BEDCON 2013

JSR 354 – Money & Currency

Introduction

Click icon to add picture

Anatole Tresch
5th April 2013
Bio

Anatole Tresch

- Consultant, Coach
- Framework Architect
- Open Source Addicted
- Credit Suisse
- Specification Lead JSR 354

- atsticks@java.net
- Twitter: @atsticks
- anatole.tresch@credit-suisse.com
Agenda

- History and Motivation
- Overview
- Currencies and Amounts
- Precision and Rounding
- Formatting and Parsing
- Currency Conversion
- Provider and Extensions
- Demo

Platform (SE) Scope

Standalone Scope
History and Motivation

[Image of ancient gold coin]
Earlier Approaches

Martin Fowler:
A large proportion of the computers in this world manipulate money, so it’s always puzzled me that money isn’t actually a first class data type in any mainstream programming language. The lack of a type causes problems, the most obvious surrounding currencies... see http://martinfowler.com/eaaCatalog/money.html

Eric Evans – Time and Money:
On project after project, software developers have to reinvent the wheel, creating objects for simple recurring concepts such as “money” and “currency”. Although most languages have a “date” or “time” object, these are rudimentary, and do not cover many needs, such as recurring sequences of time, durations of time, or intervals of time. ...
To be quite frank, their code isn’t more than an academic POC, factories called dollars() or euros() are useless in real globally deployed frameworks, but he made a good point.
Motivation

- Monetary values are a key feature to many applications
- Existing `java.util.Currency` class is strictly a structure used for representing ISO-4217 standard currencies.
- No standard value type to represent a monetary amount
- No support for currency arithmetic or conversion
- JDK Formatting features lack of flexibility
Schedule

- Java SE 9
- Java ME/Embedded 8 oder 9

Following the EC Merge and Standard/Embedded harmonization, no JSR should be SE/EE or ME only. Money is so important, and has almost no legacy in the JDK except `java.util.Currency`, that it should be supported by all possible platforms, except maybe JavaCard for now.

- With back-port to previous versions still supported and in relevant use
- EDR: Beginning of April 2013
Overview
Overview of JSR 354

- **Core API**: `javax.money`
  
  CurrencyUnit, MonetaryAmount and exceptions

- **Conversion API**: `javax.money.conversion`
  
  ExchangeRate, CurrencyConverter

- **Formatting**: `javax.money.format`
  
  LocalizationStyle, ItemFormatter, ItemParser

- **Provider singleton**: `javax.money.provider`
  
  Monetary

- **Extensions**: `javax.money.ext`
  
  Region support, Calculations
Currencies and Amounts
javax.money
Currencies
ISO 4217

Special Codes
- Precious Metals (XAU, XAG)
- Testing (XTS)
- No Currency (XXX)
- Supranational currencies, e.g. East Caribbean dollar, the CFP franc, the CFA franc.

- CFA franc: **West African CFA** franc und **Central African CFA** franc = denotes 2 effectively interchangeable (!).

Ambiguities
- Switzerland: CHF, CHE (WIR-EURO), CHW (WIR)
- USA: USD, USN (next day), USS (same day)

Legal acceptance, e.g. Indian Rupees are legally accepted in Buthan/Nepal, but not vice versa!

Typically 1/100, rarely 1/1000, but also 1/5 (Mauritania, Madagaskar), 0.00000001 (BitCoin)
Virtual Currencies

- **Video Game Currencies** (Gold, Gil, Rupees, Credits, Gold Rings, Hearts, Zenny, Potch, Munny, Nuyen...)

- **Facebook Credits** are a virtual currency you can use to buy virtual goods in any games or apps of the Facebook platform that accept payments. You can purchase Facebook Credits directly from within an app using your credit card, PayPal, mobile phone and many other local payment methods.

- **Bitcoin** (sign: BTC) is a decentralized digital currency based on an open-source, peer-to-peer internet protocol. It was introduced by a pseudonymous developer named Satoshi Nakamoto in 2009.
Limitations of java.util.Currency

- No support for historical Currencies
- No support for non standard Currencies (e.g. cows or camels)
- No support for virtual Currencies (Lindon Dollars, BitCoin, Social Currencies)
- No support for custom schemes (e.g. legacy codes)
- Only access by currency code, or Locale
- No support for special use cases/extensions

Implementation: MoneyCurrency

```java
public interface CurrencyUnit{
    public String getCurrencyCode();
    public int getNumericCode();
    public int getDefaultFractionDigits();
    // new methods
    public String getNamespace();
    public boolean isLegalTender();
    public boolean isVirtual();
    public Long getValidFrom();
    public Long getValidUntil();
    public <T> T getAttribute(String key, Class<T> type);
}
```
**Introduction to JSR 354 - [http://java.net/projects/javamoney](http://java.net/projects/javamoney)**

```
/**
 * Shows simple creation of a CurrencyUnit for ISO, backed up by JDK
 * Currency implementation.
 */

public void forISOCurrencies() {
    CurrencyUnit currency = MoneyCurrency.of("USD");
    currency = MoneyCurrency.of("myNamespace", "myCode"); // null!
}

public void buildACurrencyUnit() {
    builder.setNamespace("myNamespace");
    builder.setCurrencyCode("myCode");
    builder.setDefaultFractionDigits(4);
    builder.setLegalTender(false);
    builder.setValidFrom(System.currentTimeMillis());
    builder.setVirtual(true);
    builder.setAttribute("test-only", true);
    CurrencyUnit unit = builder.build();
    // nevertheless MoneyCurrency.of("myNamespace", "myCode"); still returns
    // null!
    builder.build(true);
    // no it is registered
    unit = MoneyCurrency.of("myNamespace", "myCode");
}
```
Monetary Amount

**Amount = Number + Currency + Operations**

How to represent the numeric amount?
Contradictory requirements:

- Performant (e.g. for trading)
- Precise (e.g. for calculations)
- Must model small numbers (e.g. webshop)
- Must support huge Numbers (e.g. risk calculations, statistics)

**Solution: support several numeric representations!**
Rounding, Precision, Scale
Monetary Amount (continued)

```java
public interface MonetaryAmount{
    public CurrencyUnit getCurrency();
    public Class<?> getNumberType();
    public <T> T asType(Class<T>);
    public int intValue();
    public int intValueExact();
    public long longValue();
    public long longValueExact();
    [...]
    public MonetaryAmount abs();
    public MonetaryAmount min(...);
    public MonetaryAmount max(...);
    public MonetaryAmount add(...);
    public MonetaryAmount subtract(...);
    public MonetaryAmount divide(...);
    public MonetaryAmount[] divideAndRemainder(...);
    public MonetaryAmount divideToIntegralValue(...);
    public MonetaryAmount remainder(...);
    public MonetaryAmount multiply(...);
    public MonetaryAmount withAmount(Number amount);
    [...]
    public int getScale();
    public int getPrecision();
    [...]
    public boolean isPositive();
    public boolean isPositiveOrZero();
    public boolean isNegative();
    public boolean isNegativeOrZero();
    public boolean isLessThan(...);
    public boolean isLessThanOrEqualTo(...);
    [...]
}
```

Data Access.

Implementation: Money

Algorithmic Operations...

Data Representation and Comparison.
Creating Amounts

Usage

/**
 * Simplest case create an amount with an ISO currency.
 */

public void forISOCurrencies() {
    MonetaryAmount amount = Money.of("USD", 1234566.15);
}

/**
 * Create an amount using a custom currency.
 */

public void forCustomCurrencies() {
    CurrencyUnit currency = MoneyCurrency.of(
            "myNamespace", "myCode");
    MonetaryAmount amount = Money.of(currency, 1234566.15);
}
Precision and Rounding
javax.money
Numeric Precision

- **Internal** Precision (implied by internal number type)
- **External** Precision (Rounding applied, when the numeric part is accessed/passed outside)
- **Formatting** Precision (Rounding for display and output)
- **Interoperability**
  - Different precision/scale
  - Distinct numeric representations
  - Serialization

By default only internal rounding is applied automatically.
Mixing Numeric Representations

Money `amt1` = `Money.of`("CHF", 10.23d);
IntegralMoney `amt2` = `IntegralMoney.of`("CHF", 123456789);
Money `result` = `amt1.add(amt2)`;

- **Money** as representation type, since it's the class on which `add()` was called.
- Precision = 9
- Scale = 2

Mechanism applies similarly for operation chaining

Money `amt1` = `...`;
IntegralMoney `amt2` = `...`;
CurrencyConversion `conversion` = `...`;
Money `result` = `amt1.add(amt2)`
  .multiply(2)
  .with(`conversion`)  
  .round(`MoneyRounding.of`());
Rounding

External Rounding and Formatting

Rounding can be implemented in many ways, depending on the use cases.

Example for non standard-rounding Argentina:

- If the third digit is 2 or less, change it to 0 or drop it.
- If the third digit is between 3 and 7, change it to 5.
- If the third digit is 8 or more, add one to the second digit and drop the third digit or change it to 0.

**Implementation:**

```
public interface Rounding{
    public MonetaryAmount round(MonetaryAmount);
}
```

```
Rounding rounding = MoneyRounding.getRounding(MoneyCurrency.of("USD"));
MonetaryAmount myAmount = ...;
MonetaryAmount rounded = rounding.round(myAmount);
```

<table>
<thead>
<tr>
<th>Original</th>
<th>Rounded</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.452</td>
<td>123.45</td>
<td>3. digit &lt; 3 -&gt; round down</td>
</tr>
<tr>
<td>123.456</td>
<td>123.455</td>
<td>3 &lt;= 3. digit &lt;= 7 -&gt; change to 5</td>
</tr>
<tr>
<td>123.459</td>
<td>123.46</td>
<td>3. digit &gt;= 8 -&gt; round up</td>
</tr>
</tbody>
</table>

Introduction to ISR 354 -
Arithmetics & Rounding

Usage

/**
 * Mixed representations.
 */

public void mixedImplementations() {
    MonetaryAmount m1 = IntegralMoney.of("USD", 789);
    MonetaryAmount m2 = Money.of("USD", 1234566.15);

    MonetaryAmount sum = m1.add(m2);
    MonetaryAmount diff = m2.subtract(m1).negate();
}

/**
 * Round amount based on ist currency (defaultFractionUnits).
 */

public MonetaryAmount roundDefault(MonetaryAmount amount) {
    Rounding rounding =
        MoneyRounding.of(amount.getCurrency());

    return rounding.round(amount);
}
Formatting and Parsing
javax.money.format

Portfolio
Cash: 64102.56 €  Market: FRA

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Company</th>
<th>Price</th>
<th>Change</th>
<th>% Change</th>
<th>Shares</th>
<th>Open</th>
<th>Volume</th>
<th>Current Value *</th>
<th>Gain/Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>&quot;IBM&quot;</td>
<td>$115.43</td>
<td>▴0.37</td>
<td>-32%</td>
<td>50</td>
<td>▴115.80</td>
<td>2,655,471</td>
<td>3699.68 €</td>
<td>▴15.98</td>
</tr>
<tr>
<td>JAVA</td>
<td>&quot;JAVA&quot;</td>
<td>$16.56</td>
<td>▴0.44</td>
<td>273%</td>
<td>200</td>
<td>▴16.12</td>
<td>5,750,460</td>
<td>2123.08 €</td>
<td>▴545.90</td>
</tr>
<tr>
<td>DELL</td>
<td>&quot;DELL&quot;</td>
<td>$19.52</td>
<td>▴0.08</td>
<td>41%</td>
<td>200</td>
<td>▴19.44</td>
<td>14,293,015</td>
<td>2502.56 €</td>
<td>▴82.30</td>
</tr>
<tr>
<td>GOOG</td>
<td>&quot;GOOG&quot;</td>
<td>$426.88</td>
<td>▴1.62</td>
<td>38%</td>
<td>100</td>
<td>▴425.26</td>
<td>5,523,309</td>
<td>27363.97 €</td>
<td>▴38.05</td>
</tr>
<tr>
<td>MSFT</td>
<td>&quot;MSFT&quot;</td>
<td>$28.58</td>
<td>▴0.20</td>
<td>71%</td>
<td>100</td>
<td>▴28.38</td>
<td>47,317,464</td>
<td>1832.15 €</td>
<td>▴71.00</td>
</tr>
</tbody>
</table>

Currency rates from 03/08/2007 12:00pm EST

<table>
<thead>
<tr>
<th>Currency Name</th>
<th>Currency Code</th>
<th>Exchange Rate to US $</th>
<th>Exchange Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australian Dollar</td>
<td>AUD</td>
<td>1.28780006</td>
<td>128.78</td>
</tr>
<tr>
<td>Baht</td>
<td>THB</td>
<td>32.7</td>
<td>3270.00</td>
</tr>
<tr>
<td>Bolivar</td>
<td>VEB</td>
<td>2144.6</td>
<td>214460.00</td>
</tr>
</tbody>
</table>
Formatting and Parsing

Challenges

- Multiple Locale instances for Translation, Dates, Time, Numbers, Currencies
- Additional parameters
  - Currency Placement
  - Rounding, Lenient Fractions, Min, Max etc.
- Natural language support for non-decimal valuations for example
  - Lakhs, Crores (1 Lakh = 100,000)
  - INR 12,34,56,000.21 is written 12 Crore, 34 Lakh, 56 Thousand Rupees and 21 Paise

Unsupported by NumberFormat, e.g.
- INR 12,34,225.21

Introduction to JSR 354 - http://java.net/projects/javamoney

```java
public class LocalizationStyle
    implements Serializable {
    [...]
    public String getId();
    public Locale getTranslationLocale();
    public Locale getNumberLocale();
    public Locale getDateLocale();
    public Locale getTimeLocale();
    public Map<String, Object> getAttributes();
    public <T> T getAttribute(String key, Class<T> type);
    public static LocalizationStyle of(Locale locale);
    public boolean isDefaultStyle();
    [...]
}
```
public interface ItemFormat<T> {
    public Class<T> getTargetClass();
    public LocalizationStyle getStyle();
    public String format(T item);
    public void print(Appendable appendable, T item)
        throws IOException;
    public T parse(CharSequence input)
        throws ParseException;
}

public final class MonetaryFormat{
    public Collection<String>
        getSupportedStyleIds(Class<?> targetType);
    public boolean isSupportedStyle(
        Class<?> targetType, String styleId);
    public <T> ItemFormat<T>
        getItemFormat(Class<T> targetType,
                      LocalizationStyle style)
        throws ItemFormatException;
    public <T> ItemFormat<T>
        getItemFormat(Class<T> targetType,
                      Locale locale)
        throws ItemFormatException;
}
Currency Conversion
javax.money.conversion
Currency Conversion

- ExchangeRateType
- ExchangeRate:
  - ExchangeRateType
  - Base, Term currency
  - Conversion factor
  - Validity (from/until)
  - Provider (optional)
- Direct/Derived Rates
- ExchangeRateProvider
- CurrencyConverter

```java
public interface ExchangeRate {
    public ExchangeRateType getExchangeRateType();
    public CurrencyUnit getBase();
    public CurrencyUnit getTerm();
    public Number getFactor();
    public Long getValidFrom();
    public Long getValidUntil();
    public boolean isValid();
    public String getProvider();
    public ExchangeRate[] getExchangeRateChain();
    public boolean isDerived();
}
```
Currency Conversion

Usage

/**
 * Shows simple conversion of an amount.
 */

public Money convertAmountToCHF(Money amount)
{
    CurrencyUnit currency = MoneyCurrency.of(curr);
    ExchangeRateType rateType = ExchangeRateType.of("EZB");

    ConversionProvider convProvider =
    MonetaryConversion.getConversionProvider(rateType);

    CurrencyConversion chfConversion =
        convProvider.getConverter().
            getCurrencyConversion(MoneyCurrency.of("CHF"));

    return amount.with(chfConversion);
}
Provider & Extensions

d prevalen, javax.money.ext

Introduction to JSR 354 -

Note: this module has changed in the meantime

5th April 2013

29
**Provider**

Monetary Singleton

- `javax.money.Monetary` singleton provides access to all components
- By default components loaded using JDK's `java.util.ServiceLoader`.
- Alternate Loader implementations possible, e.g.
  - Using CDI standalone
  - Within a J2EE container
  - Spring
  - …

```java
public final class Monetary{
    public static CurrencyUnitProvider getCurrencyUnitProvider();
    public static ConversionProvider getConversionProvider();
    public static ItemFormatterFactory getItemFormatterFactory();
    public static ItemParserFactory getItemParserFactory();
    public static RoundingProvider getRoundingProvider();
}
```

Note: this module has changed in the meantime
Extensions

Allow registration of additional functionalities into Monetary:

- Calculation Utilities
- Compound Values
- Statistical Modules
- Financial Modules
- Regions/Regional Providers, e.g. for mapping accepting currencies, legal tenders etc.
- ...

To be discussed:

- if and what extensions are part of the JSR

Note: this module has changed in the meantime
Extensions

Usage

/**
 * Shows simple usage of an extension, e.g. calculating
 * the total of all amounts, that have a certain currency.
 */

public MonetaryAmount total(MonetaryAmount... amount,
                           String curr){

    AmountUtils utils = MoneyCurrency.getExtensions(
                           AmountUtils.class);

    return utils.total(utils.filter(amount, curr));
}

Note: this module has changed in the meantime
Stay Tuned!

- JSR 354: http://jcp.org
- Java.net Project: http://java.net/projects/javamoney
- GitHub Project: https://github.com/JavaMoney/javamoney
- Twitter: @jsr354