Chapter 1: Using the BigDecimal Class

In This Chapter

- **Understanding the problem with double arithmetic in Java**
- **Solving the double problem with the BigDecimal class**
- **Creating BigDecimal objects**
- **Doing arithmetic with BigDecimal objects**
- **Discovering other things you can do with BigDecimal**

You would think that arithmetic would be one area in which any programming language would excel. After all, computers were originally conceived as enormous calculating machines.

Alas, Java doesn't always do arithmetic the way we'd like. In this chapter, you discover why. Then you work around Java's inherent arithmetic limitations by using a special class designed for just that purpose: the `BigDecimal` class. `BigDecimal` is a pain to use, but just about any program that does financial calculations should use it.

Why Java Can't Add

Every schoolboy knows that ten pennies makes a dime. You can write a Java program to prove this simple assertion. I created a `double` variable named `dime` and assigned it the value `0.10`, and then I created another `double` variable named `penny` and assigned it the value `0.01`. Then I made a `for` loop that added the `penny` to a third `double` variable named `tenPennies` ten times. Finally, I used an `if` statement to see whether ten pennies really do equal a dime. The code for this program is shown in Listing BC1-1.

Listing BC1-1: The New Math

```
public class TenPennies
{
    public static void main (String[] args)
    {
        System.out.println("Welcome to the New Math..."
            + "Or, Why Java Can't Add\n");
        double penny = 0.01;          // this is a penny
```

(continued)

Listing BC1-1 *(continued)*

```
        double dime = 0.10;           // this is a dime

        double tenPennies = 0;
        for (int i = 0; i < 10; i++)  // add up 10 pennies
            tenPennies += penny;

        System.out.println("A dime is " + dime);
        System.out.println("Ten pennies is " +
            tenPennies);

        if (tenPennies == dime)
            System.out.println(
                "Ten Pennies is equal to a dime.");
        else
            System.out.println(
                "Ten pennies is NOT equal to a dime!");
    }
}
```

Place your bets now — is 0.01 added ten times equal to 0.10? Here is the console output from this program:

```
Welcome to the New Math...Or, Why Java Can't Add

A dime is 0.1
Ten pennies is 0.09999999999999999
Ten pennies is NOT equal to a dime!
```

Apparently not. Somehow Java managed to come up with 0.0999999999999999 instead of 0.10. And although 0.09999999999999999 is *close* to 0.10, the two values aren't equal.

Double trouble

The root problem here is that Java's `double` type stores values in binary (base 2), not in decimal (base 10). The mathematicians among us know that in any counting system, certain fraction values can't be represented exactly. The best known of these values in decimal is 1/3. Pull out a pocket calculator and divide 1 by 3, and then multiply the result by 3. You should end up with 1, but instead you get 0.999 . . ., with however many nines your calculator can display.

As it turns out, each counting system has a different combination of fractions that can't be represented exactly. And in binary, one of the problem fractions is $\frac{1}{10}$. Thus `double` and `float` types can't accurately represent 0.1.

In most cases, you don't notice these errors because they are insignificant and are often hidden by rounding. For example, try this code:

```
float val = 0.1f;
System.out.println(val);
```

Here Java's default behavior is to format the number so that the inaccuracy is hidden. Thus the output from this program is simply `0.1`. However, you can see the inaccuracy if you use the `NumberFormat` class to format the number with ten significant digits, like this:

```
float val = 0.1f;
NumberFormat nf = NumberFormat.getNumberInstance();
nf.setMinimumFractionDigits(10);
System.out.println(nf.format(val));
```

If you run this code, the value displayed on the console is `0.1000000015`.

Note that in this example I used a `float` instead of a `double` to illustrate the problem. With a `double`, the value of 0.1 is much closer to 0.1 than it is with a `float` because a `double` has twice as many bits to work with. Trust me, though, the inaccuracy is still there.

Another example

The inaccuracy of double arithmetic is especially bad in programs that do financial calculations. Imagine trying to balance your checkbook if every once in a while your calculator makes a one-penny error while doing simple addition or subtraction. (Actually, I think that might happen at my house. Maybe my calculator uses an embedded Java program.)

Listing BC1-2 shows a simple program that lets you enter a number, calculates 5 percent sales tax, and then displays the total and asks if you want to enter another number. Here's a sample of its console:

```
Welcome to the sales tax calculator.

Enter subtotal: 19.95
Subtotal:   $19.95
Sales tax:  $1.00
Total:      $20.95

Again? (Y or N) y

Enter subtotal: 3.49
Subtotal:   $3.49
Sales tax:  $0.17
Total:      $3.66

Again? (Y or N) y

Enter subtotal: 0.70
```

```
Subtotal:   $0.70
Sales tax:  $0.03
Total:      $0.74

Again? (Y or N) n
```

All is well with the first two entries. However, the third one has an embarrassing arithmetic error: 70 plus 3 is *not* 74.

Listing BC1-2: Bad Tax!

```
import java.text.*;
import java.util.*;

public class BadTax
{    static Scanner sc = new Scanner(System.in);
    static NumberFormat cf =
        NumberFormat.getCurrencyInstance();

    public static void main (String[] args)
    {
        double subTotal, salesTax, invoiceTotal;
        double taxRate = 0.05;

        System.out.println(
            "Welcome to the sales tax calculator.");
        do
        {
            System.out.print("\nEnter subtotal: ");
            subTotal = sc.nextDouble();
            sc.nextLine();

            salesTax = subTotal * taxRate;
            invoiceTotal = subTotal + salesTax;

            System.out.print("Subtotal:   ");
            System.out.println(cf.format(subTotal));
            System.out.print("Sales tax:  ");
            System.out.println(cf.format(salesTax));
            System.out.print("Total:      ");
            System.out.println(cf.format(invoiceTotal));
        } while (getAnother());
    }

    static boolean getAnother()
    {
        System.out.print("\nAgain? (Y or N) ");
        if (sc.nextLine().equalsIgnoreCase("Y"))
            return true;
        else
            return false;
    }
}
```

The moral of the story is that you should *never* use `double` to represent decimal values in any application that needs decimal arithmetic to be accurate. That includes most applications that have anything to do with money. You don't want to send your customer an invoice that is inaccurate, do you?

BigDecimal to the Rescue!

If you can't use `double` for decimal values when you need accurate results, what can you use? Fortunately, the Java designers have provided a class that solves the problem for you: `BigDecimal`. The `BigDecimal` class accurately represents a decimal number and provides methods you can use to perform arithmetic and do comparisons with the value.

Internally, the `BigDecimal` class represents decimal values using integers. Because integer arithmetic isn't prone to the same conversion errors that floating-point arithmetic is, the `BigDecimal` class can provide accurate results. In addition, the `BigDecimal` class uses some fancy coding tricks to store numbers of virtually any size. In fact, the `BigDecimal` class is quite capable of representing the national debt down to the penny.

Like strings, `BigDecimal` objects are immutable. Once you create one, you can't change its value. However, many of the `BigDecimal` methods perform some operation on the value (such as addition or subtraction) and return the result. Thus, you often find yourself writing statements like this one when you work with the `BigDecimal` class:

```
totalSales = totalSales.add(quarterlySales);
```

Here the `add` method of the `BigDecimal` `totalSales` is called, and the return value, which is a new `BigDecimal` object, is assigned back to `totalSales`.

Creating BigDecimal Objects

To create a `BigDecimal` object, you call one of the constructors listed in Table BC1-1. Each of these constructors takes a value and converts it to a `BigDecimal` object.

Although you can create `BigDecimal` values from a `double` or `float` value, I recommend against it. The whole point of using `BigDecimal` is to avoid the accuracy errors that are inherent with `double` and `float` values. The only way to do that is to avoid `double` and `float` altogether. And there's an old computer saying: "Garbage In, Garbage Out."

This round goes to C#

As you probably know, C# is Microsoft's answer to Java. If you take a casual look at C#, it looks a lot like Java. And if you take a really close look at C#, it still looks a lot like Java. The class libraries used by Java and C# are similar, but C#'s class library (called the .NET Framework) and Java's API have plenty of differences. But the languages itself are very similar.

Because C# is intended to compete with Java, computer prognosticators frequently compare the two languages. Java usually comes out on top, primarily because it is open source, so you're not tied to one vendor if you use it.

However, C# has a seemingly minor feature Java really should have had from the start: a native `decimal` type. In C#, you can write code like this:

```
public decimal calculateOrderTax(decimal
         subTotal, decimal taxRate)
{
    decimal tax = subTotal * taxRate;
    return subTotal + tax
}
```

No matter how you slice it, that's much easier to code and understand than Java's `BigDecimal` class. Plus it's much more efficient, because decimal is a primitive type.

Unfortunately, there's little chance we'll ever get a primitive decimal type in Java. Adding a new primitive type to the JVM would be a major undertaking.

However, perhaps in the next version of Java, we'll get the ability to overload mathematical operators for classes. Then `BigDecimal` can be enhanced to support the arithmetic operators directly instead of via cumbersome methods such as add, subtract, and so on.

Take these statements, for example:

```
BigDecimal value = new BigDecimal(0.01);
System.out.println(value);
```

Here's what gets printed on the console:

```
0.01000000000000000020816681711721685132943093776702880859375
```

The best way to create a `BigDecimal` object with an initial decimal value is via a string, like this:

```
BigDecimal value = new BigDecimal("0.01");
```

Here `value` has a value of exactly `0.01`.

If the initial value is an integer, you can safely pass it to the constructor. Remember, integers don't have the same accuracy problems that `doubles` and `floats` do. Also, as you see later, you can convert a `BigDecimal` to a `double` solely for the purpose of using the `NumberFormat` class to format the result — as long as you don't use the `double` in any calculations, you won't have to worry about floating-point inaccuracies.

Note the `BigDecimal` class has no default constructor. That's because you can't have a `BigDecimal` object without a value.

Table BC1-1 Constructors BigDecimal Class

Constructor	*Explanation*
`BigDecimal(double val)`	Creates a `BigDecimal` from the `double` value.
`BigDecimal(float val)`	Creates a `BigDecimal` from the `float` value.
`BigDecimal(int val)`	Creates a `BigDecimal` from the `int` value.
`BigDecimal(long val)`	Creates a `BigDecimal` from the `long` value.
`BigDecimal(String val)`	Creates a `BigDecimal` from the `String` value. The string must contain a valid representation of a decimal number.

Doing BigDecimal Arithmetic

The worst part about using the `BigDecimal` class is that you can't use normal arithmetic operators with `BigDecimal` objects. For example, the following code won't compile:

```
BigDecimal subTotal, taxRate, tax, total;
subTotal = new BigDecimal("32.50");
taxRate = new BigDecimal("0.05");
tax = subTotal * taxRate;  // error: won't compile
total = subTotal + tax     // this won't compile either
```

Instead, you have to call methods of the `BigDecimal` class to perform basic arithmetic. These methods all return the result of the calculation as `BigDecimal` objects. For example, here's how you can perform the preceding tax calculation:

```
BigDecimal subTotal, taxRate, tax, total;
subTotal = new BigDecimal("32.50");
taxRate = new BigDecimal("0.05");
tax = subTotal.multiply(taxRate);
total = subTotal.add(tax);
```

Table BC1-2 lists all the arithmetic methods for the `BigDecimal` class. As you can see, there are methods for basic operations such as `add`, `subtract`, `multiply`, and `divide`, as well as some additional operations such as `abs` (absolute power), `pow` (raising the number to a power), and `negative` (changes the sign of the number).

Notice that in addition to normal division, you can use the `divideto IntegralValue` method to return the integer part of the result. Here's an example:

```
BigDecimal a = new BigDecimal("23.5");
BigDecimal b = new BigDecimal("7.0");
BigDecimal c; c = a.divideTo
   IntegralValue(b);
```

After these statements execute, the value of `c` is `3`.

You can also use the `remainder` method to get the remainder from a division:

```
c = a.remainder(b);
```

Here the value of `c` is `2.5`.

BigDecimal reminds me of COBOL

I hate to admit it, but long ago, in a galaxy far away, I was a COBOL programmer. Call me crazy, but there's something about the arithmetic methods of `BigDecimal` that remind me of COBOL.

In COBOL, there were separate statements for doing basic arithmetic operations. Thus you could write statements like this:

```
MULTIPLY SUBTOTAL BY TAX-RATE GIVING TAX.
ADD TAX TO SUBTOTAL GIVING TOTAL.
```

Many COBOL programs had long sections with page after page of statements like that.

The fact that you can't use `BigDecimal` in arithmetic expressions makes you spell out calculations step by step like you had to in COBOL. For example:

```
tax = subTotal.multiply(taxRate);
total = subTotal.add(tax);
```

Sure, there are differences. COBOL was written in uppercase letters and statements ended with periods; Java uses lowercase letters and semicolons. Oh, and there's the whole object-oriented thing, though I hear that the most recent versions of COBOL support that too.

Anyway, it takes me back to the days of *Welcome Back, Kotter* and disco. Sigh.

A common mistake when first learning the `BigDecimal` class is forgetting to assign the result of an arithmetic operation. For example, you might stare at this code loop for hours, wondering why it doesn't seem to work:

```
BigDecimal totalWinnings = new BigDecimal("1000.00");
BigDecimal winningsThisGame = new BigDecimal("200.00");
totalWinnings.add(winningsThisGame);
System.out.println(totalWinnings);
```

The third statement adds `winningsThisGame` to `totalWinnings`, but discards the result. What you probably meant was this:

```
totalWinnings = totalWinnings.add(winningsThisGame);
```

Table BC1-2 Arithmetic Methods of the BigDecimal Class

Method	*Explanation*
`BigDecimal abs()`	Returns the absolute value of this `BigDecimal`.
`BigDecimal add (BigDecimal val)`	Adds the specified `BigDecimal` to this `BigDecimal` and returns the result.
`BigDecimal divide (BigDecimal val)`	Divides this `BigDecimal` by the specified `BigDecimal` and returns the result. May throw `ArithmeticException`.
`BigDecimal[] divideAndRemainder (BigDecimal val)`	Divides this `BigDecimal` by the specified `BigDecimal`. The result and the remainder are returned as a two-element `BigDecimal` array.
`BigDecimal[] divideToIntegralValue (BigDecimal val)`	Divides this `BigDecimal` by the specified `BigDecimal` and returns the integer result.
`BigDecimal max (BigDecimal val)`	Returns the larger of this `BigDecimal` and the specified `BigDecimal`.
`BigDecimal min (BigDecimal val)`	Returns the smaller of this `BigDecimal` and the specified `BigDecimal`.
`BigDecimal multiply (BigDecimal val)`	Multiplies this `BigDecimal` by the specified `BigDecimal` and returns the result.
`BigDecimal negate()`	Negates this `BigDecimal` and returns the result.
`BigDecimal pow (int power)`	Raises this `BigDecimal` to the power specified by the `int` value and returns the result.
`BigDecimal remainder (BigDecimal val)`	Divides this `BigDecimal` by the specified `BigDecimal` and returns the remainder.
`BigDecimal subtract (BigDecimal val)`	Subtracts the specified `BigDecimal` from this `BigDecimal` and returns the result.

Rounding BigDecimal Values

Multiplication and division introduce the need for rounding. For example, suppose your sales tax rate is 5 percent. The tax calculation on a sale of $32.55 would be $1.6275, but not too many people know how to make change for ¾ of a penny. As a result, this sales tax calculation should be rounded up to $1.63.

The `NumberFormat` class automatically rounds results when it converts numbers to strings. That's not real rounding, however, because it doesn't change the underlying values used in the calculations. In other words, you shouldn't leave the sales tax as $1.6275 and just print it as $1.63. Instead, you should actually *change* the sales tax to $1.63. That's the only way to avoid the possibility of additional calculation errors down the line.

Understanding scale

To use rounding with the `BigDecimal` class, you need to understand the idea of *scale.* The scale of a `BigDecimal` number is the number of digits that appear to the right of the decimal point. The `BigDecimal` classes provide several methods that let you work with the scale; they're shown in Table BC1-3.

Table BC1-3 Scaling Methods of the BigDecimal Class

Method	*Explanation*
`int scale()`	Returns the scale of this `BigDecimal`.
`setScale(int scale)`	Returns a new `BigDecimal` with the specified scale.
`setScale(int scale, RoundingMode mode)`	Returns a new `BigDecimal` with the specified scale and rounding mode. See Table BC1-4 for `RoundingMode` values.
`BigDecimal divide (BigDecimal val, int scale, RoundingMode mode)`	Divides this `BigDecimal` by the specified `BigDecimal` and returns the result using the specified scale and rounding mode. See Table BC1-4 for rounding mode.

When you create a `BigDecimal` object, the starting value determines the object's scale. For example, look at this code:

```
BigDecimal num0 = new BigDecimal("1");
BigDecimal num1 = new BigDecimal("1.0");
BigDecimal num2 = new BigDecimal("1.00");
BigDecimal num3 = new BigDecimal("1.000");

int scale0 = num0.scale();
```

```
int scale1 = num1.scale();
int scale2 = num2.scale();
int scale3 = num3.scale();

System.out.println("Scale of " + num0 + " is " + scale0);
System.out.println("Scale of " + num1 + " is " + scale1);
System.out.println("Scale of " + num2 + " is " + scale2);
System.out.println("Scale of " + num3 + " is " + scale3);
```

When you run this code, the following lines appear on the console:

```
Scale of 1 is 0
Scale of 1.0 is 1
Scale of 1.00 is 2
Scale of 1.000 is 3
```

When you use addition and subtraction, the scale of the result is the largest scale between the two values that were added. For example, if you add 25.0 to 1.625, the result is 26.625 with a scale of 3, because the result has three digits to the right of the decimal point.

With multiplication, the resulting scale is the sum of the scales of the two numbers. Thus 1.25 times 1.75 is 2.1875 with a scale of 4. Note that the scales are added even if the number ends with zeros. For example, the result of 1.00 times 2.50 is 2.5000 with a scale of 4.

With division, the scale is exactly what it needs to be to hold the result. For example, the result of 25 divided by 5 is 5, with a scale of 0. But the result of 25 divided by 8 is 3.125, with a scale of 3.

The how-to of rounding

Now that you know about scale, you can see how it plays an important role in rounding. When you say you want to round a number to two decimal places, what you mean is you want the result to have a scale of 2 — and any digits past the second decimal digit should be used to decide whether to round the result up or down.

The `BigDecimal` class accomplishes this using the `setScale` method. The name for this method is a little misleading, so don't be confused. Remember that `BigDecimal` objects are immutable, so calling the `setScale` method doesn't change the scale of an existing `BigDecimal` object. Instead, it creates a new `BigDecimal` object whose scale is different from that of the original. Here's an example:

```
BigDecimal value1 = new BigDecimal("1.0000");
BigDecimal value2 = value1.setScale(2);
```

Here `value2` is assigned the value `1.00` with the scale set to `2`.

Reducing the scale with this version of the `setScale` method always runs the risk of cutting important digits. For example, if `value1` had been set to 1.6125, the last two digits would be cut off. The `BigDecimal` class doesn't like to let that happen, so it throws an `ArithmeticException` in that case.

However, you can use the other version of `setScale` to round the result to a specified scale. Then you have to specify one of the `RoundingMode` enumerations listed in Table BC1-4. In almost all cases, you use `RoundingMode.HALF_UP`. The other rounding modes are strange. I include them here primarily for your enjoyment, so you can see what kinds of things mathematicians and computer scientists dream up while we're trying to figure out how to pay our bills.

Table BC1-4 Members of the RoundingMode Enumeration

Rounding Mode	*Explanation*
`RoundingMode.HALF_UP`	This is the normal way to round things. Excess values that are 5 or greater are rounded up; 4 or less are rounded down. For example, 5.2459 rounded to a scale of 2 is 5.25.
`RoundingMode.HALF_DOWN`	Excess values that are 5 or less are rounded down; 6 or greater are rounded up.
`RoundingMode.HALF_EVEN`	This one is strange. Excess values of exactly 5 are rounded towards the nearest even number. Thus 5.5 rounds up to 6, but 4.5 is rounded down to 4.
`RoundingMode.CEILING`	Numbers are always rounded up regardless of the value of the excess digits.
`RoundingMode.FLOOR`	Numbers are always rounded down regardless of the value of the excess digits.
`RoundingMode.DOWN`	Numbers are always rounded towards zero regardless of the value of the excess digits. That means that positive numbers are rounded down, negative numbers are rounded up.
`RoundingMode.UP`	Numbers are always rounded away from zero regardless of the value of the excess digits. That means positive numbers are rounded up, negative numbers are rounded down.
`RoundingMode.UNNECESSARY`	Numbers are never rounded. The operation throws `ArithmeticException` if the scaling causes digits to be lost.

Getting back to the sales tax calculation, here's how you can avoid charging someone $1.6275 on a $32.55 purchase:

```
salesTax = subTotal.multiply(taxRate);
salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
invoiceTotal = subTotal.add(salesTax);
```

First, you multiply the subtotal by the tax rate. Assuming the scale of both these `BigDecimal` objects is 2, the result now has a scale of 4. Then you use the `setScale` method to change the scale of `salesTax` back to 2, with `HALF_UP` specified as the rounding mode. When that's done, you can continue with your other calculations.

If you're working with an application that does a lot of rounding, you may want to make a little helper method to save you some tedious coding. For example, you can create a `round` method like this:

```
static BigDecimal round(BigDecimal d)
{
    return d.setScale(2, RoundingMode.HALF_UP);
}
```

Then you can change the second line in the previous example to this:

```
salesTax = round(salesTax);
```

Comparing BigDecimal Values

Another annoyance of using the `BigDecimal` class is that you can't use the equality operator (`==`) to compare values. For example, the following code won't work:

```
BigDecimal big1 = new BigDecimal("1.0");
BigDecimal big2 = new BigDecimal("1.0");
if (big1 == big2)
    System.out.println("One equals one");
```

The reason it doesn't work is the same reason you can't compare strings in this way: The equals operator checks for reference equality, not value equality. Thus it always returns `false` unless both variables point to the same instance of the `BigDecimal` class.

To test for value equality, you should use either the `equals` method or the `compareTo` method, as listed in Table BC1-5. The `equals` method returns `true` if both `BigDecimal` objects have the same value and the same scale. The `compareTo` method returns an integer that indicates whether one `BigDecimal` is less than (returns `-1`), equal to (returns `0`), or greater than (returns `+1`) another `BigDecimal`. You use this method in place of the < and > operators.

Table BC1-5	Comparison Methods of the BigDecimal Class
Method	*Explanation*
`int compareTo (BigDecimal val)`	Compares this `BigDecimal` to the specified `BigDecimal`. Returns `-1`, `0`, or `+1` if this `BigDecimal` is less than, equal to, or greater than the specified `BigDecimal`.
`boolean equals (Object val)`	Returns a boolean that indicates whether or not this `BigDecimal` is equal to the specified object.

Consider the following snippet of code:

```
BigDecimal One1 = new BigDecimal("1.0");
BigDecimal One2 = new BigDecimal("1.00");

if (One1.equals(One2))
    System.out.println(
        "Scale doesn't matter with equals");
else
    System.out.println("Scale matters with equals");

if (One1.compareTo(One2) == 0)
    System.out.println(
        "Scale doesn't matter with compareTo");
else
    System.out.println("Scale matters with compareTo");
```

Here both `One1` and `One2` have a value of `1`, but with a different scale. When you run this code, the following lines are displayed on the console:

```
Scale matters with equals
Scale doesn't matter with compareTo
```

As you can see, the scale is considered when you compare values with the `equals` method. As a result, you should use `compareTo` if you aren't sure the scale of the values will be the same.

Here's a little trick for using the `compareTo` method: You can perform the same comparisons provided by the regular relational operators (`<`, `<=`, `==`, `=>`, and `>`) by using these operators to compare the result of the `compareTo` method with 0. For example, to check if `value1` is greater than or equal to `value2`, use an `if` statement like this:

```
if (value1.compareTo(value2) >= 0)   // value1 >= value2
```

Here's a list of how these expressions match up:

Comparison You Want to Make	*Equivalent `compareTo` Expression*
`value 1 == value2`	`value1.compareTo(value2) == 0`
`value 1 != value2`	`value1.compareTo(value2) != 0`
`value 1 > value2`	`value1.compareTo(value2) > 0`
`value 1 >= value2`	`value1.compareTo(value2) >= 0`
`value 1 < value2`	`value1.compareTo(value2) < 0`
`value 1 <= value2`	`value1.compareTo(value2) <= 0`

Converting BigDecimals to Strings

The `BigDecimal` class has several methods that let you convert `BigDecimal` values to strings, doubles, or integers. Table BC1-6 lists those methods.

In most cases, the only reason to convert a `BigDecimal` to a `double` is so you can then use the `double` with the `NumberFormat` class to format the value for output. As long as you don't do any intermediate calculations with the `double`, you won't encounter any accuracy errors unless the numbers are large. (You start to run into problems at about 13 digits, which is large enough even for Bill Gates's bank balance.)

Here's a snippet of code that converts a `BigDecimal` object named `salesTax` to a `double` and uses it with the `NumberFormat` class:

```
NumberFormat cf = NumberFormat.getCurrencyInstance();

double taxD = salesTax.doubleValue();
System.out.println(cf.format(taxD));
```

You can also use the `toString` method to format a `BigDecimal` as a string, but this method doesn't add commas or any other formatting niceties. Note also that this method sometimes switches to exponential notation if the number is large. To avoid that, you can use the `toPlainString` method instead.

Table BC1-6	Conversion Methods of the BigDecimal Class
Method	*Explanation*
`double doubleValue()`	Returns the value of this `BigDecimal` as a `double`.
`int intValue()`	Returns the value of this `BigDecimal` as an `int`.
`long longValue()`	Returns the value of this `BigDecimal` as a `long`.
`String toPlainString()`	Returns a string representation of this `BigDecimal` that doesn't use scientific notation.
`String toString()`	Returns a string representation of this `BigDecimal` that uses scientific notation if necessary.

Sales Tax Revisited

Now that you've seen what you can do with the `BigDecimal` class, Listing BC1-3 presents a `BigDecimal` version of the sales tax program from Listing BC1-2. To prove that `BigDecimal` has solved the addition problems of the first version, here's a sample of the console output from this improved version:

```
Welcome to the sales tax calculator.

Enter subtotal: 19.95
Subtotal:   $19.95
Sales tax:  $1.00
Total:      $20.95

Again? (Y or N) y

Enter subtotal: 3.49
Subtotal:   $3.49
Sales tax:  $0.17
Total:      $3.66

Again? (Y or N) y

Enter subtotal: 0.70
Subtotal:   $0.70
Sales tax:  $0.04
Total:      $0.74

Again? (Y or N) n
```

As you can see, this version of the program properly calculates the tax on a $0.70 purchase at four cents, and then adds the subtotal and tax correctly.

Listing BC1-3: Good Tax!

```
import java.text.*;
import java.math.*;                                          →2
import java.util.*;

public class GoodTax
{    static Scanner sc = new Scanner(System.in);
    static NumberFormat cf
        = NumberFormat.getCurrencyInstance();

    public static void main (String[] args)
    {
        BigDecimal subTotal, salesTax, invoiceTotal;        →13
        BigDecimal taxRate = new BigDecimal("0.05");        →14

        double subD, taxD, totalD;                          →16

        System.out.println(
            "Welcome to the sales tax calculator.");
        do
        {
            System.out.print("\nEnter subtotal: ");

            subTotal = new BigDecimal(sc.nextLine());       →23
            salesTax = subTotal.multiply(taxRate);          →25
            salesTax = round(salesTax);                     →26
            invoiceTotal = subTotal.add(salesTax);          →27

            subD = subTotal.doubleValue();                  →29
            taxD = salesTax.doubleValue();
            totalD = invoiceTotal.doubleValue();

            System.out.print("Subtotal:   ");
            System.out.println(cf.format(subD));
            System.out.print("Sales tax:  ");
            System.out.println(cf.format(taxD));
            System.out.print("Total:      ");
            System.out.println(cf.format(totalD));
        } while (getAnother());
    }

    static BigDecimal round(BigDecimal d)                   →42
    {
        return d.setScale(2, RoundingMode.HALF_UP);
    }

    static boolean getAnother()
```

(continued)

Listing BC1-3 *(continued)*

```
    {
        System.out.print("\nAgain? (Y or N) ");
        if (sc.nextLine().equalsIgnoreCase("Y"))
            return true;
        else
            return false;
    }
}
```

The following paragraphs describe some of the high points of this program:

→2 The `BigDecimal` class is in the `java.math` package, so this `import` statement is required to use it.

→13 This line declares three of the `BigDecimal` variables used by the program.

→14 This line declares the fourth `BigDecimal` variable and initializes it. (The other three are set inside the `do` loop.)

→16 These `double` variables are used only to format the `BigDecimal` values after the values have been calculated.

→23 The subtotal is obtained directly from the `Scanner` object's `nextLine` method. Note that this program throws an exception if you enter invalid data. In an actual program, you want to do some data validation here.

→25 The subtotal is multiplied by the tax rate to calculate the amount of sales tax for the order.

→26 The sales tax is rounded using a helper method named `round`.

→27 The sales tax is added to the subtotal, giving the total for the invoice.

→29 The `BigDecimal` values are converted to `doubles` so they can be formatted with the `NumberFormat` class and printed on the console.

→42 The `round` method uses the `setScale` method to round any `BigDecimal` value to two places.

Whew! That's a lot of work for a measly four cents, isn't it?