JSR 354

Money and Currency API

Java Money Expert Group

Specification Lead
Anatole Tresch, Credit Suisse

Version
0.7 (Public Review)
14th October 2013

Evaluation license

JSR-000354 Money and Currency API 1.0 Public Review

CREDIT SUISSE AG IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS LICENSE AGREEMENT ("AGREEMENT"). PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY THEM, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE AND THE DOWNLOADING PROCESS WILL NOT CONTINUE.

Specification: JSR-354 Money and Currency API ("Specification")
Version: 0.7
Status: Public Review
Release: 7th October 2013

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Credit Suisse AG ("the Specification Lead") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, the Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under the Specification Lead's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes:
(i) developing implementations of the Specification for your internal, non-commercial use;
(ii) discussing the Specification with any third party; and
(iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:
(a) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;
(b) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and
(c) includes the following notice: "This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."
The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.
No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other intellectual property of the Specification Lead, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from the Specification Lead if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.
"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Credit Suisse AG through the Java Community Process, or any recognized successors or replacements thereof.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Credit Suisse AG or Credit Suisse AG's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY THE

SPECIFICATION LEADS. THE SPECIFICATION LEADS MAKE NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product. THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. THE SPECIFICATION LEADS MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL THE SPECIFICATION LEADS AND/OR THEIR LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF CREDIT SUISSE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold the Specification Lead (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions)

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in *connection with your evaluation of the Specification ("Feedback"). To the extent that you provide the Specification Lead with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant the Specification Lead a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.*

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply. The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee. This Agreement is the parties' entire a greement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

1. Introduction

This document is the specification of the Java API for Money and Currency. The technical objective is to provide a money and currency API for Java, targeted at all users of currencies and monetary amounts, both simple but also extendible.
The API will provide support for standard [ISO-4217] and custom currencies, and a representation of monetary amounts. It will support currency arithmetic, also across different currencies, and will support foreign currency exchange.
Additionally, this JSR includes recommendations on interoperability and thread safety.

1.1 Expert group

This work is being conducted as part of JSR 354 under the Java Community Process Program. This specification is the result of the collaborative work of the members of the JSR 354 Expert Group and the community at large. The following persons have actively contributed to Java Money in alphabetical order:

- Greg Bakos
- Matthias Buecker (Credit Suisse)
- Stephen Colebourne
- Benjamin Cotton
- Jeremy Davies
- Thomas Huesler
- Scott James (Credit Suisse)
- Tony Jewell
- Werner Keil
- Bob Lee
- Simon Martinelli
- Sanjay Nagpal (Credit Suisse)
- Christopher Pheby
- Jefferson Prestes
- Arumugam Swaminathan
- Anatole Tresch (Credit Suisse, Spec Lead)

1.2 Specification goals

Monetary values are a key feature of many applications, yet the JDK provides little or no support. The existing `java.util.Currency` class is strictly a structure used for representing current [ISO-4217] currencies, but not associated values or custom currencies. The JDK also provides no support for monetary arithmetic or currency conversion, nor for a standard value type to represent a monetary amount.

1.2.1 Specification Targets

JSR 354 targets to support all general application types, e.g.
- eCommerce
- banking
- financial
- investment
- insurance and pension
- ERP systems
- etc.

This specification will not discuss Low Latency concerns as required for example by algorithmic trading applications. Nevertheless the API was designed to support different implementations of monetary amounts and allows to be extended in several ways. So it should be flexible enough that corresponding implementations can be used transparently to accommodate such applications.

1.3 Scope

JSR 354 targets a standalone scope, since it will not be possible to include it into Java 8. Nevertheless it should be added to JDK 9, so its design must consider integration into the JDK. During the development of the JSR a wide set of features were implemented. Most of these features will not end up within the JSR itself, as the JSR now has scope limited to interoperation, enabling feature innovation elsewhere. The corresponding libraries were published under [JavaMoney] as an Apache 2 licensed open source project. Compared to the early draft review the following features are no longer in the scope of the JSR:
    - currency conversion
    - complex formatting (replaced by a simple formatter for amounts)
    - region API
    - validity API
    - predicate API
    - additional financial functions

Though the features above were removed from the JSR, their development ensured that scope was fully evaluated and that the parts best suited to standardization were identified. Where beneficial to the community parts of the JavaMoney project may also use Java 8 features like Lambdas when Java 8 goes final, while the JSR remains backward-compatible with Java 7 in first release, see below.

1.4 Required Java version

The specification is based on Java SE 7.0 language features, but should be compatible with Java SE 6.0 also, due to the fact that still many financial institutions and clients are using this environment[1]. Implementations may target any suitable Java SE version, or given an increasing SE/ME correlation also matching ME versions like CLDC 8.

1.5 How this document is organized

There are five main section in this document:
- Use cases.
- Requirements.
- Specification.
- Implementation Recommendations.
- An appendix.

---

[1] Corresponding ballots on conferences have shown that about 20% of users are planning to switch to Java 8 within the next two years.

2. Use Cases

This section describes some, but not all, of the use cases that should be covered with this JSR.

2.1 Scenario eCommerce (Online-Shop)

One basic scenario that must be covered is a traditional web shop. Hereby products are presented and collected in a shopping cart. Each product can be added once or multiple times to the cart. Some sites also need to represent non integral amounts, such as 1.5kg of a product. Additionally a site may be internationalized handling multiple currencies, perhaps controlled by user settings or address.
Summarizing this scenario implies the following requirements:

- Prices for each item must be modelled by some monetary amount, representing a numeric amount in a single currency.
- The prices for all items in the cart must be calculated, this requires sum up all monetary amounts.
- The user may change the number of each items to purchase, either by defining an integral number (e.g. 2 products) or a decimal point number (e.g. 1.5 kg). This requires multiplication with integer and decimal numbers.
- Each item’s price must be presented to the customer with the required target currency and in the format expected. This requires formatting of amounts and currencies according to the user’s Locale.
- When changing the currency of a shopping cart, the catalog prices must be recalculated in the new target currency. This requires accessing an exchange rate to be used and calculating the item amounts with the new currency, including multiplication and division.
- When a customer finally places an order, the total amount must be calculated, which may involve tax calculation. This also requires multiplication of prices and rounding to a bookable amount (depending on the target currency).
- Finally the amount to withdrawn from the credit card must be passed to a server system, that handles credit card payment. This includes serialization of the amount.

2.2 Scenario Trading Site

On a financial trading system or a site displaying several financial information such as quotes, additional aspects must be considered. Basically, since for real time data must be paid, often data is displayed that is so called deferred. Customers may be able to create virtual portfolios with arbitrary instruments for simulation of investment strategies. To estimate a possible investment historic charts and timelines are shown, which includes current, as well as statistical data. Depending on the simulated investment also different precisions of the monetary amounts must be possible. Finally also for evaluation of complex investment strategies or products very detailed arithmetic precision may be required.
Summarizing this scenario implies the following requirements:

- A monetary amount representing a stock quote or other financial instrument, may have arbitrary additional data attached, such as mapped quote keys, the origin stock exchange, the accuracy of the of data (validity, current or deferred), as well as the data's provider. Additionally the internal logic typically requires that the data types used, such as currencies and exchange rates, can be extended with additional data, that is specific to the concrete use cases/implementation.
- An exchange rate can be current, deferred or even historic and typically has a defined validity scope.
- Legal requirements may restrict the information presented (e.g. the currencies available) to the user based on several aspects:
    - geographic location of the client
    - legal aspects, such as the client's contract
    - others

This implies that access to financial data may be restricted based on several not predictable classifications that must not match a country or locale.

2.3 Scenario Virtual Worlds and Game Portals

Virtual worlds, e.g. online games, define their own game money (but also Facebook has its own money). User's may obtain such virtual money by paying some real amount, e.g. by credit card. This usage scenario implies the following requirements:

- It must be possible to model completely virtual currencies. Since virtual money also can be converted (paid) with real money, the price effectively defines an exchange rate.
- Since several virtual game portals exist, also the number of virtual currencies can not be foreseen. Additionally a virtual world may even define different currencies (e.g. Bitcoin).
- Since such exchange rates may change during time, historization must also be supported.

2.4 Scenario Social Markets

Within social markets things are exchanged using a completely virtual currency, which has no relation to any real currency. It is used as an arbitrary measurement of something meaningful only to that social community. This usage scenario implies the following requirements:

- It must be possible to model virtual currencies that are able to completely replace any real currency schemes.

2.5 Scenario Banking & Financial Applications

Applications in financial institutes, such as a bank or insurance companies must model monetary information in several ways: exchange rates, interest rates, stock quotes, current as well as historic currencies must be supported. Typically in such companies also internal systems exist that define additional schemas of financial data representation, e.g. for historic currencies, exchange rates, risk analysis etc. Often such aspects can not be covered by the

ISO 4217 currency standard. As example imagine historic currencies, such as “Deutsche Reichsmark”, gold nuggets or even completely other things.
Additionally also within [ISO-4217] there are countries in Africa that share a common ISO code (e.g. CFA), but nevertheless have different banknotes and coins per country. Also there are ambiguities that may be confusing, such as USD, USS, USN, which all describe US dollars.
This usage scenario implies the following requirements:

- Currencies as well as exchange rates must be historic, and define their time validity range. The same may also be true for rounding algorithms.
- Customized or legacy system in big financial institutions may define additional, arbitrary currency variants.
- Such system may have additional data not covered by the JSR’s currency model, so it is important that the model will be designed to be extendible.
- Currencies of different type, must be mappable to each other.

2.6 Scenario Insurance & Pension

Complex calculation models are used within insurance and pension solutions, e.g. for scenario simulation and forecasting. Different countries, companies or even investment strategies, have rather different models implemented, that also may change quickly dependent on legal changes. Such systems are built of several isolated building blocks of different granularity size and complexity, starting from simple totalization of amounts until to complex investment strategy forecasts on an enterprise level. Such systems imply the following requirements:

- Building blocks should be modelled/organized in a common repository and accessible by a common API, that also allows introspection of the functionality available. This is a precondition so insurance solutions can reuse the blocks for modeling the required business cases.
- Input and Output data of calculations can be multivalued, e.g. for forecast scenarios, or statistical data. Hereby the (value) types used can be completely different, such as numbers, amounts, currencies, strategy identifiers, dates, time ranges, interest and exchange rates etc. So there must be a structure to model such *compound data*.

3. Requirements

3.1 Core Requirements

Based on the scope and use cases described above the following core requirements can be identified:

1. The JSR must specify a minimal set of interfaces for interoperability, since concrete usage scenarios do not allow to define an implementation that is capable of covering all aspects identified. Consequently it must be possible that implementations can provide several implementations for monetary amounts
2. Implementations must provide *value types* for currencies and amounts, implementing `CurrencyUnit` and `MonetaryAmount.`
3. Implementations must provide a minimal set of roundings, modeled as `MonetaryAdjuster`. This should include basic roundings for ISO currencies, roundings defined by `java.math.MathContext` or `java.math.RoundingMode`.
4. Custom roundings should also be supported, but are not a required part of an implementation

3.2 Formatting Requirements

It must be possible to format and parse monetary amounts:

1. A formatter can format an item into a `String` or into an `Appendable`.
2. A formatter can parse an item from `CharSequence` input.
3. A formatter may support different formatting styles for the currency part, but must at least parse the currency code.
4. A formatter supports flexible number formatting similar to `java.text.DecimalFormat`.
5. Hereby the formatter supports flexible grouping sizes and different grouping separators, so, e.g. also Indian Rupees, can be formatted correctly[2].
6. A formatter should support rounding of amounts for display and reverse rounding during parsing.

3.3 EE Support

1. This JSR must avoid restrictions that prevents its use in different runtime environments, such as EE or ME.

3.4 Non Functional Requirements

1. Any possible changes to the Java platform must be fully backward compatible.

---

[2] The JDK `NumberFormat` only supports a fixed grouping size, e.g. 3. Indian Rupees have different grouping sizes applied, e.g. `INR 12,34,56,000.21`

2. Implementation requirements for currencies must require only minimal extensions on the existing `java.util.Currency`.
3. Interfaces defined should enable interoperability between different implementations. Users should not reference the interfaces, instead the value types should be used.
4. The interface for amounts must not expose its numeric internal representation during compile time.
5. Where feasible method naming and style for currency modelling should be in alignment with parts of the Java Collection API or `java.time` / [JodaMoney]:
    a. same method name prefixes - `of()` for all factories, unless their inheritance e.g. from `java.lang.Enum` - mandates otherwise, such as `valueOf()`.
    b. basic creational factory methods with little/no conversion are named `of(...)`
    c. more complex factory methods, with some conversion, or requiring a specific name for clarity are named `ofXxx(...)`
    d. factories that extract/convert from a broadly specified input (where there is a good chance of error) are named `from(...)`
    e. parsing is explicitly named, as it is generally special, named `parse(...)`
    f. overall monetary API "feel" should be similar to `java.lang.BigDecimal`.
6. The JSRs design should acceptably work with other JVM languages (Groovy, Scala, Clojure...).
7. UTC timestamps in APIs must be modelled as `long`. SPIs are allowed to model timestamps as `java.lang.Long`, to support `null`, when a timestamp is not defined.
8. Though performance aspects can not directly targeted by this JSR, it is important that the JSR considers performance aspects where possible, so provided implementations are able optimizing performance as required by the usage scenarios they are targeting.

4. Specification

4.1 Package and Project Structure

4.1.1 Package Overview

The JSR defines one single package `javax.money.`

This package contains the platform artifacts, such as `CurrencyUnit, MonetaryAmount, MonetaryAdjuster, MonetaryQuery, MonetaryException.`

4.1.2 Module/Repository Overview

The JSR's source code repository under [Source] provides several artifacts:

- `jsr354-api` contains the API interfaces and the `MonetaryException` class.
- `jsr354-ri` contains the reference implementation[3]
- `jsr354-tck` contains the technical compatibility kit (TCK)[4]
- `javamoney-parent,javamoney-lib` contain a financial library (JavaMoney) adding comprehensive support for several extended functionalities, built on top of this JSR, but not part of the JSR.
- `javamoney-examples` finally contains the examples and demos, and also is not part of this JSR.

---

[3] Note that the reference implementation is not a required be part for public review, so it may still change.

[4] Note that the TCK is not a required part for public review.

4.2 Money and Currency API

The package `javax.money` contains the types representing currencies and monetary amounts, the core exceptions as well as supporting types for rounding and the extensions SPI. This package is a candidate for inclusion in the JDK 9 SE platform.

4.2.1 Interface javax.money.CurrencyUnit

The interface `CurrencyUnit` models a minimal set of functionality required for interoperability, since a currency code is defined to be unique, exposing this code is sufficient:

```
public interface public CurrencyUnit{

      public String getCurrencyCode();
}
```

These methods cover the following requirements:

- The method `getCurrencyCode()` returns the unique currency code. Nevertheless since `CurrencyUnit` also models *non* ISO currencies, the semantics for other currency types may be different:
- Hereby it is required that a currency code is unique. For ISO currencies this will the 3-letter uppercase ISO code. For non ISO currencies no constraints are defined.

Implementations of `CurrencyUnit`

1. must implement `equals/hashCode`, considering the concrete implementation type and currency code (which is defined to be unique).
2. must be Comparable
3. must be immutable and thread safe.
4. should be serializable.

4.2.2 Interface javax.money.MonetaryAmount

The *interface* `MonetaryAmount` defines the properties and operations of a monetary amount for interoperability.

```
public interface public MonetaryAmount{

        public CurrencyUnit getCurrency();
        public long getAmountWhole();
        public long getAmountFractionNumerator();
        public long getAmountFractionDenominator();
        public MonetaryAmount with(MonetaryAdjuster adjuster);
        public <T> T query(MonetaryQuery<T> query);
}
```

Hereby

- `getCurrency` return the amount's *currency*, modelled as `CurrencyUnit`. Implementations may co-variantly change the return type to a more specific implementation of `CurrencyUnit` if desired.
- `getAmountWhole, getAmountFractionNominator, getAmountFractionDenominator` exposes a portable *numeric representation* of the amount's numeric value.
- monetary adjusters and monetary queries can be applied on a `MonetaryAmount` instance by passing a `MonetaryAdjuster, MonetaryQuery<T>` instance to the `with` or `query` method.
- Additionally for the numeric representation
  given

  ```
  w = getAmountWhole()
  n = getFractionNominator()
  d = getFractionDenominator()
  ```

  the following must be always true:

  ```
  !(w<0 && n>0)   and
  !(w>0 && n<0)   and
  d>0             and
  |n| < d         // || = absolute value
  ```

The specification and interface do not define precisely how the amount is stored. Implementations could use a `BigDecimal`, `long` or something else entirely. The only constraint is that it can be exposed as three `long` elements of a rational number.

The representation of the amount was one of the key design decisions. The final design is intended to provide for implementors to handle very different use cases with distinct requirements. For example, use of a `BigDecimal` would cover many use cases and could be expressed in the API using a denominator with a power of ten.

Implementations of `MonetaryAmount` (of type `T`)

1. must implement `equals/hashCode`, hereby it is recommended considering
   a. Implementation type
   b. `CurrencyUnit`
   c. Numeric value.

   This also means that two different implementations types with the same currency and numeric value are *NOT* equal.
2. must be `Comparable`.
3. should be serializable.
4. should be immutable and thread safe.
17. To enable interoperability a `static` method `from(MonetaryAmount)` is recommended to be implemented, that allows conversion of a `MonetaryAmount` to a concrete type `T`.

    ```
    public static T from(MonetaryAmount amount);
    ```

    This is particularly useful when implementing monetary adjusters or queries, since arithmetic operations are not available on the `MonetaryAmount` interface, which is defined for interoperability only.
18. Finally implementations should not implement a method `getAmount()`. This method is reserved for future integration into the JDK.

This specification does not further constrain the constructor or factory methods to be implemented, or the method signatures to be used.

Two further interfaces, `MonetaryAdjuster` and `MonetaryQuery`, provide a powerful extension mechanism. The two interfaces operate as a form of the strategy pattern, allowing the algorithm of a query or adjustment to be external to the implementation of `MonetaryAmount`. Their design and naming matches JSR-310.

4.2.3 Interface javax.money.MonetaryAdjuster

This interface defines an arbitrary adjustment on a monetary amount, such as rounding or monetary calculations:

```
public interface MonetaryAdjuster{

        public MonetaryAmount adjustInto(MonetaryAmount amount);
}
```

Adjusters can be used to make any kind of change to the amount based on the original amount. For example, the following requirements (not complete listing) would be covered:

- rounding of amounts
- currency conversion
- financial calculations
- other monetary conversions

Implementations of `MonetaryAdjuster` should be

- immutable and
- thread-safe

The adjuster is typically invoked on the instance of the money class, passing the adjuster as a parameter.

```
MonetaryAmount amount = …
MonetaryAdjuster convertedToUsd = ...
MonetaryAmount usdAmount = amount.with(convertedToUsd)
```

The adjuster interface is equivalent to the `UnaryOperator` interface in JDK 8 which is a functional interface suitable for use with lambdas.

4.2.4 Interface javax.money.MonetaryQuery

This interface defines an arbitrary monetary query.

```
public interface MonetaryQuery<R> {

      public R queryFrom(MonetaryAmount amount);
}
```

Queries can be used to make any kind of query against the data held in the amount. For example, the following requirements (not complete listing) would be covered:

- type conversion
- boolean queries, such as ‘is negative’, ‘is zero’ or ‘is currency widely traded’
- splitting the amount into smaller amounts
- serialization to string/bytes

Implementations of `MonetaryQuery` should be

- immutable and
- thread-safe

The query is typically invoked on the instance of the money class, passing the query as a parameter.

```
MonetaryAmount amount = …
MonetaryQuery<Boolean> negativeQuery = ...
boolean negative = amount.query(negativeQuery)
```

The query interface is equivalent to the `Function` interface in JDK 8 which is a functional interface suitable for use with lambdas.

It is recommended that the result type of a query is not `MonetaryAmount`, as `MonetaryAdjuster` is normally more suited to those cases. However, there is no technical restriction enforcing this.

4.2.5 Exception javax.money.MonetaryException

This *runtime* exception is the base exception for other currency related exceptions. Any monetary exception added by an implementation must inherit from this class.

4.3 Formatting

As defined in 3. Requirements, Implementations of this JSR should provide a formatter for `MonetaryAmount` instances. Nevertheless formatting is a very complex field and the JSR's expert group has decided to not define any concrete formatting API at this stage. This should be considered in a coordinated way for JDK 9, especially for factory methods like `NumberFormat.getCurrencyInstance()`, but it is likely this JSR would see at least an MR or new release where the expert group may address formatting in more detail. Nevertheless some important aspects were identified that should be considered:

1. rounding of amount values can be done by applying a `MonetaryAdjuster` before formatting/printing.
2. some currencies like INR can be formatted[5] with flexible grouping sizes and/or grouping characters.
3. A currency of an amount can be also formatted in different ways:
    a. as currency code, e.g. `USD`
    b. as numeric currency code, if such a code is defined.
    c. as a (localized) currency symbol, e.g. `$`
    d. as a (localized) currency name, e.g. `Schweizer Franken`
    e. as a special case, the currency also can be omitted completely.
4. a `MonetaryAdjuster` instance can also be configured to allow a formatter instance to format/parse symmetrically.

In financial applications additional formatting requirements are quite common (see also [JavaMoney]). Nevertheless the expert group decided to require only a bare minimum of functionalities that also are easily to be included as part of JDK 9.

---

[5] `INR 123456000.21` is formatted as `INR 12,34,56,000.21`

5. Implementation Recommendations

There are a couple of best practices in the area of financial applications and frameworks. This JSR does not require most of them for the following reasons:

- The overall API design is very similar to the Date/Time API introduced with JDK 9 (JSR-310). E.g. `TemporalAdjuster` and `MonetaryAdjuster` model a similar concept for temporals and for monetary amounts. Therefore the corresponding models in this JSR define similar implementation constraints.
- More complex constraints would be difficult or impossible to ensure by a TCK, so they are defined as recommendations.
- Finally there is always the possibility that no common ground can be found for the way some functionality can be modelled generically across implementations. It would then be the responsibility of the implementers to follow best, or at least de-facto, practice.

Nevertheless we think some practices are important and should be followed by implementations, so we added the most relevant ones in the following sections.

5.1 Rounding

Rounding should be modeled by an implementation of `MonetaryAdjuster`. Hereby beside mathematical roundings, also non standard variants with arbitrary rules and constraints are quite common in the financial area.

Implementations of this JSR may provide rounding adjusters based on one or more of the following:

1. a target `CurrencyUnit`, hereby providing default rounding based on the currency's fraction units
2. a `java.math.RoundingMode` or `java.math.MathContext`, providing mathematical roundings.
3. a `String` identifier, for customized roundings.

It should be possible to add additional roundings, e.g. for use with formatters. One way to manage roundings would be via a service loader.

5.2 Monetary Arithmetic

When dealing with monetary amounts the following aspects should be considered:

- Arithmetic operations should throw an `ArithmeticException`, if performing arithmetic operations between amounts exceeds the capabilities of the numeric representation type used. Any implicit truncating, that would lead to complete invalid and useless results, should be avoided, since it may result to invalid results, which are very difficult to trace. This recommendation does not affect internal rounding, as required by the internal numeric representation of a monetary amount.

- When adding or subtracting amounts, best practice recommends to use parameters that are instances of `MonetaryAmount`.
- When multiplying or dividing amount, best practice recommends parameters that are simple numeric values.
- Arguments of type `java.lang.Number` should be avoided, since it does not allow to extract its numeric value in a feasible way.
- Arithmetic operations should honor the advanced rules how rounding and truncation should be handled. Refer to the following sections for further details.

5.3 Numeric Precision

For financial applications precision and rounding is a very important aspect. Additionally that an incorrect arithmetic obviously has direct financial consequences, also legal aspects require specific precision and rounding to by applied.

The JSR's expert group identified the following important and distinct precision types:

- Internal precision
- External precision
- Formatting precision

The following sections will explain things in more detail.

5.3.1 Internal Precision

5.3.1.1 Overview

This precision type is the most important one, since it is directly related/determined by the internal numeric representation of the class implementing `MonetaryAmount`. Hereby:

- The internal numeric capabilities of a `MonetaryAmount` typically exceed the scale implied by the corresponding currency. Internal rounding must be done after each operation, but this rounding has nothing in common with the rounding implied by the currency attached. Basically the monetary arithmetics are completely independent of the currency, or in other words rounding should only be done implicitly when required by the internal numeric representation to minimize the loss of numeric precision.
- For calculations that require high scaled results, e.g. financial product calculations, it is recommended to work with relatively high scales, e.g. 64 or even higher scales, as provided by the `BigDecimal` class[6]. On the other hand when monetary arithmetics must be fast, e.g. in trading, scale requirements are often reduced in favor of fast data manipulation. This contradictory requirements were basically the key reason, why the

---

[6] Therefore the default reference implementation class, `Money`, is based on `BigDecimal` and allows to explicitly configure its `MathContext` used on creation.

model for `MonetaryAmount` does not explicitly specify the numeric representation to be used.
- Additionally during a financial calculation, the points, where rounding is feasible, are basically use case dependent and therefore should not be performed by a `MonetaryAmount` implementation *implicitly*. Instead of, roundings can be applied as useful as monetary adjustments *explicitly*, when useful.
- Also worth to mention is that for the same currency different roundings may be defined (default rounding, cash rounding, special roundings for presentation purposes), so there is no such concept as *THE* rounding for a monetary amount.

5.3.1.2 Configuring and Changing Internal Precision

An implementation of `MonetaryAmount` may support changing the internal precision or numeric capabilities. But any value type semantics must be strictly obeyed, meaning that changing a monetary amount's internal precision or numeric capabilities, requires creating of a new instance.

Additionally if an implementation of a `MonetaryAmount` supports different numeric capabilities, it is useful to allow the default capabilities to be configurable. Hereby a mechanism should be used, that is not shared in EE runtime context, such as a property file in the classpath.

5.3.1.3 Inheriting Numeric Representation Capabilities

When performing calculations with the value type semantics new instances of amounts are created for each calculation performed. This implies additional constraints:
- By inheriting the `MonetaryAmount` implementation type to its *return* types of all arithmetic operations, also the numeric capabilities must be inherited.
- Finally a `MonetaryAmount` implementation is required to throw an `ArithmeticException`, if a client tries to create a new instance with a numeric value that exceeds its internal representation capabilities. Since each arithmetic operation requires the creation of a new amount instance, as a consequence, all operations that exceed the numeric capabilities must throw an `ArithmeticException` (basically no implicit truncation is allowed).

5.3.2 External Precision

External precision is the precision applied, when the numeric part of a `MonetaryAmount` is externalized, meaning a numeric part of an amount is accessed/converted into another numeric representation. This externalized representation may have reduced numeric capabilities compared to the internal numeric representation, so truncation must be performed, or some exception can be thrown. Generally a precision or scale reduction on externalization should never throw an exception, despite the method variants are defined to be exact, similar to `BigDecimal.longValueExact()`. The *exact* methods should then throw an exception, if the externalization would result in data loss (some sort of truncation must be performed).

5.3.3 Display Precision

The precision used for displaying of monetary amounts on the screen, a printout or for passing values through technical systems, is completely dependent on the use cases. This JSR supports these scenarios with the possibility to apply arbitrary monetary adjustments (modeled as `MonetaryAdjuster`).

APPENDIX

References

[Bitcoin] http://bitcoin.org/en/
[ICU] http://site.icu-project.org/
[ISO-4217] http://www.iso.org/iso/home/standards/currency_codes.htm
[ISO-20022] www.iso20022.org
[JodaMoney] http://www.joda.org/joda-money/
[JavaMoney] https://github.com/JavaMoney/javamoney-lib
[java.net] http://java.net/projects/javamoney/
[JSR354] http://jcp.org/en/jsr/detail?id=354
[Source] Public Source Code Repository on GitHub: GitHub Repository,
Branch / Tag matching PDR is **0.7**

Links

- JSR 354 on jcp.org
- JSR 354 on Java.net
- JSR 354 on GitHub
- Java Practices -> Representing Money
- Working with Money in Java
- Java currency by Roedy Green, Canadian Mind Products
- https://github.com/JavaMoney/jsr354-api
- UOMo Business, based on ICU4J and concepts by JScience Economics
- MoneyDance API
- JavaMoney is the Apache 2.0 licensed OSS project that evolved from JSR 354 development. It provides concrete implementations for currency conversion and mapping, advanced formatting, historic data access, regions and a set of financial calculations and formulas.
- Joda Money can be referred to as an inspiration for API and design style. it is based on real-world use cases in an e-commerce application for airlines
- Grails Currencies uses BigDecimal as internal representation, but API only exposes Number in all Money operations like *plus()*, *minus()* or similar.
- ICU4J Uses Number for all operations and internal storage in its Money type.
- Why not to use BigDecimal for Money
- M-Pesa-Mobile Money in Africa
- Currency Internationalization (i18n), Multiple Currencies and Foreign Exchange (FX).
- http://en.wikipedia.org/wiki/Japanese_units_of_measurement#Money: Discussion of internationalization of currencies, rounding, grouping and formatting, separators etc]
- http://speleotrove.com/decimal/
- http://sourceforge.net/projects/oquote/
- Karatsuba Algorithm for Fast Big Decimal Multiplication

Related Initiatives

- Eric Evans Time and Money Library
- Bitcoin Java Client
- Java and Monetary Data (PDF)