Theory: BigDecimal

57 minutes

Verify to skip   Start practicing

396 users solved this topic. Latest completion was **about 4 hours ago**.

1 required topic

✓ BigInteger In project

Table of contents:

§1. Large numbers in Java

Sometimes programmers have to work with extremely large numbers. Since standard primitive types cannot store them, there are two special classes for this purpose: **BigInteger** for integer numbers and **BigDecimal** for floating-point numbers.

You're already familiar with `BigInteger`, and in this topic, we are going to look at `BigDecimal`.

The size of numbers isn't limited by anything but the physical memory of your computer. In the case of `BigDecimal`, you can have as many digits after the decimal point as you want, which is important for accurate calculations. There are programs where the accuracy of computation is crucial, for example, aircraft or medical software, or the ones for storing and processing finances.

§2. Creating objects of BigDecimal

To create an instance of `BigDecimal`, the first thing you need to do is import this class from `java.math` package using the following statement:

```
import java.math.BigDecimal;
```

The recommended way is to create an instance of `BigDecimal` from the `String` object and specify the desired number in double quotes.

```
BigDecimal firstBigDecimal = new BigDecimal("10000000000000.5897654329");
```

As you can see, the syntax is consistent and pretty simple.

It is worth mentioning that `BigDecimal` has several useful constants, just like `BigInteger`:

```
BigDecimal zero = BigDecimal.ZERO; // 0
BigDecimal one = BigDecimal.ONE;   // 1
BigDecimal ten = BigDecimal.TEN;   // 10
```

§3. Arithmetic operations

It is extremely important to keep in mind that `BigDecimal` is an **immutable** class. Immutability implies that you cannot change an existing instance of `BigDecimal`. If you try to modify an existing object, a new object is created.

Remember: `BigDecimal` numbers are immutable.

You might remember that in the case of `double` and `float`, there are a few potential problems with the floating-point representation. For instance, the result of adding `0.2` to `0.1` won't be `0.3`, which affects the accuracy of further calculations:

```
System.out.println(0.1 + 0.2); // 0.30000000000000004
```

`BigDecimal` has no such problem: the results of all operations will be absolutely correct.

In the code snippet below, you can see some examples of binary methods with `BigDecimal`:

```java
BigDecimal first = new BigDecimal("0.2");
BigDecimal second = new BigDecimal("0.1");

BigDecimal addition = first.add(second);            // 0.3
BigDecimal subtraction = first.subtract(second);    // 0.1
BigDecimal multiplication = first.multiply(second); // 0.02
BigDecimal division = first.divide(second);         // 2
BigDecimal remainder = first.remainder(second);     // 0.0
```

Now, let's take a look at some unary methods:

```java
BigDecimal first = new BigDecimal("0.2");

// absolute value
BigDecimal module = first.abs();  //  0.2
// raise to the power
BigDecimal power = first.pow(3); // 0.008
```

§4. Rounding control

When we need to tweak the accuracy by specifying the number of digits after the point, `setScale()` method comes to the rescue. It allows us to adjust the precision of large fractional numbers:

```java
bigDecimal.setScale(newScale, RoundingMode);
```

The first parameter is `newScale`. It sets the number of digits after the decimal point. Here is how you may receive the scale of your number:

```java
BigDecimal fractionalNumber = new BigDecimal("123.4567");
System.out.println(fractionalNumber.scale()); // 4
```

The second parameter — `roundingMode` — allows us to specify the way your number will be rounded. To use it, you need to perform the import:

```java
import java.math.RoundingMode;
```

You can find the list of all the possible `BigDecimal` rounding modes along with their brief descriptions in the table below:

Mode	Description
CEILING	Rounds upwards, to positive infinity. That is, if the number is positive, it will be equivalent to ROUND_UP; if negative, to ROUND_DOWN
DOWN	Rounds with discarding digits.
FLOOR	Rounds down, to negative infinity. That is, if the number is positive, it will be equivalent to ROUND_DOWN; if negative, to ROUND_UP.
HALF_DOWN	Rounds towards the “nearest neighbor”; if both neighbors are equidistant, rounds down.
HALF_EVEN	Rounds depends on the number to the left of the point. If the number on the left is even, then the rounding will be down. If it is odd, it will be up.
HALF_UP	Rounds towards the “nearest neighbor”; if both neighbors are equidistant, rounds up.
UNNECESSARY	Used when the number does not need rounding, but a rounding mode is expected.
UP	Rounds up, away from zero.

This might seem a little abstract, so let's look at some examples that will help us sort things out.

§5. Rounding mode examples

As you now know, you can adjust the accuracy of your large numbers and choose the rules by which they will be rounded. The following code snippet shows some examples of using `BigDecimal` rounding with different rounding modes:

```java
BigDecimal bigDecimal = new BigDecimal("100.5649");
System.out.println(bigDecimal.setScale(3, RoundingMode.CEILING));     // 100.

bigDecimal = new BigDecimal("0.55");
System.out.println(bigDecimal.setScale(1, RoundingMode.HALF_DOWN));   // 0.5
System.out.println(bigDecimal.setScale(3, RoundingMode.UNNECESSARY)); // 0.55
```

Remember that `BigDecimal` numbers are immutable, so it is not enough to simply apply `setScale()` in order for your number to retain the new value after rounding. You need to assign:

```java
BigDecimal bigDecimal = new BigDecimal("999999999999999999.99999999999999");
bigDecimal.setScale(3, RoundingMode.HALF_UP);
System.out.println(bigDecimal); // 999999999999999999.99999999999999

bigDecimal = bigDecimal.setScale(3, RoundingMode.HALF_UP);
System.out.println(bigDecimal); // 1000000000000000000.000
```

You can easily compare the difference in behavior depending on rounding modes using the table below. These are the examples of Different Rounding Modes, precision is set to 0.

Input Number	CELLING	DOWN	FLOOR	HALF_DOWN	HALF_EVEN	HALF_UP	UP	UNNECESSARY
3.5	4	3	3	3	4	4	4	error
2.5	3	2	2	2	2	3	3	error
1.6	2	1	1	2	2	2	2	error
1.1	2	1	1	1	1	1	2	error
1.0	1	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1	-1
-1.1	-1	-1	-2	-1	-1	-1	-2	error
-1.6	-1	-1	-2	-2	-2	-2	-2	error
-2.5	-2	-2	-3	-2	-2	-3	-3	error
-3.5	-3	-3	-4	-3	-4	-4	-4	error

Note that `UNNECESSARY` will add insignificant zeros to the number if you specify too many digits in `setScale()`. Then again, if you specify too few digits, an error will occur.

§6. Rounding in arithmetic operations

Finally, let's discuss something a bit more advanced: at this point, you should have enough background knowledge for that.

If the result of a division has a non-terminating decimal expansion, it cannot be represented as a `BigDecimal` and `ArithmeticException` happens:

```java
BigDecimal dividend = new BigDecimal("1");
BigDecimal divisor = new BigDecimal("3");

// java.lang.ArithmeticException: Non-terminating decimal expansion; no exact representable decimal result
BigDecimal quotient = dividend.divide(divisor);
```

To avoid it, you need to determine the accuracy of the division result.

```
BigDecimal dividend = new BigDecimal("1");
BigDecimal divisor = new BigDecimal("3");

BigDecimal quotient = dividend.divide(divisor, 2, RoundingMode.HALF_EVEN); //
```

An exact scale is used if the result can be represented by a finite decimal expansion:

```
BigDecimal first = new BigDecimal("20.002");
BigDecimal second = new BigDecimal("10");

BigDecimal division = first.divide(second);   // 2.0002
```

Addition, subtraction, and multiplication have no such peculiarities. Even though precision also matters there and is used in arithmetic operations, it's quite intuitive:

```
BigDecimal first = new BigDecimal("7.7777");
BigDecimal second = new BigDecimal("3.3");

BigDecimal addition = first.add(second);   // 11.0777; The result scale is 4
BigDecimal subtraction = first.subtract(second);   // 4.4777; The result scal
BigDecimal multiplication = first.multiply(second); // 25.66641; The result s
```

Here is how we can describe the accuracy of the result:

- **Addition**: the maximum scale of the addends;
- **Subtraction**: the maximum scale of the minuend and subtrahend;
- **Multiplication**: the sum of the multiplier and multiplicand scales;
- **Division**: the resulting scale, or set manually in the case of the non-terminating decimal expansion.

§7. Conclusion

The `BigDecimal` class is useful for storing large fractional numbers. Standard arithmetic operations are also available for `BigDecimal` numbers. You can manage the rounding behavior of the objects of this class with `setScale()` indicating the desired number of digits as the first parameter and the rounding mode as the second parameter.

Report a typo

59 users liked this piece of theory. 2 didn't like it. **What about you?**

Start practicing   Verify to skip

Comments (2)   Hints (0)   Useful links (2)   Show discussion