



ONLY AT
CLOSETMAID

Get Your Free Design Now!

Log in / create account



navigation

- [Main Page](#)
- [Opentaps 2](#)
- [Users Manual](#)
- [Reference Manual](#)
- [Testing Manual](#)
- [Technical Reference](#)
- [Training Videos](#)
- [Recent changes](#)
- [Donations](#)

search

toolbox

- [What links here](#)
- [Related changes](#)
- [Upload file](#)
- [Special pages](#)
- [Printable version](#)
- [Permanent link](#)
- [Print as PDF](#)

How to Use Java BigDecimal: A Tutorial

Contents [\[hide\]](#)

- 1 The Problem
- 2 Primer on Financial Issues
- 3 Introducing BigDecimal
- 4 Rounding and Scalin
- 5 Immutability and Arithmetic
- 6 Comparison
- 7 When to Round: Thoughts on Precision
- 8 Appendix
 - 8.1 API Reference
 - 8.2 OFBiz Framework
 - 8.3 Minilang

The Problem

When we started building the General Ledger services for accounting, we discovered that there were errors of 0.01 cent or more in many places. This made accounting practically impossible. Who would want to bill a customer for \$4.01 when his order says \$4.00?

The reasons for these errors soon became clear: Computations that yielded amounts, quantities, adjustments, and many other things were generally done with little or no attention to the special precision and rounding concerns that arise when dealing with financial issues.

All of these computations used Java doubles, which offer no way to control how the number is rounded or to limit the precision in computation. We came up with a solution involving the use of [java.math.BigDecimal](#), which gives us this control.

This document serves as a primer on financial math issues and as a tutorial on the use of BigDecimal in general

Primer on Financial Issues

Currency calculations require precision to a specific degree, such as two digits after the decimal for most currencies. They also require a specific type of rounding behavior, such as always rounding up in the case of taxes.

For example, suppose we have a product which costs 10.00 in a given currency and the local sales tax is 0.0825, or 8.25%. If we work it out on paper, the tax amount is,

$$10.00 * 0.0825 = 0.825$$

Because our precision for the currency is two digits after the decimal, we need to round the 0.825 figure. Also, because this is a tax, it is good practice to always round up to the next highest cent. That way when the accounts are balanced at the end of the day, we never find ourselves underpaying taxes.

$$0.825 \rightarrow 0.83$$

And so the total we charge to the customer is 10.83 in the local currency and pay 0.83 to the tax collector. Note that if we sold 1000 of these, we would have overpaid the collector by this much,

$$1000 * (0.83 - 0.825) = 5.00$$

Another important issue is where to do the rounding in a given computation. Suppose we sold Liquid Nitrogen at 0.528361 per liter. A customer comes in and buys 100.00 liters, so we write out the total price,

$$100.0 * 0.528361 = 52.8361$$

Because this isn't a tax, we can round this either up or down at our discretion. Suppose we round according to standard rounding rules: If the next significant digit is less than 5, then round down. Otherwise round up. This gives us a figure of 52.84 for the final price.

Now suppose we want to give a promotional discount of 5% off the entire purchase. Do we apply this

Estrogen & Birth Control

Ask Your Doctor About Contraceptive Methods. Learn More.

Prescription contrac.



discount on the 52.8361 figure or the 52.84 figure? What's the difference?

```
Calculation 1: 52.8361 * 0.95 = 50.194295 = 50.19
Calculation 2: 52.84 * 0.95 = 50.198 = 50.20
```


Note that we rounded the final figure by using the standard rounding rule.

See how there's a difference of one cent between the two figures? The old code never bothered to consider rounding, so it always did computations as in Calculation 1. But in the new code we always round before applying promotions, taxes, and so on, just like in Calculation 2. This is one of the main reasons for the one cent error.

Introducing BigDecimal

From the examples in the previous section, it should be clear that we need two things:

1. Ability to specify a scale, which represents the number of digits after the decimal place
2. Ability to specify a rounding method

The `java.math.BigDecimal` class handles both of these considerations. See [BigDecimal Javadocs](#) 

Creating a big decimal from a (scalar) double is simple:

```
bd = new BigDecimal(1.0);
```

To get a `BigDecimal` from a `Double`, get its `doubleValue()` first.

However it is a good idea to use the string constructor:

```
bd = new BigDecimal("1.5");
```

If you don't, then you'll get the following,

```
bd = new BigDecimal(1.5); // is actually 1.4999...
bd.toString(); // => 0.1499999999999999944488848768742172978818416595458984375
```

Rounding and Scaling

To set the number of digits after the decimal, use the `.setScale(scale)` method. However, it is good practice to also specify the rounding mode along with the scale by using `.setScale(scale, roundingMode)`. The rounding mode specifies how to round the number.

Why do we also want to specify the rounding mode? Let's use the BD of 1.5 from above as an example,

```
bd = new BigDecimal(1.5); // is actually 1.4999...
bd.setScale(1); // throws ArithmeticException
```

It throws the exception because it does not know how to round 1.49999. So it is a good idea to always use `.setScale(scale, roundingMode)`.

There are eight choices for rounding mode,

ROUND_CEILING: Ceiling function

```
0.333 > 0.34
-0.333 > -0.33
```

ROUND_DOWN: Round towards zero

```
0.333 > 0.33
-0.333 > -0.33
```

ROUND_FLOOR: Floor function

```
0.333 > 0.33
-0.333 > -0.34
```

ROUND_HALF_UP: Round up if decimal $\geq .5$

```
0.5 > 1.0
0.4 > 0.0
```

ROUND_HALF_DOWN: Round up if decimal $> .5$

```
0.5 > 0.0
0.6 > 1.0
```

ROUND_HALF_EVEN:

Round half even will round as normal. However, when the rounding digit is 5, it will round down if the digit to the left of the 5 is even and up otherwise. This is best illustrated by example,

Estrogen & Birth Control

Ask Your Doctor About Contraceptive Methods.

Learn More.

Prescription contrac.



```
a = new BigDecimal("2.5"); // digit left of 5 is even, so round down
b = new BigDecimal("1.5"); // digit left of 5 is odd, so round up
a.setScale(0, BigDecimal.ROUND_HALF_EVEN).toString() // => 2
b.setScale(0, BigDecimal.ROUND_HALF_EVEN).toString() // => 2
```

The Javadoc says about `ROUND_HALF_EVEN`: Note that this is the rounding mode that minimizes cumulative error when applied repeatedly over a sequence of calculations.

ROUND_UNNECESSARY:

Use `ROUND_UNNECESSARY` when you need to use one of the methods that requires input of a rounding mode but you know the result won't need to be rounded.

When dividing `BigDecimal`s, be careful to specify the rounding in the `.divide(...)` method. Otherwise, you could run into an `ArithmeticException` if there is no precisely rounded resulting value, such as $1/3$. Thus, you should always do:

```
a = b.divide(c, decimals, rounding);
```

Immutability and Arithmetic

`BigDecimal` numbers are immutable. What that means is that if you create a new BD with value "2.00", that object will remain "2.00" and can never be changed.

So how do we do math then? The methods `.add()`, `.multiply()`, and so on all return a new BD value containing the result. For example, when you want to keep a running total of the order amount,

```
amount = amount.add( thisAmount );
```

Make sure you don't do this,

```
amount.add( thisAmount );
```

THIS IS THE MOST COMMON MISTAKE MADE WITH BIGDECIMALS!

Comparison

It is important to never use the `.equals()` method to compare `BigDecimal`s. That is because this equals function will compare the scale. If the scale is different, `.equals()` will return false, even if they are the same number mathematically.

```
BigDecimal a = new BigDecimal("2.00");
BigDecimal b = new BigDecimal("2.0");
print(a.equals(b)); // false
```

Instead, we should use the `.compareTo()` and `.signum()` methods.

```
a.compareTo(b); // returns (-1 if a < b), (0 if a == b), (1 if a > b)
a.signum(); // returns (-1 if a < 0), (0 if a == 0), (1 if a > 0)
```

When to Round: Thoughts on Precision

Now that you can control how to round your calculations, what precision should they be rounded to? The answer depends on how you plan to use the resulting number.

You know what the precision needed for the final result from your user requirements. For numbers which would be added or subtracted to arrive at the final result, you should add one more decimal of precision, so that $0.0144 + 0.0143$ will be rounded to 0.03, whereas if you rounded both to 0.01, you would get 0.02

If you need numbers which would be multiplied to arrive at the final result, you should preserve as many decimal places as possible. Ratios and unit costs, for example, should not be rounded. After the multiplication, you should round your final result.

Appendix

API Reference

- <http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigDecimal.html> 

OFBiz Framework

GenericValues have a method to get any numeric field as a `BigDecimal`,

```
GenericValue orderItem = delegator.findByPrimaryKey(...);
BigDecimal amount = orderItem.getBigDecimal("amount");
```

You may also set values as `BigDecimal`,

```
orderItem.set("amount", new BigDecimal("0"));
```

Sometimes you may want to define a service that accepts BigDecimals instead of Doubles, in which case you'll have a servicedef element like this,

```
<attribute name="amount" type="java.math.BigDecimal" ... />
```

If you write methods in the simple-method minilang, the internal type for numeric values is already BigDecimal. However, it is a good idea to use Java instead so that you may have control over the scale and rounding mode.

Because the rounding mode and scale may be different throughout the applications, it is important to set up separate properties for separate usages. There are utility methods to help you extract the properties.

You may wish to define at the top of each Java class the following code snippet,

```
// set some BigDecimal properties
private static BigDecimal ZERO = new BigDecimal("0");
private static int taxDecimals = -1;
private static int taxRounding = -1;
static {
    decimals = UtilNumber.getBigDecimalScale("myconfig.properties",
        "tax.decimals");
    rounding = UtilNumber.getBigDecimalRoundingMode("myconfig.properties",
        "tax.rounding");
}
```

```
// set zero to the proper scale
ZERO.setScale(decimals);
}
```

The getBigDecimalScale() and getBigDecimalRoundingMode() methods will return the OFBiz-wide default scale and rounding mode if it is unable to read the properties file. These values are scale=2 and roundingMode=ROUND_HALF_UP. If the method fails to read the configured numbers, a warning will appear in the log.

When you store back into the database via entity engine, you'll need to re-cast it back into a double like this:

```
myValue.set("amount", new Double(amountBigDecimal.doubleValue()));
```

This, however, is not really necessary: you can just force the BigDecimal object into the entity engine. Just ignore the warning message.

Minilang

Minilang supports BigDecimal in both the <calculate> and <set> operations. When calculating with minilang, you can use:

```
<calculate field-name="postedBalance" type="BigDecimal">
```

and the calculation will be performed in BigDecimal. The default precision is 2, and the default rounding mode is ROUND_HALF_EVEN. You can also set those yourself by rounding-mode="" and decimal-scale="" attribute tags in <calculate>.

When setting values, you can also use the type="" to convert values to and from BigDecimal, such as:

```
<set from-field="postedBalance" field="updateGlAccountParams.postedBalance" type="Double"/>
```

which converts a BigDecimal result to a Double.

