Java & Monetary Data

A standalone class to tackle the problem

John N. Armstrong

Java and J2EE have become ubiquitous as a platform of choice for creating robust enterprise e-commerce applications. Although these kinds of applications typically involve operations on monetary data (computing unit and extended prices, tax amounts, shipping costs, and so on), Java does not provide suitable mechanisms for dealing with this type of data.

Consequently, programmers are often faced with the problem of how to reliably represent, store, and manipulate monetary data. Obviously, when dealing with money, the importance of "getting it right" cannot be overstated. For these and similar reasons, an e-commerce application architecture should include a consistent mechanism for dealing with monetary data.

In this article, I'll examine some of the more common techniques for dealing with monetary data, and show why these techniques are less than optimal. I also present *Money*, a standalone Java class that addresses the problem of dealing with monetary data.

John is a senior partner and owner of Objective Logic L.L.P. in McKinney, Texas, a software consulting firm specializing in Java-centric architectures and solutions. He can be reached at john@objectivelogic.com.

The Nature of Money

The unit of measurement for U.S. currency is the cent; 100 cents make up one dollar. By convention, a monetary value is represented as a whole dollar amount and a fractional dollar amount (cents), normally written as a decimal value with exactly two digits to the right of the decimal point. It is important to understand that monetary values are not "real" numbers in the strict sense of the word. Whereas real numbers form a continuum of values, monetary data forms a discrete set of values. For instance, the fractional component of a dollar amount (cents) can take on any one of exactly one hundred possible discrete values—0,1,2,3,…99.

Most programming languages (including Java) offer so-called "native data types" that are directly or indirectly related to the hardware's ability to represent, store, and perform arithmetic operations on numeric data. For example, most languages provide integer data types for representing integral numbers, and floating-point data types for representing real numbers. Unfortunately, native data types are generally not well suited for use with monetary data. (See the accompanying text box entitled "Decimal Data.")

Integer Data Types

Since an integer data type can only represent integral values, using an integer data type with a monetary value normally requires the assumption of an implicit decimal point. To accommodate both whole dollars and cents, a monetary value must be scaled (multiplied) by some factor (such as 100) and stored as an integer value. For example, the monetary value $19.95 could be stored as the integer value 1995; in this case, there would be an implicit decimal point between the dollars (19) and the cents (95).

The use of integer data types to represent monetary values suffers from a number of significant drawbacks. Integer data types are limited in the magnitude of values that they can represent. More importantly, doing arithmetic on scaled integer values (in particular, multiplication and division) can be messy and prone to programming errors. Intermediate results must be appropriately scaled to ensure the preservation of all significant digits, and you must keep track of where the implied decimal point is located. Because of these and other issues, integer data types are not a practical choice for use with monetary data.

Floating-Point Data Types

Floating-point data types represent real numbers (numbers having both an integral and a fractional component) and would seem to be the logical choice for use with monetary data. However, the use

(continued from page 52)

of floating-point data types with monetary data is somewhat problematic.

While floating-point data types are capable of representing extremely large positive and negative numbers and offer the equivalent of many decimal digits of precision, they are nonetheless inexact when it comes to representing decimal numbers. The reason for this stems from the fact that computers are binary, not decimal, machines. Internally, the hardware must represent a decimal number using a binary format, and most real decimal numbers cannot be exactly represented in a fixed-length binary format.

To illustrate, Example 1 uses Java's 32-bit *float* data type to store a unit cost of $9.48 for an item. Java's *BigDecimal* class is used to obtain a true rendering of the value of the *unitCost* variable. When the code is executed, the following output is produced:

```
The unit cost is $9.48
The float value is $9.479999542236328125
```

Because the decimal value 9.48 cannot be represented exactly as a binary floating-point value, it must be represented as a close approximation—in this case, 9.479999542236328125. Fortunately, Java's built-in *float*-to-*String* conversion methods can identify such an approximation and display the value as 9.48, instead of 9.479999542236328125.

Now, suppose you wish to compute the total cost of 100 items having a unit cost of $9.48:

```
float totalCost = unitCost * 100.0f;
System.out.println("Total Cost: $" + totalCost);
```

When this code is executed, the following output is produced:

```
Total Cost: $947.99994
```

In this case, the effects of inexactness when using a floating-point data type to represent a monetary value are more obvious. If the computed value is simply truncated to two decimal places, the result is $947.99, which is definitely wrong, and would cost you a penny if you were preparing a customer invoice. The problem only gets worse as the computations become more complex.

Java's *BigDecimal* Class

Java's *BigDecimal* class is specifically designed to address the problems of dealing with decimal numbers. It offers the ability to deal with very large decimal values, while maintaining accuracy and precision. *BigDecimal* can be used with monetary data and avoids some of the pitfalls associated with using native data types. In essence, *BigDecimal* implements a virtual signed decimal integer having arbitrary precision and an implied decimal point.

BigDecimal does overcome many of the limitations of using Java's native data types. For one thing, a *BigDecimal* object can store and manipulate very large values. Furthermore, since a *BigDecimal* object represents a value internally as a decimal (not binary) number, it can represent a monetary value exactly. And methods are provided for performing the usual types of manipulation that are required when working with monetary data, such as addition, subtraction, multiplication, division, and so forth.

However, using *BigDecimal* directly with monetary data is a somewhat cumbersome and inelegant solution because the *BigDecimal* class is designed to address the

Decimal Data

Some machines have special hardware instructions and native data types specifically designed for dealing with decimal data (of which monetary data is the most common example). IBM's venerable 360/370/390 series of mainframes has an entire instruction set devoted to handling decimal data. A special variable-length data type known as "packed decimal" encodes a decimal value as a series of 4-bit decimal digits (0–9) with each byte containing two digits. Cobol, widely used for data-processing applications on IBM mainframes, makes extensive use of the packed decimal data format and the decimal instruction set to process monetary data.

—J.N.A.

```
float unitCost = 9.48f;
System.out.println("The unit cost is $" + unitCost);
BigDecimal fpValue = new BigDecimal(unitCost);
System.out.println("The float value is $" + fpValue);
```

Example 1: *Using the* float *data type with monetary data.*

SHOW WHAT YOU KNOW.

OBJECT MANAGEMENT GROUP • CERTIFICATION PROGRAM • UML™

The new UML Certification Program from OMG

If you're using OMG's Unified Modeling Language™ (UML®), here's your opportunity to show the world you know your stuff. OMG's new worldwide UML Certification Program provides an objective measure of your UML 2.0 specification knowledge—and tells the rest of the IT community that you have serious credentials.

■ Three Certification Levels
The new UML Certification Program offers three levels of certification:

- *OMG-Certified UML Professional Fundamental*
- *OMG-Certified UML Professional Intermediate*
- *OMG-Certified UML Professional Advanced*

Each level has its own examination, testing progressively more complex concepts and usage of the UML 2.0 Specification.

■ Regional Test Centers
UML Certification examinations will be conducted by OMG's testing partner, Thompson Prometric, which has more than 3,500 test centers around the world. Examinations are computer-based and multiple-choice.

■ Time to Get Serious
As the use of UML grows by leaps and bounds, more and more development organizations are looking for developers with UML expertise. Show What You Know, get certified today and bring the value of being an *OMG-Certified UML Professional* to yourself and your organization.

For more information on the OMG/UTI UML Certification Program, please visit: http://www.omg.org/ddj or call OMG at (781) 444-0404.

OMG® OBJECT MANAGEMENT GROUP

generic problem of dealing with decimal data. As such, it doesn't offer a semantically clean solution for dealing with monetary data. For example, users must explicitly deal with issues such as round-off, scaling, formatting, and so on. This adds extra code to the application, and obscures the application's use of monetary data.

The *Money* Class

The *Money* class (available electronically; see "Resource Center," page 5) offers the ability to deal with large monetary values while implicitly maintaining maximum precision throughout complex computations. It avoids the introduction of intermediate round-off errors during computations and provides a straightforward mechanism for rounding off final results.

Internally, a *Money* object stores a monetary value in the form of a *BigDecimal* object. Thus, extremely large positive and negative monetary values are supported, while maintaining many digits of precision during computation, ensuring accurate results. Round-off to the nearest whole cent is strictly controlled; incoming values (used to construct a *Money* object) are never rounded off, nor are intermediate computational results. Round-off occurs only when the monetary value is externalized for use in the "real world" outside the *Money* class. A common example of this occurs when a monetary value is formatted as a *String* for display (in a report or on an invoice, for example). In such cases, you normally want the monetary value rounded to the nearest whole cent, accurate to two decimal places.

Money objects, like *String* objects, are immutable and, therefore, have similar semantics. In particular, methods never modify the state of a *Money* object, but instead return a new *Money* object with a state that reflects the operation performed. For example, the statement *totalCost.add(itemCost);* does not modify the value of *totalCost*. The return value is discarded, and the statement has no effect. To add the value of *itemCost* to *totalCost*, you would write this as *totalCost = totalCost.add(itemCost);*, which replaces *totalCost* with a reference to a new *Money* object representing the sum of *totalCost* and *itemCost*.

The class provides a number of constructors (Table 1) that allow a *Money* object to be created from a variety of common sources.

A *double* floating-point value specifies a monetary value in dollars and cents. For example, a value of 19.95 would represent the monetary value $19.95.

A *long* integer, in the absence of an explicit scaling factor, represents whole dollars only (no cents). For example, a value of 1995 would represent $1995.00. However, an optional scaling factor of 0, 1, or 2 specifies the number of decimal digits to the right of an implicit decimal point. For example, the value of 1995 with a scaling factor of 2 would represent $19.95.

A *String* representation of a monetary value, such as "$19.95," can be also be used to construct a *Money* object. In this case, the *String* must be consistent with the currency format specified for the *Money* class.

Arithmetic Operations On Monetary Values

The *Money* class provides methods (Table 2) to perform the usual arithmetic operations required on monetary data, including add, subtract, multiply, and divide. Whereas the *add* and *subtract* methods operate on two monetary values, *multiply* and *divide* operations involve a double, or long, multiplier or divisor and facilitate operations such as tax computation.

Converting Monetary Values to Native Data Types

There may be situations in which it is desirable to convert a monetary value to a native Java data type. For example, this might be required when storing the value in a database. The *Money* class provides two methods for this purpose.

The *toDouble( )* method converts a monetary value to a double floating-point value. However, you should avoid using the converted value for computational purposes, owing to the problems associated with the use of floating-point data types with monetary data.

You can also convert a monetary value to a *long* integer data type (having an implied scaling factor of 2). In this case, an automatic round-off to the nearest whole cent is performed.

A variety of methods are provided that can be used to determine if a monetary value is zero, positive, or negative, or to compare two monetary values (as in Table 3).

Constructor	Description
Money()	Default constructor (monetary value $0.00).
Money(double value)	Value in dollars and cents.
Money(long value)	Value in whole dollars only.
Money(long value, int scale)	Value in dollars and cents, scale (0, 1, 2) specifies location of an implied decimal point.
Money(String value)	Value in dollars and cents, includes formatting characters.
Money(BigDecimal value)	A BigDecimal value in dollars and cents.
Money(Money value)	Copy constructor.

Table 1: *Constructors.*

Method	Returns
add(Money value)	Sum of this monetary value and another value.
subtract(Money value)	Difference of this monetary value and another value.
multiply(double value)	Product of this monetary value and value.
multiply(long value)	Product of this monetary value and value.
divide(double value)	This monetary value divided by value.
divide(long value)	This monetary value divided by value.
negate()	Negated monetary value.
abs()	Absolute monetary value.

Table 2: *Arithmetic operations.*

Method	Returns
IsZero()	True if monetary value is zero ($0.00).
IsPositive()	True if monetary value is greater than or equal to zero (>= $0.00).
IsNegative()	True if monetary value is less than zero (< $0.00).
isEqual(Money value)	True if this monetary value is equal to the specified value.
isLessThan(Money value)	True if this monetary value is less than the specified value.
isLessThanOrEqual(Money value)	True if this monetary value is less than or equal to the specified value.
isGreaterThan(Money value)	True if this monetary value is greater than the specified value.
isGreaterThanOrEqual(Money value)	True if this monetary value is greater than or equal to the specified value.

Table 3: *Comparison methods.*

Rounding Mode	Description
ROUND_DOWN (default)	Truncates to 2 decimal places, discarding any fractional cent.
ROUND_UP	Rounds any fractional cent upward to the next whole cent.
ROUND_HALF_UP	Rounds upward if 1/2 cent or more; otherwise, rounds downward.

Table 4: *Rounding modes used with monetary data.*

Currency Formats

The *Money* class makes use of a currency format (a special instance of Java's *DecimalFormat* class) to control the formatting of monetary values for display, as well as the parsing of a *String* representing a monetary value. A currency format specifies format attributes such as the currency symbol, the decimal separator, how positive and negative values are displayed, and whether decimal digits in the dollar amount will include a grouping character.

Unless otherwise specified, the currency format associated with the current default locale is used. For the United States, the default currency format specifies that monetary values be formatted using the dollar sign ("$") as the currency symbol and the decimal point (".") as the decimal separator. The dollar amount includes the comma (",") as a delimiter between each group of three digits, and negative amounts are indicated by enclosure within parentheses.

Examples of monetary values formatted using the default currency format for the United States include:

```
$123.45    $1,234,567.89    ($1,234.56)
```

The *getCurrencyFormat()* method is used to obtain the currency format of a *Money* object. The *setCurrencyFormat()* method is used to change the currency format. See the Java API documentation for the *DecimalFormat* class for detailed information on how to specify and use decimal formats.

Formatting a Monetary Value for Display

The *Money* class overrides the *toString()* method to return a *String* representation of a monetary value. Consequently, *Money* objects can conveniently be used in *String* expressions. A monetary value is formatted according to the conventions specified by the currency format for the object.

Parsing a Monetary Value

A *Money* object can be created from a *String* representation of a monetary value using either the *Money(String)* constructor or the *parse()* method. The currency format controls the parsing of the string (see the previous section on currency formats). If the *String* representing the monetary value is inconsistent with the currency format, a *ParseException* is recognized.

The default United States currency format, for example, requires that monetary values include the dollar sign ("$") and use the decimal point (".") as a separator between the dollars and cents. The dollar amount must also include the comma (",") as a delimiter between each group of three digits, and negative amounts must be enclosed in parentheses. The section on currency formats illustrates some monetary values that conform to the default format.

Monetary values provided as input by a user (on a web form, for example) do not usually include formal formatting characters, such as the dollar sign and grouping characters. For example, a user would probably enter "1234.56" instead of "$1,234.56." Simple monetary values such as these are easily parsed by first creating a *BigDecimal* object, from which you can then create a *Money* object:

```
BigDecimal value = new BigDecmal ("1234.56");
Money money = new Money(value);
```

Rounding

Rounding off a monetary value to an integral cent occurs only in select situations; namely, when formatting a monetary value for display as a *String*, and when exporting a monetary value to a *long* data type. In both cases, the monetary value of the object itself is never rounded; only the value returned is rounded.

As noted earlier, the *Money* class maintains a monetary value internally as a *BigDecimal* value. Consequently, all but one of the rounding modes offered by Java's *BigDecimal* class are supported. In practice, however, only a few of the available rounding modes are appropriate for monetary values. Table 4 illustrates several rounding modes that are relevant for monetary data.

The default rounding mode for a *Money* object is *BigDecimal.ROUND_DOWN*, which simply truncates the monetary value to exactly two decimal places, discarding any fractional part of a cent. Thus, by default, monetary values are never rounded upward.

For certain situations, however, it may be appropriate to use a different rounding mode. For example, you might wish to round a tax amount to the nearest whole cent. Or, you might want to round a grand total up to the next whole cent. Setting the rounding mode to the desired value allows you to control the manner in which monetary values are rounded off.

Many operations on *BigDecimal* values discard precision (trailing digits) and therefore require that a valid rounding mode be specified; otherwise, an *ArithmeticException* may be recognized. For this reason, you can specify any valid rounding mode other than *BigDecimal.ROUND_UNNECESSARY*, which suppresses rounding.

Handling Exceptions

Certain methods of the *Money* class throw runtime (nonchecked) exceptions to signal error conditions. Because these are runtime exceptions, you are not required to enclose certain method invocations in a *try-catch* block. However, you are strongly encouraged to use a *try-catch* block when parsing a *String* representation of a monetary value.

A *ParseException* is thrown by the *Money(String)* constructor and the *parse(String)* method if the specified *String* representation of a monetary value is inconsistent with the currency format.

The *InvalidScaleFactorException* is thrown by the *Money(long)* constructor if the specified scale factor is not valid (valid scale factors are 0, 1, and 2). The *InvalidScaleFactorException* is an embedded static class of the *Money* object.

The *InvalidRoundingModeException* is thrown by the *setRoundingMode()* method if an invalid rounding mode is specified (all rounding modes are valid with the exception of *BigDecimal.ROUND_UNNECESSARY*). The *InvalidRoundingModeException* is an embedded static class of the *Money* object.

Conclusion

Having a consistent, accurate, and reliable mechanism for dealing with monetary data is an important element of any robust e-commerce architecture. Java's native data types, such as integer and floating point, are not suitable for use with monetary data, and the standard Java class library does not directly address the issue of monetary data.

The *Money* class effectively encapsulates the issues associated with monetary data and offers a simple and reliable solution to the problem of handling monetary data in the Java programming environment.

DDJ

Next Article   Table of Contents