Decimal Class

Contains methods for the Decimal primitive data type.

Namespace

System

Usage

Note

Two Decimal objects that are numerically equivalent but differ in scale (such as 1.1 and 1.10) generally do not have the same hashcode. Use caution when such Decimal objects are used in Sets or as Map keys.

For more information on Decimal, see Decimal Data Type.

- **Rounding Mode**
  Rounding mode specifies the rounding behavior for numerical operations capable of discarding precision.
- **Decimal Methods**

Rounding Mode

Rounding mode specifies the rounding behavior for numerical operations capable of discarding precision.

Each rounding mode indicates how the least significant returned digit of a rounded result is to be calculated. The following are the valid values for *roundingMode*.

Name	Description

`CEILING`	Rounds towards positive infinity. That is, if the result is positive, this mode behaves the same as the `UP` rounding mode; if the result is negative, it behaves the same as the `DOWN` rounding mode. Note that this rounding mode never decreases the calculated value. For example:
	• Input number 5.5: `CEILING` round mode result: 6
	• Input number 1.1: `CEILING` round mode result: 2
	• Input number -1.1: `CEILING` round mode result: -1
	• Input number -2.7: `CEILING` round mode result: -2
	`Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};`
	`Long[] expected = new Long[]{6, 2, -1, -2};`
	`for(integer x = 0; x < example.size(); x++){`
	`    System.assertEquals(expected[x],`

`DOWN`	Rounds towards zero. This rounding mode always discards any fractions (decimal points) prior to executing. Note that this rounding mode never increases the magnitude of the calculated value. For example:
	• Input number 5.5: `DOWN` round mode result: 5
	• Input number 1.1: `DOWN` round mode result: 1
	• Input number -1.1: `DOWN` round mode result: -1
	• Input number -2.7: `DOWN` round mode result: -2
	`Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};`
	`Long[] expected = new Long[]{5, 1, -1, -2};`
	`for(integer x = 0; x < example.size(); x++){`
	`    System.assertEquals(expected[x],`
	`        example[x].round(System.RoundingMode.DOWN));`
	`}`

`FLOOR`	Rounds towards negative infinity. That is, if the result is positive, this mode behaves the same as the `DOWN` rounding mode; if negative, this mode behaves the same as the `UP` rounding mode. Note that this rounding mode never increases the calculated value. For example:
	• Input number 5.5: `FLOOR` round mode result: 5
	• Input number 1.1: `FLOOR` round mode result: 1
	• Input number -1.1: `FLOOR` round mode result: -2
	• Input number -2.7: `FLOOR` round mode result: -3
	`Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};`
	`Long[] expected = new Long[]{5, 1, -2, -3};`
	`for(integer x = 0; x < example.size(); x++){`
	`    System.assertEquals(expected[x],`
	`        example[x].round(System.RoundingMode.FLOOR));`
	`}`

`HALF_DOWN`	Rounds towards the "nearest neighbor" unless both neighbors are equidistant, in which case this mode rounds down. This rounding mode behaves the same as the `UP` rounding mode if the discarded fraction (decimal point) is > 0.5; otherwise, it behaves the same as `DOWN` rounding mode. For example:
	• Input number 5.5: `HALF_DOWN` round mode result: 5
	• Input number 1.1: `HALF_DOWN` round mode result: 1
	• Input number -1.1: `HALF_DOWN` round mode result: -1
	• Input number -2.7: `HALF_DOWN` round mode result: -3
	`Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};`
	`Long[] expected = new Long[]{5, 1, -1, -3};`
	`for(integer x = 0; x < example.size(); x++){`
	`    System.assertEquals(expected[x],`
	`        example[x].round(System.RoundingMode.HALF_DOWN));`
	`}`

`HALF_EVEN`	Rounds towards the "nearest neighbor" unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor. This rounding mode behaves the same as the `HALF_UP` rounding mode if the digit to the left of the discarded fraction

- Input number 5.5: `HALF_EVEN` round mode result: 6
- Input number 1.1: `HALF_EVEN` round mode result: 1
- Input number -1.1: `HALF_EVEN` round mode result: -1
- Input number -2.7: `HALF_EVEN` round mode result: -3

```
Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};
Long[] expected = new Long[]{6, 1, -1, -3};
for(integer x = 0; x < example.size(); x++){
    System.assertEquals(expected[x],
        example[x].round(System.RoundingMode.HALF_EVEN));
}
```

Note that this rounding mode statistically minimizes cumulative error when applied repeatedly over a sequence of calculations.

`HALF_UP`

Rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds up. This rounding method behaves the same as the `UP` rounding method if the discarded fraction (decimal point) is >= 0.5; otherwise, this rounding method behaves the same as the `DOWN` rounding method. For example:

- Input number 5.5: `HALF_UP` round mode result: 6
- Input number 1.1: `HALF_UP` round mode result: 1
- Input number -1.1: `HALF_UP` round mode result: -1
- Input number -2.7: `HALF_UP` round mode result: -3

```
Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};
Long[] expected = new Long[]{6, 1, -1, -3};
for(integer x = 0; x < example.size(); x++){
    System.assertEquals(expected[x],
        example[x].round(System.RoundingMode.HALF_UP));
}
```

`UNNECESSARY`

Asserts that the requested operation has an exact result, which means that no rounding is necessary. If this rounding mode is specified on an operation that yields an inexact result, a MathException is thrown. For example:

- Input number 5.5: `UNNECESSARY` round mode result: MathException
- Input number 1.1: `UNNECESSARY` round mode result: MathException
- Input number 1.0: `UNNECESSARY` round mode result: 1
- Input number -1.0: `UNNECESSARY` round mode result: -1
- Input number -2.2: `UNNECESSARY` round mode result: MathException

```
Decimal example1 = 5.5;
Decimal example2 = 1.0;
system.assertEquals(1,
    example2.round(System.RoundingMode.UNNECESSARY));
try{
    example1.round(System.RoundingMode.UNNECESSARY);
} catch(Exception E) {
    system.assertEquals('System.MathException', E.getTypeName());
}
```

`UP`

Rounds away from zero. This rounding mode always truncates any fractions (decimal points) prior to executing. Note that this rounding mode never decreases the magnitude of the calculated value. For example:

- Input number -1.1: `UP` round mode result: -2
- Input number -2.7: `UP` round mode result: -3

```
Decimal[] example = new Decimal[]{5.5, 1.1, -1.1, -2.7};
Long[] expected = new Long[]{6, 2, -2, -3};
for(integer x = 0; x < example.size(); x++){
    System.assertEquals(expected[x],
        example[x].round(System.RoundingMode.UP));
}
```

Decimal Methods

The following are methods for `Decimal`.

- **abs()**
  Returns the absolute value of the Decimal.
- **divide(divisor, scale)**
  Divides this Decimal by the specified divisor, and sets the scale, that is, the number of decimal places, of the result using the specified scale.
- **divide(divisor, scale, roundingMode)**
  Divides this Decimal by the specified divisor, sets the scale, that is, the number of decimal places, of the result using the specified scale, and if necessary, rounds the value using the rounding mode.
- **doubleValue()**
  Returns the Double value of this Decimal.
- **format()**
  Returns the String value of this Decimal using the locale of the context user.
- **intValue()**
  Returns the Integer value of this Decimal.
- **longValue()**
  Returns the Long value of this Decimal.
- **pow(exponent)**
  Returns the value of this decimal raised to the power of the specified exponent.
- **precision()**
  Returns the total number of digits for the Decimal.
- **round()**
  Returns the rounded approximation of this Decimal. The number is rounded to zero decimal places using half-even rounding mode, that is, it rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor.
- **round(roundingMode)**
  Returns the rounded approximation of this Decimal. The number is rounded to zero decimal places using the rounding mode specified by the rounding mode.
- **scale()**
  Returns the scale of the Decimal, that is, the number of decimal places.
- **setScale(scale)**
  Returns the Decimal scaled to the specified number of decimal places, using half-even rounding, if necessary. Half-even rounding mode rounds toward the “nearest neighbor.” If both neighbors are equidistant, the number is rounded toward the even neighbor.
- **setScale(scale, roundingMode)**
  Returns the Decimal scaled to the specified number of decimal places, using the specified rounding mode, if necessary.

- valueOf(doubleToDecimal)
  Returns a Decimal that contains the value of the specified Double.
- valueOf(longToDecimal)
  Returns a Decimal that contains the value of the specified Long.
- valueOf(stringToDecimal)
  Returns a Decimal that contains the value of the specified String. As in Java, the string is interpreted as representing a signed Decimal.

abs()

Returns the absolute value of the Decimal.

Signature

```
public Decimal abs()
```

Return Value

Type: Decimal

Example

```
Decimal myDecimal = -6.02214129;
System.assertEquals(6.02214129, myDecimal.abs());
```

divide(divisor, scale)

Divides this Decimal by the specified divisor, and sets the scale, that is, the number of decimal places, of the result using the specified scale.

Signature

```
public Decimal divide(Decimal divisor, Integer scale)
```

Parameters

divisor

Type: Decimal

scale

Type: Integer

Return Value

Type: Decimal

Example

```
Decimal decimalNumber = 19;
Decimal result = decimalNumber.divide(100, 3);
System.assertEquals(0.190, result);
```

divide(divisor, scale, roundingMode)

Divides this Decimal by the specified divisor, sets the scale, that is, the number of decimal places, of the result using the specified scale, and if necessary, rounds the value using the rounding mode.

divisor

Type: Decimal

scale

Type: Integer

roundingMode

Type: System.RoundingMode

Return Value

Type: Decimal

Example

```
Decimal myDecimal = 12.4567;
Decimal divDec = myDecimal.divide(7, 2, System.RoundingMode.UP);
System.assertEquals(divDec, 1.78);
```

doubleValue()

Returns the Double value of this Decimal.

Signature

```
public Double doubleValue()
```

Return Value

Type: Double

Example

```
Decimal myDecimal = 6.62606957;
Double value = myDecimal.doubleValue();
System.assertEquals(6.62606957, value);
```

format()

Returns the String value of this Decimal using the locale of the context user.

Signature

```
public String format()
```

Return Value

Type: String

Usage

Scientific notation will be used if an exponent is needed.

Example

intValue()

Returns the Integer value of this Decimal.

Signature

```
public Integer intValue()
```

Return Value

Type: Integer

Example

```
Decimal myDecimal = 1.602176565;
system.assertEquals(1, myDecimal.intValue());
```

longValue()

Returns the Long value of this Decimal.

Signature

```
public Long longValue()
```

Return Value

Type: Long

Example

```
Decimal myDecimal = 376.730313461;
system.assertEquals(376, myDecimal.longValue());
```

pow(exponent)

Returns the value of this decimal raised to the power of the specified exponent.

Signature

```
public Decimal pow(Integer exponent)
```

Parameters

exponent

Type: Integer

The value of *exponent* must be between 0 and 32,767.

Return Value

Type: Decimal

Usage

If you use `MyDecimal.pow(0)`, 1 is returned.

The `Math.pow` method does accept negative values.

```
    Decimal powDec = myDecimal.pow(2);
    System.assertEquals(powDec, 16.9744);
```

precision()

Returns the total number of digits for the Decimal.

Signature

`public Integer precision()`

Return Value

Type: Integer

Example

For example, if the Decimal value was 123.45, `precision` returns 5. If the Decimal value is 123.123, `precision` returns 6.

```
    Decimal D1 = 123.45;
    Integer precision1 = D1.precision();
    system.assertEquals(precision1, 5);
    Decimal D2 = 123.123;
    Integer precision2 = D2.precision();
    system.assertEquals(precision2, 6);
```

round()

Returns the rounded approximation of this Decimal. The number is rounded to zero decimal places using half-even rounding mode, that is, it rounds towards the “nearest neighbor” unless both neighbors are equidistant, in which case, this mode rounds towards the even neighbor.

Signature

`public Long round()`

Return Value

Type: Long

Usage

Note that this rounding mode statistically minimizes cumulative error when applied repeatedly over a sequence of calculations.

Example

```
    Decimal D = 4.5;
    Long L = D.round();
    System.assertEquals(4, L);

    Decimal D1 = 5.5;
    Long L1 = D1.round();
    System.assertEquals(6, L1);

    Decimal D2 = 5.2;
    Long L2 = D2.round();
    System.assertEquals(5, L2);

    Decimal D3 = -5.7;
```

Returns the rounded approximation of this Decimal. The number is rounded to zero decimal places using the rounding mode specified by the rounding mode.

Signature

```
public Long round(System.RoundingMode roundingMode)
```

Parameters

roundingMode

Type: System.RoundingMode

Return Value

Type: Long

scale()

Returns the scale of the Decimal, that is, the number of decimal places.

Signature

```
public Integer scale()
```

Return Value

Type: Integer

Example

```
Decimal myDecimal = 9.27400968;
system.assertEquals(8, myDecimal.scale());
```

setScale(scale)

Returns the Decimal scaled to the specified number of decimal places, using half-even rounding, if necessary. Half-even rounding mode rounds toward the “nearest neighbor.” If both neighbors are equidistant, the number is rounded toward the even neighbor.

Signature

```
public Decimal setScale(Integer scale)
```

Parameters

scale

Type: Integer

The value of *scale* must be between −33 and 33. If the value of *scale* is negative, your unscaled value is multiplied by 10 to the power of the negation of *scale*. For example, after this operation, the value of *d* is `4*10^3`.

```
Decimal d = 4000;
d = d.setScale(-3);
```

Return Value

Type: Decimal

returned from the query.

- If the Decimal is created from a String, the scale is the number of characters after the decimal point of the String.
- If the Decimal is created from a non-decimal number, the number is first converted to a String. The scale is then set using the number of characters after the decimal point.

Example

```
Decimal myDecimal = 8.987551787;
Decimal setScaled = myDecimal.setscale(3);
System.assertEquals(8.988, setScaled);
```

setScale(scale, roundingMode)

Returns the Decimal scaled to the specified number of decimal places, using the specified rounding mode, if necessary.

Signature

`public Decimal setScale(Integer scale, System.RoundingMode roundingMode)`

Parameters

scale

Type: Integer

The value of *scale* must be between –33 and 33. If the value of *scale* is negative, your unscaled value is multiplied by 10 to the power of the negation of *scale*. For example, after this operation, the value of *d* is `4*10^3`.

```
Decimal d = 4000;
d = d.setScale(-3);
```

roundingMode

Type: System.RoundingMode

Return Value

Type: Decimal

Usage

If you do not explicitly set the scale for a Decimal, the item from which the Decimal is created determines the scale.

- If the Decimal is created as part of a query, the scale is based on the scale of the field returned from the query.
- If the Decimal is created from a String, the scale is the number of characters after the decimal point of the String.
- If the Decimal is created from a non-decimal number, the number is first converted to a String. The scale is then set using the number of characters after the decimal point.

stripTrailingZeros()

Returns the Decimal with any trailing zeros removed.

Type: Decimal

Example

```
Decimal myDecimal = 1.10000;
Decimal stripped = myDecimal.stripTrailingZeros();
System.assertEquals(stripped, 1.1);
```

toPlainString()

Returns the String value of this Decimal, without using scientific notation.

Signature

```
public String toPlainString()
```

Return Value

Type: String

Example

```
Decimal myDecimal = 12345.6789;
System.assertEquals('12345.6789', myDecimal.toPlainString());
```

valueOf(doubleToDecimal)

Returns a Decimal that contains the value of the specified Double.

Signature

```
public static Decimal valueOf(Double doubleToDecimal)
```

Parameters

doubleToDecimal

Type: Double

Return Value

Type: Decimal

Example

```
Double myDouble = 2.718281828459045;
Decimal myDecimal = Decimal.valueOf(myDouble);
System.assertEquals(2.718281828459045, myDecimal);
```

valueOf(longToDecimal)

Returns a Decimal that contains the value of the specified Long.

Signature

```
public static Decimal valueOf(Long longToDecimal)
```

Return Value

Type: Decimal

Example

```
Long myLong = 299792458;
Decimal myDecimal = Decimal.valueOf(myLong);
System.assertEquals(299792458, myDecimal);
```

valueOf(stringToDecimal)

Returns a Decimal that contains the value of the specified String. As in Java, the string is interpreted as representing a signed Decimal.

Signature

```
public static Decimal valueOf(String stringToDecimal)
```

Parameters

stringToDecimal
Type: String

Return Value

Type: Decimal

Example

```
String temp = '12.4567';
Decimal myDecimal = Decimal.valueOf(temp);
```

DID THIS ARTICLE SOLVE YOUR ISSUE?
Let us know so we can improve!

Share your feedback

MuleSoft	Component Library	Events and Calendar
Tableau	APIs	Partner Community
Commerce Cloud	Trailhead	Blog
Lightning Design System	Sample Apps	Salesforce Admins
Einstein	Podcasts	Salesforce Architects
Quip	AppExchange	

© Copyright 2025 Salesforce, Inc. All rights reserved. Various trademarks held by their respective owners. Salesforce, Inc. Salesforce Tower, 415 Mission Street, 3rd Floor, San Francisco, CA 94105, United States

Privacy Information Terms of Service Legal Use of Cookies Trust Cookie Preferences

Your Privacy Choices Responsible Disclosure Contact