Money and Currency in Java

Best practices, Libraries & JSR-354

Stephen Colebourne
Member of Technical Staff
OpenGamma
scolebourne@joda.org

Money and Currency in Java

Best practices, Libraries & JSR-354

Anatole Tresch
Framework Architect
Spec Lead JSR 354
Credit Suisse
atsticks@java.net

CREDIT SUISSE

What best practices are there for handling money in Java?

Should the JDK have a money class?

If so, what should it look like?

How flexible does it need to be?

Do existing libraries help?

What about JSR-354?

Introduction

- Money very common domain concept
- No dedicated support in JDK, even version 8
  - But there is a Currency class

Currency

- java.util.Currency class added in JDK4
- Immutable
- Based on ISO 4217 three letter codes
  - USD = US dollars
  - GBP = British pound
  - EUR = Euro
- ISO three digit numeric code accessible from JDK7
  - USD = 840

Currency

- Enhanced in JDK7 and JDK8
- From JDK7, the set of currencies can be overridden
- From JDK8, currency switchover dates are supported

Currency - best practice

- Use Currency class where possible
- Avoid defining currency as a String

- What is missing?
- What could make it better?

Money

- Money = Amount + Currency
- No JDK money class
- Variety of things to consider
  - type to store amount
  - immutability
  - methods
- Most developers probably just store a primitive number
  - perhaps with a assumed currency

Money - store using double?

- double is a bad type for storing money
- Floating point is binary approximation of decimal
- 0.1 + 0.2 = 0.30000000000000004
- 0.1 * 0.2 = 0.020000000000000004
- Small differences, but not appropriate for money
- http://floating-point-gui.de/
- Acceptable to use double if amounts are estimates
- Acceptable if whole team understands FP and rounds correctly

Money - store using BigDecimal?

- BigDecimal is an acceptable type for storing money
- Won't lose data
- No literal or operator support
- Don't use new BigDecimal(double)
- Rounding mode often needed
  - 1bd / 3bd -> ArithmeticException unless rounding mode specified
- Slow, but does that matter?

Money - store using long?

- long is a good type for storing money
- Store data in terms of the smallest unit, eg cents
- Literal and operator support
- Easy to confuse dollars/euros with cents

Money - use a library?

- Joda-Money by Stephen Colebourne - wraps a BigDecimal
- Money by Tom Gibara - wraps a BigDecimal
- JSR-354 - neutral about amount type, by default also wraps BigDecimal

- Others
  - https://github.com/JodaOrg/joda-money/blob/master/Notes.txt
  - TimeAndMoney - dormant
  - JScience - more focussed on units

Joda-Money

- Aims to provide basic support for money
  - money classes
  - extended currency support
  - formatting
- Not providing domain specific extensions
  - no algorithms beyond most obvious (min/max)
  - no support for gross/net/tax
- http://www.joda.org/joda-money/

Money

- Immutable money class
- Represents currency and BigDecimal amount
- Fixed to decimal places of currency

```
Money amount = Money.parse("EUR 1.20");
amount = amount.multipliedBy(2);  // EUR 2.40
amount = amount.plusMajor(3);     // EUR 5.40
amount = amount.plusMinor(5);     // EUR 5.45
int cents = amount.getAmountMinorInt();  // 545
boolean positive = amount.isPositive();  // true
String str = amount.toString();   // "EUR 5.45"
```

BigMoney

- Immutable money class
- Represents currency and BigDecimal amount
- Any number of decimal places

```
BigMoney amount = BigMoney.parse("EUR 1.20567");
amount = amount.multipliedBy(2);  // EUR 2.41134
amount = amount.plusMajor(3);     // EUR 5.41134
amount = amount.plusMinor(5);     // EUR 5.46134
boolean negative = amount.isNegative();  // false
amount = amount.rounded(4, RoundingMode.UP);
String str = amount.toString();   // "EUR 5.4614"
```

CurrencyUnit

- Replacement currency class
- Allows applications to control currency data
- Includes 3 digit numeric ISO code

```
CurrencyUnit cur = CurrencyUnit.of("GBP");
int dp = cur.getDecimalPlaces();       // 2
String code = cur.getCurrencyCode();   // "GBP"
int ncode = cur.getNumericCode();      //
String str = cur.toString();           // "GBP"
```

Formatting

- Printing and parsing
- Flexible builder, like Joda-Time and JSR-310

```
MoneyFormatterBuilder b = new MoneyFormatterBuilder();
b.appendCurrencyCode().appendLiteral(": ").appendAmount(
  MoneyAmountStyle.ASCII_DECIMAL_POINT_GROUP3_COMMA);
MoneyFormatter f = b.toFormatter();

String str = f.print(money);   // eg "GBP: 1,234.56"
```

Design

- Immutable classes, so extend Object and final
- BigMoneyProvider interface unifies Money & BigMoney
- Classes are simple value types – BigDecimal + CurrencyUnit
- Follow older JSR-310 date/time design
- Super-compact serialization

JSR 354

- Standalone JSR, but targeting Java platform SE 9
- Core (targeting Java platform)
  - Money classes, currency support, rounding and monetary functions
  - Minimal algorithmic support
- Currency Conversion
- Extended Formatting
- Extensions
  - Validities (Historic Access), Region API

http://java.net/projects/javamoney/

http://jcp.org/en/jsr/summary?id=354

MoneyCurrency

- Immutable currency class, implements CurrencyUnit
- Support for different currency schemes (aka namespaces)
- Backward compatible signatures similar to java.util.Currency

```
MoneyCurrency chf = MoneyCurrency.of(“CHF”);
MoneyCurrency bitcoin = MoneyCurrency.of(“Bitcoin”);
int minorUnits = chf.getDefaultFractionUnits(); // 2
int cashRounding = chf.getCashRounding();       // 5
String code = chf.getCurrencyCode();            // “CHF"
int ncode = bitcoin.getNumericCode();           // -1
String str = chf.toString();                    // “CHF“
```

Money, MonetaryAdjuster, MonetaryQuery

- Immutable money class, implements MonetaryAmount
- Represents CurrencyUnit and BigDecimal amount
- Adaptable internal rounding and precision
- Basic Arithmetics and MonetaryAdjuster, MonetaryQuery support

```
Money amount = Money.of("EUR”, 1.20);
Money bcAmount = Money.of(MoneyCurrency.of(“Bitcoin”), 1.20);
amount = amount.multiply(2);                    // EUR 2.40
amount = amount.add(Money.of("EUR”, 3));        // EUR 5.40
amount = amount.add(Money.of("EUR”, 0.05));     // EUR 5.45
int cents = amount. getAmountFractionNumerator(); // 45
boolean positive = amount.isPositive();         // true
String str = amount.toString();                 // "EUR 5.45"
```

Formatting

- Printing and parsing
- Flexible builder using arbitrary tokens
- Customizable managed Formatters

```
ItemFormatBuilder<Money> b = new ItemFormatBuilder();          // builder
b.append(new CurrencyToken()).append(": ").append(new NumberToken());
ItemFormat<Money> f = b.build();
String str = f.format(money, Locale.ENGLISH);   // eg "GBP: 1,234.56"
LocalizationStyle style = new LocalizationStyle.Builder(
                                  Money.class, "short")
  .withAttribute("grouping",new int[]{3,2}).build();
ItemFormat<Money> f2 = MonetaryFormats.getItemFormat(          // managed format
                                                    Money.class, style);
```

Conversion

- Access to ExchangeRates, different ExchangeRate types
- Conversion modelled as MonetaryAdjuster
- Single and derived rates

```
ConversionProvider prov = MonetaryConversion.getConversionProvider(
                                                    ExchangeRateType.of("EZB"));
CurrencyConversion chfConv = prov.getConverter().getCurrencyConversion("CHF");
CurrencyConversion gbpConv = prov.getConverter().getCurrencyConversion("GBP");

Money eurAmount = Money.of("EUR", 1000);
Money chfAmount = eurAmount.with(chfConv);
Money gbpAmount = eurAmount.with(gbpConv);
```

Extensions

- Currency/Currency namespace mapping
- Access to historic data (Single and related Validities)
- Region API

```
MoneyCurrency cur = MoneyCurrency.of(CHF);
MoneyCurrency alt = MonetaryCurrencies.map(cur, “CS”);


Collection<RelatedValidity<CurrencyUnit,Region> validities =
     Validities.getRelatedValidityInfo(
        new RelatedValidityQuery(CurrencyUnit.class, Region.class)
           .withRelatedToPredicate(
             InstancesPredicate.of(Regions.getRegion(Locale.GERMANY))));
```

Design JSR 354

- Immutable classes, so extend Object and final
- Core Part should enhance/extend Java SE (Currency class)
- MonetaryAmount, CurrencyUnit interfaces for inter-op and extensibility, inter-operation constraints
- Simple value type – Money (using BD by default) + MoneyCurrency
- Follow JSR-310 naming conventions
- Modularized Design: Core, Conversion, Formatting, Extensions, (Extras), EE support
- Extendible and flexible API/SPI

Discussion

- What are your best practices?
- What do you use today?
- What does the JDK need?
- Will you use Joda-Money or JSR-354?