Four common pitfalls of the BigDecimal class and how to avoid them

Pitfall #1: The double constructor

Pitfall #2: The static valueOf(double) method

Pitfall #3: The equals(bigDecimal) method

Pitfall #4: The round(mathContext) method

A new class extending BigDecimal

Double constructors

Adding new methods

Handling rounding issues

A real-world example

Conclusion

Dig deeper

CODING

Four common pitfalls of the BigDecimal class and how to avoid them

When doing currency calculations in Java, you might use java.math.BigDecimal—but beware of some of that class's unique challenges.

by Frank Kiwy

September 11, 2020

When doing business calculations in Java, especially for currencies, you would preferably use the `java.math.BigDecimal` class to avoid the problems related to floating-point arithmetic, which you might experience if you're using one of the two primitive types: float or double (or one of their boxed type counterparts).

Indeed, the `BigDecimal` class contains a number of methods that can meet most of the requirements of common business calculations.

However, I would like to draw your attention to four common pitfalls of the `BigDecimal` class. They can be avoided when you use the regular BigDecimal API and when you use a new customized class that extends `BigDecimal`.

So, let's start with the regular BigDecimal API.

Pitfall #1: The double constructor

Consider the following example:

```
BigDecimal x = new BigDecimal(0.1);
System.out.println("x=" + x);
```

Here’s the console output:

```
x=0.10000000000000000555111512312578270211815
```

As you can see, the result of this constructor can be somewhat unpredictable. This is because floating-point numbers are represented in computer hardware as base 2 (binary) fractions. However, most decimal fractions cannot be represented exactly as binary fractions. Therefore, the binary floating-point numbers actually stored in the machine only approximate the decimal floating-point numbers you enter. Hence, the value passed to the double constructor is not exactly equal to 0.1.

In contrast, the String constructor is perfectly predictable and produces a `BigDecimal` that is exactly equal to 0.1, as expected.

```
BigDecimal y = new BigDecimal("0.1");
System.out.println("y=" + y);
```

Now the console output is this:

```
y=0.1
```

Therefore, you should use the String constructor in preference to the double constructor.

If, for any reason, a double must be used to create a `BigDecimal`, consider using the static `BigDecimal.valueOf(double)` method. This will give the same result as converting the double to a String using the `Double.toString(double)` method and then using the `BigDecimal(String)` constructor.

Pitfall #2: The static valueOf(double) method

If you are using the static `BigDecimal.valueOf(double)` method to create a `BigDecimal`, be aware of its limited precision.

```
BigDecimal x = BigDecimal.valueOf(1.012345678
BigDecimal y = new BigDecimal("1.012345678901
System.out.println("x=" + x);
System.out.println("y=" + y);
```

The code above produces this console output:

```
x=1.0123456789012346
y=1.01234567890123456789
```

Here, the `x` value has lost four decimal digits because a double has a precision of only 15–17 digits (a float has a precision of only 6–9 digits), while a `BigDecimal` is of arbitrary precision (limited only by memory).

Therefore, it is actually a good idea to use the String constructor, since two major problems caused by the double constructor are effectively avoided.

Pitfall #3: The equals(bigDecimal) method

Let's take a look at this example:

```
BigDecimal x = new BigDecimal("1");
BigDecimal y = new BigDecimal("1.0");
System.out.println(x.equals(y));
```

The console output is the following:

```
False
```

This output is due to the fact that a `BigDecimal` consists of an unscaled integer value with arbitrary precision and a 32-bit integer scale, both of which must be equal to the corresponding values of the other `BigDecimal` that's being compared. In this case

- `x` has an unscaled value of 1 and a scale of 0.
- `y` has an unscaled value of 10 and a scale of 1.

Hence, `x` is not equal to `y`.

For this reason, two instances of `BigDecimal` shouldn't be compared using the `equals()` method, but instead the `compareTo()` method should be used, because it compares the numerical values (`x` = 1; `y` = 1.0) represented by the two instances of `BigDecimal`. Here's an example:

```
System.out.println(x.compareTo(y) == 0);
```

Now the console output is this:

```
True
```

Pitfall #4: The round(mathContext) method

Some developers might be tempted to use the `round(new MathContext(precision, roundingMode))` method to round a `BigDecimal` to (let's say) two decimal places. That's not a good idea.

```
BigDecimal x = new BigDecimal("12345.6789");
x = x.round(new MathContext(2, RoundingMode.H
System.out.println("x=" + x.toPlainString());
System.out.println("scale=" + x.scale());
```

The code above produces the following console output, so `x` is not the expected value of 12345.68 and the scale is not the expected value of 2:

```
x=12000
scale=-3
```

The method doesn't round the fractional part, but it does round the unscaled value to the given number of significant digits (counting from left to right), leaving the decimal point untouched, which results, in the example above, of a negative scale of -3.

So, what happened here?

The unscaled value (123456789) was rounded to two significant digits (12), which represents a precision of 2. However, because the decimal point was left untouched, the real value represented by this `BigDecimal` is 12000.0000. This can also be written as 12000 since the four zeros to the right of the decimal point are meaningless.

But what about the scale? Why is it -3 and not 0, as you would expect for a value of 12000?

That's because the unscaled value of this `BigDecimal` is 12 and, thus, it has to be multiplied by 1000, which is 10 to the power of 3, and (12×10^3) equals 12000.

Hence, a positive scale represents the number of fraction digits (that is, the number of digits to the right of the decimal point), whereas a negative scale represents the number of insignificant digits to the left of the decimal point (in this case, the trailing zeros, since they are only placeholders to indicate the scale of the number).

Finally, the number represented by a BigDecimal is, therefore, unscaledValue x 10^{-scale}.

Also note that the code above used the `toPlainString()` method, which doesn't display the result in scientific notation (1.2E+4).

To get the expected result of 12345.68, try the `setScale(scale, roundingMode)` method, for example:

```
BigDecimal x = new BigDecimal("12345.6789");
x = x.setScale(2, RoundingMode.HALF_UP);
System.out.println("x=" + x));
```

Now the console output is what's expected:

```
x=12345.68
```

The `setScale(scale, roundingMode)` method rounds the fraction part to two decimal places according to the specified rounding mode.

By the way, you could use the `round(new MathContext(precision, roundingMode))` method for conventional rounding. But that would require you to know the total number of digits to the left of the decimal point of the calculation result. Consider the following example:

```
BigDecimal a = new BigDecimal("12345.12345");
BigDecimal b = new BigDecimal("23456.23456");
BigDecimal c = a.multiply(b);
System.out.println("c=" + c);
```

Now the console output is the following:

```
c=289570111.3153564320
```

To round `c` to two decimal places, you would have to use a `MathContext` object with a precision of 11, for example:

```
BigDecimal d = c.round(new MathContext(11, Ro
System.out.println("d=" + d);
```

The code above produces this console output:

```
d=289570111.32
```

The total number of digits to the left of the decimal point can be calculated like this:

```
bigDecimal.precision() - bigDecimal.scale() +
```

where

- `bigDecimal.precision()` is the precision of the unrounded result.

- `bigDecimal.scale()` is the scale of the unrounded result.
- `newScale` is the scale you want to round to.

So, this code

```
BigDecimal e = c.round(new MathContext(c.prec
```

produces this console output

```
e=289570111.32
```

However, if you compare this expression

```
c.round(new MathContext(c.precision() - c.sca
```

to the following expression

```
c.setScale(2, RoundingMode.HALF_UP);
```

it's obvious which one you would choose to ensure readable and concise code.

A new class extending BigDecimal

So far, I've shown the most common pitfalls of the `BigDecimal` class and how they can be avoided. But wouldn't it be better to have a class that could handle most of these issues so you don't risk falling into one of those traps? Well, that's possible by extending the `BigDecimal` class.

I'm going to show you one way this can be achieved.

First, I need a name for the new class; I'm going use "Decimal." The `Decimal` class is going to extend `BigDecimal` and, thus, it inherits all public fields and methods from its superclass.

However, to call your own methods on an instance of the new class, you need to override every method of the `BigDecimal` class that returns a `BigDecimal` instance, so it returns a `Decimal` instance instead. Because there are quite a few methods, I'm going to generate delegate methods with the help of my IDE and then change the code so it returns the correct type.

For instance, this code

```
@Override
public BigDecimal add(BigDecimal augend) {
    return super.add(augend);
}
```

becomes the following

```
@Override
public Decimal add(BigDecimal augend) {
    return new Decimal(super.add(augend));
}
```

Note that besides the new `BigDecimal` instance created by the `super.add()` method, I am also creating a new instance of the `Decimal` class here. But since `BigDecimal` is immutable, I would prefer the new class to be immutable too.

You could also imagine a mutable `Decimal` class, which would avoid the creation of a second object. In that case, you have to be aware that mutability changes the behavior of the new class, which in turn can lead to other pitfalls.

So, the method `a.add(b)` will now return a `Decimal` instance instead of a `BigDecimal` instance.

```
Decimal a = new Decimal("12345.12345");
Decimal b = new Decimal("23456.23456");
Decimal c = a.add(b);
```

On object `c`, I am now able to call my own methods, which do not exist yet. But before creating some new methods, I want to add some constructors to the newly created `Decimal` class.

Since constructors cannot be inherited in Java, because the constructor of the subclass has to have a different name than the constructor of the superclass (because the name of the constructor has to be the name of the class), I have to implement the constructors with the same arguments as those in the superclass.

Below, I'm going to add only those constructors that make sense from a business logic perspective and that will avoid most of the issues discussed above.

- Here's a constructor to create a `Decimal` instance from an int:

  ```
  public Decimal(int val) {
     super(val);
  }
  ```

A constructor with the same argument exists in the superclass, which is also called in the body of the above constructor.

- Here's a constructor to create a `Decimal` instance from a long:

  ```
  public Decimal(long val) {
     super(val);
  }
  ```

  A constructor with the same argument exists in the superclass, which is also called in the body of the above constructor.

- Here's a constructor to create a `Decimal` instance from a double:

  ```
  public Decimal(double val) {
     super(Double.toString(val));
  }
  ```

  A constructor with the same argument exists in the superclass but is not called in the body of the above constructor. Instead, the String constructor is called after the double has been converted to a string to avoid the issues seen previously in pitfall #1. I will come back to this constructor later to discuss possible further issues.

- Here's a constructor to create a `Decimal` instance from a `BigDecimal`:

  ```
  public Decimal(BigDecimal val) {
     super(val.unscaledValue(), val.scale()
  }
  ```

  There is no equivalent constructor in the superclass, because this would not make any sense, except maybe for cloning a `BigDecimal`. To clone, you can use the same constructor as the one called above, which creates a new `BigDecimal` from an unscaled value and a scale.

- Here's a constructor to create a `Decimal` instance from a string representation of a number:

  ```
  public Decimal(String val) {
     super(val);
  }
  ```

  A constructor with the same argument exists in the superclass, which is also called in the body of the above constructor.

- Finally, here's a constructor to create a `Decimal` instance from a formatted string representation of a number:

```
public Decimal(String val, FormatInfo inf
   super(getFormatInstance(info).parse(va
}
```

There is no equivalent constructor in the superclass, since the `BigDecimal` class does not handle any formatting issues.

Double constructors

I'm aware that the new double constructor may lead to issues due to a double's limited precision. Nevertheless, I must admit that I personally prefer the double constructor to the String constructor, because it feels more natural to write numbers as what they are, namely as numbers and not as strings. It is also less error-prone because the numbers you enter are recognized as numbers by the compiler.

I'll give you an example for this: Most countries in Europe use a comma instead of a period as a decimal separator, which could lead to the following error at runtime:

```
new Decimal("1000,45") -> java.lang.NumberFor
```

Another example error might be using nonnumeric characters due to typos with the String constructor:

```
new Decimal("100o.45") -> java.lang.NumberFor
```

These issues can be avoided when you use the double constructor, because your IDE precompiles the code while saving it and, thus, it immediately gives feedback on the code's syntactic correctness.

Although the piece of code `new BigDecimal("1000,45")`, which is syntactically correct, will lead to an exception only at runtime, the code `new BigDecimal(1000,45)`, which is syntactically wrong, produces an error at compile time and, thus, it can be corrected immediately.

Another argument in favor of the double constructor is that you can use the thousands separator to increase the readability of the numbers in your code, which is not possible with the String constructor, for example:

```
new Decimal(1_000_000.45) // -> works fine
new Decimal("1_000_000.45") // -> java.lang.N
```

And the following won't work either because the regular thousands separator is not recognized as such by the String constructor:

```
new Decimal("1,000,000.45") // -> java.lang.N
```

To handle a possible loss of precision, you can check the resulting precision of the newly created `Decimal` and throw an exception if it is greater than 14 (which doesn't mean that it has already lost precision, but it possibly could have).

```
public Decimal(double val) {
   super(Double.toString(val));
   if (precision() > 14) {
       throw new IllegalArgumentException("Po
   }
}
```

Note that the above constructor should help to avoid pitfall #1 and also pitfall #2 in the sense that an `IllegalArgumentException` is thrown in case there's a possible loss of precision, which then would at least not happen but go unnoticed.

However, if you still think the use of the double constructor is too risky, just omit it and use the String constructor instead. That way, you are sure to avoid both pitfalls #1 and #2.

Adding new methods

Now that I have added constructors, I'm going to add some new methods to make it easier to compare two decimal numbers. The recommended way to compare two instances of `BigDecimal` is to use the `compareTo()` method, for example:

```
Decimal a = new Decimal(12345.12345);
Decimal b = new Decimal(23456.23456);
a.compareTo(b) == 0 // false
a.compareTo(b) >= 0 // false
a.compareTo(b) <= 0 // true
a.compareTo(b) > 0  // false
a.compareTo(b) < 0  // true
```

But frankly, this way of comparing numbers is neither obvious nor very readable, and it could potentially lead to misinterpretation. That's why I'd like to introduce the following five new methods for comparing decimal numbers in a clearer and more concise way:

```
•  a.equalTo(b)
•  a.greaterOrEqualTo(b)
```

- `a.lessOrEqualTo(b)`
- `a.greaterThan(b)`
- `a.lessThan(b)`

I believe that anyone reading code that uses these new methods will immediately understand the methods correctly. Note that the `equalTo()` method should help to avoid pitfall #3.

In the implementation of the five new methods, I would of course use the `compareTo()` method, for example:

```
public boolean equalTo(Decimal decimal) {
    return this.compareTo(decimal) == 0;
}
public boolean greaterOrEqualTo(Decimal decim
    return this.compareTo(decimal) >= 0;
}
public boolean lessOrEqualTo(Decimal decimal)
    return this.compareTo(decimal) <= 0;
}
public boolean greaterThan(Decimal decimal) {
    return this.compareTo(decimal) > 0;
}
public boolean lessThan(Decimal decimal) {
    return this.compareTo(decimal) < 0;
}
```

Handling rounding issues

Now, let's address the rounding issues of the `BigDecimal` class by adding the following new method:

```
public Decimal rounding(RoundingInfo info) {
    return setScale(info.scale(), info.mode()
}
```

This method uses an interface, which is implemented by an enum to specify a scale and a rounding mode. The interface lets you create your own `Rounding` enum with rounding types according to your needs.

```
public interface RoundingInfo {

    int scale();
    RoundingMode mode();

}

public enum Rounding implements RoundingInfo

    AMOUNT(2, RoundingMode.HALF_UP),
    RATE(6, RoundingMode.HALF_UP),
    SURFACE(4, RoundingMode.HALF_UP);
    private final int scale;

    private final RoundingMode mode;

    private Rounding(int scale, RoundingMode
        this.scale = scale;
        this.mode = mode;
    }

    @Override
    public int scale() {
        return scale;
    }

    @Override
    public RoundingMode mode() {
        return mode;
    }

}
```

This capability allows you to specify preset rounding settings, the ones you use in your daily work. That way, you do not have to specify either the scale or the rounding mode yourself. (Developers who are not familiar with the different rounding modes, or who ignore the one currently in use, could easily choose the wrong one.) You could even completely ignore the scale and rounding modes as long as you know what type of number you are dealing with (currency, rate, and so on).

```
Decimal a = new Decimal(12345.12345);
Decimal b = new Decimal(23456.23456);
Decimal c = a.multiply(b).round(Rounding.AMOU
System.out.println("c=" + c);
```

The code above produces the following console output:

```
c=289570111.32
```

Note that the new rounding method should help to avoid pitfall #4.

Finally, let's add a method to format the `Decimal`. Why should a class not be able to render itself in an appropriate way? Here's the method to format a Decimal according to the specified format information:

```
public String format(FormatInfo info) {
    return getFormatInstance(info).format(thi
}

private static DecimalFormat getFormatInstanc
    DecimalFormat format = (DecimalFormat) De
    format.applyPattern(info.pattern());
    return format;
}
```

In the same way as for rounding, the following method uses an interface that is implemented by an enum to specify a pattern and a locale:

```
public interface FormatInfo {

    String pattern();

    Locale locale();

}
public enum Format implements FormatInfo {

    AMOUNT("#,##0.00", Locale. US),
    RATE("#,##0.000000", Locale. US),
    SURFACE("#,##0.0000", Locale. US);

    private final String pattern;

    private final Locale locale;

    private Format(String pattern, Locale loc
        this.pattern = pattern;
        this.locale = locale;
    }

    @Override
    public String pattern() {
        return pattern;
    }

    @Override
    public Locale locale() {
        return locale;
    }

}
```

Doing that allows you again to specify preset formats, matching the previously defined rounding settings. In this case, you also do not have to deal with the correct pattern and locale used for the formatting, which in turn leads to less error-prone code. For example, this code

```
Decimal a = new Decimal(12345.12345);
Decimal b = new Decimal(23456.23456);
Decimal c = a.multiply(b).round(Rounding.AMOU
System.out.println("c=" + c.format(Format.AMO
```

produces this console output

```
c=289,570,111.32
```

A real-world example

Now that you are able to address the four pitfalls discussed above and also do some basic formatting operations on the new class, let's see how this all works together in a real-world example.

First, here's an example that calculates compound interest:

```
Decimal principal = new Decimal(12_345.67);
Decimal rate = new Decimal(0.0456);
int compounds = 12;
int years = 7;

Decimal amount = principal.multiply(
        Decimal.ONE.add(
                rate.divide(new Decimal(compo
        ).pow(compounds * years)
).rounding(Rounding.AMOUNT);

assertTrue(amount.equalTo(new Decimal(16_977.

System.out.println("amount=" + amount.format(
```

The code above produces the following console output:

```
amount=16,977.70
```

The following is the same example using the regular BigDecimal API:

```
BigDecimal principal = new BigDecimal("12345.
BigDecimal rate = new BigDecimal("0.0456");
int compounds = 12;
int years = 7;

BigDecimal amount = principal.multiply(
        BigDecimal.ONE.add(
                rate.divide(new BigDecimal(co
        ).pow(compounds * years)
).setScale(2, RoundingMode.HALF_UP);

assertTrue(amount.compareTo(new BigDecimal("1

DecimalFormat formatter = (DecimalFormat) Dec
formatter.applyPattern("#,##0.00");

System.out.println("amount=" + formatter.form
```

The code above produces the following console output:

```
amount=16,977.70
```

You can decide which of these two versions is more readable, more concise, and less error-prone.

Conclusion

You have seen the most common pitfalls of the `BigDecimal` class and learned how to avoid them using either the regular BigDecimal API or a custom `Decimal` class that extends `BigDecimal`.

When it is used correctly, the `BigDecimal` class is well suited for any calculations where decimal values need to remain exact, especially when you are dealing with currencies. Thus, it meets most of the core requirements for business logic developers.

Dig deeper

- "Is it time for operator overloading in Java?"
- "Jakarta EE: Building microservices with Java EE's successor"
- Class BigDecimal
- Uses of Class java.math.BigDecimal
- How BigDecimal extends Number
- "Records come to Java"

Frank Kiwy

Frank Kiwy is a senior software developer and project leader who works for a government IT center in Europe. His focus is on Java SE, Java EE, and web technologies. Kiwy is also interested in software architecture and is committed to continuous integration and delivery. He is currently involved in implementing the European Union¿s Common Agricultural Policy, where he's in charge of several projects. When programming, he values well-designed software with clear and easy-to-understand APIs.

Share this Page