 456 789.12345 6789'  
  
BigNumber.config({  
 FORMAT: {  
 decimalSeparator: ',',  
 groupSeparator: '.',  
 groupSize: 3,  
 secondaryGroupSize: 2  
 }  
})  
  
x.toFormat(6) // '12.34.56.789,123'

##### toFraction.toFraction([max]) ***⇒ [string, string]***

max: *number|string|BigNumber*: integer >= 1 and < Infinity

Returns a string array 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 max.

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

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range max values.

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.

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.

Type coercion with, for example, the unary plus operator will also work, except that a BigNumber with the value minus zero will be converted to positive zero.

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 // Infinity  
1 / z.toNumber() // -Infinity

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

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

m: *number|string|BigNumber*

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

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

If n is not an integer or is out of range:

If ERRORS is true a BigNumber Error is thrown,

else if n is greater than 9007199254740991, it is interpreted as Infinity;

else if n is less than -9007199254740991, it is interpreted as -Infinity;

else if n is otherwise a number, it is truncated to an integer;

else it is interpreted as NaN.

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](#26in1rg) setting (unless a modulus m is specified).

By default [POW\_PRECISION](#26in1rg) 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.

Negative exponents will be calculated to the number of decimal places specified by [DECIMAL\_PLACES](#tyjcwt) (but not to more than [POW\_PRECISION](#26in1rg) significant digits).

If m is specified and the value of m, n and this BigNumber are positive integers, then a fast modular exponentiation algorithm is used, otherwise if any of the values is not a positive integer the operation will simply be performed as x.toPower(n).modulo(m) with a [POW\_PRECISION](#26in1rg) of 0.

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

##### 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.

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range sd or rm values.

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 64 inclusive

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, values from 10 to 35 are represented by a-z (as with Number.prototype.toString), 36 to 61 by A-Z, and 62 and 63 by $ and \_ respectively.

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

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.

See [Errors](#2iq8gzs) for the treatment of other non-integer or out of range base values.

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'

##### truncated.trunc() ***⇒ BigNumber***

Returns a BigNumber whose value is the value of this BigNumber truncated to a whole number.

x = new BigNumber(123.456)  
x.truncated() // '123'  
y = new BigNumber(-12.3)  
y.trunc() // '-12'

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

As toString, 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 |
| isBigNumber | type identifier | *boolean* | true |

\*significand

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

From v2.0.0 of this library, the value of the coefficient of a BigNumber is stored in a normalised base 100000000000000 floating point format, as opposed to the base 10 format used in v1.x.x

This change means the properties of a BigNumber are now best considered to be read-only. Previously it was acceptable to change the exponent of a BigNumber by writing to its exponent property directly, but this is no longer recommended as the number of digits in the first element of the coefficient array is dependent on the exponent, so the coefficient would also need to be altered.

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

Checking if a value is a BigNumber instance:

x = new BigNumber(3)  
x instanceof BigNumber // true  
x.isBigNumber // true  
  
BN = BigNumber.another();  
  
y = new BN(3)  
y instanceof BigNumber // false  
y.isBigNumber // true

#### 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 errors that are thrown are generic Error objects with name *BigNumber Error*.

The table below shows the errors that may be thrown if ERRORS is true, and the action taken if ERRORS is false.

|  |  |  |
| --- | --- | --- |
| Method(s) | ERRORS: true  Throw BigNumber Error | ERRORS: false  Action on invalid argument |
| BigNumber  comparedTo  dividedBy  dividedToIntegerBy  equals  greaterThan  greaterThanOrEqualTo  lessThan  lessThanOrEqualTo  minus  modulo  plus  times | number type has more than  15 significant digits | Accept. |
| not a base... number | Substitute NaN. |
| base not an integer | Truncate to integer.  Ignore if not a number. |
| base out of range | Ignore. |
| not a number\* | Substitute NaN. |
| another | not an object | Ignore. |
| config | DECIMAL\_PLACES not an integer | Truncate to integer.  Ignore if not a number. |
| DECIMAL\_PLACES out of range | Ignore. |
| ROUNDING\_MODE not an integer | Truncate to integer.  Ignore if not a number. |
| ROUNDING\_MODE out of range | Ignore. |
| EXPONENTIAL\_AT not an integer  or not [integer, integer] | Truncate to integer(s).  Ignore if not number(s). |
| EXPONENTIAL\_AT out of range  or not [negative, positive] | Ignore. |
| RANGE not an integer  or not [integer, integer] | Truncate to integer(s).  Ignore if not number(s). |
| RANGE cannot be zero | Ignore. |
| RANGE out of range  or not [negative, positive] | Ignore. |
| ERRORS not a boolean  or binary digit | Ignore. |
| CRYPTO not a boolean  or binary digit | Ignore. |
| CRYPTO crypto unavailable | Ignore. |
| MODULO\_MODE not an integer | Truncate to integer.  Ignore if not a number. |
| MODULO\_MODE out of range | Ignore. |
| POW\_PRECISION not an integer | Truncate to integer.  Ignore if not a number. |
| POW\_PRECISION out of range | Ignore. |
| FORMAT not an object | Ignore. |
| precision | argument not a boolean  or binary digit | Ignore. |
| round | decimal places not an integer | Truncate to integer.  Ignore if not a number. |
| decimal places out of range | Ignore. |
| rounding mode not an integer | Truncate to integer.  Ignore if not a number. |
| rounding mode out of range | Ignore. |
| shift | argument not an integer | Truncate to integer.  Ignore if not a number. |
| argument out of range | Substitute ±Infinity. |
| toExponential  toFixed  toFormat | decimal places not an integer | Truncate to integer.  Ignore if not a number. |
| decimal places out of range | Ignore. |
| rounding mode not an integer | Truncate to integer.  Ignore if not a number. |
| rounding mode out of range | Ignore. |
| toFraction | max denominator not an integer | Truncate to integer.  Ignore if not a number. |
| max denominator out of range | Ignore. |
| toDigits  toPrecision | precision not an integer | Truncate to integer.  Ignore if not a number. |
| precision out of range | Ignore. |
| rounding mode not an integer | Truncate to integer.  Ignore if not a number. |
| rounding mode out of range | Ignore. |
| toPower | exponent not an integer | Truncate to integer.  Substitute NaN if not a number. |
| exponent out of range | Substitute ±Infinity. |
| toString | base not an integer | Truncate to integer.  Ignore if not a number. |
| base out of range | Ignore. |

\*No error is thrown if the value is NaN or 'NaN'.

The message of a *BigNumber Error* will also contain the name of the method from which the error originated.

To determine if an exception is a *BigNumber Error*:

try {  
 // ...  
} catch (e) {  
 if ( e instanceof Error && e.name == 'BigNumber Error' ) {  
 // ...  
 }  
}

#### 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.