JSR 354

Money and Currency API

Java Money Expert Group

Specification Lead
Anatole Tresch, Credit Suisse

Version
0.8 (Public Review 2)
March 2014

Table of Contents

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: 0.8
Status: Public Review 2
Release: March 2014
Copyright 2013-2014 Credit Suisse AG
8070 Zurich, Switzerland
All rights reserved.

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 model for monetary amounts and roundings. It will have extension points for adding additional features like currency exchange. financial calculations and formulas.
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
- Mohamed Taman
- 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. Nevertheless it may be included into the JDK later, so its design and scope must consider integration into the JDK. Additionally the work on the JSR has shown, that it is possible to define a flexible and comprehensive API that is also compatible with most Java ME profiles. Since with the Internet of Things small devices are getting more important, and there is high probability that monetary aspects must be implemented, the expert group decided to keep the API independent of JDK artifacts that are not supported on ME, especially `java.math` and `java.text`. Nevertheless the reference implementation is free to use existing functionalities and the JSR also includes requirements (also checkable by the TCK) to ensure a minimal set of functionalities on Java SE.

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[1]
    - 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

---

[1] Refer also to section Currency Conversion for further details,

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. Implementations may target any suitable Java SE version, or given an increasing SE/ME correlation also matching ME versions like CLDC 8.

Hereby this decision was done with caution. There are many financial applications and products that will require years until they were migrated to Java 8. Depending on Java 8 on the API side, would make it impossible to use them in such scenarios for a very long time and would definitely decrease the adoption rate of this JSR significantly. Additionally there are only a few aspects within the API that would be affected by building everything right based on Java 8. Especially the usage of functional interfaces is already part of this specification and will be supported without any change, when this JSR is used with Java 8. Another aspect is the usage of JSR 310 date and time types. This JSR does not depend on these types in the API, but provides mechanisms to enable usage of these types. One reason is that it has shown highly arguable if JSR 310 will be included into the Java ME at a later stage due to several reasons. So the decision was to avoid usage of that types, for wider compatibility of the JSR with different runtime environments.

1.5 How this document is organized

There are five main section in this document:

- Use cases.
- Requirements.
- Specification.
- Implementation Recommendations.
- An appendix.

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 provide an API for handling and calculating with monetary amounts.
2. The JSR must support different numeric capabilities and guarantees to be provided by the monetary amount implementations. These data is called *monetary context* and must be accessible from an amount instance during runtime.
3. 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.
4. The JSR must specify extension points for adding additional logic, e.g. for extending the arithmetic capabilities, rounding etc.
5. The API for monetary amounts must allow to externalize the numeric part of an amount to the most useful representation on a runtime platform. Similarly it must be possible to create a new amount instance using an existing amount as a template, hereby changing currency and/or numeric part as required. This ensures maximal portability and allows externalization of complex financial calculations.
6. The JSR must provide a minimal set of roundings. This should include basic roundings for ISO currencies, or roundings defined by a monetary context.
7. The JSR must also support arbitrary custom roundings.

3.2 Formatting Requirements

It must be possible to format and parse monetary amounts. Therefore the JSR defines a `MonetaryAmountFormat`, which:

1. can format an amount into a `String` or into an `Appendable`.
2. can parse an amount from a `CharSequence` input.
3. supports different formatting styles and placement strategies for the currency part.
4. supports flexible number formatting similar to `java.text.DecimalFormat`.
5. supports flexible grouping sizes and different grouping separators, so, e.g. also Indian Rupees, can be formatted correctly[2].
6. supports rounding of amounts for display and reverse rounding during parsing.

3.3 EE and ME Support

---

[2] `java.text.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`

1. This JSR must avoid restrictions that prevents its use in different runtime environments, such as EE or ME. Thus e.g. direct references to elements in `java.math` and `java.text` which is not supported by Java ME so far must be avoided.

3.4 Non Functional Requirements

1. Since this JSR may be a good candidate to be included into the JDK later, any possible extensions to the Java platform must be fully backward compatible.
2. Implementation requirements for currencies must require only minimal (if any) extensions on the existing `java.util.Currency`.
3. The JSR must be self-contained, meaning it must be possible to use the JSR, without acquiring of external resources, e.g. accessing resources in the internet.
4. Interfaces defined should enable interoperability between different implementations, both for data as well as functional interoperability. The interfaces must cover all typical use cases, so casting to concrete types should not be necessary normally.
5. The API for monetary amounts must not expose its concrete numeric internal representation during compile time.
6. 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.math.BigDecimal`.
7. POSIX timestamps (the JSRs relies on millisecond resolution as returned by `System.currentTimeMillis()`) 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. As several use cases for this JSR include (business) **critical** software like real time trading and similar systems, those usually must be independent of local time or system time that could be manipulated. Thus no **untrusted** time sources like `System.currentTimeMillis(), java.util.Date` or Java 8 equivalents like `LocalTime` and similar types are permitted. The JSR is not responsible for providing a reliable time source, but where required the use of UTC time stamps makes it compatible with relevant reliable time sources, e.g. atomic clock servers, etc.
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 three packages:

- `javax.money` contains the main artifacts, such as `CurrencyUnit`, `MonetaryAmount`, `MonetaryOperator`, `MonetaryQuery`, accessors for rounding etc.
- `javax.money.format` contains the formatting artifacts.
- `javax.money.spi` contains the SPI interfaces provided by the JSR 354 API and the bootstrap logic, to support different runtime environments and component loading mechanisms.

4.1.2 Module/Repository Overview

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

- `money-api` contains the JSR 354 API as described also be this specification.
- `moneta` contains the reference implementation[3]
- `money-tck` contains the technical compatibility kit (TCK)[4]
- `javamoney-parent` is a root "POM" project for all modules under "org.javamoney". This includes the RI/TCK projects, but not `jsr354-api`.
- `javamoney-lib` contains 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.

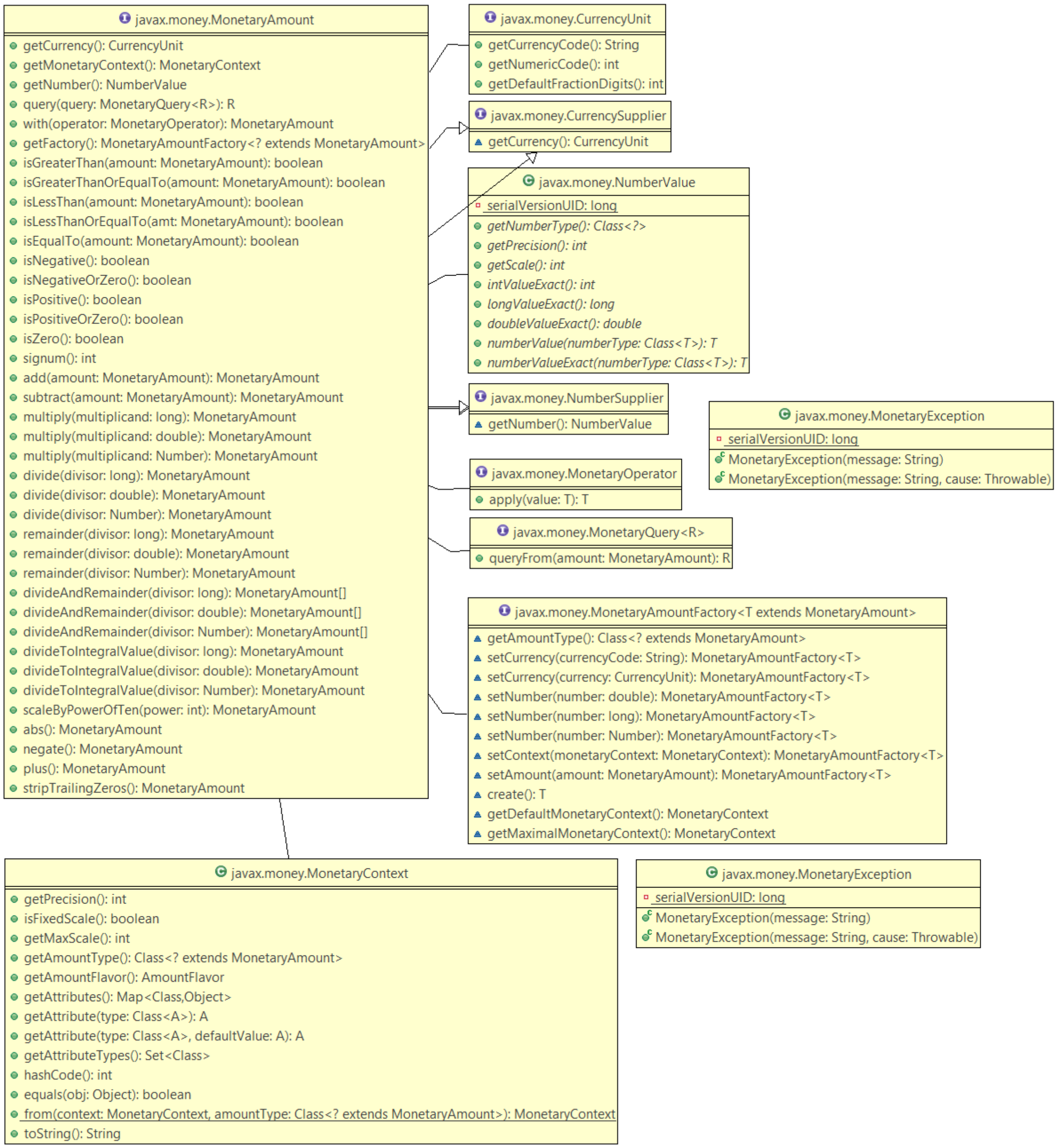
---

[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 Core 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 API. Hereby the main artifacts are as follows:


javax.money.MonetaryAmount
getCurrency(): CurrencyUnit
getMonetaryContext(): MonetaryContext
getNumber(): NumberValue
query(query: MonetaryQuery<R>): R
with(operator: MonetaryOperator): MonetaryAmount
getFactory(): MonetaryAmountFactory<? extends MonetaryAmount>
isGreaterThan(amount: MonetaryAmount): boolean
isGreaterThanOrEqualTo(amount: MonetaryAmount): boolean
isLessThan(amount: MonetaryAmount): boolean
isLessThanOrEqualTo(amt: MonetaryAmount): boolean
isEqualTo(amount: MonetaryAmount): boolean
isNegative(): boolean
isNegativeOrZero(): boolean
isPositive(): boolean
isPositiveOrZero(): boolean
isZero(): boolean
signum(): int
add(amount: MonetaryAmount): MonetaryAmount
subtract(amount: MonetaryAmount): MonetaryAmount
multiply(multiplicand: long): MonetaryAmount
multiply(multiplicand: double): MonetaryAmount
multiply(multiplicand: Number): MonetaryAmount
divide(divisor: long): MonetaryAmount
divide(divisor: double): MonetaryAmount
divide(divisor: Number): MonetaryAmount
remainder(divisor: long): MonetaryAmount
remainder(divisor: double): MonetaryAmount
remainder(divisor: Number): MonetaryAmount
divideAndRemainder(divisor: long): MonetaryAmount[]
divideAndRemainder(divisor: double): MonetaryAmount[]
divideAndRemainder(divisor: Number): MonetaryAmount[]
divideToIntegralValue(divisor: long): MonetaryAmount
divideToIntegralValue(divisor: double): MonetaryAmount
divideToIntegralValue(divisor: Number): MonetaryAmount
scaleByPowerOfTen(power: int): MonetaryAmount
abs(): MonetaryAmount
negate(): MonetaryAmount
plus(): MonetaryAmount
stripTrailingZeros(): MonetaryAmount
javax.money.CurrencyUnit
getCurrencyCode(): String
getNumericCode(): int
getDefaultFractionDigits(): int
javax.money.CurrencySupplier
getCurrency(): CurrencyUnit
javax.money.NumberValue
serialVersionUID: long
getNumberType(): Class<?>
getPrecision(): int
getScale(): int
intValueExact(): int
longValueExact(): long
doubleValueExact(): double
numberValue(numberType: Class<T>): T
numberValueExact(numberType: Class<T>): T
javax.money.NumberSupplier
getNumber(): NumberValue
javax.money.MonetaryException
serialVersionUID: long
MonetaryException(message: String)
MonetaryException(message: String, cause: Throwable)
javax.money.MonetaryOperator
apply(value: T): T
javax.money.MonetaryQuery<R>
queryFrom(amount: MonetaryAmount): R
javax.money.MonetaryAmountFactory<T extends MonetaryAmount>
getAmountType(): Class<? extends MonetaryAmount>
setCurrency(currencyCode: String): MonetaryAmountFactory<T>
setCurrency(currency: CurrencyUnit): MonetaryAmountFactory<T>
setNumber(number: double): MonetaryAmountFactory<T>
setNumber(number: long): MonetaryAmountFactory<T>
setNumber(number: Number): MonetaryAmountFactory<T>
setContext(monetaryContext: MonetaryContext): MonetaryAmountFactory<T>
setAmount(amount: MonetaryAmount): MonetaryAmountFactory<T>
create(): T
getDefaultMonetaryContext(): MonetaryContext
getMaximalMonetaryContext(): MonetaryContext
javax.money.MonetaryContext
getPrecision(): int
isFixedScale(): boolean
getMaxScale(): int
getAmountType(): Class<? extends MonetaryAmount>
getAmountFlavor(): AmountFlavor
getAttributes(): Map<Class,Object>
getAttribute(type: Class<A>): A
getAttribute(type: Class<A>, defaultValue: A): A
getAttributeTypes(): Set<Class>
hashCode(): int
equals(obj: Object): boolean
from(context: MonetaryContext, amountType: Class<? extends MonetaryAmount>): MonetaryContext
toString(): String
javax.money.MonetaryException
serialVersionUID: long
MonetaryException(message: String)
MonetaryException(message: String, cause: Throwable)

- `CurrencyUnit` models the minimal properties of a currency.
- `MonetaryAmount` defines what an amount^s capabilities are. It provides interoperability

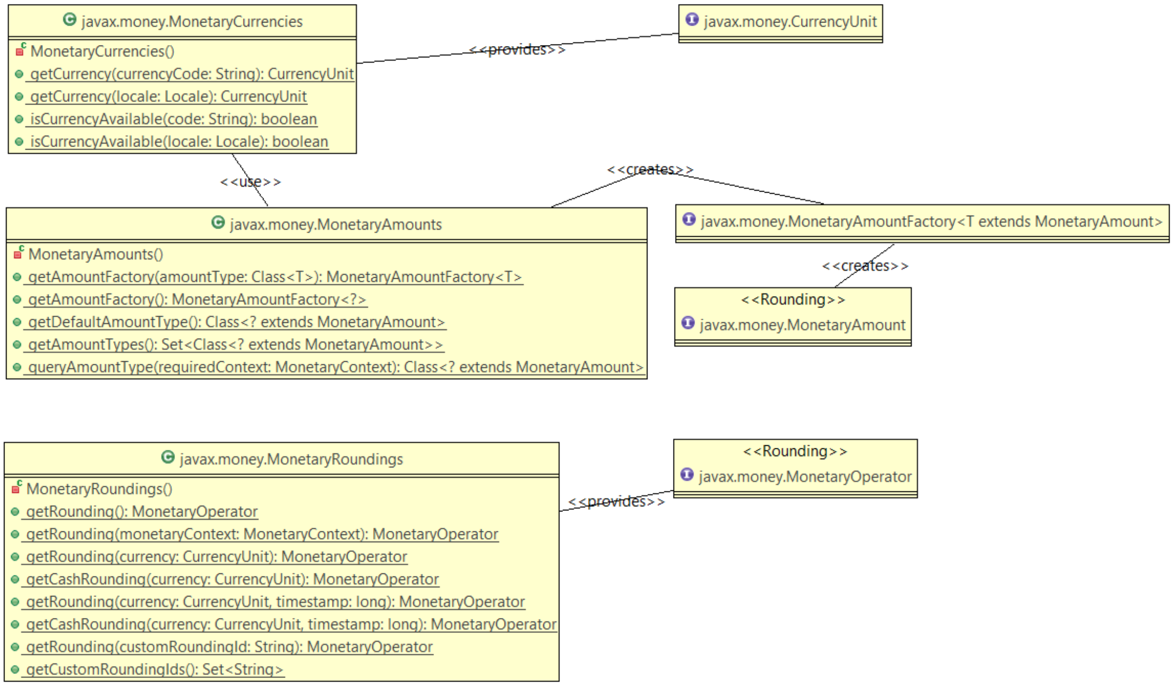
between different implementations on functional level. Interoperability on data level is ensured by `getNumber()` and `getCurrency()`. As a consequence amount can be implemented in different ways, focusing on the behavioural and data representation requirements implied by the concrete use cases.

- `NumberValue` returns the numeric part of an amount, so it can be accessed and externalized in different ways. Its purpose is to ensure maximal interoperability with existing functionalities in the JDK. Therefore it also extends `java.lang.Number`.
- `NumberSupplier` and `CurrencySupplier` model functional interfaces as defined by JDK 8.
- `MonetaryOperator` and `MonetaryQuery` model the extension points for monetary logic. They allow to implement external functionalities, either adding operations returning a MonetaryAmount (`MonetaryOperator`), or returning any arbitrary other value (`MonetaryQuery`).
- the `MonetaryAmount` factory finally represents an abstraction for creating new instances of amounts. Besides setting an amount currency and number value, it allows also to change the numeric capabilities, if the underlying implementation supports doing this. The capabilities available for a concrete factory can be queried by accessing maximal `MonetaryContext`.
- `MonetaryContext` defines the numeric capabilities of an instance as an immutable and platform independent type.
- `MonetaryException` is the base exception class for the money API, it extends `RuntimeException`.

The overview diagram above shows that the main abstraction is modeled as interfaces. There are people that would argue, that concrete immutable value types should be used to model a monetary amount. This topic was discussed intensively in the expert group, some of the aspects considered include:

- Using a concrete type as the model for a monetary amount implies a string relation to a numeric representation. Unfortunately, as seen in the use cases and requirements sections, performance and precision are conflicting requirements. Additionally, though not explicitly in scope, low latency systems may even require amounts to be mutable to able to cover the strong performance requirements. So modelling the amount as a concrete type would effectively prevent the flexibility that is required.
- Also using self-referencing template parameters was considered. The disadvantage is that you still have to know the concrete class. In that case you could also use the concrete class directly, instead of using non trivial generics semantics. Additionally in many cases these complex semantics would lead quite probably to broad usage of raw types, which will make the design quite counterproductive.
- So finally the interface based design gives maximum flexibility, ensures interoperability on data and operational level and still does not prevent its use in high performance, low latency scenarios. As a side effect it also allowed us to design it completely platform independent. Though not primarily in focus the JSR 354 API is completely platform independent.

Nevertheless for an API to be complete, you need some type of concrete classes as entry points. Since the API is designed as a standalone APIs the singleton accessor patterns are a good choice, so also this API provides according accessor classes:


javax.money.MonetaryCurrencies
MonetaryCurrencies()
getCurrency(currencyCode: String): CurrencyUnit
getCurrency(locale: Locale): CurrencyUnit
isCurrencyAvailable(code: String): boolean
isCurrencyAvailable(locale: Locale): boolean
<<provides>>
javax.money.CurrencyUnit
<<use>>
<<creates>>
javax.money.MonetaryAmounts
MonetaryAmounts()
getAmountFactory(amountType: Class<T>): MonetaryAmountFactory<T>
getAmountFactory(): MonetaryAmountFactory<?>
getDefaultAmountType(): Class<? extends MonetaryAmount>
getAmountTypes(): Set<Class<? extends MonetaryAmount>>
queryAmountType(requiredContext: MonetaryContext): Class<? extends MonetaryAmount>
javax.money.MonetaryAmountFactory<T extends MonetaryAmount>
<<creates>>
<<Rounding>>
javax.money.MonetaryAmount
javax.money.MonetaryRoundings
MonetaryRoundings()
getRounding(): MonetaryOperator
getRounding(monetaryContext: MonetaryContext): MonetaryOperator
getRounding(currency: CurrencyUnit): MonetaryOperator
getCashRounding(currency: CurrencyUnit): MonetaryOperator
getRounding(currency: CurrencyUnit, timestamp: long): MonetaryOperator
getCashRounding(currency: CurrencyUnit, timestamp: long): MonetaryOperator
getRounding(customRoundingId: String): MonetaryOperator
getCustomRoundingIds(): Set<String>
<<provides>>
<<Rounding>>
javax.money.MonetaryOperator

Basically the diagram above illustrates well the accessors available:

- `MonetaryCurrencies` provides `CurrencyUnit` instances.
- `MonetaryAmounts` provides factories for creating `MonetaryAmount`. To mention is also a query functionality, where given a required `MonetaryContext` the best matching implementation type can be queried.
- `MonetaryRoundings` finally provides access to roundings, modelled as `MonetaryOperator`.

The following sections will describe these artifacts in more detail.

4.2.1 Modeling of Currencies

When thinking of monetary values it is inevitable to think on how a currency must be modeled. Although the JDK already provides a `java.util.Currency` class, this JSR's expert group discussed, if the existing class is sufficient or what kind of additions are necessary. Fortunately a minimal interface `CurrencyUnit` could be extracted, that models a subset of the existing functionality on `java.util.Currency`, so the existing class could easily implement the new interface. Compared to the interface does not provide methods for localizing a currency instrances such as `getDisplayName(Locale), getSymbol(Locale)`. This allows to separate the different concerns of data modelling and formatting. Modelling the currency as an interface also has additional advantages:

- An interface can be implemented multiple times. There are use cases, where additional data must be stored along the common currency data, which now can be done by implementing according currencies.
- Interoperability between a standalone implementation of this JSR and the JDK's `Currency` class can be ensured, even when this JSR would be integrated into the JDK later, since the references to the interface must not change.

So the interface for currencies is modelled only with 3 methods as follows:

```
public interface CurrencyUnit{
 String getCurrencyCode();
 int getNumericCode();
 int getDefaultFractionDigits();
}
```

Hereby:

- 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: For ISO currencies this will the 3-letter uppercase ISO code. For non ISO currencies no constraints are defined.
- The numeric code returned by `getNumericCode()` is optional. If not defined it must be `-1.`
- The default fraction digits define the typical scale of values with a given currency.

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. must be serializable.

4.2.2 Modeling of Monetary Amounts

Modeling of monetary amount agnostic to its concrete numeric representation 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. This was necessary since it has shown that different usage scenarios of money can result in rather different requirements to the numeric representation of amounts, which quite probably may not fit into *one-fits-it-all* implementation.
One key aspect is that a monetary amount is always related to its currency. Mixing of currencies makes typically no sense for arithmetic operations on amount or, even worse, results in useless and incorrect results.
As a consequence the properties and operations of monetary amounts for data and functional interoperability are modelled by an *interface, called* `javax.money.MonetaryAmount`.
In general the following aspects are modelled:

- **Data interoperability** allowing access to the amount's
  - currency modeled as `CurrencyUnit`.
  - number value, for externalization, modeled as `NumberValue`.
  - accessing basic numeric state such as *negative, positive* etc.
  - Methods for evaluating the numeric capabilities of the concrete type.
- **Prototyping support** for creating new amount instances based on the same implementation, modeled by `MonetaryAmountFactory.`
- **Comparison methods** for comparing two arbitrary amounts of the same currency, hereby comparing based on the (effective) numeric value (e.g. ignoring trailing zeroes).
- **Basic arithmetic operations** like addition, subtraction, division, multiplication.
- **Functional exension points** modeled as `MonetaryOperator` (returning amount instances of the same implementation type) and `MonetaryQuery` (returning any result type).

Summarizing the interface is defined as follows:

```
public interface MonetaryAmount{

  CurrencyUnit getCurrency();
  NumberValue getNumber();
  MonetaryContext getMonetaryContext();

  // Create an factory that allows to create a new amount based on this amount
  MonetaryAmountFactory<?> getFactory();

  // Create an instance as a result of an external monetary operation
  MonetaryAmount with(MonetaryOperator operator);

  // QUery data from an amount
  <R> R query(MonetaryQuery<R> query);
```

```
    // Comparison methods
    boolean isGreaterThan(MonetaryAmount amount);
    boolean isGreaterThanOrEqualsTo(MonetaryAmount amount);
    boolean isLessThan(MonetaryAmount amount);
    boolean isLessThanOrEqualsTo(MonetaryAmount amount);
    …
    boolean isEqualTo(MonetaryAmount amount);
    boolean isNegative();
    boolean isPositive();
    boolean isZero();
    int signum();

    // Algorithmic functions and calculations
    MonetaryAmount add(MonetaryAmount amount);
    MonetaryAmount subtract(MonetaryAmount amount);
    MonetaryAmount multiply(long amount);
    MonetaryAmount multiply(double amount);
    MonetaryAmount multiply(Number amount);
    MonetaryAmount divide(long amount);
    MonetaryAmount divide(double amount);
    MonetaryAmount divide(Number amount);
    MonetaryAmount remainder(long amount);
    MonetaryAmount remainder(double amount);
    MonetaryAmount remainder(Number amount);
    MonetaryAmount divideAndRemainder(long amount);
    MonetaryAmount divideAndRemainder(double amount);
    MonetaryAmount divideAndRemainder(Number amount);
    MonetaryAmount scaleByPowerOfTen(int power);
    MonetaryAmount abs();
    MonetaryAmount negate();
  }
```

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.
- `NumberValue getNumber()` returns a `NumberValue` (discussed within the next section) that models the numeric part of an amount for data interoperability.
- `getMonetaryContext` allows to access the monetary context of the numeric part, similar to `java.math.MathContext`. The corresponding class is discussed later in this document.
- Instances of `MonetaryOperator` and `MonetaryQuery<R>` can be applied on a `MonetaryAmount` instance by passing them to the `with` or `query` method. Whereas an *operator* takes calculates a new amount based on a amount (an instance of an unary function), a *query* can return arbitrary result types.
- `isGreaterThan,isLessThan,isGreaterThanOrEqualTo etc` model basic

comparison methods, which are required to work also when comparing different implementation types. This is possible, since the numeric representation as well as the `MonetaryContext` can be accessed in a implementation agnostic way.
Also is important that the comparisons are based on the least *significant* numeric scale, e.g. `CHF 1.05` and `CHF 1.05000` are considered to be *equal*.

- The rest of the methods model common arithmetic operations that are often used in financial applications. Adding and subtracting hereby is only possible with amounts that are of the same currency (aka being currency compatible[5]) that the amount on which the operation is executed. The arithmetic methods should basically behave similar to `java.math.BigDecimal`.

Implementation Requirements

The specification and interface do not define precisely how the amount is stored. Implementations could use a `BigDecimal`, `long` or something else. The only constraint is that the numeric value can be exposed as `NumberValue` and that the `MonetaryContext` returned reflects the numeric capabilities accordingly.

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

1. must implement `equals/hashCode`, hereby it is recommended considering
    a. Implementation type
    b. `CurrencyUnit`
    c. Numeric value, with any *non significant trailing zeros truncated*.
    d. `MonetaryContext`

   This also means that two different implementations types with the same currency and numeric value are *NOT* equal. For comparing two `MonetaryAmount` instances during financial calculations the amount's comparison methods should be used. E.g. `isEqualTo` must return true, if they have equal currencies and equal numeric values, hereby *ignoring non-significant trailing zeros and different monetary contexts.*
2. must be comparable.
3. must be serializable.
4. should be immutable and thread safe.
5. To enable interoperability a `static` method `from(MonetaryAmount)` is recommended to be implemented on the concrete type, that allows conversion of a `MonetaryAmount` to a concrete type `T`.

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

6. Finally implementations should not implement a method `getAmount()`. This method is reserved for future integration into the JDK.
7. If the numeric representation allows to model `-0`, this value is also considered to be

---

[5] Note that currency conversion is a complex aspect that can not be performed implicitly or automatically. E.g. a conversion rate is dependent from the timestamp, the currencies involved, the provider, the amount ...

`isZero()==true`, and additionally should be equal to `0`.

The interfaces, `MonetaryOperator` and `MonetaryQuery<R>`, provide a powerful extension mechanism. The two interfaces operate as a form of the strategy pattern, allowing the algorithm of a query or operation to be external to the implementation of `MonetaryAmount`. Their design matches JSR-310.

This specification does no further constrain the constructor or factory methods to be implemented, or the method signatures to be used.

4.2.3 Externalizing the Numeric Value of an Amount

In the previous section we have discussed the basic model of a monetary amount. For data interoperability between different implementations it is very important that the numeric value of an amount can be effectively be externalized. Hereby the API was aimed to be platform independent, which disallows the usage of `java.math.BigDecimal`. Nevertheless simply returning `java.lang.Number`, is also not desired, since conversion to known types may imply rounding errors or truncation. So the solution was to extend `java.lang.Number`, since it is the basic type used in the JDK, but adding additional methods that help users to better identify the risks of different externalization operations and provide functionality for effective access to the numeric data:

```
public abstract class NumberValue extends java.lang.Number{
 public abstract Class<?> getNumberType();
 public abstract int intValueExact();
 public abstract long longValueExact();
 public abstract double doubleValueExact();
 public abstract <T extends Number> T numberValue(Class<T> numberType);
 public abstract <T extends Number> T numberValueExact(Class<T> numberType);
 public abstract int getPrecision();
 public abstract int getScale();
}
```

Hereby

- `getNumberType()` provides information about the numeric representation used internally. It does explicitly not constraint the type returned to be a subtype of `java.lang.Number` to allows also alternate implementations used.
- `intValueExact(), longValueExact(), doubleValueExact()` extend the methods defined in `java.lang.Number`, with their *exaxt* variants. *Exact* means, that it is required to throw an `ArithmeticException`, if the current numeric value must be truncated to fit into the required target type.
- `numberValue(Class)` allows accessing the numeric value hereby defining the required numeric representation type. If needed the numeric value may be truncated to fit into the required type. The following types must be supported:
  - `Integer, Long, Float, Double`

  - If available in the current runtime environment also: `BigDecimal, BigInteger`
- `numberValueExact(Class)` works similarly to `numberValue(Class)`, but the value returned must be *exact*. It is required to throw an `ArithmeticException,` if the current numeric value must be truncated to fit into the required target type. The types supported are similar to `numberValue(Class)`.
- `getPrecision(), getScale()` allows to access the current precision and scale of the numeric value.

4.2.4 Functional Extension Points: Operators and Queries

Since the model for monetary amounts only defines a minimal set of algorithmic functions and a prototyping mechanism additional extension points are required to allow easily external functionalities, e.g. more complex financial operations, being applied on amounts. This is modelled by

- *monetary operators,* which model a function *f(M1) -> M2*, that converts an amount to another amount, and
- *monetary queries*, which model a function *f(M1) -> T*, that converts an amount to any type of result.

Monetary Operators

The interface `javax.money.MonetaryOperator` defines an arbitrary function a function *f(M1) -> M2*, that converts an amount to another amount. Examples of such operations are rounding or monetary calculations:

```
public interface MonetaryOperator{
  <T extends MonetaryAmount> T apply(T amount);
}
```

Operators 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 `MonetaryOperator` should be

- immutable and
- thread-safe

The operator is typically invoked on the instance of an amount, passing the operator as a parameter.

```
MonetaryAmount amount = …
MonetaryOperator op = ...
MonetaryAmount result = amount.with(op);
```

Hereby, also looking at the signature, the returned (implementation) type must be the same as the type passed. This is also the case, when working with interfaces, so given the example above the following is required to be `true`:

```
amount.getClass()==result.getClass()
```

Fortunately this can be achieved easily, since the same constraint applies similarly

- to the type returned by the arithmetic operations on `MonetaryAmount` (1).
- the type returned by the `MonetaryAmountFactory` accessible from each `MonetaryAmount` (2).

So the following statements must always be `true`:

```
(1) amount.getClass() == amount.multiply(2.5).getClass()
(2) amount.getClass() ==
        amount.getFactory().with(2.5).create().getClass()
```

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

Monetary Queries

The interface `javax.money.MonetaryQuery` models a function *f(M1) -> T*, that converts an amount to any type of result:

```
public interface MonetaryQuery<R> {
  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 (predicates), such as ‘is negative’, ‘is zero’ or ‘is currency widely traded’
- splitting the amount into smaller amounts
- serialization to string/bytes
- accessing the amounts currency or properties in a functional way.

Implementations of `MonetaryQuery<R>` should be

- immutable and
- thread-safe

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

```
MonetaryAmount amount = …
MonetaryQuery<Boolean> check4eyesPrincipleNeeded = ...
boolean is4eyesPrincipleNeeded = amount.query(check4eyesPrincipleNeeded);
```

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

4.2.5 The Monetary Context

This monetary context models the numeric capabilities of an monetary amount (implementation) in a platform independent way. Though it is similar to `java.math.MathContext` for `BigDecimal` it is far more flexible, since different implementations may add several attributes that be relevant.
A monetary context (modeled as `javax.money.MonetaryContext`) is basically used on the following distinct use cases:

- It can be accessed on each instance of `MonetaryAmount`, hereby **providing information about the numeric capabilities of a concrete amount implementation instance** without having to reference to the concrete implementation class.
- Similarly a `MonetaryContext` can be passed to `MonetaryAmounts.queryAmountType(MonetaryContext ctx)` to **evaluate the implementation type that is covering a required monetary context best** (refer to the section discussing the `MonetaryAmounts` singleton and the `MonetaryAmountsSpi` SPI interface for further details on how the selection algorithm is specified). The returned implementation type `M` (aka amount type) then can be used to acquire a corresponding `MonetaryAmountFactory<M>` by calling `MonetaryAmounts.getAmountFactory(Class<M>)` to create instances of the given amount type `M`.
- Finally each `MonetaryAmountFactory<T>` allows **creation of `MonetaryAmount`** instances, without passing a `MonetaryContext` instance explicitly. In such a case the factory uses a ***default*** **monetary context,** accessible also by calling `getDefaultMonetaryContext()` on the factory. Similarly the **maximal supported capabilities** of a `MonetaryAmountFactory<T>` can be determined by calling `getMaximalMonetaryContext()`.

The `MonetaryContext` is modeled as an immutable type as follows:

```
public final class MonetaryContext
implements Serializable{
  public static enum Flavor{
    PRECISE,
    PERFORMANT,
    UNKNOWN
  }
  ...
  private MonetaryContext(Class<? extends MonetaryAmount> amountType, ...);

    public int getPrecision();
    public int getMaxScale();
    public Flavor getAmountFlavor();
    public <A> A getAttribute(Class<A> type);
    public <A> A getAttribute(Class<A> type, A defaultValue);
    public Map<Class,Object> getAttributes();
    public Set<Class> getAttributeTypes();
    public Class<? extends MonetaryAmount> getAmountType();

    public final static class Builder{
         ...
    }
}
```

Hereby

- `getPrecision()`, `getMaxScale()`, `isFixedScale()` define common numric capabilities.
- `getAmountType()` gives access to the amount's implementation type used.
- `getAmountFlavor()` allows to define a behavioural flavor, one of:
  - `PERFORMANT`: the implementation is optimized for fast computation. In favour of the performance optimization the precision and/or scale supported may be limited.
  - `PRECISE`: the implementation is optimized for providing correct result at all possible, but it may not perform as well as performance optimized implementations.
  - `UNDEFINED`: it is not possible to define a clear flavor, the `MonetaryContext` is used to determine the amount type that optimally suits the current requirements, but no specific flavor is required. Amount factories that are
- also a `MonetaryContext` provides additional attributes, identified by the attribute's type. This creates a type safe interface for adding properties, without duplicating artifacts or creating non portable dependencies.

  The example below creates a `MonetaryContext` matching amount implementations that are performance optimized, that have a maximal precision of 12, with a maximal scale of 2 and should be rounded up. Interesting hereby is that though the type `java.math.RoundingMode` is used (which would not available on Java ME), no API dependency on Java SE is implied:

```
MonetaryContext ctx = new MonetaryContext.Builder()
  .setMaxScale(2)
```

```
.setFixedScale(true)
.setPrecision(12)
.setAttribute(RoundingMode.UP)
.setFlavor(AmountFlavor.PERFORMANT)
.build();
```

4.2.6 Creating Monetary Amount Instances

Basically new instances of monetary amounts can be created in different ways. One way[6] will be by using factories, modeled by the interface `javax.money.MonetaryAmountFactory<T>.` Instances can be obtained in different ways

- calling `getFactory()` on an instance of `MonetaryAmount`, returns an instance that is *initialized with the current amount instance's values*, allowing for easily creation of similar amount instances, with some or multiple properties changed. This is known as using prototype pattern [Gof]. This is useful for `MonetaryOperator` implementations, where the default operations available on `MonetaryAmount` are not sufficient for implementing the logic/result required, or calculations are done externally and a new amount is created with the numeric result of that calculation.
- the `MonetaryAmounts` singleton also provides access to `MonetaryAmountFactory` instances, hereby also allowing to bind to a specific implementation type:

```
MonetaryAmountFactory<MyMoney> fact = MonetaryAmounts.
                                        getAmountFactory(MyMoney.class);
fact.withCurrency("USD").with(10.50);
…
MyMoney money = fact.create();
```

The signature of `MonetaryAmountFactory` is modelled as a builder also supporting a fluent programming style:

```
public interface MonetaryAmountFactory<T extends MonetaryAmount> {
  Class<T> getAmountType();
  MonetaryContext getDefaultMonetaryContext();
  MonetaryContext geMaximalMonetaryContext();

  MonetaryAmountFactory<T> setCurrency(CurrencyUnit currency);
  MonetaryAmountFactory<T> setCurrency(String code);
  MonetaryAmountFactory<T> setNumber(double number);
  MonetaryAmountFactory<T> setNumber(long number);
  MonetaryAmountFactory<T> setNumber(Number number);
  MonetaryAmountFactory<T> setContext(MonetaryContext ctx);
  MonetaryAmountFactory<T> setAmount(MonetaryAmount amount);
```

---

[6] This is the mechanism that will be interoperabel and will be tested by the TCK for each registered amount type and therefore is the recommended way of doing. Nevertheless it is still possible to define final value types with static factory methods on it, additionally to this mechanism.

```
  T create();
}
```

Hereby

- `create` returns a new instance of `T` based on the current data set on the factory.
- If no `MonetaryContext` has been set explicitly a default `MonetaryContext` is used, which can be determined by calling `getDefaultMonetaryContext()`.
- The maximal supported `MonetaryContext` can also be determined by calling `getMaximalMonetaryContext()`.
- `getAmountType()` returns the amount implementation class that will be created by a given factory instance.
- `setAmount(MonetaryAmount)` allow to initialize the factory with the values from any arbitrary amount. If the amount passed hereby exceeds the maximal `MonetaryContext` that can be supported, a `MonetaryException` must be thrown.
- the other `setXXX` methods allow to set other aspects of the `MonetaryAmount` to be created, such as
  - the `CurrencyUnit` (either directly or by passing a currency code)
  - the number value, hereby if a numeric value passed, that exceeds the representation capabilities of the targeted amount implementation (or more precise: exceed the capabilities of the maximal `MonetaryContext`), the following strategy should be implemented:
    - If the current implementation supports extending the `MonetaryContext` used, the `MonetaryContext` should be extended to accommodate the precision and scale required, e.g. an implementation based on `java.math.BigDecimal`, can be constrained to a `MathContext.DECIMAL64`, but can be easily extended to support bigger precisions.
    - If the current implementation is not able to reflect the numeric value required without doing any truncation, it must throw an `ArithmeticException`.

4.2.7 Accessing Currencies, Amounts and Roundings

Accessing Currencies

The `javax.money.MonetaryCurrencies` *singleton* class implements an accessor for `CurrencyUnit` instances. By default it is backed up by `java.util.Currency`, but allows registration of additional currencies by implementing an instance of `CurrencyProviderSpi` (explained later in this document).

```
public static CurrencyUnit getCurrency(String currencyCode){...}
public static CurrencyUnit getCurrency(Locale locale){...}
public static boolean isCurrencyAvailable(String code){...}
public static boolean isCurrencyAvailable(Locale locale) {...}
```

Hereby

- access is provided based on `Locale`, or by using the currency code. Implementations must at least provide the same locales and codes as supported by `java.util.Currency`.
- additional `CurrencyUnit` can be added by registering instances of the `CurrencyProviderSpi` as explained within the SPI section later.
- whereas, similar to `java.util.Currency` accessing a currency that does not exist, throws an `IllegalArgumentException`, the `isCurrencyAvailable` methods allow to check if a currency `code` or `Locale` is defined.

One may consider also adding access to historic currencies here. The problem hereby is that the existence of a currency is related to multiple attributes:

- the target timestamp, when it should be valid, e.g. as UTC timestamp
- the target country or region, as it was existing at that time
- the timezones of the country or region, to determine the exact timeranges related to the given target timestamp
- additionally also countries change during history

Summarizing adding historic currency support was considered to be not appropriate for being added to a core API. Nevertheless in the JavaMoney library historic currencies can be accessed, related to corresponding countries, modeled as so called regions.

Accessing Monetary Amount Factories

The `javax.money.MonetaryAmounts` *singleton* class implements an accessor for `MonetaryAmountFactory` instances. Hereby for not hard-coding the selection algorithm and for enabling contextual behaviour in a EE context, the singleton is backed up by a `MonetaryAmountsSpi`, that can be registered using the JSR's `Bootloader`.

```
public static <T extends MonetaryAmount> MonetaryAmountFactory<T>
                                getAmountFactory(Class<T> amountType);
public static MonetaryAmountFactory<?> getDefaultAmountFactory();
public static Set<Class<? extends MonetaryAmount>> getAmountTypes();
public static Class<? extends MonetaryAmount> queryAmountType(
                        MonetaryContext requiredContext);
```

Hereby

- `getAmountFactory(Class)` provides access to the corresponding `MonetaryAmountFactory<T>` matching the amount type `T`.
- additionally a default `MonetaryAmountFactory` can be accessed, by calling `getDefaultAmountFactory()`. Hereby the default type is the provided amount class of the `MonetaryAmountFactory` with the highest priority (determined by the `Bootstrap` implementation). This can be overridden by adding a `javamoney.properties` file to the classpath as follows:

```
# Defaults for java money
#------------------------------
javax.money.defaults.amount.class=my.fully.qualified.MonetaryAmountType
```

- `getAmountTypes()` returns all amount implementation classes currently available.
- Finally `queryAmountType(MonetaryContext)` allow to query the implementation class that best covers the given required `MonetaryContext`.

Implementations of this JSR must at least provide one[7] implementation of `MonetaryAmountFactoryProviderSpi` with a query policy equal to `QueryInclusionPolicy.ALWAYS`.

Accessing Roundings

Rounding is modeled by implementations of `MonetaryOperator`. Hereby beside mathematical roundings, also non standard variants with arbitrary rules and constraints are quite common in the financial area.
This JSR provides several roundings accessible from the `javax.money.MonetaryRoundings` singleton based on:

1. a target `CurrencyUnit`,. By default the rounding is based on the currency's default fraction units. Additionally also a cash rounding can be accessed, which may be different than the default currency rounding (e.g. for CHF/Swiss Francs).
2. a `MonetaryContext`, which defines the maximal precision and scale. Where available the `MonetaryContext` can have an additional attribute of type `java.math.RoundingMode`, providing a definition of the required mathematical

---

[7] If `MonetaryContext.AmountFlavor` does not equal `AmountFlavor.UNDEFINED`, it is recommended to provide also a second amount type, either with the alternate specified `AmountFlavor`, or with `AmountFlavor.UNDEFINED`, which then is used as default.

rounding. If not defined `HALF_EVEN` rounding should be used.

3. a name (`String`), for customized roundings.

The `MonetaryRoundings` singleton provides access to all these roundings with a couple of methods:

```
public static MonetaryOperator getRounding();
public static MonetaryOperator getRounding(MonetaryContext context);
public static MonetaryOperator getRounding(CurrencyUnit currency);
public static MonetaryOperator getCashRounding(CurrencyUnit currency);
public static MonetaryOperator getRounding(CurrencyUnit currency,
                                           long timestamp);
public static MonetaryOperator getCashRounding(CurrencyUnit currency,
                                           long timestamp);
public static MonetaryOperator getRounding(String customRoundingId);
public static Set<String> getCustomRoundingIds();
```

Hereby,

- `getRounding()` returns a general rounding instance that is dynamically implementing the default currency rounding, as required by the currency passed, when called.
- `getRounding(CurrencyUnit)` returns the default rounding for the given CurrencyUnit, whereas `getCashRounding(CurrencyUnit)` returns the cash rounding for the given currency, which may be different from the default rounding. E.g. for Swiss Francs the cash rounding will be in 5 minor unit steps: `1.00, 1.05, 1.10` etc..
- `getRounding(CurrencyUnit, long)`, `getCashRounding(CurrencyUnit, long)` provide access to currency related rounding and cash rounding for a certain timestamp.
- `getRounding(int, RoundingMode)` returns a general mathematical rounding instance.
- finally `getCustomRounding(String)` allows to access custom roundings, as defined by the registered `RoundingProviderSpi` implementations. `getCustomRoundingIds()` provides access to the names of the currently registered custom roundings.

4.2.8 Additional Functional Support

Though this JSR is too early to be built using JDK 8, functional aspects are already considered in its design. For example monetary operators and monetary queries basically are functional interfaces. Additional access the the numeric part as well as to the currency of an amount is modeled with corresponding functional interfaces:

CurrencySupplier

The interface `javax.money.CurrencySupplier` is a functional interface (the `CurrencyUnit`-producing specialization of a `Supplier` as defined in Java 8), whose functional method is `getCurrency()`:

```
// @FunctionalInterface
public interface CurrencySupplier {
  CurrencyUnit getCurrency();
}
```

Hereby

- There is no requirement that a distinct result be returned each time the supplier is invoked.

NumberSupplier

The interface `javax.money.NumberSupplier` is a functional interface (the `NumberValue`-producing specialization of a `Supplier` as defined in Java 8), whose functional method is `getCurrency()`:

```
// @FunctionalInterface
public interface NumberSupplier {
  NumberValue getNumber();
}
```

Hereby

- There is no requirement that a distinct result be returned each time the supplier is invoked.

4.2.9 Exception Types

javax.money.MonetaryException

`javax.money.MonetaryException` is a *runtime* exception, which models the base exception for all other exceptions.. Any monetary exception added by an implementation must inherit from this class.

javax.money.UnknownCurrencyException

This *runtime* exception extends `MonetaryException` and is thrown whenever

- a currency code given cannot be resolved into a corresponding `CurrencyUnit` instance. The invalid currency code passed is provided as a property on the exception:

```
public String getCurrencyCode();
```

- a `Locale` given cannot be resolved into a corresponding `CurrencyUnit` instance. The unresolvalbe `Locale` passed is provided as a property on the exception:

```
public Locale getLocale();
```

4.3 Currency Conversion

Currency conversion is an important aspects when dealing with monetary amounts. Unfortunately currency conversion has a great variety of how it is implemented. Whereas a web shop may base its logic on an API provided by a financial backend, that make explicit conversion even not necessary, in the financial industry, conversion is a very complex aspects, since

- conversion may be different based on the use case
- conversion may be different based on the provided of the exchange rates
- conversion rates may vary based on the amount to be converted
- conversion rates may vary based on contract or business unit
- conversion rates are different related to the target timestamp

Hereby this list is not complete. Different companies may have further requirements and aspects to be considered.

4.3.1 `MonetaryConversions` Singleton

The API defines a singleton accessor, called `MonetaryConversions`, which provides access to all different aspects related to currency conversion, such as

- access to providers that offer conversion (exchange) rates.
- access to conversion operators (extending `MonetaryOperator`), that can be used with any `MonetaryAmount` instances.
- access to further information about the providers currently available.

The following sections give an overview about the functionalities in more detail. Similar to other singletons in this API the singleton is backed up by a `MonetaryConversionsSingletonSpi` SPI to allow customized (contextual) implementation of the functionalities defined. Refer to the SPI section in this document for more details.

4.3.2 Converting Amounts

Basically converting of amounts is modelled by the `CurrencyConversion` interface which extends `MonetaryOperator`. Hereby a conversion is always bound to a specific terminating (target) currency. So basically a `MonetaryAmount` can simply be converted by

```
MonetaryAmount amount = …;
CurrencyConversion conversion = MonetaryConversions.getConversion(“CHF”);
MonetaryAmount amount2 = amount.with(conversion);
```

Using a fluent API style this can be written even shorter as:

```
MonetaryAmount amount2 = amount.with(MonetaryConversions.getConversion(“CHF”));
```

A `CurrencyConversion` instance hereby also allows to extract the `ExchangeRate` instances used:

```
CurrencyConversion conversion = MonetaryConversions.getConversion(“CHF”);
MonetaryAmount amount = …;
ExchangeRate rate = conversion.getExchangeRate(amount);
```

4.3.3 Exchange Rates and Rate Providers

The `ExchangeRate` models the details of a conversion applied:

- the base and terminating (target) `CurrencyUnit`.
- the conversion factor used[8] modeled as `NumberValue`.
- additional information if the rate is derived, meaning built up the result of rate chain. If a rate is derived `getExchangeRateChain()` returns the rate chain that is used to derive the given (final) exchange rate.
- a `ConversionContext`, which can contain arbitrary additional information about the provider that issued the rate and arbitrary further aspects concerning the rate/conversion.

We have seen in the previous section that an `ExchangeRate` can be obtained from a `CurrencyConversion`. Hereby a currency conversion is backed up by an `ExchangeRateProvider`. Such a provider allows

- to access `ExchangeRate` instances, providing a base and a terminating (target) currency.
- to access `CurrencyConversion` instances, providing a terminating (target) currency.

Both functionalities allow additionally to pass a `ConversionContext`, which allow to pass any additional attributes/parameters that may be required by a concrete `ExchangeRateProvider` instance. This allows to support arbitrary complex use cases, as an example[9] an implementation require/allow to pass

- the target amount
- a customer id
- a contract id
- a fallback strategy
- a deferred rate should be obtained

The parameters then can be included in an instannce of`ConversionContext`. This context then can be used to pass additional parameters to all rate providers that answer a given conversion query:

---

[8] Note that the conversion rate can be dependent on the `MonetaryAmount` passed.

[9] This example is completely arbitrary.

```
ConversionContext ctx = new ConversionContext.Builder()
        .setRateType(RateType.DEFERRED).
        .set(“customerID” 1234)
        .set(“contractID”, “213453-GFDT-02”)
        .set(FallbackStragey.PROVIDER)
        .set(amount)
        .create();
```

The built context then can be passed to parametrize the `CurrencyConversion` or `ExchangeRate` instance as follows:

```
ConversionContext ctx = …;
CurrencyConversion conversion = MonetaryConversions.getConversion(“CHF”, ctx);

ExchangeRateProvider prov = MonetaryConversions.getExchangeRateProvider();
CurrencyConversion conversion = prov.getCurrencyConversion(“CHF”, ctx);
ExchangeRate rate = prov.getExchangeRate();
```

Important to understand is that its the responsibility of the used `ExchangeRateProvider` implementation to interpret the attributes passed within a `ConversionContext`, Unknown parameters should simply be ignored, since a provider can be used in a provider chain (explaned later).

4.3.4 Provider Chains

Reading the previous sections one might ask, how multiple providers can be used or how an individual rate provider can be accessed. In fact all the examples seen so far rely on the default provider chain that may be accessed by calling

```
List<String> providerIds = MonetaryConversions.getDefaultProviderChain();
```

Hereby the chain contains an ordered list of provider names, which correspond to the provider names that identify each registered `ExchangeRateProvider` uniquly. The provider name is defined by each registered `ExchageRateProvider` and can be accesssed as a mandatary attribute on the `ProviderContext`. Similar to the `ConversionContext` the `ProviderContext` may contain additional data about the rate provider, such as the range and type of rates provided etc. E.g. the output of the European Central Bank (ECB) provider context, shipped with the *moneta*, reference implementation, prints out the following when accessing `toString()`:

```
ProviderContext [attributes={class java.lang.String={PROVIDER=Compound: ECB}}]
```

Each `ProviderContext` can also be obtained from the `MonetaryConversions` singleton, passing the corresponding provider name:

```
ProviderContext ctx = MonetaryConversions.getProviderContext(“ECB”);
```

As mentioned accessing a currency conversion or rate provider, without passing the providers required defaults to default provider chain. So the following two statements are equivalent, given the default chain is `“ECB”, “IMF”, “ECB-HIST”`:

```
// equivlent calls when the default provider chain equals to
// {“ECB”, “IMF”, “ECB-HIST”}
CurrencyConversion conversion = MonetaryConversions.getConversion(“CHF”, ctx);
CurrencyConversion conversion = MonetaryConversions.getConversion(“CHF”, ctx,
                                          “ECB”, “IMF”, ECB-HIST);
```

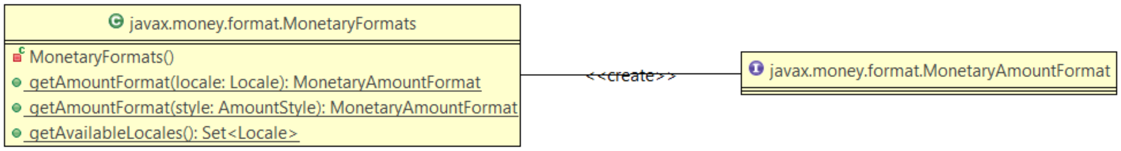
Within a provider chain, the first provider that returns a non-null result determines the final value requested, e.g. the exchange rate to be used to calculate the currency conversion. By passing the chain or providers to be used different usage scenarios can be easily separated/supported, but still kepping the API simple for the trivial use cases. Finally additional methods on the `MonetaryConversions` singleton allow to get more information on the providers available in the current context:

```
public static Collection<String> getProviderNames();
public static boolean isProviderAvailable(String providerName);
```

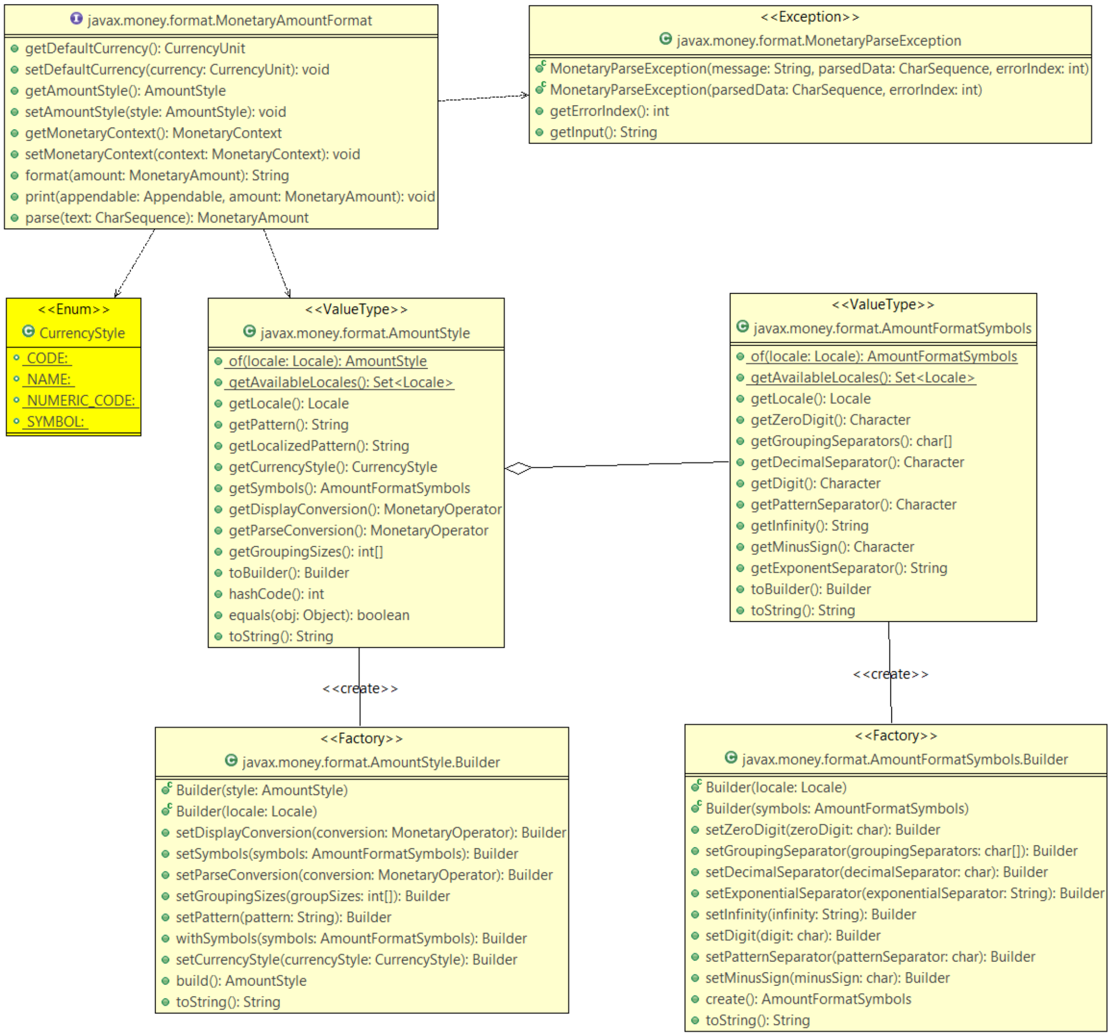
4.4 Money and Currency Formatting API

The formatting aspects modeled by several artefacts. Hereby some similarities with artifacts from JDK’s `java.text` package are not accidentally. Basically the formatter instance behaves similarly (e.g. is also mutable), whereas the underlying style and symbols were modeled as immutable value types.

Hereby like to the core APIs of the JSR a `MonetaryFormats` singleton provides access to the formatter instances:


javax.money.format.MonetaryFormats
MonetaryFormats()
getAmountFormat(locale: Locale): MonetaryAmountFormat
getAmountFormat(style: AmountStyle): MonetaryAmountFormat
getAvailableLocales(): Set<Locale>
<<create>>
javax.money.format.MonetaryAmountFormat

The following model illustrates the types involved:


javax.money.format.MonetaryAmountFormat
getDefaultCurrency(): CurrencyUnit
setDefaultCurrency(currency: CurrencyUnit): void
getAmountStyle(): AmountStyle
setAmountStyle(style: AmountStyle): void
getMonetaryContext(): MonetaryContext
setMonetaryContext(context: MonetaryContext): void
format(amount: MonetaryAmount): String
print(appendable: Appendable, amount: MonetaryAmount): void
parse(text: CharSequence): MonetaryAmount
<<Exception>>
javax.money.format.MonetaryParseException
MonetaryParseException(message: String, parsedData: CharSequence, errorIndex: int)
MonetaryParseException(parsedData: CharSequence, errorIndex: int)
getErrorIndex(): int
getInput(): String
<<Enum>>
CurrencyStyle
CODE:
NAME:
NUMERIC_CODE:
SYMBOL:
<<ValueType>>
javax.money.format.AmountStyle
of(locale: Locale): AmountStyle
getAvailableLocales(): Set<Locale>
getLocale(): Locale
getPattern(): String
getLocalizedPattern(): String
getCurrencyStyle(): CurrencyStyle
getSymbols(): AmountFormatSymbols
getDisplayConversion(): MonetaryOperator
getParseConversion(): MonetaryOperator
getGroupingSizes(): int[]
toBuilder(): Builder
hashCode(): int
equals(obj: Object): boolean
toString(): String
<<ValueType>>
javax.money.format.AmountFormatSymbols
of(locale: Locale): AmountFormatSymbols
getAvailableLocales(): Set<Locale>
getLocale(): Locale
getZeroDigit(): Character
getGroupingSeparators(): char[]
getDecimalSeparator(): Character
getDigit(): Character
getPatternSeparator(): Character
getInfinity(): String
getMinusSign(): Character
getExponentSeparator(): String
toBuilder(): Builder
toString(): String
<<create>>
<<create>>
<<Factory>>
javax.money.format.AmountStyle.Builder
Builder(style: AmountStyle)
Builder(locale: Locale)
setDisplayConversion(conversion: MonetaryOperator): Builder
setSymbols(symbols: AmountFormatSymbols): Builder
setParseConversion(conversion: MonetaryOperator): Builder
setGroupingSizes(groupSizes: int[]): Builder
setPattern(pattern: String): Builder
withSymbols(symbols: AmountFormatSymbols): Builder
setCurrencyStyle(currencyStyle: CurrencyStyle): Builder
build(): AmountStyle
toString(): String
<<Factory>>
javax.money.format.AmountFormatSymbols.Builder
Builder(locale: Locale)
Builder(symbols: AmountFormatSymbols)
setZeroDigit(zeroDigit: char): Builder
setGroupingSeparator(groupingSeparators: char[]): Builder
setDecimalSeparator(decimalSeparator: char): Builder
setExponentialSeparator(exponentialSeparator: String): Builder
setInfinity(infinity: String): Builder
setDigit(digit: char): Builder
setPatternSeparator(patternSeparator: char): Builder
setMinusSign(minusSign: char): Builder
create(): AmountFormatSymbols
toString(): String

The following section describe the relevant artifacts in more detail.

4.4.1 Formatting of Monetary Amounts

As defined in 3. Requirements, Implementations of this JSR must provide a formatter for `MonetaryAmount` instances. Nevertheless formatting is a very complex field the JSR's expert group has decided to provide a simple formatting API only, which covers the following aspects:

1. Amount values can be rounded for *display* by applying a `MonetaryOperator` before formatting/printing.
2. Similarly amount values can be operated after *parsing* by applying a `MonetaryOperator`. This is the reciprocal operation to the display rounding above.
3. It is possible to define number grouping with flexible group sizes and different grouping characters. as for example needed to format `INR`[10].
4. The currency part of an amount can be formatted in different ways:

---

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

   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`

5. The overall formatting and parsing pattern can be defined similar to `java.text.DecimalFormat`. As consequence, if defining a pattern without any currency placeholder `'¤'` (`'\u00A4'`), the currency part can also be omitted from the output.

In financial applications additional formatting requirements are quite common (see also [JavaMoney]), but these aspects will be beyond the scope of this JSR.
Nevertheless most of the use cases should be coverable by the implementations of the `MonetaryAmountFormat` interface:

```
public inteface MonetaryAmountFormat {
  String format(MonetaryAmount<?> amount);
  void print(Appendable appendable, MonetaryAmount<?> amount)
    throws IOException;
  MonetaryAmount<?> parse(CharSequence text)
    throws ParseException;

  AmountStyle getAmountStyle();
  void setAmountStyle(AmountSTyle amountStyle);
  MonetaryContext getMonetaryContext();
  void setMonetaryContext(MonetaryContext monetaryContext);
  CurrencyUnit getDefaultCurrency();
  void setDefaultCurrency(CurrencyUnit defaultCurrency);
}
```

Hereby

- an amount can be formatted to a `String` or an `Appendable`, or parsed from a `String`.
- The details of the format are managed within an immutable `AmountStyle` configuration value object.
- A `MonetaryContext` defines which type of implementation should be returned as result from a parsing operation.
- A default `CurrencyUnit` can be set, that will be used as a currency to create an amount on parsing, when no currency information can be read from the input data.

Similar to the formatters in the JDK implementations of this interface must not be thread-safe. So use of them should be synchronized.

Examples

Given the API above, acquiring a `MonetaryAmountFormat` instance is simple, the most simple usage is just creating one for a given `Locale`:

```
MonetaryAmountFactory<?> f = MonetaryAmounts.getDefaultAmountFactory();
MonetaryAmount amount = f.setCurrency(“CHF”).setNumber(12.50).create();
MonetaryAmountFormat format =
                        MonetaryAmountFormats.getAmountFormat(Locale.GERMANY);
String formatted = format.format(amount); // result: CHF 12,50
amount = f.setCurrency(“INR”).setNumber(123456789101112.123456).create();
formatted  = format.format(amount); // result: INR 123.456.789.101.112,12
```

For Indian Rupees (`INR`) it would be, of course, better using the Indian number format and different grouping sizes, for this we must first create the corresponding `AmountStyle`:

```
AmountStyle style = new AmountStyle .Builder(new Locale(“”,”INR”))
                                .withNumberGroupSizes(3,2).build();
MonetaryAmountFormat format = MonetaryAmountFormats.getAmountFormat(style);
MonetaryAmountFactory<?> f = MonetaryAmounts.getDefaultAmountFactory();
MonetaryAmount amount =
        f.setCurrency(“INR”).setNumber(123456789101112.123456).create();
String formatted = format.format(amount);
        // result: INR 12,34,56,78,91,01,112.12
```

4.4.2 Configuring a Monetary Amount Formatter

Currency Style

The `javax.money.CurrencyStyle` is modeled as an `enum` type with the following values:

- `CODE`: render the currency code. Examples: `CHF, USD`
- `NUMERIC_CODE`: render the numeric code, Examples: `62, 10, -1`
- NAME: render the localized display name, use the currency code as default, if no localized display name is present. Examples: `Swiss Francs, Japanese Yen`
- SYMBOL: render the localized currency symbol, use the currency code as default, if no localized symbol is present. Examples: `$, €, £`

Amount Style

The `javax.money.format.AmountStyle` defines how a `MonetaryAmountFormat` instance should format and/or parse `MonetaryAmount` instances. Instances of `AmountStyle` can be created using a `AmountStyle.Builder`. Summarizing the signatures look as follows:

```
    public final class AmountStyle implements Serializable{
     private AmountStyle(...);
     [...]
  public Locale getLocale();
  public CurrencyStyle getCurrencyStyle();
  public String getPattern();
  public String getLocalizedPattern();
     public AmountFormatSymbols getSymbols();
  public MonetaryOperator getDisplayConversion();
  public MonetaryOperator getParseConversion();
  public int[] getGroupingSizes();
  public Builder toBuilder();

  public static final class Builder {
   public Builder(Locale locale);
   public Builder(AmountStyle amountStyle);
   public Builder setCurrencyStyle(CurrencyStyle style);
   public Builder setGroupingSizes(int... groupSizes);
   public Builder setPattern(String pattern)
   public Builder setSymbols(AmountFormatSymbols synbols);
   public Builder setDisplayConversion(MonetaryOperator conversion);
   public Builder setParseConversion(MonetaryOperator conversion);
   public AmountStyle build();
      [...]
 }
    }
```

Hereby the above summary illustrates quite well, what are the properties that define an amount style:

- a `Locale`
- a *pattern*, defining the basic *number format*, similar as defined by `java.text.DecimalFormat.`
- *grouping sizes,* allowing to set flexible grouping sizes. Hereby the order reflects the grouping starting from the decimal point going up the significant digits. the last member of the grouping definition is used for all subsequent grouping as a default. This can be easily illustrated by setting the grouping characters to `a,b,c` and rendering the amount `112233445566778899`. Assuming a default grouping size and character this number might be formatted as `112’233’445’566’778’899.` With the grouping characters set to `a,b,c` this will be rendered as `112c233c445c566b778a899.` Now applying the same schema for grouping sizes, lets assume `3,2,5,4,1`. This will lead in combination with before to the following output: `1c1c2c2c3344c55667b78a899`.
- a `CurrencyStyle`, defining the basic *currency format* of the currency being rendered.
- a `MonetaryOperator` to be applied as *display* conversion, applied before the amount is formatted or printed.
- a `MonetaryOperator` to be applied as *parse* conversion, after the amount was parsed, e.g. for performing a symmetric reverse conversion to the rounding done during formatting.

4.4.3 Accessing Monetary Amount Formats

The class `javax.money.format.MonetaryFormats` models a singleton accessor for `MonetaryAmountFormat` instances as provided by the `MonetaryAmountFormatProviderSpi` instances registered. It provides access to `MonetaryAmountFormat` instances based on

- a `Locale, or`
- an `AmountStyle`.

It defines the following access methods:

```
public static MonetaryAmountFormat getAmountFormat(Locale locale);
public static MonetaryAmountFormat getAmountFormat(AmountStyle amountStyle);
public static Set<Locale> getAvailableLocales();
```

4.4.4 Formatting Exceptions

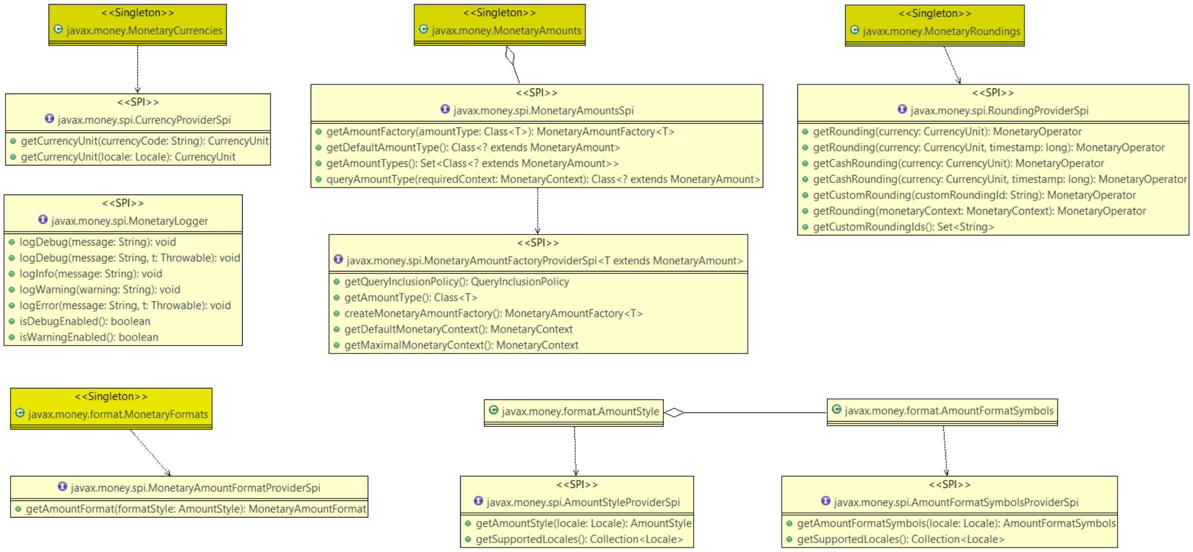
javax.money.format.MonetaryParseException

This *runtime* exception extends `MonetaryException` and is thrown whenever a MonetaryAmount could not be parsed successfully. It provides hereby additional info:

- the original input String passed to the MonetaryAmountFormat.
- the error index within the input String, where parsing failed unrecoverably.

4.5. Money and Currency SPI

JSR 354 defines a complete API and provides a default reference implementation. An implementation of this API must provide several *implementation services*, called the SPI, to provide the effective functionality. The following diagram illustrate the SPIs in place:


<<Singleton>>
javax.money.MonetaryCurrencies
<<Singleton>>
javax.money.MonetaryAmounts
<<Singleton>>
javax.money.MonetaryRoundings
<<SPI>>
javax.money.spi.CurrencyProviderSpi
getCurrencyUnit(currencyCode: String): CurrencyUnit
getCurrencyUnit(locale: Locale): CurrencyUnit
<<SPI>>
javax.money.spi.MonetaryAmountsSpi
getAmountFactory(amountType: Class<T>): MonetaryAmountFactory<T>
getDefaultAmountType(): Class<? extends MonetaryAmount>
getAmountTypes(): Set<Class<? extends MonetaryAmount>>
queryAmountType(requiredContext: MonetaryContext): Class<? extends MonetaryAmount>
<<SPI>>
javax.money.spi.RoundingProviderSpi
getRounding(currency: CurrencyUnit): MonetaryOperator
getRounding(currency: CurrencyUnit, timestamp: long): MonetaryOperator
getCashRounding(currency: CurrencyUnit): MonetaryOperator
getCashRounding(currency: CurrencyUnit, timestamp: long): MonetaryOperator
getCustomRounding(customRoundingId: String): MonetaryOperator
getRounding(monetaryContext: MonetaryContext): MonetaryOperator
getCustomRoundingIds(): Set<String>
<<SPI>>
javax.money.spi.MonetaryLogger
logDebug(message: String): void
logDebug(message: String, t: Throwable): void
logInfo(message: String): void
logWarning(warning: String): void
logError(message: String, t: Throwable): void
isDebugEnabled(): boolean
isWarningEnabled(): boolean
<<SPI>>
javax.money.spi.MonetaryAmountFactoryProviderSpi<T extends MonetaryAmount>
getQueryInclusionPolicy(): QueryInclusionPolicy
getAmountType(): Class<T>
createMonetaryAmountFactory(): MonetaryAmountFactory<T>
getDefaultMonetaryContext(): MonetaryContext
getMaximalMonetaryContext(): MonetaryContext
<<Singleton>>
javax.money.format.MonetaryFormats
javax.money.format.AmountStyle
javax.money.format.AmountFormatSymbols
javax.money.spi.MonetaryAmountFormatProviderSpi
getAmountFormat(formatStyle: AmountStyle): MonetaryAmountFormat
<<SPI>>
javax.money.spi.AmountStyleProviderSpi
getAmountStyle(locale: Locale): AmountStyle
getSupportedLocales(): Collection<Locale>
<<SPI>>
javax.money.spi.AmountFormatSymbolsProviderSpi
getAmountFormatSymbols(locale: Locale): AmountFormatSymbols
getSupportedLocales(): Collection<Locale>

These services must be registered to the `Bootstrap` singleton. The `Bootstrap` singleton relies, by default, on `java.util.ServiceLoader` to load the implementation services, but this mechanism can be replaced by an alternate component loading mechanism, such as CDI in a EE context.
All SPIs are contained in the package `javax.money.spi`. Summarizing the following SPIs are available:

- **Core SPI**
  - **`CurrencyProviderSpi`** (mandatory, multiple service chain) - provides instances of `CurrencyUnit`, accessible from `MonetaryCurrencies` singleton.
  - **`MonetaryAmountsSpi`** (mandatory, only one instance selected by priority) - manages instances of **`MonetaryAmountFactoryProviderSpi`**, which create instances of `MonetaryAmountFactory`, that are being accessible by `MonetaryAmounts`, Also this SPI allows to override the behaviour of `MonetaryAmounts.queryAmountType(MonetaryContext).`

  - **`RoundingProviderSpi`** (mandatory, multiple service chain) - provides instances of `MonetaryOperator`, for being accessible by `MonetaryRoundings.`
  - **`MonetaryLogger`** (optional, only one instance selected by priority), defines the logging backend used by the API implementation skeleton.
- **Formatting SPI**
  - **`AmountFormatSymbolsProviderSpi`** (mandatory, multiple service chain) - provides instances of `AmountFormatSymbols`, for being accessible by `AmountFormatSymbols.getInstance`.
  - **`AmountStyleProviderSpi`** (mandatory, multiple service chain) - provides instances of `AmountStyle`, for being accessible by `AmountStyle.getInstance`.
  - **`MonetaryAmountFormatProviderSpi`** (mandatory, multiple service chain) - provides instances of `MonetaryAmountFormat`, for being accessible by `MonetaryFormats.getAmountFormat`.
- **Bootstrap SPI**
  - **`ServiceProvider`** (optional, only one instance selected by priority), defines the singleton accessor for loading SPI components used by the `Bootstrap` class.
  -

How the implementations must be registered depends on the `ServiceProvider` that is loaded by the `Bootstrap` implementation. The default mechanism is based on the `java.util.ServiceLoader` class. By ordering the registered instances of some type along the priority (the most significant first), it is also possible to override partial aspects, as the first a non `null` result returned by a provider is taken as result of a call. The prioritization of components is implicitly defined by the order of the components returned by the `ServiceProvider` SPI implementation.

4.5.1 Core SPI

Registering Currencies

By adding instances of `javax.money.spi.CurrencyProvider` additional `CurrencyUnit` instances can be registered into the `MonetaryCurrencies` singleton:

```
    public interface CurrencyProviderSpi {
  public CurrencyUnit getCurrencyUnit(String currencyCode);
  public CurrencyUnit getCurrencyUnit(Locale locale);
    }
```

Hereby

- similar to `java.util.Currency.getInstance(String)` a currency is identified and can be accessed by its currency code
- similar to `java.util.Currency.getInstance(Locale)` a currency can also be

accessed by a `Locale`. Hereby the `Locale` typically represents an ISO country, but there are might alternate variants feasible.

- Also important is to mention that implementation of the `CurrencyProviderSpi` are responsible for caching the instances. Similarly the behviour of a `CurrencyProviderSpi` implementation can also be contextually dependent, as required when running in a Java EE container.

Registering Monetary Amount Factories

The `javax.money.spi.MonetaryAmountFactoryProviderSpi<T>` interface allows to create new instances of `MonetaryAmountFactory<T extends MonetaryAmount>.`
The signature looks as follows:

```
public interface MonetaryAmountFactoryProviderSpi<T extends MonetaryAmount> {
  public static enum QueryInclusionPolicy {
    ALWAYS,
    DIRECT_REFERENCE_ONLY,
    NEVER
  }
  QueryInclusionPolicy getQueryInclusionPolicy();
  Class<T> getAmountType();
  MonetaryContext getDefaultMonetaryContext();
  MonetaryContext geMaximalMonetaryContext();

  MonetaryAmountFactory<T> createAmountFactory();

}
```

Hereby

- `getAmountType()` returns a new implementation of `T` which is returned by a `MonetaryAmountFactory` created by an instance.
- The maximal supported `MonetaryContext` can be determined by calling `getMaximalMonetaryContext().`
- The default `MonetaryContext` used can be determined by calling `getDefaultMonetaryContext().`
- `createAmountFactory()` creates a corresponding `MonetaryAmountFactory` factory.
- `getQueryInclusionPolicy()` defines if the given spi (and hence the corresponding `MonetaryAmount` implementation type) is to be considered, when `MonetaryAmounts.queryAmountType(MonetaryContext)` is called:
  - `ALWAYS` means that given instance should be considered always as a candidate. Nevertheless the active implementation of `MonetaryAmountSpi` decides

finally, which implementation type (evaluated by calling `getAmountType()`) is returned as the result of such a query operation, based on the flavors and capabilities declared by the `MonetaryContext` provided.
  - `DIRECT_REFERENCE_ONLY` means that given instance should only be considered as a candidate, when the target type requested matches the type returned by `getAmountType()`.
  - `NEVER` signals that the corresponding implementation type is considered not to be a valid return type of a query operation. This is useful, e.g. for special amount types as decorators, which do not provide their own numeric representations.

Backing the MonetaryAmounts Singleton

Also the functionality of the `MonetaryAmounts` accessor singleton is backed up by an SPI interface, called `javax.money.spi.MonetaryAmountsSpi` singleton. An implementation should rely on the `Bootstrap` class to access the available instances of `MonetaryAmountFactory`. Nevertheless being able to register alternate implementations of this SPI would allow to support more complex rules for a couple of enterprise related functionalities such as:

- contextual availability of amount types (and related factories).
- contextual differences for default amount types, as provided by `MonetaryAmounts.getDefaultAmountType()`.
- contextual differences for default `MonetaryContext` instances applied.
- alternate implementations of the algorithm used within `MonetaryAmounts.queryAmountType(MonetaryContext)` to determine the best matching `MonetaryAmount` implementation given a `MonetaryContext` required.

The SPI provides the following methods to adapt the behaviour of `MonetaryAmounts`:

```
public <T extends MonetaryAmount> MonetaryAmountFactory<T>
                                  getAmountFactory(Class<T> amountType);
public MonetaryAmountFactory<?> getDefaultAmountFactory();
public Set<Class<? extends MonetaryAmount>> getAmountTypes();
public Class<? extends MonetaryAmount> queryAmountType(
                                                MonetaryContext requiredContext);
```

Hereby

- `getAmountFactory` should return an instance of `MonetaryAmountFactory` that creates the given `amountType`. Optionally also a required `MonetaryContext` can be passed, this is especially useful for accessing `MonetaryAmountFactory` implementations that are capable of supporting different target `MonetaryContext` instances, e.g. implementations based on `BigDecimal`.
- `getAmountTypes` should return a list of available implementation types for the current runtime context.

- `getDefaultAmountFactory` should return the default `MonetaryAmountFactory` for the current context. Hereby an implementation must never return `null`. If no `MonetaryAmountFactory` instances are registered, a `MonetaryException` should be thrown.
- `queryAmountType` allows to evaluate a `MonetaryAmount` implementation type that best covers the requirements defined by the passed `MonetaryContext.` Implementations should consider the following rules:
    - if the `MonetaryContext` passed is explicitly requiring a concrete implementation type, a factory of this type should be returned given the following conditions are met:
        - the implementation is capable to support the required maximal *scale*.
        - the implementation is capable to support the required maximal *precision*.

      If one of the conditions above fails a `MonetaryException` must be thrown[11].
    - If no concrete type is given (passing the `MonetaryAmount` interface as type), the following must be checked against each registered `MonetaryAmountFactoryProviderSpi` that are **eligible as a possible result type**[12] to be returned from a query:
        - is the `MonetaryAmountFactoryProviderSpi` capable to support the required maximal *scale* (`required scale <= maxScale`).
        - is the `MonetaryAmountFactoryProviderSpi` capable to support the required maximal *precision* (`required precision <= maxPrecision`, or `precision==0/unlimited`).
        - is the `MonetaryAmountFactoryProviderSpi` supporting the required `Flavor (PERFORMANCE, PRECISION` or `UNDEFINED`)
        - Additional attributes to consider may be provided with the `MonetaryContext` required, though this specification does not define any further aspects in detail.
    - if all of the above is `true`, the according result of `MonetaryAmountFactoryProviderSpi.getAmountType()` should be returned.

Registering Roundings

Additional roundings can be added by registering instances of `javax.money.spi.RoundingProviderSpi.` Since a monetary rounding is nothing else than a conversion from an unrounded amount to a rounded amount, ist is modeled as `MonetaryOPerator.` As a consequence the `MonetaryRoundings` singleton bascially is managing an (ordered) collection of `MonetaryOperator` factories defined as follows:

---

[11] This makes sense, since acquiring for a concrete type with invalid capabilities can be seen as a programming error, since the default and maximal capabilities of a concrete type are accessible from the according implementation factory.

[12] This is the case, if the the value from MonetaryAmountFactoryProviderSpi.getInclusionPolicy() does not equal to QueryInclusionPolicy.NEVER, or QueryInclusionPolicy.DIRECT_REF_ONLY.

```
  public interface RoundingProviderSpi {
MonetaryOperator getRounding(CurrencyUnit currency);
MonetaryOperator getRounding(CurrencyUnit currency, long timestamp);
MonetaryOperator getCashRounding(CurrencyUnit currency);
MonetaryOperator getCashRounding(CurrencyUnit currency, long timestamp);
MonetaryOperator getCustomRounding(String customRoundingId);
MonetaryOperator getRounding(MonetaryContext monetaryContext);
Set<String> getCustomRoundingIds();
  }
```

Hereby different types of rounfing are supported:

- based on the target `CurrencyUnit`. By default the digits returned from `CurrencyUnit.getDefaultFractionDigits()` are used, but implementations can provide alternate (e.g. non standard) implementations.
- based on the target `CurrencyUnit`, but explicitly querying for a cache rounding, which may be different to the default rounding. *Example:* in Switzerland default rounding is done for a scale of `2`, whereas when paying in cash, the minor units must be divisible by `5`, since `5` is the smallest coin possible.
- Also it is possible to get a rounding described by a `MonetaryContext`, e.g. you can set a maximal scale of `1` and set the `RoundingMode` (where available on the target platform) as an additional attribute.
- Finally you can also provide customized roundings by name. The names of the defined custom rounding must be returned when `getCustomRoundingIds()` is called.
- Finally it is possible to provide default and cash roundings also for past dates, hereby considering the additional UTC timestamp given.

Backing the MonetaryConversions Singleton

Currency conversion mechanisms are provided by the `MonetaryConversions` singleton accessor. This singleton is backed up by an implementation of `javax.money.spi. MonetaryConversionsSpi.` This singleton in a SE environment may implemented as a *real* singleton, sharing the same state and functionality, whereas in a EE context the implementation will likely behave contextually (providing different runtime context deoending on the current runtime context, e.g. the *ear* or *war* currently active. So implementing this SPI provides full control about the singleton's effective behaviour. As a consequence the methods basically are similar to the ones provided by the singleton class:

```
public interface MonetaryConversionsSpi {
        ExchangeRateProvider getExchangeRateProvider(String... providers);
        CurrencyConversion getConversion(CurrencyUnit termCurrency,
                        ConversionContext conversionContext, String... providers);
        CurrencyConversion getConversion(CurrencyUnit termCurrency,
                        String... providers);
        CurrencyConversion getConversion(String termCurrencyCode,
                        ConversionContext conversionContext, String... providers);
        CurrencyConversion getConversion(String termCurrencyCode,
                        String... providers);
        Collection<String> getProviderNames();
        boolean isProviderAvailable(String provider);
        ProviderContext getProviderContext(String provider);
        List<String> getDefaultProviderChain();
    }
```

Hereby:

- the main artifact defining currency conversion is an `ExchangeRateProvider`. It provides `ExchangeRate` instances defining the factor for converting an base amount to a target (aka terminating) amount.
- A `CurrencyConversion` basically is only an adapter to an `ExchangeRateProvider`, which allows simple use of conversion as a `MonetaryOperator`.
- `getExchangeRateProvider(String…)` allows to pass an ordered array of provider names. The names identify the providers to be used allow to define a composite `ExchangeRateProvider` instance (modeling a provider chain), that is able to answer requests based on multiple rate providers. As an example calling

  ExchangeRateProvider prov = getExchangeRateProvider(“EZB”, “IMF”);

  should return a composite `ExchangeRateProvider` instance, that internally first tries to resolve an `ExchangeRate` requested, using the provider named `“EZB”`. On success the `“EZB”` rate should be returned. If this fails, to whatever reason, the provider with name `“IMF”` should be tried. If no provider is able to return a valid result, a `CorrencyConversionException` must be thrown as defined in the corresponding `ExchangeRateProvider` interface API documentation.
  Additionally if no explicit provider names are passed, the provider names and ordering as defined by `getDefaultProviderChain()` is used.
- `getConversion(String…)` models the same concept as above, but for `CurrencyConversion` instances. Whereas the `ExchangeRateprovider` interface allows to pass a target `ConversionContext` explicitly, when accessing `ExchangeRate` instances, a `ConversionContext` can be passed optionally to further configure the `CurrencyConversion` instance required.

- As for other SPIs in this JSR the loading of different `ExchangeRateProvider` instances should be delegated to the `Bootstrap` implementation.

Adding Currency Conversion Capabilities

Adding additional capabilities for currency conversion equals to implementing and registering classes implementing the `ExchangeRateProvider` interface. The interface itself is part of the API and described in 4.3.3 Exchange Rates and Rate Providers. Basically the implementation of the `MonetaryConversionsSpi` determines how the implementations must be registered. Hereby the registered `ServiceProvider` implementation is responsible for loading and providing the according components. Refer also to 4.5.3 The Bootstrapping Mechanism for more details.

Adapting the Logging Backend

By default the JSR API logic uses `java.util.logging` (JUL) as logging backend. JUL allows to configure additional or customized logging `Handler` instances, so alternate logging backends can be used easily, by registering a forwarding `Handler` implementation for `javax.money` and configuring the `Logger` instance to not delegating to its parent loggers.

The implementation that implements the API's SPI may use a different logging approach.

4.5.2 Formatting SPI

Providing Monetary Amount Format Symbols

The `MonetaryAmountFormatSymbols` class provides factory methods that allow to access instances based on a `Locale`. By registering instances of **`javax.money.spi.MonetaryFormatSymbolsProviderSpi`** additional locales can be supported or adapted. Hereby at least one instance of `MonetaryFormatSymbolsProviderSpi` must be registered, which is defined as follows:

```
public interface MonetaryFormatSymbolsProviderSpi {
  AmountFormatSymbols getAmountFormatSymbols(Locale locale);
  Collection<Locale> getSupportedLocales();
}
```

Hereby

- `getSupportedLocales()` returns the set of locales that are supported by the given implementation.
- `getAmountFormatSymbols(Locale)` returns the corresponding `AmountFormatSymbols` instance. Note that the `AmountFormatSymbols` API class, that is relying on this SPI, will not cache any instances. When caching is useful, it must be implemented by the SPI.

Multiple instances of this interface can be registered hereby forming a *chain of responsibility*,

whereas the components priority define the ordering within the chain. The first component in the chain, that returns a non-`null` result, determines the final result from calling `AmountFormatSymbols.of(Locale)`.
It is also required that on the platforms were `java.text.DecimalFormatSymbols` is available, all locales that are supported by `java.text.DecimalFormatSymbols` must be also available/provided by the SPIs registered.

Providing Amount Styles

The `AmountStyle` class provides factory methods that allow to access instances based on a `Locale`. By registering instances of `javax.money.spi.AmountStyleProviderSpi` additional locales can be supported or adapted. Hereby at least one instance of `AmountStyleProviderSpi` must be registered, which is defined as follows:

```
public interface AmountStyleProviderSpi {
  AmountStyle getAmountStyle(Locale locale);
  Collection<Locale> getSupportedLocales();
}
```

Hereby

- `getSupportedLocales()` returns the set of locales that are supported by the given implementation.
- `getAmountStyle(Locale)` returns the corresponding `AmountStyle` instance. Note that the `AmountStyle` API class, that is relying on this SPI, will not cache any instances. When caching is useful, it must be implemented by the SPI.

Multiple instances of this interface can be registered hereby forming a *chain of responsibility*, whereas the components priority define the ordering within the chain. The first component in the chain, that returns a non-`null` result, determines the final result from calling `AmountStyle.of(Locale)`.
It is required that on the platforms were `java.text.DecimalFormat` is available, all locales that are supported by `java.text.DecimalFormat` must be also available/provided by the SPIs registered.

Prodivding Amount Formats

The `MonetaryFormats` singleton delegates creation of `MonetaryAmountFormat` instances to registered instances of `javax.money.spi.MonetaryAmountFormatProviderSpi`. Hereby at least one instance of `AmountStyleProviderSpi` must be registered as `Bootstrap` component, which is defined as follows:

```
public interface MonetaryAmountFormatProviderSpi {
  MonetaryAmountFormat getAmountFormat(AmountStyle style);
}
```

Hereby

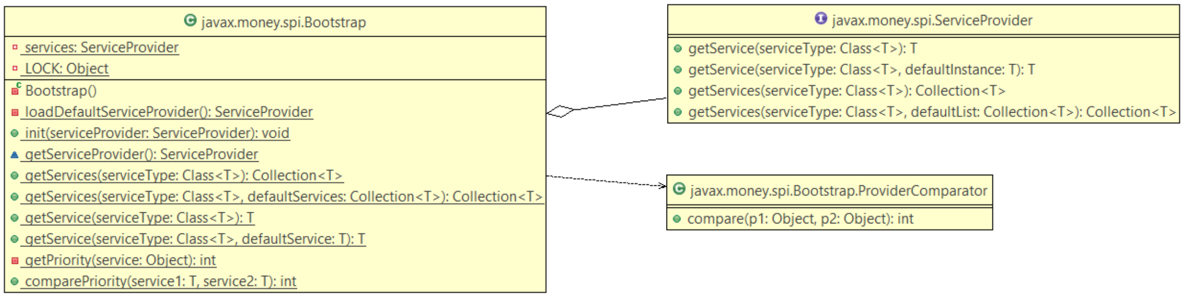
- `getAmountFormat(AmountStyle)` returns the corresponding `MonetaryAmountFormat` instance. Note that the `MonetaryFormats` API class, that is relying on this SPI, will not cache any instances. When caching is useful, it must be implemented by the SPI.

Multiple instances of this interface can be registered hereby forming a *chain of responsibility*, whereas the components priority define the ordering within the chain. The first component in the chain, that returns a non-`null` result, determines the final result from calling `MonetaryFormats`.

4.5.3 The Bootstrapping Mechanism

Overview

Basically the `Bootstrap` singleton class is used by all API components to access instances of the different pluggable components of the Money API. Hereby also the `Bootstrap` class delegates the location and loading of services to an implementation of a `javax.money.spi.ServiceProvider`, which implements the detailed logic how services are located and managed. If no such `ServiceProvider` is configured, a default implementation is used that delegates to `java.util.ServiceLoader`:


javax.money.spi.Bootstrap
services: ServiceProvider
LOCK: Object
Bootstrap()
loadDefaultServiceProvider(): ServiceProvider
init(serviceProvider: ServiceProvider): void
getServiceProvider(): ServiceProvider
getServices(serviceType: Class<T>): Collection<T>
getServices(serviceType: Class<T>, defaultServices: Collection<T>): Collection<T>
getService(serviceType: Class<T>): T
getService(serviceType: Class<T>, defaultService: T): T
getPriority(service: Object): int
comparePriority(service1: T, service2: T): int
javax.money.spi.ServiceProvider
getService(serviceType: Class<T>): T
getService(serviceType: Class<T>, defaultInstance: T): T
getServices(serviceType: Class<T>): Collection<T>
getServices(serviceType: Class<T>, defaultList: Collection<T>): Collection<T>
javax.money.spi.Bootstrap.ProviderComparator
compare(p1: Object, p2: Object): int

Hereby the methods on the `ServiceProvider`, reflect the main functionalities of the overall `Bootstrap` class:

```
public static <T> Collection<T> getServices(Class<T> serviceType){...}
public static <T> Collection<T> getServices(Class<T> serviceType,
                                        Collection<T> defaultServices){...}
public static <T> T getService(Class<T> serviceType) {...}
public static <T> T getService(Class<T> serviceType, T defaultService) {...}
public static final class ProviderComparator implements
                                                Comparator<Object>{...}
```

Summarizing the `Bootstrap` singleton

- Tries to load an instance of `ServiceProvider` using `java.util.ServiceLoader`.
- if no implementation was registered, it falls back to a default `ServiceProvider` implementation, delegating to `java.util.ServiceLoader` and with no specific order.
- if exact one implementation is registered, this implementation is used for loading/accessing the services required by the JSR 354 API. Implementation of `ServiceLoader` hereby can also implement a contextual service registry.
- if multiple implementations are registered, the implementation is not defined, Hereby a warning is logged.

To use an alternate implementation of `javax.money.spi.ServiceProvider` an alternate implementation must be registered using the java.util.`ServiceLoader`. If no instance is registered, an instance of `DefaultServiceProvider` is loaded, that relies on the `java.util.ServiceLoader`.

Implementation Requirements of ServiceProvider

Implementations of `javax.money.spi.ServiceProvider` must implement methods similar as available on the `Bootstrap` singleton class:

```
public interface ServiceProvider {
  <T> Collection<T> getServices(Class<T> serviceType);
  <T> Collection<T> getServices(Class<T> serviceType,
                                Collection<T> defaultList);
}
```

If a required serviceType can not be satisfied,

- the corresponding `defaultInstanceList` should be returned as a schedule (this also includes returning `null`).
- If the required numeric capabilities exceed the maximal supported `MonetaryContext`, a `MonetaryException` must be thrown.

5. Implementation Recommendations

5.1 Overview

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 similar to the Date/Time API introduced with JDK 8 (JSR-310) where appropriate.. E.g. `TemporalAdjuster` and `MonetaryOperator` model a similar concept for temporal 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.

5.2 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`, hereby ensuring that both amounts have the same currency.
- When multiplying or dividing amount, best practice recommends parameters that are simple numeric values.
- Arguments of type `java.lang.Number` should be used with caution, since extracting its numeric value in a feasible way is not trivial.
- 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

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[13]. 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 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.

---

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

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.

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 (e.g. calling `getNumber(Class), getNumberExact(Class)`). 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 `MonetaryOperator`).

6. Examples

The following sections illustrate the API's usage in more detail.

6.1 Working with org.javamoney.moneta.Money

A reference implementation of this JSR has to provide value type classes for monetary amounts, hereby implementing `MonetaryAmount,` and registering at least one implementation class with the `MonetaryAmounts` singleton by implementing and registering a corresponding `MonetayAmountFactory` instance.
As an example the reference implementation provides a class `org.javamoney.moneta.Money`, which is using `java.math.BigDecimal` internally:

```
public final class Money
implements MonetaryAmount, Comparable<MonetaryAmount>, Serializable {
   ...
}
```

The `MonetaryContext` (by default) hereby is defined as follows:

```
maxPrecision = 64; // may be extended arbitrarily
maxScale = -1; // unbounded
numeric class = java.math.BigDecimal
flavor = Flavor.PRECISION
attributes: RoundingMode.HALF_EVEN.
```

Since a corresponding `MonetaryAmountFactory` is registered, a new instance can be created using the typed factory:

```
MonetaryAmountFactory<Money> fact =
                                      MonetaryAmounts.getAmountFactory(Money.class);
Money m = fact.withCurrency(“USD”).with(200.50).create();
```

Also a generic `MonetaryAmount` instance can be accessed using a raw factory:

```
MonetaryAmount amt = MonetaryAmounts.getDefaultAmountFactory()
                                                 .withCurrency(“USD”).with(200.50).create();
```

Still we can evaluate the amount's type effectively:

```
if(Money.class==amt.getClass()){
  Money m = (Money)amt;
}
```

But in fact, we do not need to know the exact implementation in most cases, since we can access a `MonetaryContext`, which provides detailed information, such as maximal precision, maximal scale, the basic implementation flavor and additional attributes.

```
MonetaryContext ctx = m.getMonetaryContext();
if(ctx.getMaxPrecision()==0){
  System.out.println("Unbounded maximal precision.");
}
if(ctx.getMaxScale()>=5){
  System.out.println("Sufficient scale for our use case, go for it.");
}
```

Finally performing arithmetics in both above scenarios works similar as it is when using `java.math.BigDecimal`:

```
MonetaryAmount amt = …;
amt = amt.multiply(2.0).subtract(1.345);
```

Also the sample above illustrates how algorithmic operations can be chained together, similar to builders. As mentioned also external functionalities can be chained, using instances of `MonetaryOperator`:

```
Money amt = Money.of("CHF", 200);
amt = amt.multiply(2.12345).with(MonetaryRoundings.of())
                                          .with(MonetaryFunctions[14].minimal(100)).
                               .multiply(2.12345).with(MonetaryRoundings.of())
                               .with(MonetaryFunctions[15].percent(23));
```

Numeric Precision and Scale

Since the Money class internally uses `java.math.BigDecimal` the numeric capabilities match exact the capabilities of `BigDecimal`. By default instances of `Money` use `MathContext.DECIMAL64`. But on creation of a new `Money` instance the `MonetaryContext` required can also be passed explicitly, e.g.:

```
public static Money of(String currencyCode, Number number,
                                 MonetaryContext context);
```

Extending the API

Now, one last thing to discuss is, how users can add their own functionalities, e.g. by writing their own `MonetaryOperator` functions. Basically there are two disctinct usage scenarios:

---

[14] MonetaryFunctions is not part of the JSR, its just for illustration purposes.
[15] MonetaryFunctions is not part of the JSR, its just for illustration purposes.

- When the basic arithmetics defined on each MonetaryAmount are sufficient, it should be easy to implement such functionality, since its behaving like any other type, e.g.

  ```
  public final class DuplicateOp implements MonetaryOperator{
    public <T extends MonetaryAmount> T apply(T amount){
      return (T) amount.multiply(2);
    }
  }
  ```
- In case where the basic operations are not sufficient anymore, it is still not necessary to cast to any implementation, since
  - the numeric capabilities can be evaluated using the `MonetaryContext`
  - the numeric value can be extracted in a portable way accessing the `NumberValue`.
  - a `MonetaryFactory` can be created to create the result of the same implementation type, without having to cast to this type ever explicitly.

    ```
    public final class ToInvalid implements MonetaryOperator{
      public <T extends MonetaryAmount> T apply(T amount){
        return (T)amount.getFactory().with("XXX").with(0).create();
      }
    }
    ```

6.2 Working with org.javamoney.moneta.FastMoney

This class implements a `MonetaryAmount` using `long` as numeric representation, whereas the full amount is interpreted as minor units, with a denumerator of `100000`. As an example `CHF 2.5` is internally stored as `CHF 250000`. Addition and subtraction of values is trivial, whereas division and multiplication get complex with non integral values. Compared to `Money` the possible amounts to be modeled are limited to an overall precision of 18 and a fixed scale of 5 digits.

Beside that the overall handling of `FastMoney` is similar to `Money`. So we could rewrite the former example by just replacing `Money` with `FastMoney`:

```
FastMoney amt = FastMoney.of("CHF", 200);
amt = amt.multiply(2.12345).with(MonetaryRoundings.of())
                           .with(MonetaryFunctions.min(100))
                           .multiply(2.12345)
                           .with(MonetaryRoundings.of())
                     .with(MonetaryFunctions.percent(23));
```

Of course, given all that the `MonetaryContext` is different than for `Money`:

```
maxPrecision = 18; // hard limit
maxScale = 5; // fixed scale
numeric class = Long
flavor = Flavor.PERFORMANT
attributes: RoundingMode.HALF_EVEN
```

6.3 Calculating a Total

A total of amounts can be calculated in multiple ways, one way is simply to chain the amounts with `add()`:

```
MonetaryAmount[] params = new MonetaryAmount[]{
                Money.of(“CHF”, 100), Money.of(“CHF”, 10.20),
                                Money.of(“CHF”, 1.15),};
MonetaryAmount total = params[0];
for(int i=1; i<params.length;i++){
  total = total.add(params[i]);
}
```

As an alternate it is also possible to define a `MonetaryOperator`, which can be passed to all amounts:

```
public class Total implements MonetaryOperator{
  private MonetaryAmount total;

  public <T extends MonetaryAmount<T>> T apply(T amount){
    if(total==null){
      total = amount;
    }
    else{
          total = total.add(amount);
    }
    // ensure to return correct type, since different implementations
    // can be passed as amount parameter
    return amount.getFactory().with(total).create();
  }

  public MonetaryAmount getTotal(){
    return total;
  }

  public <T extends MonetaryAmount> T getTotal(Class<T> amountType){
    return MonetaryAmounts.getAmountFactory(amountType).with(total).create();
  }
}
```

Note, we are well aware of the fact that this implementation still has some severe drawbacks, but we decided for simplicity to not add the following features:

- the implementation can only handle one currency, a better implementation could also be multi-currency capable.
- The implementation above is not thread-safe.

Now with the `MonetaryOperator` totalizing looks as follows:

```
Total total = new Total();
for(int i=1; i<params.length;i++){
  total.with(params[i]);
}
System.out.println(“TOTAL: “ + total.getTotal());
```

A similar approach can also be used for other multi value calculations as used in statistics, e.g. average, median etc.

6.4 Calculating a Present Value

The present value (abbreviated *PV*) shows how financial formulas can be implemented based on the JSR 354 API. A *PV* models the current value of a financial in- or outflow in the future, weighted with a calculatory interest rate. The *PV* is defined as follows:

$$\frac{R_t}{(1+i)^t},$$

Hereby

t – the time of the cash flow (in periods)

i – the discount rate (the rate of return that could be earned on an investment in the financial markets with similar risk.); the opportunity cost of capital

R_t – the net cash flow i.e. cash inflow – cash outflow, at time *t* . For educational purposes,

The same financial function now can be implemented for example as folllows:

```
public <T extends MonetaryAmount> T presentValue(
                                          T amt, BigDecimal rate, int periods){
   BigDecimal divisor = BigDecimal.ONE.add(rate).pow(periods);
   // cast should be safe for implementations that adhere to this spec
   return (T)amt.divide(divisor);
}
```

This algorithm can be implemented as `MonetaryOperator`:

```
public final class PresentValue implements MonetaryOperator{
    private BigDecimal rate;
    private int periods;
    private BigDecimal divisor;

  public PresentValue(BigDecimal rate, int periods){
    Objects.requireNotNull(rate);
    this.rate = rate;
    this.periods = periods;
    this.divisor = BigDecimal.ONE.add(periods).power(periods);
  }
  public int getPeriods(){
    return periods;
  }
  public BigDecimal getRate(){
   return rate;
  }
  public <T extends MonetaryAmount> T apply(T amount){
    // cast should be safe for implementations that adhere to this spec
    return (T)amount.divide(divisor);
  }
  public String toString(){ …}
}
```

For simplicity we did not add additional feature such as caching of `PresentValue` instances using a `static` factory method, or precalculation of divisor matrices. Now given the `MonetaryOperator` a present value can be calculated as follows:

```
Money m = Money.of(“CHF”, 1000);
// present value for an amount of 100, available in two periods,
// with a rate of 5%.
Money pv = m.with(new PresentValue(new BigDecimal(“0.05”), 2));
```

6.5 Performing Currency Conversion

Currency Conversion also is a special case of a `MonetaryOperator` since it creates a new amount based on another amount. Hereby by the conversion the resulting amount will typically have a different currency and a different numeric amount:

```
MonetaryAmount inCHF =...;
CurrencyConversion conv = MonetaryConversions.getConversion(“EUR”);
MonetaryAmount inEUR = inCHF.with(conv);
```

Also we can define the providers to be used for currency conversion by passing the provider names explicitly:

```
CurrencyConversion conv = MonetaryConversions.getConversion(“EUR”, “EZB”, “IMF”);
```

To cover also more complex usage scenarios we can also pass a `ConversionContext` with additional parameters for conversion, e.g.:

```
MonetaryAmount inCHF =...;
ConversionContext ctx = new ConversionContext().Builder()
              .set(MonetaryAmount.class, MonetaryAmount.class, inCHF)
                  .setTimesampt(ts)
              .setRateType(RateType.HISTORIC)
                  .set(StockExchange.NYSE) // custom type
                  .set(“contractId”, “AA-1234.2”)
                  .create();
CurrencyConversion conv = MonetaryConversions.getConversion(“EUR”,
                                    ctx,
                                    “CS”, “EZB”, “IMF”);
```

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/ and 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 updated PDR is **0.8**

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)