8.15 Using `BigDecimal` for Precise Monetary Calculations

In earlier chapters, we demonstrated monetary calculations using values of type `double`. In Chapter 5, we discussed the fact that some `double` values are represented *approximately*. Any application that requires precise floating-point calculations—such as those in financial applications—should instead use class `BigDecimal` (from package `java.math`).[1]

[1] Dealing with currencies, monetary amounts, conversions, rounding and formatting is complex. The new JavaMoney API (`http://javamoney.github.io`) was developed to meet these challenges. At this time, JavaMoney is not part of Java SE or Java EE. Exercise 8.22 asks you to investigate Java-Money and use it to build a currency converter app.

Interest Calculations Using `BigDecimal`

Figure 8.16 reimplements the interest-calculation example of Fig. 5.6 using objects of class `BigDecimal` to perform the calculations. We also introduce class `NumberFormat` (package `java.text`) for formatting numeric values as *locale-specific* `String`s—for example, in the U.S. locale, the value 1234.56, would be formatted as `“1,234.56”`, whereas in many European locales it would be formatted as “1.234,56”.

```
// Fig. 8.16: Interest.java
// Compound-interest calculations with BigDecima
import java.math.BigDecimal;
import java.text.NumberFormat;

public class Interest {
   public static void main(String args[]) {
      // initial principal amount before interes
      BigDecimal principal = BigDecimal.valueOf(
      BigDecimal rate = BigDecimal.valueOf(0.05)

      // display headers
      System.out.printf(“%s%20s%n”, “Year”, “Amo

      // calculate amount on deposit for each of
      for (int year = 1; year <= 10; year++) {
         // calculate new amount for specified y
         BigDecimal amount =
            principal.multiply(rate.add(BigDecim

         // display the year and the amount
         System.out.printf(“%4d%20s%n”, year,
            NumberFormat.getCurrencyInstance().f
      }
   }
}
```

```
Year   Amount on deposit
   1           $1,050.00
   2           $1,102.50
   3           $1,157.62
   4           $1,215.51
   5           $1,276.28
```

```
                    6            $1,340.10
                    7            $1,407.10
                    8            $1,477.46
                    9            $1,551.33
                   10            $1,628.89
```

Fig. 8.16

Compound-interest calculations with `BigDecimal`.

Creating `BigDecimal` Objects

Lines 9–10 declare and initialize `BigDecimal` variables `principal` and `rate` using the `Big-Decimal static` method `valueOf` that receives a `double` argument and returns a `BigDecimal` object that represents the *exact* value specified.

Performing the Interest Calculations with `BigDecimal`

Lines 18–19 perform the interest calculation using

BigDecimal methods multiply, add and pow. The expression in line 19 evaluates as follows:

1. First, the expression rate.add(BigDecimal.ONE) adds 1 to the rate to produce a BigDecimal containing 1.05—this is equivalent to 1.0 + rate in line 15 of Fig. 5.6. The BigDecimal constant ONE represents the value 1. Class BigDecimal also provides the commonly used constants ZERO (0) and TEN (10).
2. Next, BigDecimal method pow is called on the preceding result to raise 1.05 to the power year—this is equivalent to passing 1.0 + rate and year to method Math.pow in line 15 of Fig. 5.6.
3. Finally, we call BigDecimal method multiply on the principal object, passing the preceding result as the argument. This returns a BigDecimal representing the amount on deposit at the end of the specified year.

Since the expression rate.add(BigDecimal.ONE) produces the same value in each iteration of the loop, we could have simply initialized rate to 1.05 in line 10 of Fig. 8.16; however, we chose to mimic the precise calculations we used in line 15 of Fig. 5.6.

Formatting Currency Values with NumberFormat

During each iteration of the loop, line 23 of Fig. 8.16

```
NumberFormat.getCurrencyInstance().format(amount)
```

evaluates as follows:

1. First, the expression uses `NumberFormat`'s `static` method `getCurrencyInstance` to get a `NumberFormat` that's preconfigured to format numeric values as locale-specific currency `String`s—for example, in the U.S. locale, the numeric value 1628.89 is formatted as $1,628.89. Locale-specific formatting is an important part of **internationalization**—the process of customizing your applications for users' various locales and spoken languages.
2. Next, the expression invokes method `NumberFormat` method `format` (on the object returned by `getCurrencyInstance`) to perform the formatting of the `amount` value. Method `format` then returns the locale-specific `String` representation. For the U.S. locale, the result is rounded to two digits to the right of the decimal point.

Rounding `BigDecimal` Values

In addition to precise calculations, `BigDecimal` gives you control over rounding—by default all calculations are exact and *no* rounding occurs. If you do not specify how to round `BigDecimal` values and a given value cannot be represented exactly—such as the result of 1 divided by 3, which is 0.3333333…—an `ArithmeticException` occurs.

Though we do not do so in this example, you can specify the *rounding mode* for `Big-Decimal` by supplying a `MathContext` object (package `java.math`) to class `BigDecimal`'s constructor when you create a `BigDecimal`. You may also provide a `MathContext` to various `BigDecimal` methods that perform calculations.

Class `MathContext` contains several preconfigured `MathContext` objects that you can learn about at

```
http://docs.oracle.com/javase/8/docs/api/java/math/Ma
```

By default, each preconfigured `MathContext` uses so-called “banker’s rounding” as explained for the `RoundingMode` constant `HALF_EVEN` at:

```
http://docs.oracle.com/javase/8/docs/api/java/math/Ro
```

Scaling `BigDecimal` Values

A `BigDecimal`’s scale is the number of digits to the right of its decimal point. If you need a `BigDecimal` rounded to a specific digit, you can call `BigDecimal` method `setScale`. For example, the following expression returns a `BigDecimal` with two digits to the right of the decimal point and using banker’s rounding:

```
amount.setScale(2, RoundingMode.HALF_EVEN)
```