BigDecimal in Java 21

In Java 21, the **java.math.BigDecimal class** represents **immutable, arbitrary-precision signed decimal numbers**, primarily used for accurate *monetary and scientific calculations* that avoid the precision limitations of standard float and double primitive types.

It remains a core part of the Java platform with no significant changes in its fundamental behavior in Java 21 compared to earlier versions.

Core Concepts:

Arbitrary Precision: Unlike float and double, BigDecimal is not bound by a fixed number of binary digits for its value representation. Instead, it stores the number as an unscaled BigInteger value and a 32-bit int representing the scale (number of digits to the right of the decimal point).

Accuracy: This internal representation ensures exact arithmetic results, which is crucial for applications where tiny floating-point errors (common with double and float) are unacceptable, such as in finance.

Immutability: Every arithmetic operation (e.g., add(), multiply(), divide()) returns a new BigDecimal instance with the result, rather than modifying the original object. This makes BigDecimal thread-safe but can impact performance compared to primitive types.

Key Methods and Usage:

To use BigDecimal effectively, developers rely on its various methods and constructors:

Creation: The recommended way to create a BigDecimal with an exact value is to use the String constructor or the static BigDecimal.valueOf(double) or BigDecimal.valueOf(long) methods.

```
    import java.math.BigDecimal;

    BigDecimal valueFromInt = BigDecimal.valueOf(100); // Value 100, scale 0
    BigDecimal valueFromString = new BigDecimal("0.01"); // Value 0.01,
exact
    // Avoid the double constructor due to potential imprecision:
    // new BigDecimal(0.1) is not exactly 0.1
```

Arithmetic Operations: The class provides methods for standard arithmetic operations:

add(BigDecimal augend)

subtract(BigDecimal subtrahend)

multiply(BigDecimal multiplicand)

divide(BigDecimal divisor, RoundingMode roundingMode) -

Division requires specifying a RoundingMode to handle non-terminating decimal expansions and avoid ArithmeticException.

Comparison: Use the compareTo(BigDecimal val) method for numerical comparison, as the equals() method considers both value and scale.

```
BigDecimal bd1 = new BigDecimal("2.0");
 BigDecimal bd2 = new BigDecimal("2.00");

 // bd1.equals(bd2) is false (different scales)
 // bd1.compareTo(bd2) == 0 is true (same numerical value)
```

Rounding and Scale Management:

The setScale(int newScale, RoundingMode roundingMode) method is used to control the number of digits after the decimal point and how the value is rounded.

The RoundingMode enum offers various rounding strategies, such as RoundingMode.HALF_UP or RoundingMode.HALF_EVEN.