a1c # java-fundamentals

BigDecimal and BigInteger

TCSS 305 Programming Practicum

This guide explains why Java's primitive numeric types (`double`, `float`, `long`) are insufficient for certain calculations and when to use `BigDecimal` for precise decimal arithmetic and `BigInteger` for arbitrarily large integers. Understanding these types is essential for any application involving money, financial calculations, or values that exceed the range of primitive types.

Why Not double or float?

Before diving into `BigDecimal`, let's understand why you can't simply use `double` or `float` for financial calculations.

The Precision Problem

Try this in any Java program:

```
double price = 0.10;
double quantity = 3;
double total = price * quantity;

System.out.println(total);  // You might expect 0.30
// Actual output: 0.30000000000000004
```

Wait, what? We multiplied 0.10 by 3 and got... not 0.30?

This isn't a bug in Java. It's how binary floating-point arithmetic works on every computer in the world. The number 0.1 cannot be represented exactly in binary, just as 1/3 cannot be represented exactly in decimal (0.333...).

The Real-World Consequences

These tiny errors accumulate. Consider a shopping cart:

```
double subtotal = 0.0;
for (int i = 0; i < 1000; i++) {
    subtotal += 0.10;  // Add 10 cents, 1000 times
```

```
}
System.out.println(subtotal);
// Expected: 100.00
// Actual: 99.9999999999986 (or similar)
```

Now imagine this is a bank processing millions of transactions daily. Those fractions of pennies add up to real money, and worse, your books won't balance.

> **Floating-Point for Money = Bugs**
>
> Using `double` or `float` for monetary calculations is a common source of bugs in software. Banks, e-commerce platforms, and financial applications **must** use `BigDecimal` or integer-based representations (like storing cents as `long`).

Why This Happens

`double` and `float` use IEEE 754 binary floating-point representation. Some decimal fractions that look simple to us (0.1, 0.2, 0.01) cannot be represented exactly in binary, just as 1/3 cannot be written exactly in decimal.

Decimal	Binary Representation	Exact?
0.5	0.1	Yes
0.25	0.01	Yes
0.1	0.00011001100110... (repeating)	No
0.01	Similar repeating pattern	No

> **Important**
>
> As Joshua Bloch states in *Effective Java*: "The `float` and `double` types are particularly ill-suited for monetary calculations because it is impossible to represent 0.1 (or any other negative power of ten) as a `float` or `double` exactly."

What is BigDecimal?

`BigDecimal` is an **immutable**, **arbitrary-precision** decimal number class. Unlike `double`, it can represent decimal values exactly, making it perfect for financial calculations.

Key Characteristics

Property	Meaning
Immutable	Operations return new objects; the original is unchanged
Arbitrary precision	Can represent any decimal value exactly, regardless of size
Explicit scale	You control how many decimal places to maintain
Explicit rounding	You specify how to round when necessary

When to Use BigDecimal

Use `BigDecimal` when:

1. **Money and currency calculations** — The classic use case. Any time you're dealing with dollars, cents, or any currency.
2. **Financial applications** — Interest calculations, tax computations, billing systems.
3. **Scientific calculations requiring exact decimals** — When you need more precision than `double` provides or exact decimal representation.
4. **Any situation where small errors are unacceptable** — Accumulating rounding errors can cause serious problems.

> **When double is Acceptable**
>
> For scientific simulations, graphics, or performance-critical code where approximate results are fine, `double` is often the better choice due to its speed. The key question: **Can my application tolerate tiny rounding errors?** For money: no. For drawing pixels: usually yes.

How to Create BigDecimal Values

The String Constructor (Preferred)

Always create `BigDecimal` from a `String` when you want exact values:

```
// GOOD: Exact representation
BigDecimal price = new BigDecimal("19.99");
BigDecimal taxRate = new BigDecimal("0.0825");
```

The valueOf Method

Use `BigDecimal.valueOf()` when converting from a `double` variable:

```
double userInput = 19.99;

// GOOD: valueOf converts via String representation internally
BigDecimal price = BigDecimal.valueOf(userInput);
```

`valueOf()` is preferable to the `double` constructor because it uses the canonical String representation of the double.

What NOT to Do

```
// DANGEROUS: Captures floating-point imprecision directly
BigDecimal price = new BigDecimal(19.99);
// Creates: 19.98999999999999843680598132777959108352661132812 5

// GOOD: Use the String constructor instead
BigDecimal price = new BigDecimal("19.99");
// Creates: 19.99 (exactly)
```

Never Use the double Constructor

`new BigDecimal(double)` captures the *exact* binary representation of the `double`, including all its imprecision. This defeats the purpose of using `BigDecimal`. Always use the `String` constructor for literals or `valueOf()` for variables.

Predefined Constants

`BigDecimal` provides constants for common values:

```
BigDecimal zero = BigDecimal.ZERO;
BigDecimal one = BigDecimal.ONE;
BigDecimal ten = BigDecimal.TEN;
```

Scale and Rounding

What is Scale?

Scale is the number of digits to the right of the decimal point:

Value	Scale
`1`	0
`1.0`	1
`1.99`	2
`1.990`	3

Setting Scale with setScale

Use `setScale()` to adjust precision:

```
BigDecimal price = new BigDecimal("19.999");

// Round to 2 decimal places
BigDecimal rounded = price.setScale(2, RoundingMode.HALF_EVEN);
// Result: 20.00
```

RoundingMode Options

Java provides several rounding modes in the `java.math.RoundingMode` enum:

RoundingMode	Description	Example: 2.5 becomes
`HALF_EVEN`	Round to nearest neighbor, tie goes to even (banker's rounding)	2
`HALF_UP`	Round to nearest neighbor, tie rounds up	3
`HALF_DOWN`	Round to nearest neighbor, tie rounds down	2
`UP`	Round away from zero	3

RoundingMode	Description	Example: 2.5 becomes
`DOWN`	Round toward zero (truncate)	2
`CEILING`	Round toward positive infinity	3
`FLOOR`	Round toward negative infinity	2
`UNNECESSARY`	Assert that no rounding is needed (throws exception otherwise)	Exception

Use HALF_EVEN for Financial Calculations

`RoundingMode.HALF_EVEN` (banker's rounding) is generally preferred for financial calculations because it minimizes cumulative rounding bias. When the value is exactly halfway between two neighbors, it rounds to the even neighbor, which statistically balances out over many operations.

Avoid Deprecated Rounding Constants

Older code may use integer constants like `BigDecimal.ROUND_HALF_EVEN`. These are deprecated. Always use the `RoundingMode` enum values instead:

```
// DEPRECATED - do not use
price.setScale(2, BigDecimal.ROUND_HALF_EVEN);

// CORRECT - use RoundingMode enum
price.setScale(2, RoundingMode.HALF_EVEN);
```

Practical Rounding Example

```
BigDecimal subtotal = new BigDecimal("99.995");

// Round to 2 decimal places for display
BigDecimal displayPrice = subtotal.setScale(2, RoundingMode.HALF_EVEN);
// Result: 100.00

// For tax calculations, you might use HALF_UP
BigDecimal tax = subtotal.multiply(new BigDecimal("0.0825"))
    .setScale(2, RoundingMode.HALF_UP);
```

BigDecimal Arithmetic

Because `BigDecimal` is immutable, arithmetic operations return **new** objects. The original values are never modified.

Basic Operations

```
BigDecimal price = new BigDecimal("19.99");
BigDecimal quantity = new BigDecimal("3");

// Addition
BigDecimal sum = price.add(new BigDecimal("5.00"));  // 24.99

// Subtraction
BigDecimal difference = price.subtract(new BigDecimal("5.00"));  // 14.99

// Multiplication
BigDecimal total = price.multiply(quantity);  // 59.97

// Division (requires scale and rounding mode for non-terminating results)
BigDecimal half = price.divide(new BigDecimal("2"), 2,
RoundingMode.HALF_EVEN);  // 10.00
```

Division Requires Rounding Mode

Division can produce non-terminating decimals (like 1/3). If you don't specify a scale and rounding mode, Java throws `ArithmeticException` when the result cannot be represented exactly.

```
BigDecimal one = BigDecimal.ONE;
BigDecimal three = new BigDecimal("3");

// WRONG: Throws ArithmeticException
BigDecimal result = one.divide(three);

// RIGHT: Specify scale and rounding
BigDecimal result = one.divide(three, 10,
RoundingMode.HALF_EVEN);
// Result: 0.3333333333
```

Method Chaining

Since each operation returns a new `BigDecimal`, you can chain operations:

```
BigDecimal subtotal = price
    .multiply(quantity)
```

```
    .add(shippingCost)
    .multiply(taxMultiplier);
```

Common Arithmetic Methods

Method	Description	Example
`add(BigDecimal)`	Addition	`a.add(b)`
`subtract(BigDecimal)`	Subtraction	`a.subtract(b)`
`multiply(BigDecimal)`	Multiplication	`a.multiply(b)`
`divide(BigDecimal, scale, RoundingMode)`	Division with rounding	`a.divide(b, 2, RoundingMode.HALF_EVEN)`
`negate()`	Returns negated value	`a.negate()` returns -a
`abs()`	Absolute value	`a.abs()`
`pow(int)`	Raise to integer power	`a.pow(2)`

Comparing BigDecimal Values

Prefer compareTo for Value Comparison

This is one of the most common mistakes with `BigDecimal`:

```
BigDecimal a = new BigDecimal("1.0");
BigDecimal b = new BigDecimal("1.00");

// WRONG: equals considers scale
System.out.println(a.equals(b));  // false!

// RIGHT: compareTo compares mathematical value
System.out.println(a.compareTo(b) == 0);  // true
```

`equals()` in `BigDecimal` checks both **value** and **scale** (number of decimal places). Two `BigDecimal` objects representing the same mathematical value but with different scales are NOT equal according to `equals()`.

The compareTo Rule

Always use `compareTo()` to compare `BigDecimal` values for mathematical equality:

- `a.compareTo(b) == 0` — a equals b
- `a.compareTo(b) < 0` — a is less than b
- `a.compareTo(b) > 0` — a is greater than b

Comparison Examples

```
BigDecimal price = new BigDecimal("19.99");
BigDecimal budget = new BigDecimal("20.00");
BigDecimal zero = BigDecimal.ZERO;

// Check if price is within budget
if (price.compareTo(budget) <= 0) {
    System.out.println("Affordable!");
}

// Check if price is positive
if (price.compareTo(zero) > 0) {
    System.out.println("Price is positive");
}

// Check if two values are mathematically equal
if (price.compareTo(new BigDecimal("19.990")) == 0) {
    System.out.println("Same value!");
}
```

Common BigDecimal Mistakes

Mistake 1: Using the double Constructor

```
// WRONG: Imprecise
BigDecimal bad = new BigDecimal(0.1);
// Actually creates: 0.1000000000000000055511151231257827021181583404541015625

// RIGHT: Exact
BigDecimal good = new BigDecimal("0.1");
```

Mistake 2: Forgetting BigDecimal is Immutable

```
BigDecimal price = new BigDecimal("10.00");

// WRONG: This does nothing! add() returns a new object
```

```
price.add(new BigDecimal("5.00"));
System.out.println(price);  // Still 10.00

// RIGHT: Assign the result
price = price.add(new BigDecimal("5.00"));
System.out.println(price);  // Now 15.00
```

Mistake 3: Using equals for Value Comparison

```
BigDecimal a = new BigDecimal("2.0");
BigDecimal b = new BigDecimal("2.00");

// WRONG: Returns false due to different scales
if (a.equals(b)) { /* ... */ }

// RIGHT: Returns 0 because values are mathematically equal
if (a.compareTo(b) == 0) { /* ... */ }
```

Mistake 4: Division Without Rounding Mode

```
BigDecimal one = BigDecimal.ONE;
BigDecimal three = new BigDecimal("3");

// WRONG: Throws ArithmeticException because 1/3 is non-terminating
BigDecimal result = one.divide(three);

// RIGHT: Specify scale and rounding mode
BigDecimal result = one.divide(three, 10, RoundingMode.HALF_EVEN);
```

Mistake 5: Not Specifying Scale for Money

```
BigDecimal price = new BigDecimal("19.99");
BigDecimal discount = new BigDecimal("0.15");

// Result has 4 decimal places: 16.9915
BigDecimal discounted = price.subtract(price.multiply(discount));

// Better: Explicitly set scale for currency
BigDecimal discounted = price.subtract(price.multiply(discount))
    .setScale(2, RoundingMode.HALF_EVEN);
// Result: 16.99
```

BigInteger: Beyond Long

Why BigInteger?

Java's `long` type can hold values from approximately -9.2 quintillion to 9.2 quintillion. That sounds like a lot, but consider:

```
long maxLong = Long.MAX_VALUE;  // 9,223,372,036,854,775,807

// What's 100 factorial (100!)?
// It's a 158-digit number. Way beyond long's range.
```

`BigInteger` provides **arbitrary-precision integers**—numbers as large as your computer's memory can hold.

When to Use BigInteger

1. **Cryptography** — RSA encryption uses prime numbers hundreds of digits long
2. **Large factorials and combinatorics** — 50! exceeds `long` range
3. **Precise integer arithmetic beyond long** — When overflow is unacceptable
4. **Scientific computing with huge integers** — Astronomical calculations, particle physics

Creating BigInteger Values

```
// From String (most common)
BigInteger huge = new BigInteger("12345678901234567890123456789");

// From long
BigInteger fromLong = BigInteger.valueOf(1000000000000L);

// Predefined constants
BigInteger zero = BigInteger.ZERO;
BigInteger one = BigInteger.ONE;
BigInteger ten = BigInteger.TEN;
```

BigInteger Arithmetic

Like `BigDecimal`, `BigInteger` is immutable and uses method calls for arithmetic:

```
BigInteger a = new BigInteger("123456789012345678901234567890");
BigInteger b = new BigInteger("987654321098765432109876543210");

BigInteger sum = a.add(b);
BigInteger difference = b.subtract(a);
BigInteger product = a.multiply(b);
BigInteger quotient = b.divide(a);
BigInteger remainder = b.remainder(a);
```

Useful BigInteger Methods

Method	Description
`add(BigInteger)`	Addition
`subtract(BigInteger)`	Subtraction
`multiply(BigInteger)`	Multiplication
`divide(BigInteger)`	Integer division
`remainder(BigInteger)`	Remainder after division
`pow(int)`	Raise to power
`mod(BigInteger)`	Modular arithmetic
`gcd(BigInteger)`	Greatest common divisor
`isProbablePrime(int)`	Probabilistic primality test
`compareTo(BigInteger)`	Compare values

Example: Computing Large Factorials

```
public static BigInteger factorial(int n) {
    BigInteger result = BigInteger.ONE;
    for (int i = 2; i <= n; i++) {
        result = result.multiply(BigInteger.valueOf(i));
    }
    return result;
}

// 100! is a 158-digit number
BigInteger oneHundredFactorial = factorial(100);
```

Summary

Concept	Key Point
Why not double for money	Binary floating-point cannot exactly represent 0.1, 0.01, etc.
BigDecimal purpose	Exact decimal arithmetic for money and precision-critical calculations
Creating BigDecimal	Use String constructor: `new BigDecimal("19.99")`
BigDecimal arithmetic	Use methods: `add()`, `subtract()`, `multiply()`, `divide()`
BigDecimal comparison	Use `compareTo()`, not `equals()` for value comparison
Scale and rounding	Use `setScale()` with appropriate `RoundingMode`
BigInteger purpose	Arbitrarily large integers beyond `long` range
Immutability	Both classes are immutable; operations return new objects

Gen AI & Learning: Numeric Precision in AI-Assisted Development

AI coding assistants often generate code using `double` for all numeric calculations because it's simpler. When working with money or financial data, you must recognize this and correct it. Understanding *why* `BigDecimal` is necessary helps you evaluate AI suggestions critically rather than accepting potentially incorrect code.

Further Reading

External Resources

- Oracle JDK: BigDecimal Javadoc - Complete API documentation
- Oracle JDK: BigInteger Javadoc - Complete API documentation
- Oracle JDK: RoundingMode Javadoc - All rounding mode options explained
- Baeldung: BigDecimal and BigInteger - Practical tutorial with examples
- IEEE 754 Floating-Point Standard - Understanding why floating-point has precision issues

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 60: Avoid float and double if exact answers are required.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 3: Fundamental Programming Structures in Java — Section on Big Numbers.

Language Documentation:

- Oracle JDK 25: BigDecimal — Official BigDecimal class documentation
- Oracle JDK 25: BigInteger — Official BigInteger class documentation
- Oracle JDK 25: RoundingMode — Rounding mode enum documentation

Additional Resources:

- IEEE 754-2019 Standard — Floating-point arithmetic specification
- What Every Computer Scientist Should Know About Floating-Point Arithmetic — Classic paper by David Goldberg

---

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.