How the BigDecimal class helps Java get its arithmetic right

When you use Java for simple business arithmetic, you may be surprised to discover that Java doesn't always produce the right answers. If, for example, you use the double data type for an invoice's subtotal, sales tax, and total, your arithmetic expressions may deliver inaccurate results. I'll illustrate this in a moment.

The problem is that floating-point numbers can't represent all decimal numbers with complete accuracy. Then, when you round the results to two decimal places, you can get errors. The best solution in a case like this is to use Java's BigDecimal class, and that's what you'll learn to do in this document.

This article has been excerpted from ***Murach's Java SE 6***, a book by Joel Murach and Andrea Steelman. One of the unique features of that book is its "paired pages" presentation method in which each topic is presented in two pages, with a text page on the left and the related figure page on the right.

To get the most from this "paired pages" method as you read this article, please place the left and right pages side-by-side, whether you view them in the Adobe Reader or print them out. You'll soon find that this approach not only helps you learn faster but also works great for reference.

Then, to learn more about this book, please go to our web site. There, you can find out **how this book differs from other Java books**, view the **table of contents**, download **sample applications**, and more.

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • **www.murach.com**

An Invoice application

To illustrate the math problems that are common when floating-point values are used for business applications, figure 1 presents a simple console application that calculates several values after the user enters an invoice subtotal. You can see the results for one user entry in this figure. This time, the arithmetic is correct.

The code for the application

Figure 1 also presents all of the Java code for this application. Here, the shaded code identifies the double values and arithmetic expressions that are used to do the math that this application requires. After those statements are executed, the results are given percent and currency formats, which round the results. Then, the results are displayed on the console.

You might notice that this application uses the Scanner class that became available with Java 1.5. You might also notice that this application doesn't provide for the exception that's thrown if the user doesn't enter a valid number at the console. Even in this simple form, though, the application will illustrate the math problems that are common with floating-point arithmetic.

The console for the formatted Invoice application

```
Enter subtotal:   150.50
Discount percent: 10%
Discount amount:  $15.05
Total before tax: $135.45
Sales tax:        $6.77
Invoice total:    $142.22

Continue? (y/n):
```

The code for the formatted Invoice application

```
import java.util.Scanner;
import java.text.NumberFormat;

public class InvoiceApp
{
    public static void main(String[] args)
    {
        // create a Scanner object and start while loop
        Scanner sc = new Scanner(System.in);
        String choice = "y";
        while (choice.equalsIgnoreCase("y"))
        {
            // get the input from the user
            System.out.print("Enter subtotal:   ");
            double subtotal = sc.nextDouble();

            // calculate the results
            double discountPercent = 0.0;
            if (subtotal >= 100)
                discountPercent = .1;
            else
                discountPercent = 0.0;
            double discountAmount = subtotal * discountPercent;
            double totalBeforeTax = subtotal - discountAmount;
            double salesTax = totalBeforeTax * .05;
            double total = totalBeforeTax + salesTax;

            // format and display the results
            NumberFormat currency = NumberFormat.getCurrencyInstance();
            NumberFormat percent = NumberFormat.getPercentInstance();
            String message =
                "Discount percent: " + percent.format(discountPercent) + "\n"
              + "Discount amount:  " + currency.format(discountAmount) + "\n"
              + "Total before tax: " + currency.format(totalBeforeTax) + "\n"
              + "Sales tax:        " + currency.format(salesTax) + "\n"
              + "Invoice total:    " + currency.format(total) + "\n";
            System.out.println(message);

            // see if the user wants to continue
            System.out.print("Continue? (y/n): ");
            choice = sc.next();
            System.out.println();
        }
    }
}
```

Figure 1 An Invoice application that will illustrate some arithmetic problems

The math problems in the Invoice application

The console at the top of figure 2 shows more output from the Invoice application in figure 1. But wait! The results for a subtotal entry of 100.05 don't add up. If the discount amount is $10.00, the total before tax should be $90.05, but it's $90.04. Similarly, the sales tax for a subtotal entry of .70 is shown as $0.03, so the invoice total should be $0.73, but it's shown as is $0.74. What's going on?

To analyze data problems like this, you can add *debugging statements* like the ones in this figure. These statements display the unformatted values of the result fields so you can see what they are before they're formatted and rounded. This is illustrated by the console at the bottom of this figure, which shows the results for the same entries as the ones in the console at the top of this figure.

If you look at the unformatted results for the first entry (100.05), you can easily see what's going on. Because of the way NumberFormat rounding works, the discount amount value of 10.005 and the total before tax value of 90.045 aren't rounded up. However, the invoice total value of 94.54725 is rounded up. With this extra information, you know that everything is working the way it's supposed to, even though you're not getting the results you want.

Now, if you look at the unformatted results for the second entry (.70), you can see another type of data problem. In this case, the sales tax is shown as .034999999999999996 when it should be .035. This happens because floating-point numbers aren't able to exactly represent some decimal fractions. As a result, the formatted value is $0.03 when it should be rounded up to $0.04. However, the unformatted invoice total is correctly represented as 0.735, which is rounded to a formatted $0.74. And here again, it looks like Java can't add.

Although trivial errors like these are acceptable in many applications, they are unacceptable in most business applications. And for those applications, you need to provide solutions that deliver the results that you want. (Imagine getting an invoice that didn't add up!)

One solution is to write your own code that does the rounding so you don't need to use the NumberFormat class to do the rounding for you. However, that still doesn't deal with the fact that some decimal fractions can't be accurately represented by floating-point numbers. To solve that problem as well as the other data problems, the best solution is to use the BigDecimal class that you'll learn about next.

Output data that illustrates a problem with the Invoice application

```
Enter subtotal:   100.05
Discount percent: 10%
Discount amount:  $10.00
Total before tax: $90.04
Sales tax:        $4.50
Invoice total:    $94.55

Continue? (y/n): y

Enter subtotal:   .70
Discount percent: 0%
Discount amount:  $0.00
Total before tax: $0.70
Sales tax:        $0.03
Invoice total:    $0.74

Continue? (y/n):
```

Statements that you can add to the program to help analyze this problem

```
// debugging statements that display the unformatted fields
// these are added before displaying the formatted results
String debugMessage = "\nUNFORMATTED RESULTS\n"
                    + "Discount percent: " + discountPercent + "\n"
                    + "Discount amount:  " + discountAmount + "\n"
                    + "Total before tax: " + totalBeforeTax + "\n"
                    + "Sales tax:        " + salesTax + "\n"
                    + "Invoice total:    " + total + "\n"
                    + "\nFORMATTED RESULTS";
System.out.println(debugMessage);
```

The unformatted and formatted output data

```
Enter subtotal:   100.05

UNFORMATTED RESULTS
Discount percent: 0.1
Discount amount:  10.005
Total before tax: 90.045
Sales tax:        4.50225
Invoice total:    94.54725

FORMATTED RESULTS
Discount percent: 10%
Discount amount:  $10.00
Total before tax: $90.04
Sales tax:        $4.50
Invoice total:    $94.55

Continue? (y/n): y

Enter subtotal:   .70

UNFORMATTED RESULTS
Discount percent: 0.0
Discount amount:  0.0
Total before tax: 0.7
Sales tax:        0.034999999999999996
Invoice total:    0.735

FORMATTED RESULTS
Discount percent: 0%
Discount amount:  $0.00
Total before tax: $0.70
Sales tax:        $0.03
Invoice total:    $0.74

Continue? (y/n):
```

Figure 2 The math problems in the Invoice application

How to use the BigDecimal class

The BigDecimal class is designed to solve two types of problems that are associated with floating-point numbers. First, the BigDecimal class can be used to exactly represent decimal numbers. Second, it can be used to work with numbers that have more than 16 significant digits. If you haven't ever used this class, it's one that you should master and use for many business applications.

The constructors and methods of the BigDecimal class

Figure 3 summarizes a few of the constructors that you can use with the BigDecimal class. These constructors accept an int, double, long, or string argument and create a BigDecimal object from it. Because floating-point numbers are limited to 16 significant digits and because these numbers don't always represent decimal numbers exactly, it's often best to construct BigDecimal objects from strings rather than doubles.

Once you create a BigDecimal object, you can use its methods to work with the data. In this figure, for example, you can see some of the BigDecimal methods that are most useful in business applications. Here, the add, subtract, multiply, and divide methods let you perform those operations. The compareTo method lets you compare the values in two BigDecimal objects. And the toString method converts the value of a BigDecimal object to a string.

This figure also includes the setScale method, which lets you set the number of decimal places (*scale*) for the value in a BigDecimal object as well as the rounding mode. For example, you can use the setScale method to return a number that's rounded to two decimal places like this:

```
salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
```

In this example, RoundingMode.HALF_UP is a value in the RoundingMode enumeration that's summarized in this figure. The scale and rounding mode arguments work the same for the divide method.

In case you aren't familiar with enumerations, they are similar to classes. For our purposes right now, you can code the rounding mode as HALF_UP because it provides the type of rounding that is normal for business applications. However, you need to import the RoundingMode enumeration at the start of the application unless you want to qualify the rounding mode like this:

```
java.math.RoundingMode.HALF_UP
```

If you look at the API documentation for the BigDecimal class, you'll see that it provides several other methods that you may want to use. This class also provides many other features that you may want to become more familiar with. But the constructors and methods in this figure will get you started right.

The BigDecimal class

`java.math.BigDecimal`

Constructors of the BigDecimal class

Constructor	Description
`BigDecimal(int)`	Creates a new BigDecimal object with the specified int value.
`BigDecimal(double)`	Creates a new BigDecimal object with the specified double value.
`BigDecimal(long)`	Creates a new BigDecimal object with the specified long value.
`BigDecimal(String)`	Creates a new BigDecimal object with the specified String object. Because of the limitations of floating-point numbers, it's often best to create BigDecimal objects from strings.

Methods of the BigDecimal class

Methods	Description
`add(value)`	Returns the value of this BigDecimal object after the specified BigDecimal value has been added to it.
`compareTo(value)`	Compares the value of the BigDecimal object with the value of the specified BigDecimal object and returns -1 if less, 0 if equal, and 1 if greater.
`divide(value, scale, rounding-mode)`	Returns the value of this BigDecimal object divided by the value of the specified BigDecimal object, sets the specified scale, and uses the specified rounding mode.
`multiply(value)`	Returns the value of this BigDecimal object multiplied by the specified BigDecimal value.
`setScale(scale, rounding-mode)`	Sets the scale and rounding mode for the BigDecimal object.
`subtract(value)`	Returns the value of this BigDecimal object after the specified BigDecimal value has been subtracted from it.
`toString()`	Converts the BigDecimal value to a string.

The RoundingMode enumeration

`java.math.RoundingMode`

Two of the values in the RoundingMode enumeration

Values	Description
`HALF_UP`	Round towards the "nearest neighbor" unless both neighbors are equidistant, in which case round up.
`HALF_EVEN`	Round towards the "nearest neighbor" unless both neighbors are equidistant, in which case round toward the even neighbor.

Description

- The BigDecimal class provides a way to perform accurate decimal calculations in Java. It also provides a way to store numbers with more than 16 significant digits.
- You can pass a BigDecimal object to the format method of a NumberFormat object, but NumberFormat objects limit the results to 16 significant digits.

Figure 3 The constructors and methods for the BigDecimal class

How to use BigDecimal arithmetic in the Invoice application

Figure 4 shows how you can use BigDecimal arithmetic in the Invoice application. To start, look at the console output when BigDecimal is used. As you can see, this solves both the rounding problem and the floating-point problem so it now works the way you want it to.

To use BigDecimal arithmetic in the Invoice application, you start by coding an import statement that imports all of the classes and enumerations of the java.math package. This includes both the BigDecimal class and the RoundingMode enumeration. Then, you use the constructors and methods of the BigDecimal class to create the BigDecimal objects, do the calculations, and round the results when necessary.

In this figure, the code starts by constructing BigDecimal objects from the subtotal and discountPercent variables, which are double types. To avoid conversion problems, though, the toString method of the Double class is used to convert the subtotal and discountPercent values to strings that are used in the BigDecimal constructors.

Since the user may enter subtotal values that contain more than two decimal places, the setScale method is used to round the subtotal entry after it has been converted to a BigDecimal object. However, since the discountPercent variable only contains two decimal places, it isn't rounded. From this point on, all of the numbers are stored as BigDecimal objects and all of the calculations are done with BigDecimal methods.

In the statements that follow, only discount amount and sales tax need to be rounded. That's because they're calculated using multiplication, which can result in extra decimal places. In contrast, the other numbers (total before tax and total) don't need to be rounded because they're calculated using subtraction and addition. Once the calculations and rounding are done, you can safely use the NumberFormat objects and methods to format the BigDecimal objects for display.

When working with BigDecimal objects, you may sometimes need to create one BigDecimal object from another BigDecimal object. However, you can't supply a BigDecimal object to the constructor of the BigDecimal class. Instead, you need to call the toString method from the BigDecimal object to convert the BigDecimal object to a String object. Then, you can pass that String object as the argument of the constructor as illustrated by the last statement in this figure.

Is this a lot of work just to do simple business arithmetic? Relative to some other languages, you would have to say that it is. In fact, it's fair to say that this is a weakness of Java when it is compared to languages that provide a decimal data type. Once you get the hang of working with the BigDecimal class, though, you should be able to solve all of your floating-point and rounding problems with relative ease.

The Invoice application output when BigDecimal arithmetic is used

```
Enter subtotal:   100.05
Subtotal:         $100.05
Discount percent: 10%
Discount amount:  $10.01
Total before tax: $90.04
Sales tax:        $4.50
Invoice total:    $94.54

Continue? (y/n): y

Enter subtotal:   .70
Subtotal:         $0.70
Discount percent: 0%
Discount amount:  $0.00
Total before tax: $0.70
Sales tax:        $0.04
Invoice total:    $0.74

Continue? (y/n):
```

The import statement that's required for BigDecimal arithmetic

```
import java.math.*;  // imports all classes and enumerations in java.math
```

The code for using BigDecimal arithmetic in the Invoice application

```
// convert subtotal and discount percent to BigDecimal
BigDecimal decimalSubtotal = new BigDecimal(Double.toString(subtotal));
decimalSubtotal = decimalSubtotal.setScale(2, RoundingMode.HALF_UP);
BigDecimal decimalDiscountPercent =
    new BigDecimal(Double.toString(discountPercent));

// calculate discount amount
BigDecimal discountAmount =
    decimalSubtotal.multiply(decimalDiscountPercent);
discountAmount = discountAmount.setScale(2, RoundingMode.HALF_UP);

// calculate total before tax, sales tax, and total
BigDecimal totalBeforeTax = decimalSubtotal.subtract(discountAmount);
BigDecimal salesTaxPercent = new BigDecimal(".05");
BigDecimal salesTax = salesTaxPercent.multiply(totalBeforeTax);
salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
BigDecimal total = totalBeforeTax.add(salesTax);
```

How to create a BigDecimal object from another BigDecimal object

```
BigDecimal total2 = new BigDecimal(total.toString());
```

Description

- With this code, all of the result values are stored in BigDecimal objects, and all of the results have two decimal places that have been rounded correctly when needed.
- Once the results have been calculated, you can use the NumberFormat methods to format the values in the BigDecimal objects without any fear of rounding problems. However, the methods of the NumberFormat object limits the results to 16 significant digits.

Figure 4 How to use BigDecimal arithmetic in the Invoice application

Summary

If you haven't used the BigDecimal class before, I hope this article has demonstrated the need for it and will get you started using it. Also, if you like this article and our "paired pages" presentation method, I hope it will encourage you to review our Java SE 6 book or some of our other books.

Unlike many Java books, ***Murach's Java SE 6*** is designed to teach you the essential skills for developing *business* applications. That's why it shows you how to use the BigDecimal class, which isn't even mentioned in many competing books. That's also why it shows you how to do data validation at a professional level and how to develop three-tier database applications, two more essentials that are commonly omitted or neglected in competing books.