Overview Package Class Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 7

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.math

Class BigDecimal

java.lang.Object
  java.lang.Number
    java.math.BigDecimal

All Implemented Interfaces:

Serializable, Comparable<BigDecimal>

```
public class BigDecimal
extends Number
implements Comparable<BigDecimal>
```

Immutable, arbitrary-precision signed decimal numbers. A `BigDecimal` consists of an arbitrary precision integer *unscaled value* and a 32-bit integer *scale*. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value of the number is multiplied by ten to the power of the negation of the scale. The value of the number represented by the `BigDecimal` is therefore $(\mathtt{unscaledValue} \times 10^{\mathtt{-scale}})$.

The `BigDecimal` class provides operations for arithmetic, scale manipulation, rounding, comparison, hashing, and format conversion. The `toString()` method provides a canonical representation of a `BigDecimal`.

The `BigDecimal` class gives its user complete control over rounding behavior. If no rounding mode is specified and the exact result cannot be represented, an exception is thrown; otherwise, calculations can be carried out to a chosen precision and rounding mode by supplying an appropriate `MathContext` object to the operation. In either case, eight *rounding modes* are provided for the control of rounding. Using the integer fields in this class (such as `ROUND_HALF_UP`) to represent rounding mode is largely obsolete; the enumeration values of the `RoundingMode` `enum`, (such as `RoundingMode.HALF_UP`) should be used instead.

When a `MathContext` object is supplied with a precision setting of 0 (for example, `MathContext.UNLIMITED`), arithmetic operations are exact, as are the arithmetic methods which take no `MathContext` object. (This is the only behavior that was supported in releases prior to 5.) As a corollary of computing the exact result, the rounding mode setting of a `MathContext` object with a precision setting of 0 is not used and thus irrelevant. In the case of divide, the exact quotient could have an infinitely long decimal expansion; for example, 1 divided by 3. If the quotient has a nonterminating decimal expansion and the operation is specified to return an exact result, an `ArithmeticException` is thrown. Otherwise, the exact result of the division is returned, as done for other operations.

When the precision setting is not 0, the rules of `BigDecimal` arithmetic are broadly compatible with selected modes of operation of the arithmetic defined in ANSI X3.274-1996 and ANSI X3.274-1996/AM 1-2000 (section 7.4). Unlike those standards, `BigDecimal` includes many rounding modes, which were mandatory for division in `BigDecimal` releases prior to 5. Any conflicts between these ANSI standards and the `BigDecimal` specification are resolved in favor of `BigDecimal`.

Since the same numerical value can have different representations (with different scales), the rules of arithmetic and rounding must specify both the numerical result and the scale used in the result's representation.

In general the rounding modes and precision setting determine how operations return results with a limited number of digits when the exact result has more digits (perhaps infinitely many in the case of division) than the number of digits returned. First, the total number of digits to return is specified by the `MathContext`'s `precision` setting; this determines the result's *precision*. The digit count starts from the leftmost nonzero digit of the exact result. The rounding mode determines how any discarded trailing digits affect the returned result.

For all arithmetic operators , the operation is carried out as though an exact intermediate result were first calculated and then rounded to the number of digits specified by the precision setting (if necessary), using the selected rounding mode. If the exact result is not returned, some digit positions of the exact result are discarded. When rounding increases the magnitude of the returned result, it is possible for a new digit position to be created by a carry propagating to a leading "9" digit. For example, rounding the value 999.9 to three digits rounding up would be numerically equal to one thousand, represented as 100×10^{1}. In such cases, the new "1" is the leading digit position of the returned result.

Besides a logical exact result, each arithmetic operation has a preferred scale for representing a result. The preferred scale for each operation is listed in the table below.

Preferred Scales for Results of Arithmetic Operations

Operation	Preferred Scale of Result
Add	max(addend.scale(), augend.scale())
Subtract	max(minuend.scale(), subtrahend.scale())
Multiply	multiplier.scale() + multiplicand.scale()
Divide	dividend.scale() - divisor.scale()

These scales are the ones used by the methods which return exact arithmetic results; except that an exact divide may have to use a larger scale since the exact result may have more digits. For example, `1/32` is `0.03125`.

Before rounding, the scale of the logical exact intermediate result is the preferred scale for that operation. If the exact numerical result cannot be represented in `precision` digits, rounding selects the set of digits to return and the scale of the result is reduced from the scale of the intermediate result to the least scale which can represent the `precision` digits actually returned. If the exact result can be represented with at most `precision` digits, the representation of the result with the scale closest to the preferred scale is returned. In particular, an exactly representable quotient may be represented in fewer than `precision` digits by removing trailing zeros and decreasing the scale. For example, rounding to three digits using the floor rounding mode,
`19/100 = 0.19 // integer=19, scale=2`
but
`21/110 = 0.190 // integer=190, scale=3`

Note that for add, subtract, and multiply, the reduction in scale will equal the number of digit positions of the exact result which are discarded. If the rounding causes a carry propagation to create a new high-order digit position, an additional digit of the result is discarded than when no new digit position is created.

Other methods may have slightly different rounding semantics. For example, the result of the `pow` method using the specified algorithm can occasionally differ from the rounded mathematical result by more than one unit in the last place, one *ulp*.

Two types of operations are provided for manipulating the scale of a `BigDecimal`: scaling/rounding operations and decimal point motion operations. Scaling/rounding operations (`setScale` and `round`) return a `BigDecimal` whose value is approximately (or exactly) equal to that of the operand, but whose scale or precision is the specified value; that is, they increase or decrease the precision of the stored number with minimal effect on its value. Decimal point motion operations (`movePointLeft` and `movePointRight`) return a `BigDecimal` created from the operand by moving the decimal point a specified distance in the specified direction.

For the sake of brevity and clarity, pseudo-code is used throughout the descriptions of `BigDecimal` methods. The pseudo-code expression `(i + j)` is shorthand for "a `BigDecimal` whose value is that of the `BigDecimal` `i` added to that of the `BigDecimal` `j`." The pseudo-code expression `(i == j)` is shorthand for "`true` if and only if the `BigDecimal` `i` represents the same value as the `BigDecimal` `j`." Other pseudo-code expressions are interpreted similarly. Square brackets are used to represent the particular `BigInteger` and scale pair defining a `BigDecimal` value; for example [19, 2] is the `BigDecimal` numerically equal to 0.19 having a scale of 2.

Note: care should be exercised if `BigDecimal` objects are used as keys in a `SortedMap` or elements in a `SortedSet` since `BigDecimal`'s *natural ordering* is *inconsistent with equals*. See `Comparable`, `SortedMap` or `SortedSet` for more information.

All methods and constructors for this class throw `NullPointerException` when passed a `null` object reference for any input parameter.

See Also:

`BigInteger`, `MathContext`, `RoundingMode`, `SortedMap`, `SortedSet`, Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description

static BigDecimal	ONE The value 1, with a scale of 0.
static int	ROUND_CEILING Rounding mode to round towards positive infinity.
static int	ROUND_DOWN Rounding mode to round towards zero.
static int	ROUND_FLOOR Rounding mode to round towards negative infinity.
static int	ROUND_HALF_DOWN Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down.
static int	ROUND_HALF_EVEN Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor.
static int	ROUND_HALF_UP Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up.
static int	ROUND_UNNECESSARY Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary.
static int	ROUND_UP Rounding mode to round away from zero.
static BigDecimal	TEN The value 10, with a scale of 0.
static BigDecimal	ZERO The value 0, with a scale of 0.

Constructor Summary

Constructors

Constructor and Description
BigDecimal(BigInteger val) Translates a BigInteger into a BigDecimal.
BigDecimal(BigInteger unscaledVal, int scale) Translates a BigInteger unscaled value and an int scale into a BigDecimal.
BigDecimal(BigInteger unscaledVal, int scale, MathContext mc) Translates a BigInteger unscaled value and an int scale into a BigDecimal, with rounding according to the context settings.
BigDecimal(BigInteger val, MathContext mc) Translates a BigInteger into a BigDecimal rounding according to the context settings.
BigDecimal(char[] in) Translates a character array representation of a BigDecimal into a BigDecimal, accepting the same sequence of characters as the BigDecimal(String) constructor.
BigDecimal(char[] in, int offset, int len) Translates a character array representation of a BigDecimal into a BigDecimal, accepting the same sequence of characters as the BigDecimal(String) constructor, while allowing a sub-array to be specified.
BigDecimal(char[] in, int offset, int len, MathContext mc) Translates a character array representation of a BigDecimal into a BigDecimal, accepting the same sequence of characters as the BigDecimal(String) constructor, while allowing a sub-array to be specified and with rounding according to the context settings.

BigDecimal(char[] in, **MathContext** mc) Translates a character array representation of a `BigDecimal` into a `BigDecimal`, accepting the same sequence of characters as the **`BigDecimal(String)`** constructor and with rounding according to the context settings.
BigDecimal(double val) Translates a `double` into a `BigDecimal` which is the exact decimal representation of the `double`'s binary floating-point value.
BigDecimal(double val, **MathContext** mc) Translates a `double` into a `BigDecimal`, with rounding according to the context settings.
BigDecimal(int val) Translates an `int` into a `BigDecimal`.
BigDecimal(int val, **MathContext** mc) Translates an `int` into a `BigDecimal`, with rounding according to the context settings.
BigDecimal(long val) Translates a `long` into a `BigDecimal`.
BigDecimal(long val, **MathContext** mc) Translates a `long` into a `BigDecimal`, with rounding according to the context settings.
BigDecimal(**String** val) Translates the string representation of a `BigDecimal` into a `BigDecimal`.
BigDecimal(**String** val, **MathContext** mc) Translates the string representation of a `BigDecimal` into a `BigDecimal`, accepting the same strings as the **`BigDecimal(String)`** constructor, with rounding according to the context settings.

Method Summary

Methods

Modifier and Type	Method and Description
BigDecimal	**abs**() Returns a `BigDecimal` whose value is the absolute value of this `BigDecimal`, and whose scale is `this.scale()`.
BigDecimal	**abs**(**MathContext** mc) Returns a `BigDecimal` whose value is the absolute value of this `BigDecimal`, with rounding according to the context settings.
BigDecimal	**add**(**BigDecimal** augend) Returns a `BigDecimal` whose value is `(this + augend)`, and whose scale is `max(this.scale(), augend.scale())`.
BigDecimal	**add**(**BigDecimal** augend, **MathContext** mc) Returns a `BigDecimal` whose value is `(this + augend)`, with rounding according to the context settings.
byte	**byteValueExact**() Converts this `BigDecimal` to a `byte`, checking for lost information.
int	**compareTo**(**BigDecimal** val) Compares this `BigDecimal` with the specified `BigDecimal`.
BigDecimal	**divide**(**BigDecimal** divisor) Returns a `BigDecimal` whose value is `(this / divisor)`, and whose preferred scale is `(this.scale() - divisor.scale())`; if the exact quotient cannot be represented (because it has a non-terminating decimal expansion) an `ArithmeticException` is thrown.
BigDecimal	**divide**(**BigDecimal** divisor, int roundingMode) Returns a `BigDecimal` whose value is `(this / divisor)`, and whose scale is `this.scale()`.

BigDecimal	divide(BigDecimal divisor, int scale, int roundingMode) Returns a BigDecimal whose value is (this / divisor), and whose scale is as specified.
BigDecimal	divide(BigDecimal divisor, int scale, RoundingMode roundingMode) Returns a BigDecimal whose value is (this / divisor), and whose scale is as specified.
BigDecimal	divide(BigDecimal divisor, MathContext mc) Returns a BigDecimal whose value is (this / divisor), with rounding according to the context settings.
BigDecimal	divide(BigDecimal divisor, RoundingMode roundingMode) Returns a BigDecimal whose value is (this / divisor), and whose scale is this.scale().
BigDecimal[]	divideAndRemainder(BigDecimal divisor) Returns a two-element BigDecimal array containing the result of divideToIntegralValue followed by the result of remainder on the two operands.
BigDecimal[]	divideAndRemainder(BigDecimal divisor, MathContext mc) Returns a two-element BigDecimal array containing the result of divideToIntegralValue followed by the result of remainder on the two operands calculated with rounding according to the context settings.
BigDecimal	divideToIntegralValue(BigDecimal divisor) Returns a BigDecimal whose value is the integer part of the quotient (this / divisor) rounded down.
BigDecimal	divideToIntegralValue(BigDecimal divisor, MathContext mc) Returns a BigDecimal whose value is the integer part of (this / divisor).
double	doubleValue() Converts this BigDecimal to a double.
boolean	equals(Object x) Compares this BigDecimal with the specified Object for equality.
float	floatValue() Converts this BigDecimal to a float.
int	hashCode() Returns the hash code for this BigDecimal.
int	intValue() Converts this BigDecimal to an int.
int	intValueExact() Converts this BigDecimal to an int, checking for lost information.
long	longValue() Converts this BigDecimal to a long.
long	longValueExact() Converts this BigDecimal to a long, checking for lost information.
BigDecimal	max(BigDecimal val) Returns the maximum of this BigDecimal and val.
BigDecimal	min(BigDecimal val) Returns the minimum of this BigDecimal and val.
BigDecimal	movePointLeft(int n) Returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the left.
BigDecimal	movePointRight(int n) Returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the right.
BigDecimal	multiply(BigDecimal multiplicand) Returns a BigDecimal whose value is (this × multiplicand), and whose scale is (this.scale() + multiplicand.scale()).

BigDecimal	**multiply**(**BigDecimal** multiplicand, **MathContext** mc) Returns a BigDecimal whose value is (this × multiplicand), with rounding according to the context settings.
BigDecimal	**negate**() Returns a BigDecimal whose value is (-this), and whose scale is this.scale().
BigDecimal	**negate**(**MathContext** mc) Returns a BigDecimal whose value is (-this), with rounding according to the context settings.
BigDecimal	**plus**() Returns a BigDecimal whose value is (+this), and whose scale is this.scale().
BigDecimal	**plus**(**MathContext** mc) Returns a BigDecimal whose value is (+this), with rounding according to the context settings.
BigDecimal	**pow**(int n) Returns a BigDecimal whose value is (this^{n}), The power is computed exactly, to unlimited precision.
BigDecimal	**pow**(int n, **MathContext** mc) Returns a BigDecimal whose value is (this^{n}).
int	**precision**() Returns the *precision* of this BigDecimal.
BigDecimal	**remainder**(**BigDecimal** divisor) Returns a BigDecimal whose value is (this % divisor).
BigDecimal	**remainder**(**BigDecimal** divisor, **MathContext** mc) Returns a BigDecimal whose value is (this % divisor), with rounding according to the context settings.
BigDecimal	**round**(**MathContext** mc) Returns a BigDecimal rounded according to the MathContext settings.
int	**scale**() Returns the *scale* of this BigDecimal.
BigDecimal	**scaleByPowerOfTen**(int n) Returns a BigDecimal whose numerical value is equal to (this * 10^{n}).
BigDecimal	**setScale**(int newScale) Returns a BigDecimal whose scale is the specified value, and whose value is numerically equal to this BigDecimal's.
BigDecimal	**setScale**(int newScale, int roundingMode) Returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
BigDecimal	**setScale**(int newScale, **RoundingMode** roundingMode) Returns a BigDecimal whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this BigDecimal's unscaled value by the appropriate power of ten to maintain its overall value.
short	**shortValueExact**() Converts this BigDecimal to a short, checking for lost information.
int	**signum**() Returns the signum function of this BigDecimal.
BigDecimal	**stripTrailingZeros**() Returns a BigDecimal which is numerically equal to this one but with any trailing zeros removed from the representation.
BigDecimal	**subtract**(**BigDecimal** subtrahend) Returns a BigDecimal whose value is (this - subtrahend), and whose scale is max(this.scale(), subtrahend.scale()).

`BigDecimal`	`subtract(BigDecimal subtrahend, MathContext mc)` Returns a `BigDecimal` whose value is `(this - subtrahend)`, with rounding according to the context settings.
`BigInteger`	`toBigInteger()` Converts this `BigDecimal` to a `BigInteger`.
`BigInteger`	`toBigIntegerExact()` Converts this `BigDecimal` to a `BigInteger`, checking for lost information.
`String`	`toEngineeringString()` Returns a string representation of this `BigDecimal`, using engineering notation if an exponent is needed.
`String`	`toPlainString()` Returns a string representation of this `BigDecimal` without an exponent field.
`String`	`toString()` Returns the string representation of this `BigDecimal`, using scientific notation if an exponent is needed.
`BigDecimal`	`ulp()` Returns the size of an ulp, a unit in the last place, of this `BigDecimal`.
`BigInteger`	`unscaledValue()` Returns a `BigInteger` whose value is the *unscaled value* of this `BigDecimal`.
`static BigDecimal`	`valueOf(double val)` Translates a `double` into a `BigDecimal`, using the `double`'s canonical string representation provided by the `Double.toString(double)` method.
`static BigDecimal`	`valueOf(long val)` Translates a `long` value into a `BigDecimal` with a scale of zero.
`static BigDecimal`	`valueOf(long unscaledVal, int scale)` Translates a `long` unscaled value and an `int` scale into a `BigDecimal`.

Methods inherited from class java.lang.Number

`byteValue, shortValue`

Methods inherited from class java.lang.Object

`clone, finalize, getClass, notify, notifyAll, wait, wait, wait`

Field Detail

ZERO

```
public static final BigDecimal ZERO
```

The value 0, with a scale of 0.

Since:

1.5

ONE

```
public static final BigDecimal ONE
```

The value 1, with a scale of 0.

Since:

1.5

TEN

```
public static final BigDecimal TEN
```

The value 10, with a scale of 0.

Since:

1.5

ROUND_UP

```
public static final int ROUND_UP
```

Rounding mode to round away from zero. Always increments the digit prior to a nonzero discarded fraction. Note that this rounding mode never decreases the magnitude of the calculated value.

See Also:

Constant Field Values

ROUND_DOWN

```
public static final int ROUND_DOWN
```

Rounding mode to round towards zero. Never increments the digit prior to a discarded fraction (i.e., truncates). Note that this rounding mode never increases the magnitude of the calculated value.

See Also:

Constant Field Values

ROUND_CEILING

```
public static final int ROUND_CEILING
```

Rounding mode to round towards positive infinity. If the `BigDecimal` is positive, behaves as for `ROUND_UP`; if negative, behaves as for `ROUND_DOWN`. Note that this rounding mode never decreases the calculated value.

See Also:

Constant Field Values

ROUND_FLOOR

```
public static final int ROUND_FLOOR
```

Rounding mode to round towards negative infinity. If the `BigDecimal` is positive, behave as for `ROUND_DOWN`; if negative, behave as for `ROUND_UP`. Note that this rounding mode never increases the calculated value.

See Also:

Constant Field Values

ROUND_HALF_UP

```
public static final int ROUND_HALF_UP
```

Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round up. Behaves as for `ROUND_UP` if the discarded fraction is ≥ 0.5; otherwise, behaves as for `ROUND_DOWN`. Note that this is the rounding mode that most of us were taught in grade school.

See Also:

Constant Field Values

ROUND_HALF_DOWN

```
public static final int ROUND_HALF_DOWN
```

Rounding mode to round towards "nearest neighbor" unless both neighbors are equidistant, in which case round down. Behaves as for `ROUND_UP` if the discarded fraction is > 0.5; otherwise, behaves as for `ROUND_DOWN`.

See Also:

Constant Field Values

ROUND_HALF_EVEN

```
public static final int ROUND_HALF_EVEN
```

Rounding mode to round towards the "nearest neighbor" unless both neighbors are equidistant, in which case, round towards the even neighbor. Behaves as for `ROUND_HALF_UP` if the digit to the left of the discarded fraction is odd; behaves as for `ROUND_HALF_DOWN` if it's even. Note that this is the rounding mode that minimizes cumulative error when applied repeatedly over a sequence of calculations.

See Also:

Constant Field Values

ROUND_UNNECESSARY

```
public static final int ROUND_UNNECESSARY
```

Rounding mode to assert that the requested operation has an exact result, hence no rounding is necessary. If this rounding mode is specified on an operation that yields an inexact result, an `ArithmeticException` is thrown.

See Also:

Constant Field Values

Constructor Detail

BigDecimal

```
public BigDecimal(char[] in,
          int offset,
          int len)
```

Translates a character array representation of a `BigDecimal` into a `BigDecimal`, accepting the same sequence of characters as the `BigDecimal(String)` constructor, while allowing a sub-array to be specified.

Note that if the sequence of characters is already available within a character array, using this constructor is faster than converting the `char` array to string and using the `BigDecimal(String)` constructor .

Parameters:

`in` - `char` array that is the source of characters.

`offset` - first character in the array to inspect.

`len` - number of characters to consider.

Throws:

`NumberFormatException` - if `in` is not a valid representation of a `BigDecimal` or the defined subarray is not wholly within `in`.

Since:

1.5

BigDecimal

```
public BigDecimal(char[] in,
          int offset,
          int len,
          MathContext mc)
```

Translates a character array representation of a `BigDecimal` into a `BigDecimal`, accepting the same sequence of characters as the `BigDecimal(String)` constructor, while allowing a sub-array to be specified and with rounding according to the context settings.

Note that if the sequence of characters is already available within a character array, using this constructor is faster than converting the `char` array to string and using the `BigDecimal(String)` constructor .

Parameters:

`in` - `char` array that is the source of characters.

`offset` - first character in the array to inspect.

`len` - number of characters to consider..

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

`NumberFormatException` - if `in` is not a valid representation of a `BigDecimal` or the defined subarray is not wholly within `in`.

Since:

1.5

BigDecimal

```
public BigDecimal(char[] in)
```

Translates a character array representation of a `BigDecimal` into a `BigDecimal`, accepting the same sequence of characters as the `BigDecimal(String)` constructor.

Note that if the sequence of characters is already available as a character array, using this constructor is faster than converting the `char` array to string and using the `BigDecimal(String)` constructor .

Parameters:

`in` - `char` array that is the source of characters.

Throws:

`NumberFormatException` - if `in` is not a valid representation of a `BigDecimal`.

Since:

1.5

BigDecimal

```
public BigDecimal(char[] in,
          MathContext mc)
```

Translates a character array representation of a `BigDecimal` into a `BigDecimal`, accepting the same sequence of characters as the `BigDecimal(String)` constructor and with rounding according to the context settings.

Note that if the sequence of characters is already available as a character array, using this constructor is faster than converting the `char` array to string and using the `BigDecimal(String)` constructor .

Parameters:

`in` - `char` array that is the source of characters.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

`NumberFormatException` - if `in` is not a valid representation of a `BigDecimal`.

Since:

1.5

BigDecimal

```
public BigDecimal(String val)
```

Translates the string representation of a `BigDecimal` into a `BigDecimal`. The string representation consists of an optional sign, `'+'` (`'\u002B'`) or `'-'` (`'\u002D'`), followed by a sequence of zero or more decimal digits ("the integer"), optionally followed by a fraction, optionally followed by an exponent.

The fraction consists of a decimal point followed by zero or more decimal digits. The string must contain at least one digit in either the integer or the fraction. The number formed by the sign, the integer and the fraction is referred to as the *significand*.

The exponent consists of the character `'e'` (`'\u0065'`) or `'E'` (`'\u0045'`) followed by one or more decimal digits. The value of the exponent must lie between -`Integer.MAX_VALUE` (`Integer.MIN_VALUE`+1) and `Integer.MAX_VALUE`, inclusive.

More formally, the strings this constructor accepts are described by the following grammar:

BigDecimalString:

Sign$_{opt}$ *Significand Exponent*$_{opt}$

Sign:

`+`

`-`

Significand:

IntegerPart `.` *FractionPart*$_{opt}$

`.` *FractionPart*

IntegerPart

IntegerPart:

Digits

FractionPart:

Digits

Exponent:

ExponentIndicator SignedInteger

ExponentIndicator:

`e`

`E`

SignedInteger:

Sign$_{opt}$ *Digits*

Digits:

Digit

Digits Digit

Digit:

any character for which `Character.isDigit(char)` returns `true`, including 0, 1, 2 ...

The scale of the returned `BigDecimal` will be the number of digits in the fraction, or zero if the string contains no decimal point, subject to adjustment for any exponent; if the string contains an exponent, the exponent is subtracted from the scale. The value of the resulting scale must lie between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`, inclusive.

The character-to-digit mapping is provided by `Character.digit(char, int)` set to convert to radix 10. The String may not contain any extraneous characters (whitespace, for example).

Examples:
The value of the returned `BigDecimal` is equal to *significand* × 10^{exponent}. For each string on the left, the resulting representation [`BigInteger`, `scale`] is shown on the right.

```
"0"            [0,0]
"0.00"         [0,2]
"123"          [123,0]
"-123"         [-123,0]
"1.23E3"       [123,-1]
"1.23E+3"      [123,-1]
"12.3E+7"      [123,-6]
"12.0"         [120,1]
"12.3"         [123,1]
"0.00123"      [123,5]
"-1.23E-12"    [-123,14]
"1234.5E-4"    [12345,5]
```

```
 "0E+7"         [0,-7]
 "-0"           [0,0]
```

Note: For values other than `float` and `double` NaN and ±Infinity, this constructor is compatible with the values returned by `Float.toString(float)` and `Double.toString(double)`. This is generally the preferred way to convert a `float` or `double` into a BigDecimal, as it doesn't suffer from the unpredictability of the `BigDecimal(double)` constructor.

Parameters:

`val` - String representation of `BigDecimal`.

Throws:

`NumberFormatException` - if `val` is not a valid representation of a `BigDecimal`.

BigDecimal

```
public BigDecimal(String val,
          MathContext mc)
```

Translates the string representation of a `BigDecimal` into a `BigDecimal`, accepting the same strings as the `BigDecimal(String)` constructor, with rounding according to the context settings.

Parameters:

`val` - string representation of a `BigDecimal`.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

`NumberFormatException` - if `val` is not a valid representation of a BigDecimal.

Since:

1.5

BigDecimal

```
public BigDecimal(double val)
```

Translates a `double` into a `BigDecimal` which is the exact decimal representation of the `double`'s binary floating-point value. The scale of the returned `BigDecimal` is the smallest value such that $(10^{scale} \times val)$ is an integer.

Notes:

1. The results of this constructor can be somewhat unpredictable. One might assume that writing `new BigDecimal(0.1)` in Java creates a `BigDecimal` which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257827021181583404541015625. This is because 0.1 cannot be represented exactly as a `double` (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed *in* to the constructor is not exactly equal to 0.1, appearances notwithstanding.
2. The `String` constructor, on the other hand, is perfectly predictable: writing `new BigDecimal("0.1")` creates a `BigDecimal` which is *exactly* equal to 0.1, as one would expect. Therefore, it is generally recommended that the `String` constructor be used in preference to this one.
3. When a `double` must be used as a source for a `BigDecimal`, note that this constructor provides an exact conversion; it does not give the same result as converting the `double` to a `String` using the `Double.toString(double)` method and then using the `BigDecimal(String)` constructor. To get that result, use the `static` `valueOf(double)` method.

Parameters:

`val` - `double` value to be converted to `BigDecimal`.

Throws:

`NumberFormatException` - if `val` is infinite or NaN.

BigDecimal

```
public BigDecimal(double val,
          MathContext mc)
```

Translates a `double` into a `BigDecimal`, with rounding according to the context settings. The scale of the `BigDecimal` is the smallest value such that ($10^{scale} \times val$) is an integer.

The results of this constructor can be somewhat unpredictable and its use is generally not recommended; see the notes under the `BigDecimal(double)` constructor.

Parameters:

`val` - `double` value to be converted to `BigDecimal`.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the RoundingMode is UNNECESSARY.

`NumberFormatException` - if `val` is infinite or NaN.

Since:

1.5

BigDecimal

```
public BigDecimal(BigInteger val)
```

Translates a `BigInteger` into a `BigDecimal`. The scale of the `BigDecimal` is zero.

Parameters:

`val` - `BigInteger` value to be converted to `BigDecimal`.

BigDecimal

```
public BigDecimal(BigInteger val,
          MathContext mc)
```

Translates a `BigInteger` into a `BigDecimal` rounding according to the context settings. The scale of the `BigDecimal` is zero.

Parameters:

`val` - `BigInteger` value to be converted to `BigDecimal`.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

BigDecimal

```
public BigDecimal(BigInteger unscaledVal,
          int scale)
```

Translates a `BigInteger` unscaled value and an `int` scale into a `BigDecimal`. The value of the `BigDecimal` is `(unscaledVal × 10`$^{-scale}$`)`.

Parameters:

`unscaledVal` - unscaled value of the `BigDecimal`.

`scale` - scale of the `BigDecimal`.

BigDecimal

```
public BigDecimal(BigInteger unscaledVal,
          int scale,
          MathContext mc)
```

Translates a `BigInteger` unscaled value and an `int` scale into a `BigDecimal`, with rounding according to the context settings. The value of the `BigDecimal` is `(unscaledVal × 10`$^{-scale}$`)`, rounded according to the `precision` and rounding mode settings.

Parameters:

`unscaledVal` - unscaled value of the `BigDecimal`.

`scale` - scale of the `BigDecimal`.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

BigDecimal

```
public BigDecimal(int val)
```

Translates an `int` into a `BigDecimal`. The scale of the `BigDecimal` is zero.

Parameters:

`val` - `int` value to be converted to `BigDecimal`.

Since:

1.5

BigDecimal

```
public BigDecimal(int val,
          MathContext mc)
```

Translates an `int` into a `BigDecimal`, with rounding according to the context settings. The scale of the `BigDecimal`, before any rounding, is zero.

Parameters:

`val` - `int` value to be converted to `BigDecimal`.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

BigDecimal

```
public BigDecimal(long val)
```

Translates a `long` into a `BigDecimal`. The scale of the `BigDecimal` is zero.

Parameters:

`val` - `long` value to be converted to `BigDecimal`.

Since:

1.5

BigDecimal

```
public BigDecimal(long val,
          MathContext mc)
```

Translates a `long` into a `BigDecimal`, with rounding according to the context settings. The scale of the `BigDecimal`, before any rounding, is zero.

Parameters:

`val` - `long` value to be converted to `BigDecimal`.

`mc` - the context to use.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

Method Detail

valueOf

```
public static BigDecimal valueOf(long unscaledVal,
                 int scale)
```

Translates a `long` unscaled value and an `int` scale into a `BigDecimal`. This "static factory method" is provided in preference to a (`long`, `int`) constructor because it allows for reuse of frequently used `BigDecimal` values..

Parameters:

`unscaledVal` - unscaled value of the `BigDecimal`.

`scale` - scale of the `BigDecimal`.

Returns:

a `BigDecimal` whose value is (unscaledVal × 10^{-scale}).

valueOf

```
public static BigDecimal valueOf(long val)
```

Translates a `long` value into a `BigDecimal` with a scale of zero. This "static factory method" is provided in preference to a (`long`) constructor because it allows for reuse of frequently used `BigDecimal` values.

Parameters:

`val` - value of the `BigDecimal`.

Returns:

a `BigDecimal` whose value is `val`.

valueOf

```
public static BigDecimal valueOf(double val)
```

Translates a `double` into a `BigDecimal`, using the `double`'s canonical string representation provided by the `Double.toString(double)` method.

Note: This is generally the preferred way to convert a `double` (or `float`) into a `BigDecimal`, as the value returned is equal to that resulting from constructing a `BigDecimal` from the result of using `Double.toString(double)`.

Parameters:

`val` - `double` to convert to a `BigDecimal`.

Returns:

a `BigDecimal` whose value is equal to or approximately equal to the value of `val`.

Throws:

`NumberFormatException` - if `val` is infinite or NaN.

Since:

1.5

add

```
public BigDecimal add(BigDecimal augend)
```

Returns a `BigDecimal` whose value is `(this + augend)`, and whose scale is `max(this.scale(), augend.scale())`.

Parameters:

`augend` - value to be added to this `BigDecimal`.

Returns:

`this + augend`

add

```
public BigDecimal add(BigDecimal augend,
             MathContext mc)
```

Returns a `BigDecimal` whose value is `(this + augend)`, with rounding according to the context settings. If either number is zero and the precision setting is nonzero then the other number, rounded if necessary, is used as the result.

Parameters:

`augend` - value to be added to this `BigDecimal`.

`mc` - the context to use.

Returns:

`this + augend`, rounded as necessary.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

subtract

```
public BigDecimal subtract(BigDecimal subtrahend)
```

Returns a `BigDecimal` whose value is `(this - subtrahend)`, and whose scale is `max(this.scale(), subtrahend.scale())`.

Parameters:

`subtrahend` - value to be subtracted from this `BigDecimal`.

Returns:

`this - subtrahend`

subtract

```
public BigDecimal subtract(BigDecimal subtrahend,
                  MathContext mc)
```

Returns a `BigDecimal` whose value is `(this - subtrahend)`, with rounding according to the context settings. If `subtrahend` is zero then this, rounded if necessary, is used as the result. If this is zero then the result is `subtrahend.negate(mc)`.

Parameters:

`subtrahend` - value to be subtracted from this `BigDecimal`.

`mc` - the context to use.

Returns:

`this - subtrahend`, rounded as necessary.

Throws:

ArithmeticException - if the result is inexact but the rounding mode is UNNECESSARY.

Since:

1.5

multiply

```
public BigDecimal multiply(BigDecimal multiplicand)
```

Returns a BigDecimal whose value is (this × multiplicand), and whose scale is (this.scale() + multiplicand.scale()).

Parameters:

multiplicand - value to be multiplied by this BigDecimal.

Returns:

this * multiplicand

multiply

```
public BigDecimal multiply(BigDecimal multiplicand,
                  MathContext mc)
```

Returns a BigDecimal whose value is (this × multiplicand), with rounding according to the context settings.

Parameters:

multiplicand - value to be multiplied by this BigDecimal.

mc - the context to use.

Returns:

this * multiplicand, rounded as necessary.

Throws:

ArithmeticException - if the result is inexact but the rounding mode is UNNECESSARY.

Since:

1.5

divide

```
public BigDecimal divide(BigDecimal divisor,
                int scale,
                int roundingMode)
```

Returns a BigDecimal whose value is (this / divisor), and whose scale is as specified. If rounding must be performed to generate a result with the specified scale, the specified rounding mode is applied.

The new divide(BigDecimal, int, RoundingMode) method should be used in preference to this legacy method.

Parameters:

divisor - value by which this BigDecimal is to be divided.

scale - scale of the BigDecimal quotient to be returned.

roundingMode - rounding mode to apply.

Returns:

this / divisor

Throws:

ArithmeticException - if divisor is zero, roundingMode==ROUND_UNNECESSARY and the specified scale is insufficient to represent the result of the division exactly.

IllegalArgumentException - if roundingMode does not represent a valid rounding mode.

See Also:

ROUND_UP, ROUND_DOWN, ROUND_CEILING, ROUND_FLOOR, ROUND_HALF_UP, ROUND_HALF_DOWN, ROUND_HALF_EVEN, ROUND_UNNECESSARY

divide

```
public BigDecimal divide(BigDecimal divisor,
                int scale,
                RoundingMode roundingMode)
```

Returns a BigDecimal whose value is (this / divisor), and whose scale is as specified. If rounding must be performed to generate a result with the specified scale, the specified rounding mode is applied.

Parameters:

divisor - value by which this BigDecimal is to be divided.

scale - scale of the BigDecimal quotient to be returned.

roundingMode - rounding mode to apply.

Returns:

this / divisor

Throws:

ArithmeticException - if divisor is zero, roundingMode==RoundingMode.UNNECESSARY and the specified scale is insufficient to represent the result of the division exactly.

Since:

1.5

divide

```
public BigDecimal divide(BigDecimal divisor,
                int roundingMode)
```

Returns a BigDecimal whose value is (this / divisor), and whose scale is this.scale(). If rounding must be performed to generate a result with the given scale, the specified rounding mode is applied.

The new divide(BigDecimal, RoundingMode) method should be used in preference to this legacy method.

Parameters:

divisor - value by which this BigDecimal is to be divided.

roundingMode - rounding mode to apply.

Returns:

`this / divisor`

Throws:

`ArithmeticException` - if `divisor==0`, or `roundingMode==ROUND_UNNECESSARY` and `this.scale()` is insufficient to represent the result of the division exactly.

`IllegalArgumentException` - if `roundingMode` does not represent a valid rounding mode.

See Also:

`ROUND_UP`, `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_FLOOR`, `ROUND_HALF_UP`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_UNNECESSARY`

divide

```
public BigDecimal divide(BigDecimal divisor,
                 RoundingMode roundingMode)
```

Returns a `BigDecimal` whose value is `(this / divisor)`, and whose scale is `this.scale()`. If rounding must be performed to generate a result with the given scale, the specified rounding mode is applied.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

`roundingMode` - rounding mode to apply.

Returns:

`this / divisor`

Throws:

`ArithmeticException` - if `divisor==0`, or `roundingMode==RoundingMode.UNNECESSARY` and `this.scale()` is insufficient to represent the result of the division exactly.

Since:

1.5

divide

```
public BigDecimal divide(BigDecimal divisor)
```

Returns a `BigDecimal` whose value is `(this / divisor)`, and whose preferred scale is `(this.scale() - divisor.scale())`; if the exact quotient cannot be represented (because it has a non-terminating decimal expansion) an `ArithmeticException` is thrown.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

Returns:

`this / divisor`

Throws:

`ArithmeticException` - if the exact quotient does not have a terminating decimal expansion

Since:

1.5

divide

```
public BigDecimal divide(BigDecimal divisor,
                MathContext mc)
```

Returns a `BigDecimal` whose value is `(this / divisor)`, with rounding according to the context settings.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

`mc` - the context to use.

Returns:

`this / divisor`, rounded as necessary.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY` or `mc.precision == 0` and the quotient has a non-terminating decimal expansion.

Since:

1.5

divideToIntegralValue

```
public BigDecimal divideToIntegralValue(BigDecimal divisor)
```

Returns a `BigDecimal` whose value is the integer part of the quotient `(this / divisor)` rounded down. The preferred scale of the result is `(this.scale() - divisor.scale())`.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

Returns:

The integer part of `this / divisor`.

Throws:

`ArithmeticException` - if `divisor==0`

Since:

1.5

divideToIntegralValue

```
public BigDecimal divideToIntegralValue(BigDecimal divisor,
                               MathContext mc)
```

Returns a `BigDecimal` whose value is the integer part of `(this / divisor)`. Since the integer part of the exact quotient does not depend on the rounding mode, the rounding mode does not affect the values returned by this method. The preferred scale of the result is `(this.scale() - divisor.scale())`. An `ArithmeticException` is thrown if the integer part of the exact quotient needs more than `mc.precision` digits.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

`mc` - the context to use.

Returns:

The integer part of `this / divisor`.

Throws:

`ArithmeticException` - if `divisor==0`

`ArithmeticException` - if `mc.precision` > 0 and the result requires a precision of more than `mc.precision` digits.

Since:

1.5

remainder

```
public BigDecimal remainder(BigDecimal divisor)
```

Returns a `BigDecimal` whose value is `(this % divisor)`.

The remainder is given by `this.subtract(this.divideToIntegralValue(divisor).multiply(divisor))`. Note that this is not the modulo operation (the result can be negative).

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

Returns:

`this % divisor`.

Throws:

`ArithmeticException` - if `divisor==0`

Since:

1.5

remainder

```
public BigDecimal remainder(BigDecimal divisor,
                            MathContext mc)
```

Returns a `BigDecimal` whose value is `(this % divisor)`, with rounding according to the context settings. The `MathContext` settings affect the implicit divide used to compute the remainder. The remainder computation itself is by definition exact. Therefore, the remainder may contain more than `mc.getPrecision()` digits.

The remainder is given by `this.subtract(this.divideToIntegralValue(divisor, mc).multiply(divisor))`. Note that this is not the modulo operation (the result can be negative).

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided.

`mc` - the context to use.

Returns:

`this % divisor`, rounded as necessary.

Throws:

`ArithmeticException` - if `divisor==0`

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`, or `mc.precision` > 0 and the result of `this.divideToIntgralValue(divisor)` would require a precision of more than `mc.precision` digits.

Since:

1.5

See Also:

`divideToIntegralValue(java.math.BigDecimal, java.math.MathContext)`

divideAndRemainder

```
public BigDecimal[] divideAndRemainder(BigDecimal divisor)
```

Returns a two-element `BigDecimal` array containing the result of `divideToIntegralValue` followed by the result of `remainder` on the two operands.

Note that if both the integer quotient and remainder are needed, this method is faster than using the `divideToIntegralValue` and `remainder` methods separately because the division need only be carried out once.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided, and the remainder computed.

Returns:

a two element `BigDecimal` array: the quotient (the result of `divideToIntegralValue`) is the initial element and the remainder is the final element.

Throws:

`ArithmeticException` - if `divisor==0`

Since:

1.5

See Also:

`divideToIntegralValue(java.math.BigDecimal, java.math.MathContext)`, `remainder(java.math.BigDecimal, java.math.MathContext)`

divideAndRemainder

```
public BigDecimal[] divideAndRemainder(BigDecimal divisor,
                                       MathContext mc)
```

Returns a two-element `BigDecimal` array containing the result of `divideToIntegralValue` followed by the result of `remainder` on the two operands calculated with rounding according to the context settings.

Note that if both the integer quotient and remainder are needed, this method is faster than using the `divideToIntegralValue` and `remainder` methods separately because the division need only be carried out once.

Parameters:

`divisor` - value by which this `BigDecimal` is to be divided, and the remainder computed.

`mc` - the context to use.

Returns:

a two element `BigDecimal` array: the quotient (the result of `divideToIntegralValue`) is the initial element and the remainder is the final element.

Throws:

ArithmeticException - if divisor==0

ArithmeticException - if the result is inexact but the rounding mode is UNNECESSARY, or mc.precision > 0 and the result of this.divideToIntgralValue(divisor) would require a precision of more than mc.precision digits.

Since:

1.5

See Also:

divideToIntegralValue(java.math.BigDecimal, java.math.MathContext), remainder(java.math.BigDecimal, java.math.MathContext)

pow

```
public BigDecimal pow(int n)
```

Returns a BigDecimal whose value is (this^n), The power is computed exactly, to unlimited precision.

The parameter n must be in the range 0 through 999999999, inclusive. ZERO.pow(0) returns ONE. Note that future releases may expand the allowable exponent range of this method.

Parameters:

n - power to raise this BigDecimal to.

Returns:

this^n

Throws:

ArithmeticException - if n is out of range.

Since:

1.5

pow

```
public BigDecimal pow(int n,
              MathContext mc)
```

Returns a BigDecimal whose value is (this^n). The current implementation uses the core algorithm defined in ANSI standard X3.274-1996 with rounding according to the context settings. In general, the returned numerical value is within two ulps of the exact numerical value for the chosen precision. Note that future releases may use a different algorithm with a decreased allowable error bound and increased allowable exponent range.

The X3.274-1996 algorithm is:

- An ArithmeticException exception is thrown if
  - abs(n) > 999999999
  - mc.precision == 0 and n < 0
  - mc.precision > 0 and n has more than mc.precision decimal digits
- if n is zero, ONE is returned even if this is zero, otherwise
  - if n is positive, the result is calculated via the repeated squaring technique into a single accumulator. The individual multiplications with the accumulator use the same math context settings as in mc except for a precision increased to mc.precision + elength + 1 where elength is the number of decimal digits in n.
  - if n is negative, the result is calculated as if n were positive; this value is then divided into one using the working precision specified above.
  - The final value from either the positive or negative case is then rounded to the destination precision.

Parameters:

`n` - power to raise this `BigDecimal` to.

`mc` - the context to use.

Returns:

`this`^{n} using the ANSI standard X3.274-1996 algorithm

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`, or `n` is out of range.

Since:

1.5

abs

```
public BigDecimal abs()
```

Returns a `BigDecimal` whose value is the absolute value of this `BigDecimal`, and whose scale is `this.scale()`.

Returns:

`abs(this)`

abs

```
public BigDecimal abs(MathContext mc)
```

Returns a `BigDecimal` whose value is the absolute value of this `BigDecimal`, with rounding according to the context settings.

Parameters:

`mc` - the context to use.

Returns:

`abs(this)`, rounded as necessary.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

negate

```
public BigDecimal negate()
```

Returns a `BigDecimal` whose value is `(-this)`, and whose scale is `this.scale()`.

Returns:

`-this`.

negate

```
public BigDecimal negate(MathContext mc)
```

Returns a `BigDecimal` whose value is `(-this)`, with rounding according to the context settings.

Parameters:

`mc` - the context to use.

Returns:

`-this`, rounded as necessary.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

plus

```
public BigDecimal plus()
```

Returns a `BigDecimal` whose value is `(+this)`, and whose scale is `this.scale()`.

This method, which simply returns this `BigDecimal` is included for symmetry with the unary minus method `negate()`.

Returns:

`this`.

Since:

1.5

See Also:

`negate()`

plus

```
public BigDecimal plus(MathContext mc)
```

Returns a `BigDecimal` whose value is `(+this)`, with rounding according to the context settings.

The effect of this method is identical to that of the `round(MathContext)` method.

Parameters:

`mc` - the context to use.

Returns:

`this`, rounded as necessary. A zero result will have a scale of 0.

Throws:

`ArithmeticException` - if the result is inexact but the rounding mode is `UNNECESSARY`.

Since:

1.5

See Also:

round(MathContext)

signum

```
public int signum()
```

Returns the signum function of this `BigDecimal`.

Returns:

-1, 0, or 1 as the value of this `BigDecimal` is negative, zero, or positive.

scale

```
public int scale()
```

Returns the *scale* of this `BigDecimal`. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value of the number is multiplied by ten to the power of the negation of the scale. For example, a scale of `-3` means the unscaled value is multiplied by 1000.

Returns:

the scale of this `BigDecimal`.

precision

```
public int precision()
```

Returns the *precision* of this `BigDecimal`. (The precision is the number of digits in the unscaled value.)

The precision of a zero value is 1.

Returns:

the precision of this `BigDecimal`.

Since:

1.5

unscaledValue

```
public BigInteger unscaledValue()
```

Returns a `BigInteger` whose value is the *unscaled value* of this `BigDecimal`. (Computes `(this *` $10^{\texttt{this.scale()}}$`)`.)

Returns:

the unscaled value of this `BigDecimal`.

Since:

1.2

round

```
public BigDecimal round(MathContext mc)
```

Returns a `BigDecimal` rounded according to the `MathContext` settings. If the precision setting is 0 then no rounding takes place.

The effect of this method is identical to that of the `plus(MathContext)` method.

Parameters:

`mc` - the context to use.

Returns:

a `BigDecimal` rounded according to the `MathContext` settings.

Throws:

`ArithmeticException` - if the rounding mode is `UNNECESSARY` and the `BigDecimal` operation would require rounding.

Since:

1.5

See Also:

`plus(MathContext)`

setScale

```
public BigDecimal setScale(int newScale,
                           RoundingMode roundingMode)
```

Returns a `BigDecimal` whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this `BigDecimal`'s unscaled value by the appropriate power of ten to maintain its overall value. If the scale is reduced by the operation, the unscaled value must be divided (rather than multiplied), and the value may be changed; in this case, the specified rounding mode is applied to the division.

Note that since BigDecimal objects are immutable, calls of this method do *not* result in the original object being modified, contrary to the usual convention of having methods named `setX` mutate field *X*. Instead, `setScale` returns an object with the proper scale; the returned object may or may not be newly allocated.

Parameters:

`newScale` - scale of the `BigDecimal` value to be returned.

`roundingMode` - The rounding mode to apply.

Returns:

a `BigDecimal` whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this `BigDecimal`'s unscaled value by the appropriate power of ten to maintain its overall value.

Throws:

`ArithmeticException` - if `roundingMode==UNNECESSARY` and the specified scaling operation would require rounding.

Since:

1.5

See Also:

`RoundingMode`

setScale

```
public BigDecimal setScale(int newScale,
                           int roundingMode)
```

Returns a `BigDecimal` whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this `BigDecimal`'s unscaled value by the appropriate power of ten to maintain its overall value. If the scale is reduced by the operation, the unscaled value must be divided (rather than multiplied), and the value may be changed; in this case, the specified rounding mode is applied to the division.

Note that since BigDecimal objects are immutable, calls of this method do *not* result in the original object being modified, contrary to the usual convention of having methods named `set`*X* mutate field *X*. Instead, `setScale` returns an object with the proper scale; the returned object may or may not be newly allocated.

The new `setScale(int, RoundingMode)` method should be used in preference to this legacy method.

Parameters:

`newScale` - scale of the `BigDecimal` value to be returned.

`roundingMode` - The rounding mode to apply.

Returns:

a `BigDecimal` whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this `BigDecimal`'s unscaled value by the appropriate power of ten to maintain its overall value.

Throws:

`ArithmeticException` - if `roundingMode==ROUND_UNNECESSARY` and the specified scaling operation would require rounding.

`IllegalArgumentException` - if `roundingMode` does not represent a valid rounding mode.

See Also:

`ROUND_UP`, `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_FLOOR`, `ROUND_HALF_UP`, `ROUND_HALF_DOWN`, `ROUND_HALF_EVEN`, `ROUND_UNNECESSARY`

setScale

```
public BigDecimal setScale(int newScale)
```

Returns a `BigDecimal` whose scale is the specified value, and whose value is numerically equal to this `BigDecimal`'s. Throws an `ArithmeticException` if this is not possible.

This call is typically used to increase the scale, in which case it is guaranteed that there exists a `BigDecimal` of the specified scale and the correct value. The call can also be used to reduce the scale if the caller knows that the `BigDecimal` has sufficiently many zeros at the end of its fractional part (i.e., factors of ten in its integer value) to allow for the rescaling without changing its value.

This method returns the same result as the two-argument versions of `setScale`, but saves the caller the trouble of specifying a rounding mode in cases where it is irrelevant.

Note that since `BigDecimal` objects are immutable, calls of this method do *not* result in the original object being modified, contrary to the usual convention of having methods named `set`*X* mutate field *X*. Instead, `setScale` returns an object with the proper scale; the returned object may or may not be newly allocated.

Parameters:

`newScale` - scale of the `BigDecimal` value to be returned.

Returns:

a `BigDecimal` whose scale is the specified value, and whose unscaled value is determined by multiplying or dividing this `BigDecimal`'s unscaled value by the appropriate power of ten to maintain its overall value.

Throws:

ArithmeticException - if the specified scaling operation would require rounding.

See Also:

setScale(int, int), setScale(int, RoundingMode)

movePointLeft

```
public BigDecimal movePointLeft(int n)
```

Returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the left. If n is non-negative, the call merely adds n to the scale. If n is negative, the call is equivalent to movePointRight(-n). The BigDecimal returned by this call has value (this × 10^{-n}) and scale max(this.scale()+n, 0).

Parameters:

n - number of places to move the decimal point to the left.

Returns:

a BigDecimal which is equivalent to this one with the decimal point moved n places to the left.

Throws:

ArithmeticException - if scale overflows.

movePointRight

```
public BigDecimal movePointRight(int n)
```

Returns a BigDecimal which is equivalent to this one with the decimal point moved n places to the right. If n is non-negative, the call merely subtracts n from the scale. If n is negative, the call is equivalent to movePointLeft(-n). The BigDecimal returned by this call has value (this × 10^{n}) and scale max(this.scale()-n, 0).

Parameters:

n - number of places to move the decimal point to the right.

Returns:

a BigDecimal which is equivalent to this one with the decimal point moved n places to the right.

Throws:

ArithmeticException - if scale overflows.

scaleByPowerOfTen

```
public BigDecimal scaleByPowerOfTen(int n)
```

Returns a BigDecimal whose numerical value is equal to (this * 10^{n}). The scale of the result is (this.scale() - n).

Throws:

ArithmeticException - if the scale would be outside the range of a 32-bit integer.

Since:

1.5

stripTrailingZeros

```
public BigDecimal stripTrailingZeros()
```

Returns a `BigDecimal` which is numerically equal to this one but with any trailing zeros removed from the representation. For example, stripping the trailing zeros from the `BigDecimal` value `600.0`, which has [`BigInteger`, `scale`] components equals to [6000, 1], yields `6E2` with [`BigInteger`, `scale`] components equals to [6, -2]

Returns:

a numerically equal `BigDecimal` with any trailing zeros removed.

Since:

1.5

compareTo

```
public int compareTo(BigDecimal val)
```

Compares this `BigDecimal` with the specified `BigDecimal`. Two `BigDecimal` objects that are equal in value but have a different scale (like 2.0 and 2.00) are considered equal by this method. This method is provided in preference to individual methods for each of the six boolean comparison operators (<, ==, >, >=, !=, <=). The suggested idiom for performing these comparisons is: `(x.compareTo(y)` <*op*> `0)`, where <*op*> is one of the six comparison operators.

Specified by:

`compareTo` in interface `Comparable`<`BigDecimal`>

Parameters:

`val` - `BigDecimal` to which this `BigDecimal` is to be compared.

Returns:

-1, 0, or 1 as this `BigDecimal` is numerically less than, equal to, or greater than `val`.

equals

```
public boolean equals(Object x)
```

Compares this `BigDecimal` with the specified `Object` for equality. Unlike `compareTo`, this method considers two `BigDecimal` objects equal only if they are equal in value and scale (thus 2.0 is not equal to 2.00 when compared by this method).

Overrides:

`equals` in class `Object`

Parameters:

`x` - `Object` to which this `BigDecimal` is to be compared.

Returns:

`true` if and only if the specified `Object` is a `BigDecimal` whose value and scale are equal to this `BigDecimal`'s.

See Also:

`compareTo(java.math.BigDecimal)`, `hashCode()`

min

```
public BigDecimal min(BigDecimal val)
```

Returns the minimum of this `BigDecimal` and `val`.

Parameters:

`val` - value with which the minimum is to be computed.

Returns:

the `BigDecimal` whose value is the lesser of this `BigDecimal` and `val`. If they are equal, as defined by the `compareTo` method, `this` is returned.

See Also:

`compareTo(java.math.BigDecimal)`

max

```
public BigDecimal max(BigDecimal val)
```

Returns the maximum of this `BigDecimal` and `val`.

Parameters:

`val` - value with which the maximum is to be computed.

Returns:

the `BigDecimal` whose value is the greater of this `BigDecimal` and `val`. If they are equal, as defined by the `compareTo` method, `this` is returned.

See Also:

`compareTo(java.math.BigDecimal)`

hashCode

```
public int hashCode()
```

Returns the hash code for this `BigDecimal`. Note that two `BigDecimal` objects that are numerically equal but differ in scale (like 2.0 and 2.00) will generally *not* have the same hash code.

Overrides:

`hashCode` in class `Object`

Returns:

hash code for this `BigDecimal`.

See Also:

`equals(Object)`

toString

```
public String toString()
```

Returns the string representation of this `BigDecimal`, using scientific notation if an exponent is needed.

A standard canonical string form of the `BigDecimal` is created as though by the following steps: first, the absolute value of the unscaled value of the `BigDecimal` is converted to a string in base ten using the characters `'0'` through `'9'` with no leading zeros (except if its value is zero, in which case a single `'0'` character is used).

Next, an *adjusted exponent* is calculated; this is the negated scale, plus the number of characters in the converted unscaled value, less one. That is, `-scale+(ulength-1)`, where `ulength` is the length of the absolute value of the unscaled value in decimal digits (its *precision*).

If the scale is greater than or equal to zero and the adjusted exponent is greater than or equal to `-6`, the number will be converted to a character form without using exponential notation. In this case, if the scale is zero then no decimal point is added and if the scale is positive a decimal point will be inserted with the scale specifying the number of characters to the right of the decimal point. `'0'` characters are added to the left of the converted unscaled value as necessary. If no character precedes the decimal point after this insertion then a conventional `'0'` character is prefixed.

Otherwise (that is, if the scale is negative, or the adjusted exponent is less than `-6`), the number will be converted to a character form using exponential notation. In this case, if the converted `BigInteger` has more than one digit a decimal point is inserted after the first digit. An exponent in character form is then suffixed to the converted unscaled value (perhaps with inserted decimal point); this comprises the letter `'E'` followed immediately by the adjusted exponent converted to a character form. The latter is in base ten, using the characters `'0'` through `'9'` with no leading zeros, and is always prefixed by a sign character `'-'` (`'\u002D'`) if the adjusted exponent is negative, `'+'` (`'\u002B'`) otherwise).

Finally, the entire string is prefixed by a minus sign character `'-'` (`'\u002D'`) if the unscaled value is less than zero. No sign character is prefixed if the unscaled value is zero or positive.

Examples:

For each representation [*unscaled value*, *scale*] on the left, the resulting string is shown on the right.

```
[123,0]       "123"
[-123,0]      "-123"
[123,-1]      "1.23E+3"
[123,-3]      "1.23E+5"
[123,1]       "12.3"
[123,5]       "0.00123"
[123,10]      "1.23E-8"
[-123,12]     "-1.23E-10"
```

Notes:

1. There is a one-to-one mapping between the distinguishable `BigDecimal` values and the result of this conversion. That is, every distinguishable `BigDecimal` value (unscaled value and scale) has a unique string representation as a result of using `toString`. If that string representation is converted back to a `BigDecimal` using the `BigDecimal(String)` constructor, then the original value will be recovered.
2. The string produced for a given number is always the same; it is not affected by locale. This means that it can be used as a canonical string representation for exchanging decimal data, or as a key for a Hashtable, etc. Locale-sensitive number formatting and parsing is handled by the `NumberFormat` class and its subclasses.
3. The `toEngineeringString()` method may be used for presenting numbers with exponents in engineering notation, and the `setScale` method may be used for rounding a `BigDecimal` so it has a known number of digits after the decimal point.
4. The digit-to-character mapping provided by `Character.forDigit` is used.

Overrides:

`toString` in class `Object`

Returns:

string representation of this `BigDecimal`.

See Also:

`Character.forDigit(int, int)`, `BigDecimal(java.lang.String)`

toEngineeringString

```
public String toEngineeringString()
```

Returns a string representation of this `BigDecimal`, using engineering notation if an exponent is needed.

Returns a string that represents the `BigDecimal` as described in the `toString()` method, except that if exponential notation is used, the power of ten is adjusted to be a multiple of three (engineering notation) such that the integer part of nonzero values will be in the range 1 through 999. If exponential notation is used for zero values, a decimal point and one or two fractional zero digits are used so that the scale of the zero value is preserved. Note that unlike the output of `toString()`, the output of this method is *not* guaranteed to recover the same [integer, scale] pair of this `BigDecimal` if the output string is converting back to a `BigDecimal` using the string constructor. The result of this method meets the weaker constraint of always producing a numerically equal result from applying the string constructor to the method's output.

Returns:

string representation of this `BigDecimal`, using engineering notation if an exponent is needed.

Since:

1.5

toPlainString

```
public String toPlainString()
```

Returns a string representation of this `BigDecimal` without an exponent field. For values with a positive scale, the number of digits to the right of the decimal point is used to indicate scale. For values with a zero or negative scale, the resulting string is generated as if the value were converted to a numerically equal value with zero scale and as if all the trailing zeros of the zero scale value were present in the result. The entire string is prefixed by a minus sign character '-' (`'\u002D'`) if the unscaled value is less than zero. No sign character is prefixed if the unscaled value is zero or positive. Note that if the result of this method is passed to the string constructor, only the numerical value of this `BigDecimal` will necessarily be recovered; the representation of the new `BigDecimal` may have a different scale. In particular, if this `BigDecimal` has a negative scale, the string resulting from this method will have a scale of zero when processed by the string constructor. (This method behaves analogously to the `toString` method in 1.4 and earlier releases.)

Returns:

a string representation of this `BigDecimal` without an exponent field.

Since:

1.5

See Also:

`toString()`, `toEngineeringString()`

toBigInteger

```
public BigInteger toBigInteger()
```

Converts this `BigDecimal` to a `BigInteger`. This conversion is analogous to the *narrowing primitive conversion* from `double` to `long` as defined in section 5.1.3 of *The Java™ Language Specification*: any fractional part of this `BigDecimal` will be discarded. Note that this conversion can lose information about the precision of the `BigDecimal` value.

To have an exception thrown if the conversion is inexact (in other words if a nonzero fractional part is discarded), use the `toBigIntegerExact()` method.

Returns:

this `BigDecimal` converted to a `BigInteger`.

toBigIntegerExact

```
public BigInteger toBigIntegerExact()
```

Converts this `BigDecimal` to a `BigInteger`, checking for lost information. An exception is thrown if this `BigDecimal` has a nonzero fractional part.

Returns:

this `BigDecimal` converted to a `BigInteger`.

Throws:

`ArithmeticException` - if `this` has a nonzero fractional part.

Since:

1.5

longValue

```
public long longValue()
```

Converts this `BigDecimal` to a `long`. This conversion is analogous to the *narrowing primitive conversion* from `double` to `short` as defined in section 5.1.3 of *The Java™ Language Specification*: any fractional part of this `BigDecimal` will be discarded, and if the resulting "`BigInteger`" is too big to fit in a `long`, only the low-order 64 bits are returned. Note that this conversion can lose information about the overall magnitude and precision of this `BigDecimal` value as well as return a result with the opposite sign.

Specified by:

`longValue` in class `Number`

Returns:

this `BigDecimal` converted to a `long`.

longValueExact

```
public long longValueExact()
```

Converts this `BigDecimal` to a `long`, checking for lost information. If this `BigDecimal` has a nonzero fractional part or is out of the possible range for a `long` result then an `ArithmeticException` is thrown.

Returns:

this `BigDecimal` converted to a `long`.

Throws:

`ArithmeticException` - if `this` has a nonzero fractional part, or will not fit in a `long`.

Since:

1.5

intValue

```
public int intValue()
```

Converts this `BigDecimal` to an `int`. This conversion is analogous to the *narrowing primitive conversion* from `double` to `short` as defined in section 5.1.3 of *The Java™ Language Specification*: any fractional part of this `BigDecimal` will be discarded, and if the resulting "`BigInteger`" is too big to fit in an `int`, only the low-order 32 bits are returned. Note that this conversion can lose information about the overall magnitude and precision of this `BigDecimal` value as well as return a result with the opposite sign.

Specified by:

`intValue` in class `Number`

Returns:

this `BigDecimal` converted to an `int`.

intValueExact

```
public int intValueExact()
```

Converts this `BigDecimal` to an `int`, checking for lost information. If this `BigDecimal` has a nonzero fractional part or is out of the possible range for an `int` result then an `ArithmeticException` is thrown.

Returns:

this `BigDecimal` converted to an `int`.

Throws:

`ArithmeticException` - if `this` has a nonzero fractional part, or will not fit in an `int`.

Since:

1.5

shortValueExact

```
public short shortValueExact()
```

Converts this `BigDecimal` to a `short`, checking for lost information. If this `BigDecimal` has a nonzero fractional part or is out of the possible range for a `short` result then an `ArithmeticException` is thrown.

Returns:

this `BigDecimal` converted to a `short`.

Throws:

`ArithmeticException` - if `this` has a nonzero fractional part, or will not fit in a `short`.

Since:

1.5

byteValueExact

```
public byte byteValueExact()
```

Converts this `BigDecimal` to a `byte`, checking for lost information. If this `BigDecimal` has a nonzero fractional part or is out of the possible range for a `byte` result then an `ArithmeticException` is thrown.

Returns:

this `BigDecimal` converted to a `byte`.

Throws:

`ArithmeticException` - if `this` has a nonzero fractional part, or will not fit in a `byte`.

Since:

1.5

floatValue

```
public float floatValue()
```

Converts this `BigDecimal` to a `float`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` as defined in section 5.1.3 of *The Java™ Language Specification*: if this `BigDecimal` has too great a magnitude to represent as a `float`, it will be converted to `Float.NEGATIVE_INFINITY` or `Float.POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the `BigDecimal` value.

Specified by:

`floatValue` in class `Number`

Returns:

this `BigDecimal` converted to a `float`.

doubleValue

```
public double doubleValue()
```

Converts this `BigDecimal` to a `double`. This conversion is similar to the *narrowing primitive conversion* from `double` to `float` as defined in section 5.1.3 of *The Java™ Language Specification*: if this `BigDecimal` has too great a magnitude represent as a `double`, it will be converted to `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` as appropriate. Note that even when the return value is finite, this conversion can lose information about the precision of the `BigDecimal` value.

Specified by:

`doubleValue` in class `Number`

Returns:

this `BigDecimal` converted to a `double`.

ulp

```
public BigDecimal ulp()
```

Returns the size of an ulp, a unit in the last place, of this `BigDecimal`. An ulp of a nonzero `BigDecimal` value is the positive distance between this value and the `BigDecimal` value next larger in magnitude with the same number of digits. An ulp of a zero value is numerically equal to 1 with the scale of `this`. The result is stored with the same scale as `this` so the result for zero and nonzero values is equal to `[1,` `this.scale()]`.

Returns:

the size of an ulp of `this`

Since:

1.5

Overview Package Class Use Tree Deprecated Index Help

***Java™ Platform
Standard Ed. 7***

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Submit a bug or feature

For further API reference and developer documentation, see Java SE Documentation. That documentation contains more detailed, developer-targeted descriptions, with conceptual overviews, definitions of terms, workarounds, and working code examples.