Go for the Money!
Introduction to JSR 354 Money & Currency

Anatole Tresch
11th August 2013
Bio

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

- Twitter: @atsticks
- [atsticks@java.net](mailto:atsticks@java.net)
- [anatole.tresch@credit-suisse.com](mailto:anatole.tresch@credit-suisse.com)
Agenda

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

(Platform (SE) Scope)¹

Standalone Scope

¹ See http://mreinhold.org/blog/secure-the-train
this may no longer be part of Money API 1.0
History and Motivation
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
- No support for historic currencies
- JDK Formatting features lack of flexibility

**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](http://martinfowler.com/eaaCatalog/money.html)
Schedule

- Standalone JSR
- EDR finished 31st May 2013
- PDR until September 2013
- 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.

- Minimum Java Version: 7, with reasonable support of previous Java versions still relevant (e.g. Java 6)

- Java SE 9: Early 2016
  - Either a Maintenance Release or new version of this JSR shall be integrated with OpenJDK 9.
Overview of JSR 354

- **Core API**: `javax.money`
  - `CurrencyUnit`, `MonetaryAmount`, monetary functions and exceptions
- **Conversion API**: `javax.money.conversion`
  - `ExchangeRate`, `CurrencyConverter`, `CurrencyConversion`
- **Formatting**: `javax.money.format`
  - `LocalizationStyle`, `ItemFormat`, `ItemFormatBuilder`
- **Extensions**: `javax.money.ext`
  - Region API, Currency services, calculation utilities

- **Reference Implementation**: `net.java.javamoney.ri`
- **TCK**: `net.java.javamoney.tck`
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.

Ambiguities
- CFA franc: West African CFA franc und Central African CFA franc = denotes 2 effectively interchangeable (!).
- Switzerland: CHF, CHE (WIR-EURO), CHW (WIR)
- USA: USD, USN (next day), USS (same day)

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

Minor units
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.
Improve `java.util.Currency`

Objectives

- Support for standard Currencies (e.g. cows or camels)
- Support virtual Currencies (Lindon Dollars, BitCoin, Social Currencies)
- Support custom schemes (e.g. legacy codes)
- Allow access by currency code, namespace, or `Locale`
- Provide a flexible design
- Be backward compatible, add a minimum of functionality to existing `Currency` class

```java
public interface CurrencyUnit{
    public String getCurrencyCode();
    public int getNumericCode();
    public int getDefaultFractionDigits();
    // new methods
    public String getNamespace();
    public boolean isLegalTender();
    public boolean isVirtual();
}
```

```java
public interface Localizable{
    public String getDisplayName(Locale locale);
}
```

```java
public final class MoneyCurrency implements CurrencyUnit, Localizable{
    public String getDisplayName(Locale locale);
}
```
/**
 * Shows simple creation of a CurrencyUnit for ISO, backed up by JDK Currency implementation.
 */

public void forISOCurrencies() {

    MoneyCurrency currency1 = MoneyCurrency.of("USD");

    CurrencyUnit currency2 = MoneyCurrency.of("myNamespace", "myCode");
}
/**
 * Shows creation of a non ISO CurrencyUnit using a Builder, including
 * registering of the build currency into the shared cache.
 */

public void buildACurrencyUnit() {
builder.setNamespace("myNamespace");
builder.setCurrencyCode("myCode");
builder.setDefaultFractionDigits(4);
builder.setLegalTender(false);
builder.setVirtual(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:
- Performance (e.g. for trading)
- Precision and scale (e.g. for calculations)
- Must model small numbers (e.g. web shop)
- Must support large numbers (e.g. risk calculations, statistics)
- Rounding
- Financial operations and functions, function chaining

Solution:
- support several numeric representations!
- MonetaryFunction similar to java.function.Function
- MonetaryOperator similar to java.function.UnaryOperator
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(...);
    [...] public MonetaryAmount with(MonetaryOperator op);
}
```
Monetary Function, Monetary Operator

```java
//@FunctionalInterface for Java 9
public interface MonetaryFunction<T, R> {
    public R apply(T value);
}
```

### Implementations:
- Minimum
- Maximum
- Average
- Total
- SeparateAmounts
- SeparateCurrencies
- Filters...

```java
//@FunctionalInterface for Java 9
public interface MonetaryOperator extends MonetaryFunction<MonetaryAmount, MonetaryAmount>{
}
```

### Implementations:
- CurrencyConversion
- MonetaryRounding
- Reciprocal
- MinorPart
- MajorPart
Creating Amounts
Usage

/**
 * Simplest case create an amount with an ISO currency.
 */
public void forISOCurrencies() {

    MonetaryAmount amount = Money.of("USD", 1234566.15);
    // using ISO namespace by default
}

/**
 * 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.
- the precision/scale capabilities of an MonetaryAmount are inherited to its operational results.
Mixing Numeric Representations

Money amt1 = Money.of(“CHF”, 10.23d);
IntegralMoney amt2 = IntegralMoney.of(“CHF”, 123456789);
MonetaryAmount result = amt1.add(amt2);
   // returns Money instance, since its the class on which
   // add() was called.

• Mechanism applies similarly for operation chaining

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

- External Rounding and Formatting Rounding can be implemented in many ways, depending on the use cases.
- Rounding is modeled as a MonetaryOperator.
- Cash Rounding may be different.
- Arithmetic Rounding vs. Non-standard Rounding

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.

<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>

```java
public final class MoneyRounding{
    public static MonetaryOperator getRounding(CurrencyUnit unit){
        ...
    }
}
```
Rounding

Usage

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

public MonetaryAmount roundDefault(MonetaryAmount amount) {
    Rounding rounding = MoneyRounding.of();
    return rounding.round(amount);
}

public MonetaryAmount halfConvertAndRound(MonetaryAmount amount, 
                                          CurrencyConversion conversion) {
    return amount.divide(2).with(conversion).with(MoneyRounding.of());
}
### Formatting and Parsing

**javax.money.format**

---

#### Portfolio

<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>

Cash: 64102.56 €  Market: FRA

---

#### 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>1.287830006</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, e.g. Currency Placement, Rounding, Lenient Fractions, Min, Max etc.
- Dynamic Content based on Data Formatted
- Programmatic Format/Parser Definition
- Natural language support for non-decimal valuations for example
  - Lakhs, Crores (1 Lakh = 100,000, 1 Crore = 100 Lakh)
  - INR 12,34,56,000.21 is written 12 Crore, 34 Lakh, 56 Thousand Rupees and 21 Paise

- How handle different formatting styles?
  - LocalizationStyle
  - ItemFormat
  - ItemFormatBuilder
  - MonetaryFormats Singleton

```java
public class LocalizationStyle
    implements Serializable {
    [...]
    public String getId();
    public Map<String, Object> getAttributes();
    public <T> T getAttribute(String key, Class<T> type);
    public boolean isDefaultStyle();
    [...]
}
```
Formatting and Parsing
ItemFormat, FormatToken and ItemFormatBuilder

```java
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 interface FormatToken<T> {
    public void print(Appendable appendable, T item, LocalizationStyle s)
        throws IOException;
    public void parse(ParseContext<T> context, LocalizationStyle style)
        throws ItemParseException;
}

public class ItemFormatBuilder<T> {
    [...]
    public ItemFormatBuilder() {...}
    public ItemFormatBuilder(Class<T> targetType){...}

    public ItemFormatBuilder<T> addToken(FormatToken<T> token) {...}
    public ItemFormatBuilder<T> addLiteral(String token) {...}
    public boolean isBuildable() {...}
    public ItemFormat<T> build() {...}
    [...]
}
```

Implementations:
LiteralToken
NumberToken
CurrencyToken
AmountToken
...
/**
 * Round amount based on its currency (defaultFractionUnits).
 */

public void buildAndUseItemFormat() throws IOException {
    ItemFormatBuilder<ItemFormatBuilder.<Money> b = new ItemFormatBuilder.<Money>(Money.class);
    b.addLiteral("[test] ");
    // equals: b.addToken(new LiteralToken<Number>("test- "));
    DecimalFormat df = new DecimalFormat("#0.0#");
    DecimalFormatSymbols syms = df.getDecimalFormatSymbols();
    syms.setDecimalSeparator(':');
    df.setDecimalFormatSymbols(syms);
    b.addToken(new CurrencyToken(CurrencyToken.TYPE_DISPLAYNAME));
    b.addLiteral(" ");
    b.addToken(new NumberToken(df).setNumberGroupChars(',', ' ', '
' ).setNumberGroupSizes(2, 2, 3));
    ItemFormat.<Money> f = b.build();

    System.out.println(f.format(123456789.123456789d,
            LocalizationStyle.of(Locale.ENGLISH)));
}

Output: [test] Swiss Francs 12'345'67,89:12
Currency Conversion
javax.money.conversion
Currency Conversion

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

```java
public interface ExchangeRate {
    public ExchangeRateType getExchangeRateType();
    public CurrencyUnit getBase();
    public CurrencyUnit getTerm();
    public BigDecimal getFactor();
    public Long getValidFrom();
    public Long getValidUntil();
    public boolean isValid();
    public String getProvider();
    public List<ExchangeRate> getExchangeRateChain();
    public boolean isDerived();
}
```
Currency Conversion

Usage

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

public Money convertAmountToCHF(Money amount) {
    ConversionProvider provider = MonetaryConversion
        .getConversionProvider(ExchangeRateType.of("EZB"));

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

    return amount.with(chfConversion);
}
```
Currency Services & Extensions
javax.money.ext
Currency Services & Extensions
Currency and Region Services

- `javax.money.MonetaryCurrencies` singleton provides
  - access to registered namespaces
  - access to contained currencies within a namespace
  - access to historic currencies

- `javax.money.Regions` singleton provides
  - Access to `Region` instances (similar to Unicode/CLDC)
  - Access to validities per `Region`
  - Access to the `Region` forest (`RegionNode`)
Extras

Additional Functionalities, *not part of the actual JSR*
- Multi-Currency
- Compound Values
- Statistical Modules
- Financial Modules
- CDI Extensions
- JavaFX Bindings and Components
- …

Extras are provided:
- as separate [javamoney-extras](http://java.net/projects/javamoney) GitHub repository
  - What’s in for you, e.g. like javamoney-examples, no full JCP Membership is required to contribute there, contribute if you want/can…
Demo
Stay Tuned!

- JSR 354: [http://jcp.org](http://jcp.org)
- Java.net Project: [http://java.net/projects/javamoney](http://java.net/projects/javamoney)
- GitHub Project: [https://github.com/JavaMoney/javamoney](https://github.com/JavaMoney/javamoney)
- JUG Chennai adoption (TrakStok): [https://github.com/jugchennaiadoptjava/TrakStok](https://github.com/jugchennaiadoptjava/TrakStok)
- Twitter: @jsr354

- Adopt, join our JSR, talk and tweet about it!
- In the area of reference implementation/TCK no full EC membership is required!
The End

Thank you!