**Make cents with BigDecimal**

**Write Java programs to calculate and format currency**

**Summary**  
Java programs must often calculate and format financial figures. To write Java programs that deal with finances, you must know about the BigDecimal class and numeric formatting. This introductory tutorial shows you how to use BigDecimal to calculate basic financial transactions. In addition, you'll learn about numeric formatting as it applies to currency. The result: your programs will not only calculate financials, but also display the results in an appropriate form. (*1,750 words*)

**By Robert Nielsen**

To be able to write Java programs that deal with finances, you need to know about the BigDecimal class and numeric formatting. The first step to teaching you both of those topics is to first create BigDecimal objects. 

We will use the BigDecimal class in the java.math library to hold values. You can create a BigDecimal object in the following manner:   
BigDecimal amount = new BigDecimal("1115.37");

In the above case, the String argument to the BigDecimal constructor determines the value of the object created. The value of "1115.37" might represent, for example, a monthly mortgage payment in dollars, or a checkbook balance. To display the amount, you can use the BigDecimal class's toString() method:

   System.out.println(amount.toString());

A program that creates and displays a BigDecimal amount is shown below:   
import java.math.\*;  
  
public class Mortgage {  
   public static void main(String[] args) {  
      BigDecimal payment = new BigDecimal("1115.37");  
      System.out.println(payment.toString());  
   }  
}

Output from the above program is:   
1115.37

**Formatting currency**   
Since we are dealing with money, it would be nice to have the amounts held by BigDecimal objects properly formatted, which for US currency would include a dollar sign and a comma as a thousands separator. (For other currencies, please see the section **Currency of Other Countries** below). The NumberFormat class, found in the java.text library, can create an appropriate object for US currency with the following code:   
     NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);

Note that the Locale class, used as an argument for the getCurrencyInstance() method above, is found in the java.util library.

The NumberFormat's format() method, which we will be using next, takes a double or long primitive as an argument, so first we will turn the BigDecimal object into a double using BigDecimal's doubleValue() method:   
double doublePayment = payment.doubleValue();

Now we use NumberFormat's format() method to create a String:   
String s = n.format(doublePayment);

Putting these steps in a program, we then have:   
import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Mortgage2 {  
   public static void main(String[] args) {  
      BigDecimal payment = new BigDecimal("1115.37");  
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double doublePayment = payment.doubleValue();  
      String s = n.format(doublePayment);  
      System.out.println(s);  
   }  
}

Output from the above program is:   
$1,115.37

Readers should note that creating a double value involves a small loss in the value's accuracy. While the inaccuracies are too small to be seen in this article's examples, they are visible in very large amounts. Therefore, you cannot rely upon NumericFormat to produce accurate results with very large numbers (about 13 or more digits).

**Currencies of other countries**   
In the previous example, we used Locale.US as the argument passed to the getCurrencyInstance() method to specify the currency of the country (United States) with which we'd be working. Java is not limited to working with US currency however. For example, you would use Locale.GERMANY, Locale.FRANCE, or Locale.ITALY to specify the currencies of Germany, France, and Italy, respectively. The topic of internationalization is a subject in its own right; see the [Resources](http://www.javaworld.com/javaworld/jw-06-2001/jw-0601-cents_p.html#resources#resources) section for a link to more information.

**BigDecimal operations**   
BigDecimal methods for adding and subtracting numbers are add() and subtract(), respectively. For example, to add 1,115.37 and 115.37, we could do the following:

BigDecimal balance = new BigDecimal("1115.37");  
BigDecimal transaction = new BigDecimal("115.37");  
BigDecimal newBalance = balance.add(transaction);

The BigDecimal's newBalance object now holds the value of 1,230.74. Similarly, to subtract 115.37 from 1,115.37, we could use this code:

BigDecimal balance = new BigDecimal("1115.37");  
BigDecimal transaction = new BigDecimal("115.37");  
BigDecimal newBalance2 = balance.subtract(transaction);

The BigDecimal's newBalance2 object now holds the value of 1,000.00. (Naturally, if we are talking about checkbook balances in real life, the subtract() method will be used much more often than the add() method, and the total amount subtracted from the checkbook balance will exceed the total amount added, or so it often seems.) You can accomplish multiplying and dividing with BigDecimal's multiply() and divide() methods. Multiplying is demonstrated in the following program:

import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Multiply {  
   public static void main(String[] args) {  
      BigDecimal d = new BigDecimal("1115.32");  
      BigDecimal taxRate = new BigDecimal("0.0049");  
      BigDecimal d2 = d.multiply(taxRate);  
      System.out.println("Unformatted: " + d2.toString());   
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double money = d2.doubleValue();  
      String s = n.format(money);  
      System.out.println("Formatted:  " + s);  
   }  
}

The output for the above code is shown below:   
Unformatted:   5.465068  
Formatted:      $5.46

Note the extra decimal places in the unformatted BigDecimal object as compared to the formatted output. In addition, formatting the value of the BigDecimal object causes the fraction -- greater than one half -- to be dropped. To manage the extra decimal places and the lack of rounding, we can use BigDecimal's setScale() method to set the number of decimal places. When using setScale(), we need to specify not only the number of decimal places, but how the number will be rounded, if rounding is necessary. The most common way of rounding -- round up fractions half or greater, and round down all other fractions -- can be specified with BigDecimal's constant ROUND\_HALF\_UP. Therefore, to set the number of decimal places to two and specify that fractions half and greater will be rounded up, we can write:   
d2 = d2.setScale(2, BigDecimal.ROUND\_HALF\_UP);

Modifying the above program to add setScale(), we now have:   
import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Multiply2 {  
   public static void main(String[] args) {  
      BigDecimal d = new BigDecimal("1115.32");  
      BigDecimal taxRate = new BigDecimal("0.0049");  
      BigDecimal d2 = d.multiply(taxRate);  
      d2 = d2.setScale(2, BigDecimal.ROUND\_HALF\_UP);  
      System.out.println("Unformatted: " + d2.toString());   
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double money = d2.doubleValue();  
      String s = n.format(money);  
      System.out.println("Formatted:  " + s);  
   }  
}

Now the output is:   
Unformatted:   5.47  
Formatted:      $5.47

Now the BigDecimal value is rounded to two digits, rounding the value up, and the formatted String correctly displays the rounded value. Other constants useful in rounding are ROUND\_HALF\_DOWN and ROUND\_HALF\_EVEN. The first, ROUND\_HALF\_DOWN, rounds fractions of half and under down, and all others up. The second, ROUND\_HALF\_EVEN, rounds half fractions to the even number (e.g., 2.5 rounds to 2, while 3.5 rounds to 4), and fractions greater or less than half to the closest integer. When dividing BigDecimal objects, we are required to specify how the result will be rounded. For this article, we will round halves up. The following program shows some sample division:   
import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Divide {  
   public static void main(String[] args) {  
      BigDecimal d = new BigDecimal("1115.32");  
      BigDecimal days = new BigDecimal("30");  
      BigDecimal d2 = d.divide(days, 2, BigDecimal.ROUND\_HALF\_UP);   
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double money = d2.doubleValue();  
      String s = n.format(money);  
      System.out.println(s);  
   }  
}

Output from the above program is:   
$37.18

**Calculating interest**   
For this example, assume that a sum of $9,500 will receive interest payments at an annual rate of 6.7 percent. Payments will be calculated quarterly, and we will calculate the first quarterly payment. To do so, we will use the formula *I=PRT*, where *I* is the amount of interest, *P* is the principal (9,500), *R* is the rate (6.7 percent annually), and *T* is the time (0.25 years). The program is:   
import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Interest {  
   public static void main(String[] args) {  
      BigDecimal principal = new BigDecimal("9500.00");  
      BigDecimal rate = new BigDecimal("0.067");  
      BigDecimal time = new BigDecimal("0.25");  
      BigDecimal temp = principal.multiply(rate);  
      BigDecimal interest = temp.multiply(time);  
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double money = interest.doubleValue();  
      String s = n.format(money);  
      System.out.println("First quarter interest:  " + s);    }  
}

Output from the above program is:   
First quarter interest: $159.12

**Mutual fund transactions**   
In this example, an investor owns 754.495 shares in a mutual fund. The investor makes an additional $200.00 purchase of shares at $10.38 per share. We will use the following Java program to answer two questions: How many shares does the investor own after purchase, and what is the current market value of the account after the purchase? We will assume that the mutual fund keeps track of share numbers to three decimal places:   
import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Mutual {  
   public static void main(String[] args) {  
      BigDecimal shares = new BigDecimal("754.495");  
      BigDecimal purchaseAmount = new BigDecimal("200.00");   
      BigDecimal pricePerShare = new BigDecimal("10.38");  
      BigDecimal sharesPurchased = purchaseAmount.divide(pricePerShare, 3, BigDecimal.ROUND\_HALF\_UP);  
      shares = shares.add(sharesPurchased);  
      BigDecimal accountValue = shares.multiply(pricePerShare);   
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double dAccountValue = accountValue.doubleValue();  
      String sAccountValue = n.format(dAccountValue);  
      System.out.println("Number of shares = " + shares.toString());   
      System.out.println("Account value = " + sAccountValue);   }  
}

The above program outputs:   
Number of shares = 773.763  
Account value = $8,031.66

**More formatting**   
In the above example, the number of shares happens to be less than 1,000. If it had been greater than 1,000, the program would have outputted the number without a comma to separate the thousands place from the other digits. We can create a NumberFormat object to format numbers in the US style (commas separate thousands, periods separate decimals) by using:   
NumberFormat n2 = NumberFormat.getInstance(Locale.US);

Modifying the previous program to increase the number of shares to more than 1,000 and to display the number of shares formatted as we desire, we have:   
import java.math.\*;  
import java.text.\*;  
import java.util.\*;  
  
public class Mutual2 {  
   public static void main(String[] args) {  
      BigDecimal shares = new BigDecimal("1754.495");  
      BigDecimal purchaseAmount = new BigDecimal("2000.00");   
      BigDecimal pricePerShare = new BigDecimal("10.38");  
      BigDecimal sharesPurchased = purchaseAmount.divide(pricePerShare, 3, BigDecimal.ROUND\_HALF\_UP);  
      shares = shares.add(sharesPurchased);  
      BigDecimal accountValue = shares.multiply(pricePerShare);   
      NumberFormat n = NumberFormat.getCurrencyInstance(Locale.US);   
      double dAccountValue = accountValue.doubleValue();  
      String sAccountValue = n.format(dAccountValue);  
      NumberFormat n2 = NumberFormat.getInstance(Locale.US);   
      double dShares = shares.doubleValue();  
      String sShares = n2.format(dShares);  
      System.out.println("Number of shares = " + sShares);   
      System.out.println("Account value = " + sAccountValue);   }  
}

The modified version (directly above) now outputs:   
Number of shares = 1,947.173  
Account value = $20,211.66