decimal.js

An arbitrary-precision Decimal type for JavaScript.

Hosted on GitHub.

API

See the README 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.

When the library is loaded, it defines a single function object, `Decimal`, the constructor of Decimal instances.

If necessary, multiple Decimal constructors can be created, each with their own independent configuration, e.g. precision and range, which applies to all Decimal instances created from it.

A new Decimal constructor is created by calling the `clone` method of an already existing Decimal constructor.

CONSTRUCTOR

Decimal `Decimal(value)` □ *`Decimal`*

`value`: *number|string|Decimal*

> A legitimate `value` is an integer or float, including `±0`, or is `±Infinity`, or `NaN`.
>
> The number of digits of `value` is not limited, except by JavaScript's maximum array size and, in practice, the processing time required.
>
> The allowable range of `value` is defined in terms of a maximum exponent, see maxE, and a minimum exponent, see minE.
>
> As well as in decimal, a string `value` may be expressed in binary, hexadecimal or octal, if the appropriate prefix is included: `0x` or `0X` for hexadecimal, `0b` or `0B` for binary, and `0o` or `0O` for octal.
>
> Both decimal and non-decimal string values may use exponential (floating-point), as well as normal (fixed-point) notation.
>
> In exponential notation, `e` or `E` defines a power-of-ten exponent for decimal values, and `p` or `P` defines a power-of-two exponent for non-decimal values, i.e. binary, hexadecimal or octal.

Returns a new Decimal object instance.

Throws on an invalid `value`.

```
x = new Decimal(9)                       // '9'
y = new Decimal(x)                       // '9'

new Decimal('5032485723458348569331745.33434346346912144534543')
new Decimal('4.321e+4')                  // '43210'
new Decimal('-735.0918e-430')            // '-7.350918e-428'
new Decimal('5.6700000')                 // '5.67'
new Decimal(Infinity)                    // 'Infinity'
new Decimal(NaN)                         // 'NaN'
new Decimal('.5')                        // '0.5'
new Decimal('-0b10110100.1')             // '-180.5'
new Decimal('0xff.8')                    // '255.5'

new Decimal(0.046875)                    // '0.046875'
new Decimal('0.046875000000')            // '0.046875'

new Decimal(4.6875e-2)                   // '0.046875'
new Decimal('468.75e-4')                 // '0.046875'

new Decimal('0b0.000011')                // '0.046875'
new Decimal('0o0.03')                    // '0.046875'
new Decimal('0x0.0c')                    // '0.046875'

new Decimal('0b1.1p-5')                  // '0.046875'
new Decimal('0o1.4p-5')                  // '0.046875'
new Decimal('0x1.8p-5')                  // '0.046875'
```

Methods

The methods of a Decimal constructor.

abs `.abs(x)` ⇒ *`Decimal`*

`x`: *number|string|Decimal*

See `absoluteValue`.

```
a = Decimal.abs(x)
b = new Decimal(x).abs()
a.equals(b)                    // true
```

acos `.acos(x)` ⇒ *`Decimal`*

`x`: *number|string|Decimal*

See `inverseCosine`.

```
a = Decimal.acos(x)
b = new Decimal(x).acos()
a.equals(b)                    // true
```

acosh `.acosh(x)` ⇒ *`Decimal`*

x: *number|string|Decimal*

See `inverseHyperbolicCosine`.

```
a = Decimal.acosh(x)
b = new Decimal(x).acosh()
a.equals(b)                    // true
```

add `.add(x, y)` ⇒ *`Decimal`*

x: *number|string|Decimal*
y: *number|string|Decimal*

See `plus`.

```
a = Decimal.add(x, y)
b = new Decimal(x).plus(y)
a.equals(b)                    // true
```

asin `.asin(x)` ⇒ *`Decimal`*

x: *number|string|Decimal*

See `inverseSine`.

```
a = Decimal.asin(x)
b = new Decimal(x).asin()
a.equals(b)                    // true
```

asinh `.asinh(x)` ⇒ *`Decimal`*

x: *number|string|Decimal*

See `inverseHyperbolicSine`.

```
a = Decimal.asinh(x)
b = new Decimal(x).asinh()
a.equals(b)                    // true
```

atan `.atan(x)` ⇒ *`Decimal`*

x: *number|string|Decimal*

See `inverseTangent`.

```
a = Decimal.atan(x)
b = new Decimal(x).atan()
a.equals(b)                    // true
```

atanh `.atanh(x)` ⇒ *`Decimal`*

API

CONSTRUCTOR

Methods

Properties

x: *number|string|Decimal*

See `inverseHyperbolicTangent`.

```
a = Decimal.atanh(x)
b = new Decimal(x).atanh()
a.equals(b)                    // true
```

atan2 `.atan2(y, x)` ⇒ *`Decimal`*

y: *number|string|Decimal*
x: *number|string|Decimal*

Returns a new Decimal whose value is the inverse tangent in radians of the quotient of `y` and `x`, rounded to `precision` significant digits using rounding mode `rounding`.

The signs of `y` and `x` are used to determine the quadrant of the result.

Domain: [`-Infinity, Infinity`]
Range: [`-pi, pi`]

See `Pi` and `Math.atan2()`.

```
r = Decimal.atan2(y, x)
```

cbrt `.cbrt(x)` ⇒ *`Decimal`*

x: *number|string|Decimal*

See `cubeRoot`.

```
a = Decimal.cbrt(x)
b = new Decimal(x).cbrt()
a.equals(b)                    // true
```

ceil `.ceil(x)` ⇒ *`Decimal`*

x: *number|string|Decimal*

See `ceil`.

```
a = Decimal.ceil(x)
b = new Decimal(x).ceil()
a.equals(b)                    // true
```

clone `.clone([object])` ⇒ *`Decimal constructor`*

`object`: *object*

Returns a new independent Decimal constructor with configuration settings as described by `object` (see `set`), or with the same settings as `this` Decimal constructor if `object` is omitted.

```
Decimal.set({ precision: 5 })
Decimal9 = Decimal.clone({ precision: 9 })
```

```
a = new Decimal(1)
b = new Decimal9(1)

a.div(3)                           // 0.33333
b.div(3)                           // 0.333333333

// Decimal9 = Decimal.clone({ precision: 9 }) is equivalent to:
Decimal9 = Decimal.clone()
Decimal9.set({ precision: 9 })
```

If `object` has a `'defaults'` property with value `true` then the new constructor will use the default configuration.

```
D1 = Decimal.clone({ defaults: true })

// Use the defaults except for precision
D2 = Decimal.clone({ defaults: true, precision: 50 })
```

It is not inefficient in terms of memory usage to use multiple Decimal constructors as functions are shared between them.

cos `.cos(x)` □ *Decimal*

`x`: *number|string|Decimal*

See `cosine`.

```
a = Decimal.cos(x)
b = new Decimal(x).cos()
a.equals(b)                    // true
```

cosh `.cosh(x)` □ *Decimal*

`x`: *number|string|Decimal*

See `hyperbolicCosine`.

```
a = Decimal.cosh(x)
b = new Decimal(x).cosh()
a.equals(b)                    // true
```

div `.div(x, y)` □ *Decimal*

`x`: *number|string|Decimal*
`y`: *number|string|Decimal*

See `dividedBy`.

```
a = Decimal.div(x, y)
b = new Decimal(x).div(y)
a.equals(b)                    // true
```

exp `.exp(x)` □ *Decimal*

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

`x`: *number|string|Decimal*

See `naturalExponential`.

```
a = Decimal.exp(x)
b = new Decimal(x).exp()
a.equals(b)                    // true
```

floor `.floor(x)` □ *Decimal*

`x`: *number|string|Decimal*

See `floor`.

```
a = Decimal.floor(x)
b = new Decimal(x).floor()
a.equals(b)                    // true
```

hypot `.hypot([x [, y, ...]])` □ *Decimal*

`x`: *number|string|Decimal*
`y`: *number|string|Decimal*

Returns a new Decimal whose value is the square root of the sum of the squares of the arguments, rounded to `precision` significant digits using rounding mode `rounding`.

```
r = Decimal.hypot(x, y)
```

ln `.ln(x)` □ *Decimal*

`x`: *number|string|Decimal*

See `naturalLogarithm`.

```
a = Decimal.ln(x)
b = new Decimal(x).ln()
a.equals(b)                    // true
```

isDecimal `.isDecimal(object)` □ *boolean*

`object`: *any*

Returns `true` if `object` is a Decimal instance (where Decimal is any Decimal constructor), or `false` if it is not.

```
a = new Decimal(1)
b = {}
a instanceof Decimal           // true
Decimal.isDecimal(a)           // true
Decimal.isDecimal(b)           // false
```

log `.log(x [, base])` □ *Decimal*

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto
ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

`x`: *number|string|Decimal*
`base`: *number|string|Decimal*

See `logarithm`.

The default base is `10`, which is not the same as JavaScript's `Math.log()`, which returns the natural logarithm (base `e`).

```
a = Decimal.log(x, y)
b = new Decimal(x).log(y)
a.equals(b)                    // true
```

log2 `.log2(x)` □ *Decimal*

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the base `2` logarithm of `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
r = Decimal.log2(x)
```

log10 `.log10(x)` □ *Decimal*

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the base `10` logarithm of `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
r = Decimal.log10(x)
```

max `.max([x [, y, ...]])` □ *Decimal*

`x`: *number|string|Decimal*
`y`: *number|string|Decimal*

Returns a new Decimal whose value is the maximum of the `arguments`.

```
r = Decimal.max(x, y, z)
```

min `.min([x [, y, ...]])` □ *Decimal*

`x`: *number|string|Decimal*
`y`: *number|string|Decimal*

Returns a new Decimal whose value is the minimum of the `arguments`.

```
r = Decimal.min(x, y, z)
```

mod `.mod(x, y)` □ *Decimal*

`x`: *number|string|Decimal*
`y`: *number|string|Decimal*

See `modulo`.

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto
ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

```
a = Decimal.mod(x, y)
b = new Decimal(x).mod(y)
a.equals(b)                    // true
```

mul `.mul(x, y)` □ *`Decimal`*

`x`: *number|string|Decimal*
`y`: *number|string|Decimal*

See `times`.

```
a = Decimal.mul(x, y)
b = new Decimal(x).mul(y)
a.equals(b)                    // true
```

noConflict `.noConflict()` □ *`Decimal constructor`*

Browsers only.

Reverts the `Decimal` variable to the value it had before this library was loaded and returns a reference to the original Decimal constructor so it can be assigned to a variable with a different name.

```
<script> Decimal = 1 </script>
<script src='/path/to/decimal.js'></script>
<script>
  a = new Decimal(2)      // '2'
  D = Decimal.noConflict()
  Decimal                 // 1
  b = new D(3)            // '3'
</script>
```

pow `.pow(base, exponent)` □ *`Decimal`*

`base`: *number|string|Decimal*
`exponent`: *number|string|Decimal*

See `toPower`.

```
a = Decimal.pow(x, y)
b = new Decimal(x).pow(y)
a.equals(b)                    // true
```

random `.random([dp])` □ *`Decimal`*

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

Returns a new Decimal 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 `precision` setting.

If the value of `this` Decimal constructor's `crypto` property is `true`, and the `crypto` object is available globally in the host environment, the random digits of the return value are generated by either `crypto.getRandomValues` (Web Cryptography API in modern browsers) or `crypto.randomBytes`

(Node.js), otherwise, if the the value of the property is `false` the return value is generated by `Math.random` (fastest).

To make the `crypto` object available globally in Node.js use

```
global.crypto = require('crypto')
```

If the value of `this` Decimal constructor's `crypto` property is set `true` and the `crypto` object and associated method are not available, an exception will be thrown.

If one of the `crypto` methods is used, the value of the returned Decimal should be cryptographically-secure and statistically indistinguishable from a random value.

```
Decimal.set({ precision: 10 })
Decimal.random()                    // '0.4117936847'
Decimal.random(20)                  // '0.78193327636914089009'
```

round `.round(x)` □ *`Decimal`*

`x`: *number|string|Decimal*

See `round`.

```
a = Decimal.round(x)
b = new Decimal(x).round()
a.equals(b)                    // true
```

set `.set(object)` □ *`Decimal constructor`*

`object`: *object*

Configures the 'global' settings for `this` particular Decimal constructor, i.e. the settings which apply to operations performed on the Decimal instances created by it.

Returns `this` Decimal constructor.

The configuration object, `object`, can contain some or all of the properties described in detail at Properties and shown in the example below.

The values of the configuration object properties are checked for validity and then stored as equivalently-named properties of `this` Decimal constructor.

If `object` has a `'defaults'` property with value `true` then any unspecified properties will be reset to their default values.

Throws on an invalid `object` or configuration property value.

```
// Defaults
Decimal.set({
    precision: 20,
    rounding: 4,
    toExpNeg: -7,
    toExpPos: 21,
    maxE: 9e15,
    minE: -9e15,
    modulo: 1,
    crypto: false
})

// Reset all properties to their default values
Decimal.set({ defaults: true })

// Set precision to 50 and all other properties to their default values
Decimal.set({ precision: 50, defaults: true })
```

The properties of a Decimal constructor can also be set by direct assignment, but that will by-pass the validity checking that this method performs - this is not a problem if the user knows that the assignment is valid.

```
Decimal.precision = 40
```

sign `.sign(x)` □ *number*

`x`: *number|string|Decimal*

Returns	
`1`	if the value of `x` is non-zero and its sign is positive
`-1`	if the value of `x` is non-zero and its sign is negative
`0`	if the value of `x` is positive zero
`-0`	if the value of `x` is negative zero
`NaN`	if the value of `x` is `NaN`

```
r = Decimal.sign(x)
```

sin `.sin(x)` □ *Decimal*

`x`: *number|string|Decimal*

See `sine`.

```
a = Decimal.sin(x)
b = new Decimal(x).sin()
a.equals(b)                    // true
```

sinh `.sinh(x)` □ *Decimal*

`x`: *number|string|Decimal*

See `hyperbolicSine`.

```
a = Decimal.sinh(x)
b = new Decimal(x).sinh()
a.equals(b)                    // true
```

sqrt `.sqrt(x)` □ *Decimal*

`x`: *number|string|Decimal*

See squareRoot.

```
a = Decimal.sqrt(x)
b = new Decimal(x).sqrt()
a.equals(b)                    // true
```

sub .sub(x, y) □ *Decimal*

x: *number|string|Decimal*
y: *number|string|Decimal*

See `minus`.

```
a = Decimal.sub(x, y)
b = new Decimal(x).sub(y)
a.equals(b)                    // true
```

tan .tan(x) □ *Decimal*

x: *number|string|Decimal*

See `tangent`.

```
a = Decimal.tan(x)
b = new Decimal(x).tan()
a.equals(b)                    // true
```

tanh .tanh(x) □ *Decimal*

x: *number|string|Decimal*

See `hyperbolicTangent`.

```
a = Decimal.tanh(x)
b = new Decimal(x).tanh()
a.equals(b)                    // true
```

trunc .trunc(x) □ *Decimal*

x: *number|string|Decimal*

See `truncated`.

```
a = Decimal.trunc(x)
b = new Decimal(x).trunc()
a.equals(b)                    // true
```

Properties

The properties of a Decimal constructor.

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

Configuration properties

The values of the configuration properties `precision`, `rounding`, `minE`, `maxE`, `toExpNeg`, `toExpPos`, `modulo`, and `crypto` are set using the `set` method.

As simple object properties they can be set directly without using `set`, and it is fine to do so, but the values assigned will not then be checked for validity. For example:

```
Decimal.set({ precision: 0 })
// '[DecimalError] Invalid argument: precision: 0'

Decimal.precision = 0
// No error is thrown and the results of calculations are unreliable
```

precision

number: integer, `1` to `1e+9` inclusive
Default value: `20`

The *maximum* number of significant digits of the result of an operation.

All functions which return a Decimal will round the return value to `precision` significant digits except `Decimal`, `absoluteValue`, `ceil`, `floor`, `negated`, `round`, `toDecimalPlaces`, `toNearest` and `truncated`.

See `Pi` for the precision limit of the trigonometric methods.

```
Decimal.set({ precision: 5 })
Decimal.precision                  // 5
```

rounding

number: integer, `0` to `8` inclusive
Default value: `4` `(ROUND_HALF_UP)`

The default rounding mode used when rounding the result of an operation to `precision` significant digits, and when rounding the return value of the `round`, `toBinary`, `toDecimalPlaces`, `toExponential`, `toFixed`, `toHexadecimal`, `toNearest`, `toOctal`, `toPrecision` and `toSignificantDigits` methods.

The rounding modes are available as enumerated properties of the constructor.

```
Decimal.set({ rounding: Decimal.ROUND_UP })
Decimal.set({ rounding: 0 })       // equivalent
Decimal.rounding                   // 0
```

minE

number: integer, `-9e15` to `0` inclusive
Default value: `-9e15`

The negative exponent limit, i.e. the exponent value below which underflow to zero occurs.

If the Decimal to be returned by a calculation would have an exponent lower than minE then the value of that Decimal becomes zero.

JavaScript numbers underflow to zero for exponents below -324.

```
Decimal.set({ minE: -500 })
Decimal.minE                       // -500
new Decimal('1e-500')              // '1e-500'
new Decimal('9.9e-501')            // '0'

Decimal.set({ minE: -3 })
new Decimal(0.001)                 // '0.01'       e is -3
new Decimal(0.0001)                // '0'          e is -4
```

The smallest possible magnitude of a non-zero Decimal is 1e-9000000000000000

maxE

number: integer, 0 to 9e15 inclusive
Default value: 9e15

The positive exponent limit, i.e. the exponent value above which overflow to Infinity occurs.

If the Decimal to be returned by a calculation would have an exponent higher than maxE then the value of that Decimal becomes Infinity.

JavaScript numbers overflow to Infinity for exponents above 308.

```
Decimal.set({ maxE: 500 })
Decimal.maxE                       // 500
new Decimal('9.999e500')           // '9.999e+500'
new Decimal('1e501')               // 'Infinity'

Decimal.set({ maxE: 4 })
new Decimal(99999)                 // '99999'      e is 4
new Decimal(100000)                // 'Infinity'
```

The largest possible magnitude of a finite Decimal is 9.999...e+9000000000000000

toExpNeg

number: integer, -9e15 to 0 inclusive
Default value: -7

The negative exponent value at and below which toString returns exponential notation.

```
Decimal.set({ toExpNeg: -7 })
Decimal.toExpNeg                   // -7
new Decimal(0.00000123)            // '0.00000123'       e is -6
new Decimal(0.000000123)           // '1.23e-7'

// Always return exponential notation:
Decimal.set({ toExpNeg: 0 })
```

JavaScript numbers use exponential notation for negative exponents of -7 and below.

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

Regardless of the value of `toExpNeg`, the `toFixed` method will always return a value in normal notation and the `toExponential` method will always return a value in exponential form.

toExpPos

number: integer, `0` to `9e15` inclusive
Default value: `20`

The positive exponent value at and above which `toString` returns exponential notation.

```
Decimal.set({ toExpPos: 2 })
Decimal.toExpPos                   // 2
new Decimal(12.3)                  // '12.3'        e is 1
new Decimal(123)                   // '1.23e+2'

// Always return exponential notation:
Decimal.set({ toExpPos: 0 })
```

JavaScript numbers use exponential notation for positive exponents of `20` and above.

Regardless of the value of `toExpPos`, the `toFixed` method will always return a value in normal notation and the `toExponential` method will always return a value in exponential form.

modulo

number: integer, `0` to `9` inclusive
Default value: `1` (`ROUND_DOWN`)

The modulo mode used when calculating the modulus: `a mod n`.

The quotient, `q = a / n`, is calculated according to the `rounding` mode 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, else 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(x) * floor(a / abs(x))`.

The rounding/modulo modes are available as enumerated properties of the Decimal constructor.

```
Decimal.set({ modulo: Decimal.EUCLID })
Decimal.set({ modulo: 9 })         // equivalent
Decimal.modulo                     // 9
```

crypto

boolean: true/false
Default value: false

The value that determines whether cryptographically-secure pseudo-random number generation is used.

See random.

```
// Node.js
global.crypto = require('crypto')

Decimal.crypto                     // false
Decimal.set({ crypto: true })
Decimal.crypto                     // true
```

Rounding modes

The library's enumerated rounding modes are stored as properties of the Decimal 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
EUCLID	9	Not a rounding mode, see modulo

```
Decimal.set({ rounding: Decimal.ROUND_CEIL })
Decimal.set({ rounding: 2 })       // equivalent
Decimal.rounding                   // 2
```

INSTANCE

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

Methods

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

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

Methods that return a Decimal can be chained:

```
x = new Decimal(2).times('999.999999999999999').dividedBy(4).ceil()
```

Methods do not round their arguments before execution.

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()` □ *`Decimal`*

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

The return value is not affected by the value of the `precision` setting.

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

ceil `.ceil()` □ *`Decimal`*

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

The return value is not affected by the value of the `precision` setting.

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

comparedTo `.cmp(x)` □ *`number`*

`x`: *number|string|Decimal*

Returns	
`1`	if the value of this Decimal is greater than the value of `x`
`-1`	if the value of this Decimal is less than the value of `x`
`0`	if this Decimal and `x` have the same value
`NaN`	if the value of either this Decimal or `x` is `NaN`

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

```
x = new Decimal(Infinity)
y = new Decimal(5)
x.comparedTo(y)                // 1
x.comparedTo(x.minus(1))       // 0
y.cmp(NaN)                     // NaN
```

cosine `.cos()` □ *`Decimal`*

Returns a new Decimal whose value is the cosine of the value in radians of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`-1, 1`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(0.25)
x.cosine()                      // '0.96891242171064478414'
y = new Decimal(-0.25)
y.cos()                         // '0.96891242171064478414'
```

cubeRoot `.cbrt()` □ *`Decimal`*

Returns a new Decimal whose value is the cube root of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

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 Decimal(125)
x.cubeRoot()                    // '5'
y = new Decimal(3)
y.cbrt()                        // '1.4422495703074083823'
```

decimalPlaces `.dp()` □ *`number`*

Returns the number of decimal places, i.e. the number of digits after the decimal point, of the value of this Decimal.

```
x = new Decimal(1.234)
x.decimalPlaces()              // '3'
y = new Decimal(987.654321)
y.dp()                         // '6'
```

dividedBy `.div(x)` □ *`Decimal`*

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the value of this Decimal divided by `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
x = new Decimal(355)
y = new Decimal(113)
```

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto
ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

```
x.dividedBy(y)             // '3.14159292035398230088'
x.div(5)                   // '71'
```

dividedToIntegerBy `.divToInt(x)` □ *`Decimal`*

`x`: *number|string|Decimal*

Return a new Decimal whose value is the integer part of dividing this Decimal by `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
x = new Decimal(5)
y = new Decimal(3)
x.dividedToIntegerBy(y)     // '1'
x.divToInt(0.7)             // '7'
```

equals `.eq(x)` □ *`boolean`*

`x`: *number|string|Decimal*

Returns `true` if the value of this Decimal equals the value of `x`, otherwise returns `false`.
As with JavaScript, `NaN` does not equal `NaN`.

Note: This method uses the `cmp` method internally.

```
0 === 1e-324                     // true
x = new Decimal(0)
x.equals('1e-324')               // false
new Decimal(-0).eq(x)            // true  ( -0 === 0 )

y = new Decimal(NaN)
y.equals(NaN)                    // false
```

floor `.floor()` □ *`Decimal`*

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

The return value is not affected by the value of the `precision` setting.

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

greaterThan `.gt(x)` □ *`boolean`*

`x`: *number|string|Decimal*

Returns `true` if the value of this Decimal is greater than the value of `x`, otherwise returns `false`.

Note: This method uses the `cmp` method internally.

```
0.1 > (0.3 - 0.2)                            // true
x = new Decimal(0.1)
```

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

```
x.greaterThan(Decimal(0.3).minus(0.2))       // false
new Decimal(0).gt(x)                         // false
```

greaterThanOrEqualTo `.gte(x)` □ *`boolean`*

`x`: *number|string|Decimal*

Returns `true` if the value of this Decimal is greater than or equal to the value of `x`, otherwise returns `false`.

Note: This method uses the `cmp` method internally.

```
(0.3 - 0.2) >= 0.1                       // false
x = new Decimal(0.3).minus(0.2)
x.greaterThanOrEqualTo(0.1)              // true
new Decimal(1).gte(x)                    // true
```

hyperbolicCosine `.cosh()` □ *`Decimal`*

Returns a new Decimal whose value is the hyperbolic cosine of the value in radians of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`1, Infinity`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(1)
x.hyperbolicCosine()                     // '1.5430806348152437785'
y = new Decimal(0.5)
y.cosh()                                 // '1.1276259652063807852'
```

hyperbolicSine `.sinh()` □ *`Decimal`*

Returns a new Decimal whose value is the hyperbolic sine of the value in radians of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`-Infinity, Infinity`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(1)
x.hyperbolicSine()                       // '1.1752011936438014569'
y = new Decimal(0.5)
y.sinh()                                 // '0.52109530549374736162'
```

hyperbolicTangent `.tanh()` □ *`Decimal`*

Returns a new Decimal whose value is the hyperbolic tangent of the value in radians of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`-1, 1`]

See Pi for the precision limit of this method.

```
x = new Decimal(1)
x.hyperbolicTangent()                    // '0.7615941559557648881'
y = new Decimal(0.5)
y.tanh()                                 // '0.4621171572600097585'
```

inverseCosine .acos() □ *Decimal*

Returns a new Decimal whose value is the inverse cosine in radians of the value of this Decimal, rounded to precision significant digits using rounding mode rounding.

Domain: [-1, 1]
Range: [0, pi]

See Pi for the precision limit of this method.

```
x = new Decimal(0)
x.inverseCosine()                        // '1.5707963267948966192'
y = new Decimal(0.5)
y.acos()                                 // '1.0471975511965977462'
```

inverseHyperbolicCosine .acosh() □ *Decimal*

Returns a new Decimal whose value is the inverse hyperbolic cosine in radians of the value of this Decimal, rounded to precision significant digits using rounding mode rounding.

Domain: [1, Infinity]
Range: [0, Infinity]

See Pi for the precision limit of this method.

```
x = new Decimal(5)
x.inverseHyperbolicCosine()              // '2.2924316695611776878'
y = new Decimal(50)
y.acosh()                                // '4.6050701709847571595'
```

inverseHyperbolicSine .asinh() □ *Decimal*

Returns a new Decimal whose value is the inverse hyperbolic sine in radians of the value of this Decimal, rounded to precision significant digits using rounding mode rounding.

Domain: [-Infinity, Infinity]
Range: [-Infinity, Infinity]

See Pi for the precision limit of this method.

```
x = new Decimal(5)
x.inverseHyperbolicSine()                // '2.3124383412727526203'
y = new Decimal(50)
y.asinh()                                // '4.6052701709914238266'
```

inverseHyperbolicTangent .atanh() □ *Decimal*

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto
ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

Returns a new Decimal whose value is the inverse hyperbolic tangent in radians of the value of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-1, 1`]
Range: [`-Infinity, Infinity`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(0.5)
x.inverseHyperbolicTangent()             // '0.5493061443340548457'
y = new Decimal(0.75)
y.atanh()                                // '0.97295507452765665255'
```

inverseSine `.asin()` ⇒ *`Decimal`*

Returns a new Decimal whose value is the inverse sine in radians of the value of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-1, 1`]
Range: [`-pi/2, pi/2`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(0.5)
x.inverseSine()                          // '0.52359877559829887308'
y = new Decimal(0.75)
y.asin()                                 // '0.84806207898148100805'
```

inverseTangent `.atan()` ⇒ *`Decimal`*

Returns a new Decimal whose value is the inverse tangent in radians of the value of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`-pi/2, pi/2`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(0.5)
x.inverseTangent()                       // '0.46364760900080611621'
y = new Decimal(0.75)
y.atan()                                 // '0.6435011087932843868'
```

isFinite `.isFinite()` ⇒ *`boolean`*

Returns `true` if the value of this Decimal is a finite number, otherwise returns `false`.
The only possible non-finite values of a Decimal are `NaN`, `Infinity` and `-Infinity`.

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

Note: The native method `isFinite()` can be used if `n <= Number.MAX_VALUE`.

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

isInteger `.isInt()` □ *boolean*

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

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

isNaN `.isNaN()` □ *boolean*

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

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

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

isNegative `.isNeg()` □ *boolean*

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

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

Note: `n < 0` can be used if `n <= -Number.MIN_VALUE`.

Also note that signed zeroes are implemented, following the IEEE Standard for Floating-Point Arithmetic (IEEE 754).

```
Decimal(0).valueOf()                 // '0'
Decimal(0).isNegative()              // false
Decimal(0).negated().valueOf()       // '-0'
Decimal(0).negated().isNegative()    // true
```

isPositive `.isPos()` □ *boolean*

Returns `true` if the value of this Decimal is positive, otherwise returns `false`.

```
x = new Decimal(0)
x.isPositive()                           // true
y = new Decimal(-2)
y.isPos                                  // false
```

Note: `n < 0` can be used if `n <= -Number.MIN_VALUE`.

isZero `.isZero()` □ *boolean*

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto
ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

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

```
x = new Decimal(-0)
x.isZero() && x.isNeg()                 // true
y = new Decimal(Infinity)
y.isZero()                              // false
```

Note: `n == 0` can be used if `n >= Number.MIN_VALUE`.

lessThan `.lt(x)` □ *`boolean`*

`x`: *number|string|Decimal*

Returns `true` if the value of this Decimal is less than the value of `x`, otherwise returns `false`.

Note: This method uses the `cmp` method internally.

```
(0.3 - 0.2) < 0.1                       // true
x = new Decimal(0.3).minus(0.2)
x.lessThan(0.1)                         // false
new Decimal(0).lt(x)                    // true
```

lessThanOrEqualTo `.lte(x)` □ *`boolean`*

`x`: *number|string|Decimal*

Returns `true` if the value of this Decimal is less than or equal to the value of `x`, otherwise returns `false`.

Note: This method uses the `cmp` method internally.

```
0.1 <= (0.3 - 0.2)                             // false
x = new Decimal(0.1)
x.lessThanOrEqualTo(Decimal(0.3).minus(0.2))   // true
new Decimal(-1).lte(x)                         // true
```

logarithm `.log(x)` □ *`Decimal`*

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the base `x` logarithm of the value of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

If `x` is omitted, the base 10 logarithm of the value of this Decimal will be returned.

```
x = new Decimal(1000)
x.logarithm()                           // '3'
y = new Decimal(256)
y.log(2)                                // '8'
```

The return value will *almost always* be correctly rounded, i.e. rounded as if the result was first calculated to an infinite number of correct digits before rounding. If a result is incorrectly rounded the maximum error will be `1` *ulp* (unit in the last place).

Logarithms to base `2` or `10` will always be correctly rounded.

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto
ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

See `toPower` for the circumstances in which this method may return an incorrectly rounded result, and see `naturalLogarithm` for the precision limit.

The performance of this method degrades exponentially with increasing digits.

minus `.minus(x) □ Decimal`

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the value of this Decimal minus `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
0.3 - 0.1                                 // 0.19999999999999998
x = new Decimal(0.3)
x.minus(0.1)                              // '0.2'
```

modulo `.mod(x) □ Decimal`

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the value of this Decimal modulo `x`, rounded to `precision` significant digits using rounding mode `rounding`.

The value returned, and in particular its sign, is dependent on the value of the `modulo` property of this Decimal's constructor. If it is `1` (default value), the result will have the same sign as this Decimal, and it will match that of Javascript's `%` operator (within the limits of double precision) and BigDecimal's `remainder` method.

See `modulo` for a description of the other modulo modes.

```
1 % 0.9                                   // 0.09999999999999998
x = new Decimal(1)
x.modulo(0.9)                             // '0.1'

y = new Decimal(8)
z = new Decimal(-3)
Decimal.modulo = 1
y.mod(z)                                  // '2'
Decimal.modulo = 3
y.mod(z)                                  // '-1'
```

naturalExponential `.exp() □ Decimal`

Returns a new Decimal whose value is the base `e` (Euler's number, the base of the natural logarithm) exponential of the value of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

The `naturalLogarithm` function is the inverse of this function.

```
x = new Decimal(1)
x.naturalExponential()                    // '2.7182818284590452354'
y = new Decimal(2)
y.exp()                                   // '7.3890560989306502272'
```

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

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. (The mathematical result of the exponential function is non-terminating, unless its argument is `0`).

The performance of this method degrades exponentially with increasing digits.

naturalLogarithm `.ln()` □ *`Decimal`*

Returns a new Decimal whose value is the natural logarithm of the value of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

The natural logarithm is the inverse of the `naturalExponential` function.

```
x = new Decimal(10)
x.naturalLogarithm()                    // '2.3026'
y = new Decimal('1.23e+30')
y.ln()                                  // '69.28'
```

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. (The mathematical result of the natural logarithm function is non-terminating, unless its argument is `1`).

Internally, this method is dependent on a constant whose value is the natural logarithm of `10`. This `LN10` variable in the source code currently has a precision of `1025` digits, meaning that this method can accurately calculate up to `1000` digits.

If more than `1000` digits is required then the precision of `LN10` will need to be increased to `25` digits more than is required - though, as the time-taken by this method increases exponentially with increasing digits, it is unlikely to be viable to calculate over `1000` digits anyway.

negated `.neg()` □ *`Decimal`*

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

The return value is not affected by the value of the `precision` setting.

```
x = new Decimal(1.8)
x.negated()                             // '-1.8'
y = new Decimal(-1.3)
y.neg()                                 // '1.3'
```

plus `.plus(x)` □ *`Decimal`*

`x`: *number|string|Decimal*

Returns a new Decimal whose value is the value of this Decimal plus `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
0.1 + 0.2                               // 0.30000000000000004
x = new Decimal(0.1)
y = x.plus(0.2)                         // '0.3'
new Decimal(0.7).plus(x).plus(y)        // '1.1'
```

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

precision `.sd([include_zeros])` □ *`number`*

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

If `include_zeros` 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 Decimal(1.234)
x.precision()                            // '4'
y = new Decimal(987000)
y.sd()                                   // '3'
y.sd(true)                               // '6'
```

round `.round()` □ *`Decimal`*

Returns a new Decimal whose value is the value of this Decimal rounded to a whole number using rounding mode `rounding`.

To emulate `Math.round`, set `rounding` to `7`, i.e. `ROUND_HALF_CEIL`.

```
Decimal.set({ rounding: 4 })
x = 1234.5
x.round()                                // '1235'

Decimal.rounding = Decimal.ROUND_DOWN
x.round()                                // '1234'
x                                        // '1234.5'
```

sine `.sin()` □ *`Decimal`*

Returns a new Decimal whose value is the sine of the value in radians of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`-1, 1`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(0.5)
x.sine()                                 // '0.47942553860420300027'
y = new Decimal(0.75)
y.sin()                                  // '0.68163876002333416673'
```

squareRoot `.sqrt()` □ *`Decimal`*

Returns a new Decimal whose value is the square root of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

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.

This method is much faster than using the `toPower` method with an exponent of `0.5`.

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

```
x = new Decimal(16)
x.squareRoot()                           // '4'
y = new Decimal(3)
y.sqrt()                                 // '1.7320508075688772935'
y.sqrt().eq( y.pow(0.5) )                // true
```

tangent `.tan()` □ *Decimal*

Returns a new Decimal whose value is the tangent of the value in radians of this Decimal, rounded to `precision` significant digits using rounding mode `rounding`.

Domain: [`-Infinity, Infinity`]
Range: [`-Infinity, Infinity`]

See `Pi` for the precision limit of this method.

```
x = new Decimal(0.5)
x.tangent()                              // '0.54630248984379051326'
y = new Decimal(0.75)
y.tan()                                  // '0.93159645994407246117'
```

times `.times(x)` □ *Decimal*

`x`: *number*|*string*|*Decimal*

Returns a new Decimal whose value is the value of this Decimal times `x`, rounded to `precision` significant digits using rounding mode `rounding`.

```
0.6 * 3                                  // 1.7999999999999998
x = new Decimal(0.6)
y = x.times(3)                           // '1.8'
new Decimal('7e+500').times(y)           // '1.26e+501'
```

toBinary `.toBinary([sd [, rm]])` □ *string*

`sd`: *number*: integer, `0` to `1e+9` inclusive
`rm`: *number*: integer, `0` to `8` inclusive

Returns a string representing the value of this Decimal in binary, rounded to `sd` significant digits using rounding mode `rm`.

If `sd` is defined, the return value will use binary exponential notation.

If `sd` is omitted, the return value will be rounded to `precision` significant digits.

If `rm` is omitted, rounding mode `rounding` will be used.

Throws on an invalid `sd` or `rm` value.

```
x = new Decimal(256)
x.toBinary()                             // '0b100000000'
x.toBinary(1)                            // '0b1p+8'
```

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

toDecimalPlaces `.toDP([dp [, rm]])` □ *`Decimal`*

`dp`: *number*: integer, `0` to `1e+9` inclusive
`rm`: *number*: integer, `0` to `8` inclusive.

Returns a new Decimal whose value is the value of this Decimal rounded to `dp` decimal places using rounding mode `rm`.

If `dp` is omitted, the return value will have the same value as this Decimal.

If `rm` is omitted, rounding mode `rounding` is used.

Throws on an invalid `dp` or `rm` value.

```
x = new Decimal(12.34567)
x.toDecimalPlaces(0)                      // '12'
x.toDecimalPlaces(1, Decimal.ROUND_UP)    // '12.4'

y = new Decimal(9876.54321)
y.toDP(3)                           // '9876.543'
y.toDP(1, 0)                        // '9876.6'
y.toDP(1, Decimal.ROUND_DOWN)       // '9876.5'
```

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 Decimal 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 Decimal in exponential notation has fewer than `dp` fraction digits, the return value will be appended with zeros accordingly.

If `dp` is omitted, 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, rounding mode `rounding` is used.

Throws on an invalid `dp` or `rm` value.

```
x = 45.6
y = new Decimal(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, Decimal.ROUND_DOWN)   // '4.5e+1'
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 Decimal in normal (fixed-point) notation rounded to `dp` decimal places using rounding mode `rm`.

If the value of this Decimal 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 10^{21}, this method will always return normal notation.

If `dp` is omitted, the return value will be unrounded and in normal notation. This is unlike `Number.prototype.toFixed`, which returns the value to zero decimal places, but is useful when because of the current `toExpNeg` or `toExpNeg` values, `toString` returns exponential notation.

If `rm` is omitted, rounding mode `rounding` is used.

Throws on an invalid `dp` or `rm` value.

```
x = 3.456
y = new Decimal(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, Decimal.ROUND_DOWN)  // '3.45'
x.toFixed(5)                      // '3.45600'
y.toFixed(5)                      // '3.45600'
```

toFraction `.toFraction([max_denominator]) ⇒ [Decimal, Decimal]`

`max_denominator`: *number|string|Decimal*: `1` >= integer < `Infinity`

Returns an array of two Decimals representing the value of this Decimal 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_denominator`.

If a maximum denominator is omitted, the denominator will be the lowest value necessary to represent the number exactly.

Throws on an invalid `max_denominator` value.

```
x = new Decimal(1.75)
x.toFraction()                       // '7, 4'

pi = new Decimal('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'
```

toHexadecimal `.toHex([sd [, rm]])` □ *string*

`sd`: *number*: integer, `0` to `1e+9` inclusive
`rm`: *number*: integer, `0` to `8` inclusive

Returns a string representing the value of this Decimal in hexadecimal, rounded to `sd` significant digits using rounding mode `rm`.

If `sd` is defined, the return value will use binary exponential notation.

If `sd` is omitted, the return value will be rounded to `precision` significant digits.

If `rm` is omitted, rounding mode `rounding` will be used.

Throws on an invalid `sd` or `rm` value.

```
x = new Decimal(256)
x.toHexadecimal()                        // '0x100'
x.toHex(1)                               // '0x1p+8'
```

toJSON `.toJSON()` □ *string*

As `valueOf`.

toNearest `.toNearest(x [, rm])` □ *Decimal*

`x`: *number|string|Decimal*
`rm`: *number*: integer, `0` to `8` inclusive

Returns a new Decimal whose value is the nearest multiple of `x` in the direction of rounding mode `rm`, or `rounding` if `rm` is omitted, to the value of this Decimal.

The return value will always have the same sign as this Decimal, unless either this Decimal or `x` is `NaN`, in which case the return value will be also be `NaN`.

The return value is not affected by the value of the `precision` setting.

```
x = new Decimal(1.39)
x.toNearest(0.25)                        // '1.5'

y = new Decimal(9.499)
y.toNearest(0.5, Decimal.ROUND_UP)       // '9.5'
y.toNearest(0.5, Decimal.ROUND_DOWN)     // '9'
```

toNumber `.toNumber()` □ *number*

Returns the value of this Decimal converted to a primitive number.

Type coercion with, for example, JavaScript's unary plus operator will also work, except that a Decimal with the value minus zero will convert to positive zero.

```
x = new Decimal(456.789)
x.toNumber()                   // 456.789
+x                             // 456.789
```

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

```
y = new Decimal('45987349857634085409857349856430985')
y.toNumber()                  // 4.598734985763409e+34

z = new Decimal(-0)
1 / +z                        // Infinity
1 / z.toNumber()              // -Infinity
```

toOctal `.toOctal([sd [, rm]])` *string*

`sd`: *number*: integer, `0` to `1e+9` inclusive
`rm`: *number*: integer, `0` to `8` inclusive

Returns a string representing the value of this Decimal in octal, rounded to `sd` significant digits using rounding mode `rm`.

If `sd` is defined, the return value will use binary exponential notation.

If `sd` is omitted, the return value will be rounded to `precision` significant digits.

If `rm` is omitted, rounding mode `rounding` will be used.

Throws on an invalid `sd` or `rm` value.

```
x = new Decimal(256)
x.toOctal()                             // '0o400'
x.toOctal(1)                            // '0o1p+8'
```

toPower `.pow(x)` *Decimal*

`x`: *number|string|Decimal*: integer or non-integer

Returns a new Decimal whose value is the value of this Decimal raised to the power `x`, rounded to `precision` significant digits using rounding mode `rounding`.

The performance of this method degrades exponentially with increasing digits. For non-integer exponents in particular, the performance of this method may not be adequate.

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

new Decimal(1217652.23).pow('98765.489305603941')
// '4.8227010515242461181e+601039'
```

Is the pow function guaranteed to be correctly rounded?

The return value will *almost always* be correctly rounded, i.e. rounded as if the result was first calculated to an infinite number of correct digits before rounding. If a result is incorrectly rounded the maximum error will be `1` *ulp* (unit in the last place).

For non-integer and larger exponents this method uses the formula

$$x^y = \exp(y*\ln(x))$$

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

As the mathematical return values of the `exp` and `ln` functions are both non-terminating (excluding arguments of `0` or `1`), the values of the Decimals returned by the functions as implemented by this library will necessarily be rounded approximations, which means that there can be no guarantee of correct rounding when they are combined in the above formula.

The return value may, depending on the rounding mode, be incorrectly rounded only if the first `15` rounding digits are `15` zeros (and there are non-zero digits following at some point), or `15` nines, or a `5` or `4` followed by `14` nines.

Therefore, assuming the first `15` rounding digits are each equally likely to be any digit, `0-9`, the probability of an incorrectly rounded result is less than `1` in `250,000,000,000,000`.

An example of incorrect rounding:

```
Decimal.set({ precision: 20, rounding: 1 })
new Decimal(28).pow('6.166675020000903537297764507632802193308677149')
// 839756321.64088511
```

As the exact mathematical result begins

```
839756321.64088510999999999999999999999999999998969466049426031167...
```

and the rounding mode is set to `ROUND_DOWN`, the correct return value should be

```
839756321.64088510999
```

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 Decimal 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, the return value is the same as `toString`.

If `rm` is omitted, rounding mode `rounding` is used.

Throws on an invalid `sd` or `rm` value.

```
x = 45.6
y = new Decimal(x)
x.toPrecision()                         // '45.6'
y.toPrecision()                         // '45.6'
x.toPrecision(1)                        // '5e+1'
y.toPrecision(1)                        // '5e+1'
y.toPrecision(2, Decimal.ROUND_UP)      // '46'
y.toPrecision(2, Decimal.ROUND_DOWN)    // '45'
x.toPrecision(5)                        // '45.600'
y.toPrecision(5)                        // '45.600'
```

toSignificantDigits `.toSD([sd [, rm]])` □ *Decimal*

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

`sd`: *number*: integer, `1` to `1e+9` inclusive.
`rm`: *number*: integer, `0` to `8` inclusive.

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

If `sd` is omitted, the return value will be rounded to `precision` significant digits.

If `rm` is omitted, rounding mode `rounding` will be used.

Throws on an invalid `sd` or `rm` value.

```
Decimal.set({ precision: 5, rounding: 4 })
x = new Decimal(9876.54321)

x.toSignificantDigits()                          // '9876.5'
x.toSignificantDigits(6)                         // '9876.54'
x.toSignificantDigits(6, Decimal.ROUND_UP)       // '9876.55'
x.toSD(2)                                        // '9900'
x.toSD(2, 1)                                     // '9800'
x                                                // '9876.54321'
```

toString `.toString()` □ *string*

Returns a string representing the value of this Decimal.

If this Decimal has a positive exponent that is equal to or greater than `toExpPos`, or a negative exponent equal to or less than `toExpNeg`, then exponential notation will be returned.

```
x = new Decimal(750000)
x.toString()                             // '750000'
Decimal.set({ toExpPos: 5 })
x.toString()                             // '7.5e+5'

Decimal.set({ precision: 4 })
y = new Decimal('1.23456789')
y.toString()                             // '1.23456789'
```

truncated `.trunc()` □ *Decimal*

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

The return value is not affected by the value of the `precision` setting.

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

valueOf `.valueOf()` □ *string*

As `toString`, but zero is signed.

```
x = new Decimal(-0)
x.valueOf()                              // '-0'
```

Properties

The value of a Decimal is stored in a normalised base `10000000` floating point format.

A Decimal instance is an object with three properties:

Property	Description	Type	Value
d	digits	*number*`[]`	Array of integers, each `0` - `1e7`, or `null`
e	exponent	*number*	Integer, `-9e15` to `9e15` inclusive, or `NaN`
s	sign	*number*	`-1`, `1`, or `NaN`

All the properties are best considered to be read-only.

As with JavaScript numbers, the original exponent and fractional trailing zeros of a value are not preserved.

```
x = new Decimal(0.123)                   // '0.123'
x.toExponential()                        // '1.23e-1'
x.d                                      // [ 1230000 ]
x.e                                      // -1
x.s                                      // 1

y = new Number(-123.4567000e+2)          // '-12345.67'
y.toExponential()                        // '-1.234567e+4'
z = new Decimal('-123.4567000e+2')       // '-12345.67'
z.toExponential()                        // '-1.234567e+4'
z.d                                      // [ 12345, 6700000 ]
z.e                                      // 4
z.s                                      // -1
```

Zero, NaN and Infinity

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

	±0	NaN	±Infinity
d	`[0]`	`null`	`null`
e	`0`	`NaN`	`NaN`
s	`±1`	`NaN`	`±1`

```
x = new Number(-0)                       // 0
1 / x == -Infinity                       // true

y = new Decimal(-0)
y.d                                      // '0' ( [0].toString() )
y.e                                      //  0
```

```
y.s                                       // -1
y.toString()                              // '0'
y.valueOf()                               // '-0'
```

Errors

The errors that are thrown are generic `Error` objects whose `message` property begins with `"[DecimalError]"`.

To determine if an exception is a Decimal Error:

```
try {
    // ...
} catch (e) {
    if ( e instanceof Error && /DecimalError/.test(e.message) ) {
        // ...
    }
}
```

Pi

The maximum precision of the trigonometric methods is dependent on the internal value of the constant pi, which is defined as the string `PI` near the top of the source file.

It has a precision of `1025` digits, meaning that the trigonometric methods can calculate up to just over `1000` digits, but the actual figure depends on the precision of the argument passed to them. To calculate the actual figure use:

maximum_result_precision = 1000 - argument_precision

For example, the following both work fine:

```
Decimal.set({precision: 991}).tan(123456789)
Decimal.set({precision: 9}).tan(991_digit_number)
```

as, for each, the result precision plus the argument precision, i.e. `991 + 9` and `9 + 991`, is less than or equal to `1000`.

If greater precision is required then the value of `PI` will need to be extended to about `25` digits more than the precision required. The time taken by the methods will then be the limiting factor.

The value can also be shortened to reduce the size of the source file if such high precision is not required.

To get the value of pi:

```
pi = Decimal.acos(-1)
```

FAQ

API

CONSTRUCTOR

Decimal

Methods

abs
acos
acosh
add
asin
asinh
atan
atanh
atan2
cbrt
ceil
clone
cos
cosh
div
exp
floor
hypot
isDecimal
ln
log
log2
log10
max
min
mod
mul
noConflict
pow
random
round
set
sign
sin
sinh
sqrt
sub
tan
tanh
trunc

Properties

precision
rounding
minE
maxE
toExpNeg
toExpPos
modulo
crypto

ROUND_UP
ROUND_DOWN
ROUND_CEIL
ROUND_FLOOR
ROUND_HALF_UP
ROUND_HALF_DOWN

Why are trailing fractional zeros removed from Decimals?

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.