JSR 354 (Money & Currency) - Specification

Anatole Tresch, Credit Suisse

Version 1.0
April 2015

Table of Contents

Version Information: 1
1. Introduction 2
1.1. Expert group 2
1.2. Specification goals 3
1.3. Scope 3
1.4. Required Java version 4
1.5. How this document is organized 4
2. Use Cases 5
2.1. Scenario eCommerce (Online-Shop) 5
2.2. Scenario Trading Site 5
2.3. Scenario Virtual Worlds and Game Portals 6
2.4. Scenario Social Markets 6
2.5. Scenario Banking & Financial Applications 6
2.6. Scenario Insurance & Pension 7
3. Requirements 8
3.1. Core Requirements 8
3.2. Formatting Requirements 8
3.3. Java EE Support 9
3.4. Non Functional Requirements 9
4. Specification 10
4.1. Package and Project Structure 10
4.2. Money and Currency Core API 11
4.3. Currency Conversion 37
4.4. Money and Currency Formatting API 43
4.5. Money and Currency SPI 50
5. Meta-Data Contexts and Query Models 66
5.1. Overview 66
5.2. AbstractContext 67
5.3. Abstract Class AbstractContextBuilder 68
5.4. Abstract Class AbstractQuery 69
5.5. Abstract Class AbstractQueryBuilder 70
6. Implementation Recommendations 70
6.1. Overview 70
6.2. Monetary Arithmetic 71
6.3. Numeric Precision 71
7. Examples 73
7.1. Working with org.javamoney.moneta.Money 73
7.2. Working with org.javamoney.moneta.FastMoney 76
7.3. Calculating a Total 77
7.4. Calculating a Present Value 79
7.5. Performing Currency Conversion 80
APPENDIX 81

Bibliography 81
Links 82
Related Initiatives 83

Version Information:

```
Specification: JSR-354 Money and Currency API ("Specification")
Version:       1.0
Status:        Final
Release:       March 2015
Copyright:     2012-2015
               Credit Suisse AG
               P.O.Box
               8070 Zurich
               Switzerland
               All rights reserved.
```

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 expandable. The API will provide support for standard [ISO-4217] and custom currencies, and a model for monetary amounts and rounding. 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. 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
- Manuela Grindei
- 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 areas, e.g.

- eCommerce
- Banking
- Finance & 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 for extension in several ways. So it should be flexible enough that corresponding implementations can be used transparently to accommodate such applications.

As many applications in the financial area will quite probably use Java 7 for several years this JSR supports both platforms, Java 7 as well as Java 8.

1.3. Scope

JSR 354 targets a standalone scope. Nevertheless we considered a later integration into the JDK, so its design and scope must follow JDK patterns. Additionally the work on the JSR has shown, that it is possible to define a flexible and comprehensive API that is not only be compatible with Java 7 and Java 8, but also with Java Embedded. Basically this affects usage of `java.math` and `java.text`. Nevertheless the reference implementations are free to use existing functionality and the JSR also includes requirements (also checkable by the TCK) to ensure a minimal set of functionality on Java SE 7 and SE 8. 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, enabling feature innovation elsewhere. The corresponding

libraries were avaailable under [JavaMoney] as an Apache 2 licensed open source project. Though these libraries were removed from the JSR, their development ensured that scope was fully evaluated and that the parts best suited to standardization were identified.

1.4. Required Java version

The specification supports Java SE platforms version 7 and 8 (in fact the Java 7 based API is even compatible with Java 6). The Java 8 based API is backward compatible with the Java 7 version since its only adding additional default method implementations to the API. Implementations hereby may target any suitable Java SE version, or given an increasing SE/ME correlation also future ME versions. The JSR consequently provides two reference implementations, one based on Java 7, one based on Java 8 language features. Since the Java 8 version is backward compatible the same TCK can be used for testing both reference implementations. This allows to design an API, which will easily fit into Java 8 and beyond, but still supporting older releases.

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 by performing *currency conversion*.

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 flexible 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 payments. This includes serialization of the amount and/or special formatting of the amount into the format required by the remote server.

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, regional, and define their time validity range. Currencies available may depend further from contract, current tenant or other aspects. The same may also be `true` for rounding algorithms. Access to these features must be very flexible and capable of behaving different depending on the current runtime context.

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 extensible.

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 depending on legal changes. Such systems are built of several isolated building blocks of different granularity size and complexity, starting from simple sum 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, currencies, conversions, formats, statistics, filtering etc.
5. Meta-data must be accessible using a generic API, so custom requirements can be implemented and context information not explicitly defined by this JSR is accessible using a unified access mechanism.
6. 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.
7. The JSR must provide a minimal set of roundings. This should include basic roundings for ISO currencies, or roundings defined by a monetary context.
8. 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, e.g. *Indian Rupees* can be formatted correctly. [`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`]

6. supports rounding of amounts for display and reverse rounding during parsing.

3.3. Java EE Support

1. This JSR must avoid restrictions that prevents its use in different runtime environments, such as Java EE. Refer also to the section [Boostrap] for more details on possible EE/CDI integration.

3.4. Non Functional Requirements

1. Since this JSR may be a candidate to be included into the JDK later, any possible extension 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, 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. Queries and contexts may require adding additional time related data, such as POSIX timestamps based on millisecond resolution as returned by `System.currentTimeMillis()` or other time types based on new Java 8 date/time API. These aspects are not explicitly modelled, since they depend on the capabilities of the corresponding providers and the Meta-Data Contexts and Query Models capabilities provide good flexibility to implement these things effectively.

8. This JSR will probably also be used also for (business) critical software like real time trading and similar systems. These systems and use cases require very specific parameters, which are impossible to model by this JSR and may vary for different use cases, provider and/or companies. As a solution attributable contexts and queries can be passed optionally that can contain arbitrary parameters needed.

9. Though performance aspects can not directly targeted by this JSR, it is important that the JSR considers performance aspects where possible, so that provided implementations are able to optimize 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 4 packages:

`javax.money`
: contains the main artifacts, such as `CurrencyUnit`, `MonetaryAmount`, `MonetaryContext`, `MonetaryOperator`, `MonetaryQuery`, `MonetaryRounding`, and the singleton accessor `Monetary`. It is discussed in section Money and Currency Core API. The meta-data context and query features are discussed in Meta-Data Contexts and Query Models.

`javax.money.conversion`
: contains the conversion artifacts `ExchangeRate`, `ExchangeRateProvider`, `CurrencyConversion` and the according `MonetaryConversions` accessor singleton. It is discussed in section Currency Conversion.

`javax.money.format`
: contains the formatting artifacts `MonetaryAmountFormat`, `AmountFormatContext` and the according `MonetaryFormats` accessor singleton. It is discussed in section Money and Currency Formatting API.

`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. It is discussed in section Money and Currency SPI.

4.1.2. Module/Repository Overview

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

jsr354-api

contains the JSR 354 API based on Java 8 as described by this specification.

jsr354-api-bp

contains the JSR 354 API based on Java 7 as described by this specification.

jsr354-ri

contains the *'moneta'* reference implementation based on Java 8 language features.

jsr354-ri-bp

contains the *'moneta'* reference implementation based on Java 7 language features.

jsr354-tck

contains the technical compatibility kit (TCK). The TCK is built using Java 7 but can be seemlessly be used to test implementations based on Java 8.

javamoney-parent

is a root "POM" project for all modules under `org.javamoney`. This includes the RI/TCK projects, but not jsr354-api (which is standalone).

javamoney-library

contains a financial library (JavaMoney) adding comprehensive support for several extended functionality, 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.

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:

- `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.

- The abstract type `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 functionality 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 functional interfaces providing extension points for monetary logic. They allow to implement external functionality, either adding operations returning an amount (`MonetaryOperator`), or returning any arbitrary other value (`MonetaryQuery`).

- the `MonetaryAmountFactory` 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 the *default* and the *maximal* `MonetaryContext`

- `MonetaryContext` models the meta-data of `MonetaryAmount` instances, including a representation ot the numeric capabilities of an instance as an immutable and platform independent type.

- `CurrencyContext` models the meta-data of a `CurrencyUnit` instance as an immutable and platform independent type.

- `RoundingContext` models the meta-data of a `MonetaryRounding` instance as an immutable and platform independent type.

- `MonetaryAmountFactoryQuery` models a query for evaluating instances of `MonetatyAmountFactory` given concrete requirements/required capabilities.

- `CurrencyQuery` models a query for evaluating instances of `CurrencyUnit` given concrete requirements/required capabilities.

- `RoundingQuery` models a query for evaluating instances of `MonetaryRounding` given concrete requirements/required capabilities.

- `MonetaryContextBuilder`, `CurrencyContextBuilder`, `RoundingContextBuilder`, `MonetaryAmountFactoryQueryBuilder`, `CurrencyQueryBuilder`, `RoundingQueryBuilder` all model the builders necessary for creating instances of the several context and query classes.

- `MonetaryException` is the base exception class for the money API, it extends `java.lang.RuntimeException`.

Finally the core module also contains base classes used for metadata and query modeling:

- `AbstractContext` models the abstract basic value type for additional context data, used in several parts of this JSR. It provides the basic logic for implementing an immutable context internally using a `Map<String,Object>` store.

- `AbstractQuery` models the abstract query value type for querying monetary data from the different singleton accessors provided. `AbstractQuery` extends `AbstractContext`.

- `AbstractContextBuilder, AbstractQueryBuilder` model the abstract basic builder types for builders that create instances of `AbstractContext, AbstractQuery`, used in several parts of this JSR.

Refer to section Meta-Data Contexts and Query Models for more details.

NOTE

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 strong coupling to a numeric representation. Unfortunately, as seen in the use cases and requirements sections, performance and precision are conflicting 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.
- 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.

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 this API provides according accessor classes. Summarizing the following singletons are available as part of the JSR's core module:

- `Monetary`
  - provides access and query functionality to `CurrencyUnit` instances.
  - provides access and query functionality to factories for creating `MonetaryAmount` instances.
  - provides provides access and query functionality to `MonetaryRounding` instances.

Additionally the conversion and formatting module also provide singletons:

- `MonetaryConversions` for accessing `CurrencyConversion` and `ExchangeRateProvider` instances.
- `MonetaryFormats` for accessing `MonetaryAmountFormat` instances.

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 abstraction is sufficient or what kind of additions are necessary.

Fortunately a minimal interface `CurrencyUnit` could be extracted, that models almost a subset of the existing functionality on `java.util.Currency`, so the existing class could easily implement the new interface. Compared to `java.util.Currency` the new currency interface does not provide methods for localizing a currency instance such as `getDisplayName(Locale)`, `getSymbol(Locale)`. This allows to separate the different concerns of data modelling and formatting. Additionally the JSR's currency interface provides access to a `CurrencyContext` meta-data class, which is capable of providing arbitrary meta-data on the current instance. This meta-data container can be used to store additional data, such as the validity time range, corresponding regions or territories or provider data.

So the `CurrencyUnit` interface for currencies is modelled only with 4 methods as follows:

Interface CurrencyUnit

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

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`. In case of ISO currencies the code must match the value of the corresponding ISO code. For alternate currency scheme, if useful numeric code is defined for the currency, this code should be reflected accordingly. A numeric code is defined to be unique within an underlying currency scheme, though the JSR dies only support accessing currencies using their (unique) currency code.

- the default fraction digits define the typical scale of values with a given currency.

- the `CurrencyContext` models additional metadata of a currency unit (refer to section [metadata modelling] for more details on contexts). It basically allows to evaluate the data provider of a currency unit, but can also contain additional data as useful, determined by the implementation that provided the currency instance. This context allows to support also more complex use cases for extended currency meta-data such as:

- validity range, e.g. modelled as from/to `LocalDate`
- regional validity constraints
- provider validity constraints, e.g. the target stock exchange
- internal provider reference ids
- conversion service URLs
- related customer or contract information
- etc.

Furthermore implementations of `CurrencyUnit`

1. must implement `equals/hashCode`, considering the concrete implementation type, currency code (which is defined to be unique) and the `CurrencyContext`.
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 amounts 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 a *one-fits-it-all* implementation.

One key aspect is that a monetary amount must always be related to a currency. Mixing of currencies makes typically no sense for arithmetic operations on amount or, even worse, results in useless and incorrect results. Properties and operations of monetary amounts are modeled by an interface, called `javax.money.MonetaryAmount`. This enables effective data and functional interoperability. 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 amount meta-data, such as *numeric capabilities* of the concrete type (`MonetaryContext`).

- *Prototyping support* for creating new `MonetaryAmount` instances based on the same implementation, modeled by a `MonetaryAmountFactory`, which is accessible from each instance calling `MonetaryAmount.getFactory()`.
- *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 extension points* modeled as `MonetaryOperator` (returning amount instances of the same implementation type) and `MonetaryQuery` (returning any result type).

The interface is defined as follows:

Interface MonetaryAmount

```
public interface MonetaryAmount{
  CurrencyUnit getCurrency();
  NumberValue getNumber();
  MonetaryContext getContext();

  // Create a 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();
  MonetaryAmount plus();
  MonetaryAmount stripTrailingZeros();
```

```
}
```

Hereby

- `getCurrency()` returns 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.
- `getContext()` allows to access the monetary meta-data context of an amount, which may include data similar to `java.math.MathContext` but also other arbitrary attributes determined by the implementation (refer to section [metadata modelling] for more details on contexts).
- Instances of `MonetaryOperator` and `MonetaryQuery<R>` can be applied on a `MonetaryAmount` instance by passing them to the `with(MonetaryOperator)` or `query(MonetaryQuery)` method. Whereas an operator calculates a new amount based on a amount (an instance of an unary function), a query can return arbitrary result types.
- `isGreaterThan(MonetaryAmount)`, `isLessThan(MonetaryAmount)`, `isGreaterThanOrEqualTo(MonetaryAmount)` 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* [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 target date and time, the currencies involved, the provider, the amount ...]) with the amount, on which the operation is executed. The arithmetic methods should basically behave similar to `java.math.BigDecimal`, always returning amounts with the same `CurrencyUnit`.
- 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.

When dealing with `double` values additional aspects must be considered:

- multiplying/adding/subtracting with POSITIVE_INFINITY should throw ArithmeticException because it overflows
- multiplying/adding/subtracting with NEGATIVE_INFINITY should throw ArithmeticException because it overflows

- multiplying/adding/subtracting with NaN should throw ArithmeticException because the result is NaN
- dividing by POSITIVE_INFINITY returns 0
- dividing by NEGATIVE_INFINITY returns 0
- dividing/multiplying/adding/subtracting by NaN should throw ArithmeticException because the result is NaN

Finally implementations of `MonetaryAmount<T>`

1. must implement `equals/hashCode`, hereby it is recommended considering
   a. its implementation type
   b. its `CurrencyUnit`
   c. its numeric value, with any *non significant trailing zeros truncated*.
   d. its meta-data context, modeled as `MonetaryContext`
2. must be thread safe and immutable.
3. must be comparable.
4. should be serializable.
5. should be final.
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 `isZero()==true`, and additionally should be equal to `0`.
8. This specification does no further constrain the constructor or factory methods to be implemented, or the method signatures to be used.

NOTE 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(MonetaryAmount)` must return `true`, if they have equal currencies and equal numeric values, hereby ignoring non-significant trailing zeros and different monetary contexts.

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 (date & time).

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 externalized. This can be achieved by calling `NumberValue getNumber();` on `MonetaryAmount`.

Nevertheless simply returning `java.lang.Number`, is also not desired, since conversion to known types may imply rounding errors or truncation. So `NumberValue` extends `java.lang.Number`, which is the basic type used in the JDK, but `NumberValue` adds methods that help users to better identify the risks of different externalization operations and provide functionality for effective access to the numeric data:

Abstract Class NumberValue

```
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();
  public abstract long getAmountFractionNumerator();
  public abstract long getAmountFractionDenominator();
}
```

Hereby

1. `getNumberType()` provides information about the numeric representation used internally. It explicitly does not constrain the type returned to be a subtype of `java.lang.Number` to allow alternate implementations to be used.

2. `intValueExact()`, `longValueExact()`, `doubleValueExact()` extend the methods defined in `java.lang.Number`, with their exact 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. So in the following cases an exception must be thrown:

   a. the current amount's value exceeds the overall maximal value of the target type (overflow)

   b. the current amount's fraction value cannot be mapped into the target type (underflow)

3. the methods `getAmountFractionNumerator()` and `getAmountFractionDenominator` allow to extract the fraction part of an amount in a flexible way.

4. `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:

a. `Integer`

b. `Long`

c. `Float`

d. `Double`

e. If available in the current runtime environment also: `BigDecimal, BigInteger`

5. `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)`.

6. `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 functionality, e.g. more complex financial operations, being applied on amounts. This is modelled by

- `javax.money.MonetaryOperator`, which models a function `f(M1)  M2`, that converts an amount to another amount, and

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

Note that interfaces in Java 7 and Java 8 have similar signatures, whereas Java 8 additionally is annotated with the `@FunctionalInterface` annotation.

Monetary Operators

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

Interface MonetaryOperator (Java 7)

```
// Java 8
@FunctionalInterface
public interface MonetaryOperator{
   MonetaryAmount apply(MonetaryAmount amount);
}

// Java 7
public interface MonetaryOperator{
   MonetaryAmount apply(MonetaryAmount amount);
}
```

Monetary 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, see section [MonetaryRounding]
- currency conversion, see section Currency Conversion
- financial calculations and formulas, see section [JavaMoney]
- other statistical use cases, e.g. by passing an operator to each element in a `Collection` of `MonetaryAmount` or using the JDK 8 *Streaming API*.
- other monetary conversions

Implementations of `MonetaryOperator` are highly recommended to be

1. immutable and
2. thread-safe

A `MonetaryOperator` is typically invoked on the instance of an `MonetaryAmount`, passing the operator as a parameter:

Example Usage of MonetaryOperator

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

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

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

assertTrue(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 also always apply:

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

NOTE The operator interface is equivalent to the `UnaryOperator` interface, 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:

Interface MonetaryQuery

```
// Java 8
@FunctionalInterface
public interface MonetaryQuery<R> {
  R queryFrom(MonetaryAmount<?> amount);
}

// Java 7
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:

- Amount 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, or other types
- accessing the amounts currency or properties in a functional way, additional to the supplier interfaces already in place

Implementations of `MonetaryQuery<R>` should be

1. immutable and
2. thread-safe

A `MonetaryQuery` is typically invoked on an instance of `MonetaryAmount`, passing the query as a parameter:

Usage Example for s MonetaryQuery

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

NOTE The query interface is equivalent to the `Function` interface, which is a functional interface suitable for use with Lambda expressions.

4.2.5. The Monetary Context

The monetary context (`javax.money.MonetaryContext`) models the monetary amount's meta-data, including the numeric capabilities (implementation) in a platform independent way (refer also to section Meta-Data Contexts and Query Models for more details on contexts). Though it has some similarities with `java.math.MathContext` for `BigDecimal` it is far more flexible, since different implementations may add several attributes that be relevant. A `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.
- Instances of `MonetaryAmountFactory<T>` supports creation of `MonetaryAmount` instances, hereby setting explicitly the `MonetaryContext` required. In such a case the factory uses this monetary context to determine the amount created. `MonetaryAmountFactory.getDefaultMonetaryContext()` returns the default context used. Similarly the maximal supported capabilities of a `MonetaryAmountFactory<T>` can be determined by calling `MonetaryAmountFactory.getMaximalMonetaryContext()`. Hereby the *maximal capabilities* are determined:
  - by the *maximal scale*, that an implementation type supports, without having to truncate any parts of the numeric fraction

- by the *maximal precision*, that an implementation type supports, without having to truncate the whole or the fractional part of an amount.
- basically additional aspects can be modelled as useful, but are not defined by this specification, e.g. the `MonetaryContext` can also contain an amount flavor or some other implementation priority, that can be used for determining, which amount type is best suited for some use case. For additional aspects to be considered a corresponding instance of `[MonetaryAmountsSingletonQuerySpi]` must be implemented and registered, with the according component registration mechanism actually loaded by the JSR's `The Bootstrapping Mechanism` component.

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

Class MonetaryContext

```
public final class MonetaryContext extends AbstractContext
implements Serializable{
  ...
  public int getPrecision();
  public int getMaxScale();
  public boolean isFixedScale();
  public Class<? extends MonetaryAmount> getAmountType();
  public MonetaryContextBuilder toBuilder();

}
```

Hereby

- `getPrecision(), getMaxScale(), isFixedScale()` define common numeric capabilities.
- `getAmountType()` gives access to the amount's implementation type used.
- new instances are built using an instance of `MonetaryContextBuilder`, which also can be accessed from each `MonetaryContext` instance.
- the inherited `AbstractContext` provides access to additional non standard context properties, see AbstractContext.

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:

Class MonetaryContext

```
enum MyFlavor{ // only an example, not part of the API
  SLOW. FAST
}

MonetaryContext ctx = MonetaryContextBuilder.of()
 .setMaxScale(2)
 .setFixedScale(true)
 .setPrecision(12)
 .set(RoundingMode.UP)
 .set(MyFlavor.FAST)
 .build();
```

For further details on contexts, related builders and meta-data modeling, refer to section Meta-Data Contexts and Query Models.

4.2.6. Creating Monetary Amount Instances

Basically new instances of `MonetaryAmount` can be created in different ways. One way [Types may also be instantiated directly depending on the implementation.] will be by using factories, modeled by the interface `javax.money.MonetaryAmountFactory<T>`. Instances can be obtained in different ways

- calling `getFactory()` on an any instance of `MonetaryAmount`, returns an instance that is pre-initialized with the current amount's values, allowing for easily creation of similar amount instances, with some or multiple properties changed. This is known as the 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 `Monetary` singleton also provides access to `MonetaryAmountFactory` instances, hereby also allowing to bind to a specific implementation type or query for matching `MonetaryAmountFactory` instances:

Usage Example for creating an Amount, using an explicit type

```
MonetaryAmountFactory<MyMoney> fact = Monetary.getAmountFactory(MyMoney.class);
MyMoney money = fact.setCurrency("USD").setNumber(10.50).create();
```

More complex evaluations of `MonetaryAmountFactory` instances can be performed as only constraint by the registered SPIs (see Money and Currency SPI) using `MonetaryAmountFactoryQuery` and its related Builder class:

Usage Example for querying for a `MonetaryAmountFactory`

```
MonetaryAmountFactory<?> fact = Monetary.getAmountFactory(
   MonetaryAmountFactoryQueryBuilder.of()
     .setMaxScale(2)
     .setPrecision(10)
     .build());
MonetaryAmount money = fact.setCurrency("USD").setNumber(10.50).create();
```

As illustrated above the signature of `MonetaryAmountFactory` is modelled as a builder also supporting a fluent programming style:

Interface MonetaryAmountFactory

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

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

  T create();
}
```

Hereby

- create returns a new instance of `T` based on the current properties of the factory instance.
- 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 significant truncation, it must throw an `ArithmeticException`.

4.2.7. Accessing Currencies, Amounts and Roundings

All JSR's main artifacts are accessible by corresponding singleton accessor classes. Hereby the exact behaviour of the singletons are delegated to corresponding SPI's. This allows to implement runtime dependent behaviour that can be different for different runtime environments, e.g. use CDI based contextual implementations, instead of the default SE ServiceLoader based component lifecycle. Refer to section Money and Currency SPI for more details.

Accessing Currencies

The `javax.money.Monetary` singleton class implements an accessor for `CurrencyUnit` instances. Each implementation must also provide/include a provider that uses `java.util.Currency` as a backend. But this JSR in addition allows registration of additional currencies by implementing instances of `CurrencyProviderSpi` (refer to section [CurrencyProviderSpi]):

Monetary Singleton (Currency related methods)

```
public final class Monetary{
  private Monetary(){}

  public static CurrencyUnit getCurrency(String currencyCode, String... providers);
  public static CurrencyUnit getCurrency(Locale locale, String... providers);
  public static Set<CurrencyUnit> getCurrencies(Locale locale, String... providers);
  public static boolean isCurrencyAvailable(String currencyCode, String... providers);
  public static boolean isCurrencyAvailable(Locale locale, String... providers);
  public static boolean isCurrencyAvailable(CurrencyQuery query);
  public static Set<CurrencyUnit> getCurrencies(String... providers);
  public static Collection<CurrencyUnit> getCurrencies(CurrencyQuery query);
  public static Set<String> getCurrencyProviderNames();
  public static List<String> getDefaultCurrencyProviderChain();
  [...]
}
```

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`. Additionally (compared to `java.util.Currency`) it is also possible to access multiple currencies per `Locale`.
- additional `CurrencyUnit` can be added by registering instances of the `CurrencyProviderSpi` as explained within the section Money and Currency SPI 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, before accessing it.
- `getCurrencies(String  )` allows to access all currencies currently known by this singleton (which delegates to the known `[MonetaryCurrencyProviderSpi]` instances).
- All access methods above also allow to pass an ellipse operator of provider names. If not set explicitly the default providers and ordering as defined by `getDefaultProviderNames()` must be used. Hereby
  - if only a single valued result is returned (`CurrencyUnit`, boolean), the provider chain is evaluated until the first provider returns `true` or a non-null `CurrencyUnit` instance.
  - in case of multi valued results all values returned by the providers are added to the result collection (`List, Set, Collection`).
- All available provider names are accessible from `getProviderNames()`. Hereby each provider name maps to exact one instance of `CurrencyProviderSpi`. Refer to section [CurrencyProviderSpi] for more details.

- The default provider names and ordering are accessible from `getDefaultProviderNames()`.
- Finally the method `getCurrencies(CurrencyQuery)` gives you maximal flexibility for accessing currencies, e.g.

Example for querying currencies

```
// Note: Enum Region only serves as an example and not part of the API
Collection<CurrencyUnit> currencies = Monetary.getCurrencies(
  CurrencyQueryBuilder.of()
    .setProvider("A", "B")
    .set(Region.EMEA)
    .set("contractNr", 12345)
    .build()
);
```

The query interface also is flexible enough to support access to historic currencies. As an example, if an according provider would be implemented and registered one could perform the following query:

Example for querying historic currencies

```
// Note: This is just an example: no historic provider is part of the API or RI currently
Collection<CurrencyUnit> currencies = Monetary.getCurrencies(
  CurrencyQueryBuilder.of()
    .set(Locale.GERMANY)
    .setTimestamp(LocalDate.of(1930,1,1))
    .build()
);
```

The default provider chain can be configured within the `javamoney.properties` configuration file, located in the classpath as follows:

javamoney.properties Configuration of default currencies provider chain

```
# Defaults for java money
...
javax.money.defaults.Monetary.currencyProviderChain=provider1,provider2,provider3
```

Accessing Monetary Amount Factories

The `javax.money.Monetary` singleton class implements also 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 `[MonetaryAmountsSingletonSpi]` and `[MonetaryAmountsSingletonQuerySpi]`, that can be registered using the JSR's `The Bootstrapping Mechanism` mechanism.

Monetary Singleton (Amount Related Methods)

```
public final class Monetary{
  private Monetary(){}

  public static <T extends MonetaryAmount> MonetaryAmountFactory<T> getAmountFactory
(Class<T> amountType);
  public static MonetaryAmountFactory<?> getDefaultAmountFactory();
  public static Collection<MonetaryAmountFactory<?>> getAmountFactories(){
  public static Set<Class<? extends MonetaryAmount>> getAmountTypes();
  public static Class<? extends MonetaryAmount> getDefaultAmountType();
  public static MonetaryAmountFactory getAmountFactory(MonetaryAmountFactoryQuery query);
  public static Collection<MonetaryAmountFactory> getAmountFactories
(MonetaryAmountFactoryQuery query);
  public static boolean isAvailable(MonetaryAmountFactoryQuery query);
  [...]
}
```

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:

javamoney.properties Configuration File

```
# Defaults for java money

javax.money.defaults.amount.class=my.fully.qualified.MonetaryAmountType
```

- `getAmountTypes()` returns all amount implementation classes currently available.
- `getAmountFactories()` returns all amount factories currently available. Compared to calling `getAmountTypes()` the factories provide also minimal and maximal monetary amount meta-data, which also includes corresponding attributes describing the numeric capabilities supported.
- `getAmountFactory(MonetaryAmountFactoryQuery query)` allow to access a `MonetaryAmountFactory` that best covers the given `MonetaryAmountFactoryQuery`.
- Finally `getAmountFactories(MonetaryAmountFactoryQuery query)` allow to query multiple instances of `MonetaryAmountFactory` using a `MonetaryAmountFactoryQuery`.

IMPORTANT	Implementations of this JSR must at least provide one implementation of `MonetaryAmountFactoryProviderSpi` with a query policy equal to `MonetaryAmountFactoryProviderSpi.QueryInclusionPolicy.ALWAYS`*. Refer to section [MonetaryAmountFactoryProviderSpi] for more details.

Accessing Roundings

Rounding is modeled by implementations of `MonetaryRounding`, which extends `MonetaryOperator` but also provides rounding meta-data, modeled as `RoundingContext`. This is very useful since in the financial area 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.Monetary` singleton based on:

1. a target `CurrencyUnit`. By default the rounding is based on the currency's default fraction units (see `CurrencyUnit.getDefaultFractionUnits()`).
2. an explicit (unique) *rounding id* that must be known (and documented) by a `RoundingProviderSpi` implementation.
3. each implementation should at least enable accessing mathematical rounding, supporting
   a. the maximal *precision* (`int`)
   b. the target *scale* (`int`)
   c. the `java.math.RoundingMode`, providing a definition of the required mathematical rounding. If not defined `HALF_EVEN` rounding mode should be used as a default.
4. Using a `RoundingContext`, which can be configured with any kind of attributes. Also other use cases can be supported, e.g. it could be possible to access special cash rounding, which may be different than the default currency rounding (e.g. for `CHF/Swiss Francs`).

The `Monetary` singleton provides access to `MonetaryRounding` instances as follows:

Monetary Singleton (Rounding related methods)

```
public final class Monetary{
  private Monetary(){}

  public static MonetaryOperator getDefaultRounding();
  public static MonetaryRounding getRounding(CurrencyUnit currencyUnit, String...
providers);
  public static MonetaryRounding getRounding(String roundingName, String... providers);
  public static MonetaryOperator getRounding(RoundingQuery query);
  public static Collection<MonetaryRounding> getRoundings(RoundingQuery roundingQuery);
  public static boolean isRoundingAvailable(String roundingName, String... providers);
  public static boolean isRoundingAvailable(RoundingQuery query);
  public static Set<String> getRoundingNames(String... providers);
  public static Set<String> getRoundingProviderNames();
  public static List<String> getDefaultRoundingProviderChain();
}
```

Hereby

- `getDefaultRounding()` returns a general rounding instance that is dynamically implementing the default currency rounding, as required by the currency passed, when called.
- `getRounding(CurrencyUnit, String )` returns the default rounding for the given `CurrencyUnit`.
- `getRounding(String, String )` returns an explicit named rounding.
- `getRoundingNames(String )` provides access to the rounding names of the currently registered roundings for the given providers.
- `isRoundingAvailable` allows to determine if the query function return corresponding roundings.
- All access methods above also allow to pass an ellipse operator of provider names. If not set explicitly the default providers and ordering as defined by `getDefaultRoundingProviderNames()` must be used. Hereby
  - if only a single valued result is returned (`MonetaryRounding`, boolean), the provider chain is evaluated until the first provider returns `true` or a non-null `CurrencyUnit` instance.
  - in case of multi valued results all values returned by the providers are added to the result collection (`List, Set, Collection`).
- `getRoundingProviderNames()` provide the names of all currently registered `RoundingProviderSpi` instances. Refer to section [RoundingProviderSpi] for more details.
- `getDefaultRoundingProviderNames()` provide the names of the current default `RoundingProviderSpi` providers in the corresponding chain order.

- `getRounding(RoundingQuery)` offers maximal flexibility for accessing roundings. It is only restricted by the capabilities provided by the registered `RoundingProviderSpi` instances. Refer to section [RoundingProviderSpi] for more details.

- `getRoundings(RoundingQuery)` offers maximal flexibility for accessing roundings, but allows accessing multiple roundings.

The `RoundingQuery` for accessing a rounding from the `Monetary` singleton is modeled as follows:

RoundingQuery Value Type

```
public final class RoundingQuery extends AbstractQuery<RoundingQuery>{
  ...
  public String getRoundingName();
  public int getScale();
  public CurrencyUnit getCurrencyUnit();

  public RoundingQueryBuilder toBuilder();
}
```

By querying `MonetaryRounding` instances with an instance of `RoundingQuery` we can model easily some rather complex use cases:

1. Access cash rounding for a `CurrencyUnit`, 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. This can be achieved by creating an instance of `RoundingContext` with *currency unit* and *cashRounding=true* explicitly yet.

2. Access to historic roundings can be achieved by setting a `CurrencyUnit` and an (optional) target `LocalDate` (or whatever time type is most appropriate).

3. by setting the *rounding id* to a non default value, custom roundings can be implemented, e.g. for support of technical formats.

Instances of this value type can be created using an instance of `RoundingQueryBuilder`. So it would be possible (if the registered provider supports this behaviour) to access special cash rounding, which may be different than the default currency rounding (e.g. for `CHF/Swiss Francs`), as follows:

Example how a cash rounding could be accessed (not part of the API)

```
LocalDate localDate = ...;
MonetaryRounding rounding = Monetary.getRounding(
                RoundingQueryBuilder.of()                                <1>
                  .setRoundingName("cashRounding")                       <2>
                  .setCurrencyUnit("CHF")                                <3>
                  .set(localDate)                                        <4>
                  .build());                                             <5>
```

① Access a rounding by passing a `RoundingQuery`

② Acquire a specific *named* rounding.

③ Set the target currency unit (predefined attribute).

④ Access a rounding valid for the given `LocalDate`.

⑤ Creates the new `RoundingQuery` instance.

Finally the default rounding provider chain can be configured within `javamoney.properties` added to the classpath:

javamoney.properties Configuration of default currencies provider chain

```
# Defaults for java money
...
javax.money.defaults.Monetary.roundingProviderChain=provider1,provider2,provider3
```

4.2.8. Additional Functional Support

Additionally to monetary operators and monetary queries access the the numeric part as well as to the currency of an amount is modeled with corresponding *functional* interfaces similarly.

CurrencySupplier

The interface `javax.money.CurrencySupplier` is a functional interface similar to `Supplier<CurrencyUnit>` as defined in Java 8), whose functional method is `getCurrency()`:

Interface CurrencySupplier

```
// Java 8
@FunctionalInterface
public interface CurrencySupplier {
  CurrencyUnit getCurrency();
}

// Java 7
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 similar to specialization of `Supplier<NumberValue>` as defined in Java 8), whose functional method is `getNumberValue()`:

Interface NumberSupplier

```
// Java 8
@FunctionalInterface
public interface NumberSupplier {
  NumberValue getNumber();
}

// Java 7
public interface NumberSupplier {
  NumberValue getNumber();
}
```

Hereby

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

4.2.9. Exception Types

The core API defines basically two 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 as `public String getCurrencyCode();`.
- a `Locale` given cannot be resolved into a corresponding `CurrencyUnit` instance. The unresolvable `Locale` passed is provided as a property on the exception as `public Locale getLocale();`.

4.3. Currency Conversion

Currency conversion is an important aspect 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 makes explicit conversion even not necessary, in the financial industry, conversion is a very complex concern, since

- conversion may be different based on the use case
- conversion may be different based on the provider 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 date/time

Hereby this list is not complete. Different companies may have further requirements and aspects to be considered. The API focuses on the common aspects of currency conversion such as:

- a source and a target currency
- an exchange rate
- providing conversion providers and having the possibiity to address and combine providers as needed.

Hereby currency conversion or the access of exchange rates can be parametrized with additional meta-data, similar to other models defined by this JSR. This allows to enrich the basic model with whatever complexity is required, hereby keeping the basic model as simple as possible.

4.3.1. Accessing Monetary Conversions

Similar to other areas of this JSR a `MonetaryConversions` singleoton is defined, which provides access to all different aspects related to currency conversion, such as

- access to providers that offer conversion rates, modelled as `ExchangeRate`.

- access to conversion operators (`CurrencyConversion extends 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 functionality in more detail. Similar to other singletons also `MonetaryConversions` is backed up by a `MonetaryConversionsSingletonSpi` SPI to allow customized (e.g. contextual) implementation of the functionality defined. Refer to the Money and Currency 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 adding meta-data support, modelled by `ConversionContext`. Hereby a **conversion is always bound to a specific terminating (target) currency**. So basically a `MonetaryAmount` can simply be converted by passing a `CurrencyConversion` to the amount's `with(MonetaryOperator)` method:

Usage Sample Currency Conversion

```
MonetaryAmount amount = ...;

// Get a default conversion to Swiss Franc
CurrencyConversion conversion = MonetaryConversions.getConversion("CHF");

// Convert the amount
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 concrete `ExchangeRate` applied. This allows further pass the `ExchangeRate` instance to any subsequent logic.

Currency Conversion, accessing exchange rates

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

Nevertheless for accessing `ExchangeRate` instances an `ExchangeRateProvider` is much more effective. It can be accessed from the `MonetaryConversions` singletons as well as from a `CurrencyConversion`.

4.3.3. Exchange Rates and Rate Providers

Exchange Rates

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

- the base and terminating (target) `CurrencyUnit`.
- the conversion factor used [Note that the conversion rate can be dependent on the `MonetaryAmount` passed.], 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.

Summarizing an `ExchangeRate` is modelled as follows:

Interface ExchangeRate

```
public interface ExchangeRate extends CurrencySupplier{
  ...
  ConversionContext getContext();
  CurrencyUnit getBaseCurrency();
  CurrencyUnit getCurrency();
  NumberValue getFactor();
  // Support for chained rates
  List<ExchangeRate> getExchangeRateChain();
  boolean isDerived();
}
```

Hereby

- `getBaseCurrency()`, `getCurrency()`, `getFactor()` model basically the mapping from the base currency to the target currency.
- `isDerived()` allows to check if the mapping in fact is backed up by a derived mapping, e.g. a triangular rate chain.
- `getExchangeRateChain()` return the full rate chain. In case of a non derived rate, this chain must contain only the single rate itself. In case of triangular rate the chain contains all contained subrates.
- the `ConversionContext` accessible from `getContext()` allows to store additional meta data (refer also to Meta-Data Contexts and Query Models for further details) about the rate instance, such as

    - the rate's provider
    - the rate's `LocalDateTime` or `ZonedDateTime`
    - any other data that may be relevant
- each instance of rate finally can easily be converted into an according `ExchangeRate.Builder` instance, so adaptations/changes on existing rates can be done easily.

Implementations of `ExchangeRate`

1. must implement `equals/hashCode`, hereby it is recommended considering
    a. its base and term `CurrencyUnit`
    b. its conversion factor
    c. its `ConversionContext`
2. must be comparable.
3. must be serializable.
4. should be immutable and thread safe.
5. should be implemented as value types, with a fluent Builder pattern.

Exchange Rate Providers

We have seen in the previous section that an `ExchangeRate` can be obtained from a `CurrencyConversion` or from its backing `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.

Summarizing an `ExchangeRateProvider` is modelled as follows:

Interface ExchangeRateProvider

```
// Java 8
public interface ExchangeRateProvider{
  ProviderContext getContext();

  boolean isAvailable(ConversionQuery conversionQuery);
  ExchangeRate getExchangeRate(ConversionQuery conversionQuery);
  CurrencyConversion getCurrencyConversion(ConversionQuery conversionQuery);

  // modelled as default methods in Java 8
  boolean isAvailable(CurrencyUnit base, CurrencyUnit term);
  boolean isAvailable(String baseCode, String termCode);
  ExchangeRate getExchangeRate(CurrencyUnit base, CurrencyUnit term);
  ExchangeRate getExchangeRate(String baseCode, String termCode);
  CurrencyConversion getCurrencyConversion(CurrencyUnit term);
  CurrencyConversion getCurrencyConversion(String termCode);
  ExchangeRate getReversed(ExchangeRate rate);
}
```

Hereby

- the `ProviderContext` allows to provide additional provider meta-data, including the (required and unique) provider name.
- the `isAvailable` methods allow to check for availability of conversion rates from this a provider instance.
- the `getExchangeRate` methods allow to access a concrete conversion rate.
- `getReversed` can be called to reverse an exchange rate (NOTE: rates can, but must not be reversible).
- the `getCurrencyConversion` methods allow to access a `CurrencyConversion` that is internally backed up by the given rate provider instance.

Conversion Query and Conversion Context

The API allows additionally to pass a `ConversionQuery`, 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 [This example is completely arbitrary.] an implementation require/allow to pass

- the target amount
- a customer id
- a contract id

- a fallback strategy
- a deferred rate should be obtained

All these parameters then can be defined as part of a `ConversionQuery`. With such a query any kind of additional parameters can be passed to the rate providers used to evaluate the required `ExchangeRate`. A `ConversionQuery` then can be used to parametrize the `Currency Conversion` as well as an `[ExchangeRateProvider]` instance acquired:

Usage Sample for configuring of a Currency Conversion / ExchangeRate (Provider)

```
ConversionQuery query = ConversionQueryBuilder.of()
       .setRateType(RateType.DEFERRED).
       .set("customerID", 1234)
       .set("contractID", "213453-GFDT-02")
       .set(FallbackStragey.PROVIDER)
       .set(amount)
       .setTermCurrency("CHF")
       .build();

// Access a conversion...
CurrencyConversion conversion = MonetaryConversions.getConversion(query);

// ... or access a rate provider.
ExchangeRateProvider prov = MonetaryConversions.getExchangeRateProvider();
CurrencyConversion conversion = prov.getCurrencyConversion(query);

// for a rate, we need also a base currency
query = query.toBuilder().setBaseCurrency("USD").build();
ExchangeRate rate = prov.getExchangeRate(query);
```

IMPORTANT	Important to understand is that its the responsibility of the used `ExchangeRateProvider` implementation to interpret the attributes passed within a `ConversionQuery`. Unknown parameters should simply be ignored, since a provider can be used in a *provider chain* (explained in the next section).

4.3.4. ExchangeRateProvider 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 can be accessed by calling `MonetaryConversions.getDefaultProviderChain()`. Hereby the chain contains an ordered list of provider names, which correspond to the provider names that identify each registered `ExchangeRateProvider` uniquely. The provider name is defined as a mandatory attribute on the `ProviderContext`, accessible from each `ExchangeRateProvider` from `ExchangeRateProvider.getContext()`.

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={PROVIDER=Compound: ECB}]
```

Usage Sample Accessing the default Exchange Rate Provider Chain IDs

```
// Accessing the default provider chain, configurable in javamoney.properties
List<String> providerIds = MonetaryConversions.getDefaultProviderChain();
```

As mentioned accessing a currency conversion or rate provider, without passing the providers required returns the 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");
CurrencyConversion conversion = MonetaryConversions.getConversion("CHF", "ECB", "IMF",
"ECB-HIST");
```

Within a provider chain, the first provider that returns a non-null result determines the final value of the method call, 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 keeping the API simple for the simple use cases.

The default rate provider chain can be configured within `javamoney.properties` added to the classpath:

javamoney.properties Configuration of default conversion provider chain

```
# Defaults for java money
...
javax.money.defaults.MonetaryConversions.providerChain=provider1,provider2,provider3
```

4.4. Money and Currency Formatting API

The formatting is modelled with a quite simple, but very flexible design. It allows the access of formats based on `java.util.Locale`, similarly to the functionality in `java.text`, but offers flexibility that goes beyond the JDKs formatting packages. In contrary to the JDK formatter the formatter defined by this API are thread-safe and arbitrarily expandable.

The entry point for the JSR formatter is the `MonetaryFormats` singleton, which provides access to different formatter API artifacts. The following section describes the relevant artifacts in more detail.

4.4.1. Formatting of Monetary Amounts

As defined in [RequirementsFormatting|Requirements], this JSR must provide an API for providing flexible and expandable formatting capabilities for `MonetaryAmount` instances. Though formatting is a very complex field the JSR's expert group has identified a minimal set of functionality, that provides an API simple to use, but still being flexible to accommodate a wide range of usage scenarios. Aspects to be considered are:

1. Amount values can be rounded for display by applying any `MonetaryOperator` before formatting/printing.
2. Similarly amount values can be operated after parsing by applying any `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` [`INR 123456000.21` is formatted as `INR 12,34,56,000.21`].
4. The currency part of an amount can be formatted in different ways:
   a. as currency code, e.g. `USD`
   b. as numeric currency code, if such a code is defined.
   c. as a (localized) currency symbol, e.g. `$`
   d. as a (localized) currency name, e.g. `Schweizer Franken`
   e. the currency part is omitted from the formatter's output (e.g. because its printed out somewhere else already).
5. The overall formatting and parsing pattern can be defined similar to `java.text.DecimalFormat`, but also completely different usage scenarios are possible.

Fortunately all this scenarios can be covered by implementing instances of the `MonetaryAmountFormat` interface as shown below:

Interface MonetaryAmountFormat

```
public interface MonetaryAmountFormat extends MonetaryQuery<String>{
  String format(MonetaryAmount<?> amount);
  void print(Appendable appendable, MonetaryAmount<?> amount) throws IOException;
  MonetaryAmount<?> parse(CharSequence text) throws ParseException;
  AmountFormatContext getContext();
}
```

Hereby

- an amount can be formatted to a String or an `Appendable`, or parsed from a `String`.

- The meta-data of the format are provided by an immutable `AmountFormatContext` value type. Refer to Meta-Data Contexts and Query Models for further details on meta-data modeling.

The power of the API now comes with the capability to pass instances of `AmountFormatQuery` to the singleton for accessing `MonetaryAmountFormat` instances. Similar to other queries defined by this JSR it is possible to pass any additional parameters that are necessary to configure the concrete formatting to be returned. Summarizing: * The `AmountFormatQuery` defines the parameters and attributes that configure a format. Hereby a format can be identified by *name* or configured on the fly. The effective behaviour depends on the concrete functionality provided by the (possibly several) registered instances of type `Adding Amount Formats`. * The `Adding Amount Formats` implementation finally must interpret the attributes in `AmountFormatContext` and create an according formatter instance.

With that simple approach, we can extend our formatting capabilities easily as needed. Nevertheless the basic API for common use cases still is simple, since we can also access formatting just using a `Locale`, similarly to `javax.text.DecimalFormat.getCurrencyInstance(Locale)`.

IMPORTANT	Implementations of this JSR must provide according default formatter for each `Locale` that is also available from `javax.text.DecimalFormat.getCurrencyInstance(Locale)`. Hereby it is not required that the format is exact the same, e.g. formatting for Indian Rupees is expected to have different grouping sizes.

Contrary to the formatter in `javax.text` implementations of this interface must be thread-safe.

Examples

Given the API above, acquiring a `MonetaryAmountFormat` instance is simple, the most simple usage is just accessing one using a `Locale`:

Usage Example Formatting a MonetaryAmount

```
MonetaryAmountFactory<?> f = Monetary.getDefaultAmountFactory();
MonetaryAmount amount = f.setCurrency("CHF").setNumber(12.50).create();

MonetaryAmountFormat format =
                        MonetaryAmountFormats.getAmountFormat(Locale.GERMANY);

// format the given amount
String formatted = format.format(amount); // result: CHF 12,50

// create another amount based on the given amount
amount = f.toBuilder().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 could configure an `AmountFormatContext` that implements this behaviour as illustrated below:

Usage Example (continued) Formatting a MonetaryAmount

```
AmountFormatQuery query = AmountFormatQueryBuilder.of(new Locale("","INR"))
                                  .set("groupSizes", new int[]{3,2}).build();
MonetaryAmountFormat format = MonetaryAmountFormats.getAmountFormat(query);

MonetaryAmount amount =
          Monetary.getDefaultAmountFactory()
            .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

As mentioned before a `MonetaryAmountFormat` can be configured using an `AmountFormatQuery` with arbitrary attributes, so also very complex and historic formats can be supported easily. Instances of `AmountFormatQuery` can be created using an `AmountFormatQueryBuilder`:

Class AmountFormatQuery

```
public final class AmountFormatQuery extends AbstractQuery{
  private AmountFormatQuery(AmountFormatQueryBuilder builder);
  ...
  public String getFormatName();
  public Locale getLocale();
  public AmountFormatQueryBuilder toBuilder();

}

public final class AmountFormatQueryBuilder extends AbstractQueryBuilder
<AmountFormatQueryBuilder,AmountFormatQuery>{
  ...
  public static AmountFormatQueryBuilder create(String formatName);
  public static AmountFormatQueryBuilder create(Locale locale);

  public AmountFormatQueryBuilder setMonetaryQuery(MonetaryQuery monetaryQuery);

  public AmountFormatQuery build();

}
```

Hereby the above listing illustrates quite well, what are the minimal properties that define an `AmountFormatQuery`:

- a format name, by default `"default"`.
- a `Locale`
- of course, additional parameters can be added as needed, such as output and input patterns, color or style settings, `MonetaryAmountFactory` instance to be used for creating amounts on parsing etc.

The configuration active for a concrete `MonetaryAmountFormat` is accessible also from the `AmountFormatContext`, which can obtained by calling `MonetaryAmountFormat.getContext()`.

4.4.3. MonetaryFormats Accessor Singleton

The class `javax.money.format.MonetaryFormats` models a singleton accessor, which is, similarly to other singleton in this JSR, backed up by an SPI instance of `Accessing Monetary Amount Formats`. The SPI implementation is responsible for collecting and managing registered instances of `MonetaryAmountFormatProviderSpi` providing them based on the `AmountFormatQuery` passed. Such a query can contain

- a `Locale`, or
- a format name
- the target providers that should be selected to handle the query to create/provide a `MonetaryAmountFormat` instance.
- any other attributes as defined by the provider that should handle the query, refer to the section Meta-Data Contexts and Query Models for more details.

Interface MonetaryFormatsSingletonSpi

```
public interface MonetaryFormatsSingletonSpi{
  Collection<MonetaryAmountFormat> getAmountFormats(AmountFormatQuery query);
<1>
  Set<Locale> getAvailableLocales(String... providers);
<2>
  Set<String> getFormatNames(String... providers);
<3>
  Set<String> getProviderNames();
<4>
  List<String> geDefaultProviderChain();
<5>

  // The following methods are modelled as default methods in Java 8
  MonetaryAmountFormat getAmountFormat(Locale locale, String... providers){...}    <6>
  MonetaryAmountFormat getAmountFormat(Locale locale, String... providers){...}    <6>
  MonetaryAmountFormat getAmountFormat(String name, String... providers);          <6>
  boolean isAvailable(Locale locale, String... providers){...}                     <7>
  boolean isAvailable(AmountFormatQuery formatQuery){...}                          <7>

}
```

① Main method called for accessing formats.

② Collect all locales available from the given providers. If no provider IDs are passed the default provider chain is used.

③ Get the available format names for the given providers. If no provider IDs are passed the default provider chain is used.

④ Access all registered provider's ids.

⑤ Access the ids of the default provider chain, in the ordering as executing.

⑥ In java 8 these methods are provided as default methods delegating to `getAmountFormats(AmountFormatQuery)`.

⑦ In java 8 these methods are default methods trying to access a formatting calling `getAmountFormats(AmountFormatQuery)`.

Similar to other singletons of this JSR overriding the `Accessing Monetary Amount Formats` allows to add contextual behaviour in EE or multi-tenancy runtime environment. Refer to Money and Currency SPI for further details.

The `MonetaryFormats` singleton finally defines the following access methods, very similar to the `MonetaryFOrmatsSingletonSpi`:

MonetaryFormats Singleton

```
public final class MonetaryFormats{
  private MonetaryFormats(){}

  public static MonetaryAmountFormat getAmountFormat(Locale locale);
  public static MonetaryAmountFormat getAmountFormat(AmountFormatContext context);
  public static MonetaryAmountFormat getAmountFormat(Locale locale, String... providers)
{...}
  public static MonetaryAmountFormat getAmountFormat(Locale locale, String... providers)
{...}
  public static MonetaryAmountFormat getAmountFormat(String name, String... providers);
  public static Collection<MonetaryAmountFormat> getAmountFormats(AmountFormatQuery
query);
  public static boolean isAvailable(Locale locale, String... providers){...}
  public static boolean isAvailable(AmountFormatQuery formatQuery){...}
  public static Set<Locale> getAvailableLocales(String... providers);
  public static Set<String> getFormatNames(String... providers);
  public static Set<String> getFormatProviderNames();
  public static List<String> geDefaultFormatProviderChain();
}
```

The design chosen ist so flexible that every kind of formatting related to monetary amounts can be easily mapped. The code below illustrates a hypothetical, but more complex example:

Advanced setup of a AmountFormatQuery

```
AmountFormatQuery query = new AmountFormatQueryBuilder.of("htmlFormat")<1>
                          .set("title", "MyTitle")                      <2>
                          .set("negativeStyle", ".negNumber")           <3>
                          .set("positiveStyle", ".posNumber+)           <3>
                          .set("styleClass", "styledAmount")            <4>
                          .build();                                     <5>
```

① Access a format with name `htmlFormat`

② Sets the format's display name

③ Sets the CSS style classes to be used for positive and negative values.

④ Sets the overall default style class.

⑤ Creates a new instance.

NOTE The example above is arbitrarily chosen. This specification does not require this behaviour to be available, or be implemented as shown before.

Similar to currency conversion the default format provider chain can be configured within `javamoney.properties` added to the classpath:

javamoney.properties Configuration of default format provider chain

```
# Defaults for java money
...
javax.money.defaults.MonetaryFormats.providerChain=provider1,provider2,provider3
```

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 `CharSequence` passed to the `MonetaryAmountFormat`.
- the error index within the input String, where parsing failed unrecoverable.

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. These services must be registered to the JSR's `Bootstrap` mechanism. The `Bootstrap` singleton internally loads an instance of `ServiceProvider` using `java.util.ServiceLoader`. The default loader used hereby relies on `java.util.ServiceLoader` to load the implementation services, so be default the JSR behaves like a normal SE based JSR. However by registering alternate implementations of `ServiceProvider` the component loading mechanism can be replaced completely, e.g. with a mechanism that also tries to get access a *CDI* `BeanManager` from JNDI thus enabling to register SPI implementations as CDI managed beans.

All SPIs are contained in the package `javax.money.spi`. Summarizing the following SPIs are available:

- `CurrencyProviderSpi` (mandatory, multiple service chain) - provides instances of `CurrencyUnit`, accessible from `Monetary` singleton.
- `MonetaryAmountsSingletonSpi` (mandatory, only one instance selected by priority) - manages instances of `MonetaryAmountFactoryProviderSpi`, which create instances of `MonetaryAmountFactory`, that are being accessible from the `Monetary` singleton.
- `MonetaryAmountFactoryProviderSpi` (mandatory, multiple service chain) - is responsible for registering and providing instances of `MonetaryAmountFactory`.
- `MonetaryAmountsSingletonQuerySpi` (mandatory, only one instance selected by priority) - this SPI allows to override/define the behaviour of `Monetary.queryAmountType(MonetaryContext)`.
- `RoundingProviderSpi` (mandatory, multiple service chain) - provides instances of `MonetaryOperator`, for being accessible from `Monetary`.

- `MonetaryRoundingsSingletonSpi` controls the loading of `RoundingProviderSpi` instances.
- `MonetaryConversionSingletonSpi` (mandatory, only one instance selected by priority) - manages instances of `ExchangeRateProvider`, for being accessible by the `Monetary` singleton and also is responsible for providing the composite provider instances as to be returned by the conversion API.
- `ExchangeRateProvider` (mandatory, multiple instances selected by API) - this class is also part of the API, but also models the huge part of the SPI required for currency conversion.
- `MonetaryAmountFormatSingletonSpi` (mandatory, only one instance selected by priority) - provides the backing bean for the `MonetaryFormats` singleton, manages instances of `MonetaryAmountFormatProviderSpi`.
- `MonetaryAmountFormatProviderSpi` (mandatory, multiple service chain) - provides instances of `MonetaryAmountFormat`, for being accessible by+ MonetaryFormats.getAmountFormat(<?>)+.
- `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.

NOTE	SPI interfaces called `XXXSingletonSpi` are generally loaded once during early boot of the JSR and are subsequently managed as static references within the singleton accessors. This may look as a constraint, but in fact you just have to ensure to delegate component loading and management to the `Bootstrap` mechanism. You can refer to the Moneta reference implementation for further details, which exactly implements this behaviour.

4.5.1. Core SPI

Accessing Currencies

Currencies are accessed from the [Monetary] singleton. This singleton is backed up by an implementation of `MonetaryCurrenciesSingletonSpi`, which must be registered to the The Bootstrapping Mechanism:

Interface MonetaryCurrenciesSingletonSpi

```
public interface MonetaryCurrenciesSingletonSpi {
  List<String> getDefaultProviderChain();
  Set<String> getProviderNames();
  Set<CurrencyUnit> getCurrencies(CurrencyQuery query);

  // The following methods are modelled as default methods in Java 8
  CurrencyUnit getCurrency(String currencyCode, String... providers);
  CurrencyUnit getCurrency(Locale country, String... providers);
  Set<CurrencyUnit> getCurrencies(Locale locale, String... providers);
  boolean isCurrencyAvailable(String code, String... providers);
  boolean isCurrencyAvailable(Locale locale, String... providers);
  Set<CurrencyUnit> getCurrencies(String... providers);
  CurrencyUnit getCurrency(CurrencyQuery query);
}
```

Hereby

- Similar to other areas multiple instances of `CurrencyProviderSpi` can be registered.
- Each `CurrencyProviderSpi` instance is identified by its (unique) provider name.
- On access the required chain of `CurrencyProviderSpi` can be defined explicitly.
- If no provider chain is explicitly passed, the default provider chain as defined by `getDefaultProviderChain()` is used.
- When accessing `CurrencyUnit` instances the first non null/not empty instances returned by a provider are used as a result by default. Implementations of `MonetaryCurrenciesSingletonSpi` may also add additional combination strategies by defining additional attributes that can be passed as part of a `CurrencyQuery` passed.
- Basically it is sufficient for implementations to provide the `getCurrencies(CurrencyQuery)` method for looking up currencies, since by default all other methods delegate to this method.

Implementations of this class should use the The Bootstrapping Mechanism suport to evaluate the instances of `CurrencyProviderSpi` for a given runtime context.

Registering Currencies

By registering instances of `javax.money.spi.CurrencyProviderSpi` to the The Bootstrapping Mechanism logic additional `CurrencyUnit` instances can be registered into the `Monetary` singleton:

Interface CurrencyProviderSpi

```
public interface CurrencyProviderSpi {
  Set<CurrencyUnit> getCurrencies(CurrencyQuery query);

  // Modelled as default methods in Java 8
  String getProviderName();
  boolean isCurrencyAvailable(CurrencyQuery query);
}
```

Hereby

- each provider must return a unique provider ID from `getProviderName()`. By default the provider's simple class name is used.
- with `getCurrencies(CurrencyQuery)` each provider can return the `CurrencyUnit` instances that are valid as defined by the `CurrencyQuery` instance passed. Hereby a query can refer to a single currency instance, or a more open query returning multiple instances. Additional parameters can be supported as needed for the concrete use cases, e.g. historic or regional reference, namespace schemas etc. If a provider cannot deliver any `CurrencyUnit` instances, an empty `Set` must be returned.
- `isCurrencyAvailable(CurrencyQuery)` allows to check for any currencies being available. By default this method calls `getCurrencies(CurrencyQuery)` and simply checks for an empty `Set` being returned. Implementations may override this behaviour with a more efficient approach.

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:

Interface MonetaryAmountFactoryProviderSpi

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

  MonetaryAmountFactory<T> createMonetaryAmountFactory();

}
```

Hereby

- `getAmountType()` returns a new implementation of `T` which is returned by a `MonetaryAmountFactory` created by an instance.
- The default `MonetaryContext` used can be determined by calling `getDefaultMonetaryContext()`.
- The maximal supported `MonetaryContext` can be determined by calling `getMaximalMonetaryContext()`.
- `getQueryInclusionPolicy()` defines if the given spi (and hence the corresponding `MonetaryAmount` implementation type) is to be considered, when `Monetary.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.
- `createAmountFactory()` finally creates a corresponding `MonetaryAmountFactory` factory.

Backing the Monetary Singleton (Amount related functionality)

Also the functionality of the `Monetary` accessor singleton is backed up by two SPI interfaces, called `javax.money.spi.MonetaryAmountsSingletonSpi` and `javax.money.spi.MonetaryAmountsSingletonQuerySpi`. Implementations of these interfaces should rely on the `Bootstrap` class to access the available instances of `MonetaryAmountFactory` and `MonetaryAmountFactoryProviderSpi` respectively. Nevertheless being able to register alternate implementations would allow to support more complex rules for a couple of enterprise related functionality such as:

Implementing `MonetaryAmountsSingletonSpi` allows * to provide amount types (and related factories) based on the current runtime context. * to configure the default amount type, as provided by `Monetary.getDefaultAmountType()`, differently based on the current runtime context. * to configure different defaults for the `MonetaryContext` used by the amount implementation types/factories.

The interface is defined as follows:

Interface MonetaryAmountsSingletonSpi

```
public interface MonetaryAmountsSingletonSpi{
  <T extends MonetaryAmount> MonetaryAmountFactory<T> getAmountFactory(Class<T>
amountType);
  Class<? extends MonetaryAmount> getDefaultAmountType();
  Collection<Class<? extends MonetaryAmount>> getAmountTypes();

  // Modelled as default methods in Java 8
  MonetaryAmountFactory<?> getDefaultAmountFactory();
  Collection<MonetaryAmountFactory<?>> getAmountFactories();
}
```

Hereby

- `getAmountFactory(Class<T>)` should return an instance of `MonetaryAmountFactory<T>` that creates the amount instances.
- `getDefaultAmountType()` returns the default implementation type created by the factory returned from `getDefaultAmountFactory()` for the current runtime context.
- `getAmountTypes()` should return a collection 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.
- Similar to `getAmountTypes()` the method `getAmountFactories()` returns all corresponding factories for the current context.

Implementing `MonetaryAmountsSingletonQuerySpi` allows to provide alternate implementations of query algorithm used within `Monetary.queryAmountType(MonetaryContext)` to evaluate the best matching `MonetaryAmount` implementation given a `MonetaryAmountFactoryQuery` required. The interface is defined as follows:

Interface MonetaryAmountsSingletonQuerySpi

```
public interface MonetaryAmountsSingletonQuerySpi{
  Collection<MonetaryAmountFactory<? extends MonetaryAmount>> getAmountFactories
(MonetaryAmountFactoryQuery query);

  // Modelled as default methods in Java 8
  MonetaryAmountFactory getAmountFactory(MonetaryAmountFactoryQuery query);
  Class<? extends MonetaryAmount> getAmountType(MonetaryAmountFactoryQuery query)
  Collection<Class<? extends MonetaryAmount>> getAmountTypes(MonetaryAmountFactoryQuery
query)
  boolean isAvailable(MonetaryAmountFactoryQuery query);
}
```

Hereby

- The only method to be overriden by default is `getAmountFactories(MonetaryAmountFactoryQuery)`, which evaluates a given `MonetaryAmountFactoryQuery` and returns the matching `MonetaryAmountFactory` instances.
- Additionally there are a couple of default methods, which reflect the overall functionality provided from the `Monetary` singleton. This gives implementation providers full control about the functionality executed. Summarizing the methods implemented by default are:
  - `getAmountType(MonetaryAmountFactoryQuery)` allows to evaluate a `MonetaryAmount` implementation type that best covers the requirements defined by the passed `MonetaryAmountFactoryQuery`. If multiple types match the query and are not resolvable, a `MonetaryException` should be thrown.
  - `getAmountFactory(MonetaryAmountFactoryQuery)` matches `getAmountType(MonetaryAmountFactoryQuery)`, but returns the `MonetaryAmountFactory` instance instead of the implementation type.
  - `getAmountTypes(MonetaryAmountFactoryQuery)` allows to evaluate all `MonetaryAmount` implementation types that covers the requirements defined by the passed `MonetaryAmountFactoryQuery`.
  - `isAvailable(MonetaryAmountFactoryQuery)` allows to determine (without any exception thrown) if the given query returns any types/factories matching.

In general implementations of this interface should consider the following rules: . if the `MonetaryAmountFactoryQuery` 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 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 [This is the case, if the the value from `MonetaryAmountFactoryProviderSpi.getInclusionPolicy()` does not equal to `QueryInclusionPolicy.NEVER`, or `QueryInclusionPolicy.DIRECT_REF_ONLY`.] 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`). . Additional attributes to consider may be provided with the `MonetaryAmountFactoryQuery` provided, though this specification does not define any further aspects in detail. . if all of the above is true, the according amount types or amount factories should be returned.

If one of the conditions above fails a `MonetaryException` must be thrown [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.], or in case of `Collection<?>` being the method's return type, an empty collection should be returned. Also `isAvailable(MonetaryAmountFactoryQuery)` should never throw an exception, but return `false`, if no matching `MonetaryAmountFactoryProviderSpi` could be determined.

Accessing Roundings

Since a monetary rounding is nothing else than a conversion from an unrounded amount to a rounded amount, it is modeled as `Monetary Operators`. Nevertheless similar to other artifacts defined by this JSR also roundings have metadata attached, so roundings extends `Monetary Operators` and additionally provide access to the so called `RoundingContext`. `MonetaryRounding` instances are accessed from the `Monetary` singleton. Similar to other singletons in this JSR the `Monetarys` singleton is backed up by an instance of `MonetaryRoundingsSingletonSpi`, which allows to control the exact logic how registered `javax.money.spi.RoundingProviderSpi` are managed:

Interface MonetaryRoundingsSingletonSpi

```
public interface MonetaryRoundingsSingletonSpi {
  Collection<MonetaryRounding> getRoundings(RoundingQuery query);
  MonetaryRounding getDefaultRounding();
  Set<String> getRoundingNames(String... providers);
  Set<String> getProviderNames();
  List<String> getDefaultProviderChain();

  // Modelled as default methods in Java 8
  MonetaryRounding getRounding(RoundingQuery roundingQuery);
  MonetaryRounding getRounding(CurrencyUnit currencyUnit, String... providers);
  MonetaryRounding getRounding(String roundingName, String... providers);
  MonetaryRounding getRounding(RoundingQuery query);
  boolean isRoundingAvailable(RoundingQuery query);
  boolean isRoundingAvailable(String roundingId, String... providers);
  boolean isRoundingAvailable(CurrencyUnit currencyUnit, String... providers);
}
```

Similar to Currency Conversion a provider based service model is defined, meaning * multiple rounding providers can be registered * when accessing `MonetaryRounding` instances the providers and ordering of the provider chain that should handle the request can be explicitly defined. * if no chain is passed explicitly a default provider chain is used, configurable within the `javamoney.properties` file (see details later). * `getRoundingNames` allows to evaluate all explicitly named roundings available from a given set of rounding providers. * `getProviderNames()` allows to evaluate the names of the registered providers for the current context. * `getDefaultProviderChain()` return the ordered list of provider names representing the default provider chain used, when no explicit rounding providers are selected.

Basically all access methods by default delegate to `getRoundings(RoundingQuery)`. Nevertheless all possible access methods from the `Monetary` singleton are reflected in the SPI, so implementations have full control of the logic executed.

Summarizing:

- `MonetaryRounding` instances can be accessed
  - based on a `CurrencyUnit`, hereby returning the default rounding for a given currency.
  - by its explicit (unique) *roundingId*.
  - by passing a `RoundingQuery` allowing to add arbitrary additional parameters to configure the rounding returned.

Registering Roundings

To register additional `MonetaryRounding` instances of `RoundingProviderSpi` must be implemented and

registered with the current The Bootstrapping Mechanism logic:

Interface RoundingProviderSpi

```
public interface RoundingProviderSpi {
  MonetaryRounding getRounding(RoundingQuery query);
  Set<String> getRoundingNames();

  // Modelled as default method in Java 8
  String getProviderName();
}
```

Hereby:

- `getRounding(RoundingQuery)` should return the matching rounding, given a `RoundingQuery`. If a provider cannot provide the requested rounding, it should simply return `null`.
- `getRoundingNames()` returns all rounding names of the explicitly accessible roundings provided by this rounding provider. It is within the responsibility of the implementation of `MonetaryRoundingsSingletonSpi` to collect all rounding names contributed by providers to built the full list of rounding names.
- Similar to other providers in this JSR each rounding provider must declare an explicit unique name provided by the `getProviderName()` method.

Implementations of this JSR should also consider additional aspects:

1. When providing roundings targeting currencies, *by default*, if no explicit rounding is available for a given `CurrencyUnit`, the digits returned from `CurrencyUnit.getDefaultFractionDigits()` and `RoundingMode.HALF_EVEN` should be used, to create a rounding for a given `CurrencyUnit`.
2. Under Java SE (or where available) reference implementations should also provide *default* arithmetic rounding instances, e.g. you can set a maximal scale of `1` and a `RoundingMode` as an additional attribute.
3. Implementations should also support cash rounding. E.g. 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.
4. Finally it may also possible to provide *historic* roundings hereby considering an additional target date/time, e.g. modelled as `LocalDate`.

4.5.2. Currency Conversion SPI

Accessing Currency Conversion Artifacts

Currency Conversion and rate providers mechanisms are provided by the `MonetaryConversions`

singleton (see Currency Conversion). This singleton is backed up by an implementation of `javax.money.spi.MonetaryConversionsSingletonSpi`. Implementing this SPI provides full control about the singleton's effective behaviour. As a consequence the methods to be implemented basically match the ones defined by the `MonetaryConversions` class:

Interface MonetaryConversionsSingletonSpi

```
public interface MonetaryConversionsSingletonSpi {
  ExchangeRateProvider getExchangeRateProvider(ConversionQuery conversionQuery);
  Collection<String> getProviderNames();
  List<String> getDefaultProviderChain();

  // Modelled as default methods in Java 8
  ExchangeRateProvider getExchangeRateProvider(String... providers);
  List<ExchangeRateProvider> getExchangeRateProviders(String... providers);
  boolean isExchangeRateProviderAvailable(ConversionQuery conversionQuery);
  CurrencyConversion getConversion(ConversionQuery conversionQuery);
  CurrencyConversion getConversion(CurrencyUnit termCurrency, String... providers);
  boolean isConversionAvailable(ConversionQuery conversionQuery);
  boolean isConversionAvailable(CurrencyUnit termCurrency, String... providers);
}
```

Hereby

- multiple `ExchangeRateProvider` instances can be registered.
- each `ExchangeRateProvider` is identified by it's (unique) name, accessible from `ExchangeRateProvider.getContext().getProviderName()`.
- `getProviderNames()` allows to evaluate the names of the registered providers for the current context.
- `getDefaultProviderChain()` return the ordered list of provider names representing the default provider chain used, when no explicit rounding providers are selected.
- `ExchangeRateProvider` instances can be accessed directly by passing the single provider name only as the target chain definition, e.g. the provider *XY* can be accessed by calling `getExchangeRateProvider("XY")`.
- 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( ECB , IMF )` should by default return a *composite* `ExchangeRateProvider` instance, that internally first tries to resolve an `ExchangeRate` requested, using the provider named "ECB". On success the "ECB" 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 `CurrencyConversionException` must be thrown as defined in the corresponding `ExchangeRateProvider` interface API documentation.

- If no explicit provider names are passed, the provider names and ordering as defined by `getDefaultProviderChain()` have to be used.

- Implementations can easily provide alternate combination policies, but defining a corresponding configuration flag that can be passed to a `ConversionQuery` and that must be interpreted by the registered `MonetaryConversionsSingletonSpi` implementation, e.g.

```
ExchangeRateProvider rateProvider = getExchangeRateProvider(ConversionQueryBuilder.of()
   .setProviders("XY", "foo")
   .set(MyProviderCombiationPolicy.CHEAPEST_CONTRACT_FIRST)
   .build());
```

Basically all access methods by default delegate to `getExchangeRateProvider(ConversionQuery)`. Nevertheless all possible access methods from the `MonetaryConversions` singleton are reflected in the SPI, so implementations have full control of the logic executed.

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 `[ExchangeRateProvider]` and `[ExchangeRate]`.

Implementations of the `MonetaryConversionsSingletonSpi` should use the current The Bootstrapping Mechanism implementation to load right providers to be used for a given runtime context.

4.5.3. Formatting SPI

Accessing Monetary Amount Formats

Amount Formats are provided by the `MonetaryFormats` singleton (see [MonetaryFormats]). This singleton is backed up by an implementation of `javax.money.spi.MonetaryFormatsSingletonSpi`. Implementing this SPI provides full control about the singleton's effective behaviour. As a consequence the methods to be implemented basically match the ones defined by the `MonetaryFormats` class:

Interface MonetaryFormatsSingletonSpi

```
public interface MonetaryFormatsSingletonSpi {
  Collection<MonetaryAmountFormat> getAmountFormats(AmountFormatQuery formatQuery);
  Set<Locale> getAvailableLocales(String... providers);
  Set<String> getProviderNames();
  List<String> getDefaultProviderChain();

  // Modelled as default methods in Java 8
  MonetaryAmountFormat getAmountFormat(Locale locale, String... providers);
  MonetaryAmountFormat getAmountFormat(String formatName, String... providers);
  MonetaryAmountFormat getAmountFormat(AmountFormatQuery formatQuery);
  boolean isAvailable(AmountFormatQuery formatQuery);
  boolean isAvailable(Locale locale, String... providers);
}
```

Hereby

- multiple `MonetaryAmountFormat` instances can be accessed, provided by registered instances of `MonetaryAmountFormatProviderSpi` (see next section for details), whereby each provider spi is identified by a unique provider name.
- `getProviderNames()` allows to evaluate the names of the registered providers for the current context.
- `getDefaultProviderChain()` return the ordered list of provider names representing the default provider chain used, when no explicit rounding providers are selected.
- Instances of `MonetaryAmountFormat` can be identified and accessed using a format name.
- Instances of `MonetaryAmountFormat` can be accessed using a target `Locale`.
- When accessing `MonetaryAmountFormat` instances, the provider chain to be used can be defined explicitly by passing the ordered provider names. If no explicit provider names are passed, the provider names and ordering as defined by `getDefaultProviderChain()` have to be used.
- `getAvailableLocales()` allows to access the locales for which providers can return formats.

Basically all access methods by default delegate to `getAmountFormat(AmountFormatQuery)`. Nevertheless all possible access methods from the `MonetaryFormats` singleton are reflected in the SPI, so implementations have full control of the logic executed.

Adding Amount Formats

The `MonetaryFormats` singleton delegates access to `MonetaryAmountFormat` instances to the registered `MonetaryFormatsSingletonSpi` instance. The ladder class is responsible to manage the registered instances of `javax.money.spi.MonetaryAmountFormatProviderSpi` to evaluate the correct results:

Interface MonetaryAmountFormatProviderSpi

```
public interface MonetaryAmountFormatProviderSpi {
   Collection<MonetaryAmountFormat> getAmountFormats(AmountFormatQuery formatQuery);
   Set<Locale> getAvailableLocales();
   Set<String> getAvailableFormatNames();

   // Modelled as default method in Java 8
   String getProviderName();
}
```

Hereby * `getProviderName()` defines the (unique) provider name that can be used to reference this format provider either explicitly when accessing format instances or when configuring the default provider chain. * at least one instance of `MonetaryAmountFormatProviderSpi` must be registered to the The Bootstrapping Mechanism logic. * `getAmountFormats(AmountFormatQuery)` returns the corresponding `MonetaryAmountFormat` instances that match the given query. If the query doe not match, an empty collection should be returned. * `getAvailableLocales()` returns the locales for which instances of `MonetaryAmountFormat` can be accessed. * `getAvailableFormatNames()` returns the explicit format names for whcih instances of `MonetaryAmountFormat` can be accessed explicitly from a provider.

4.5.4. 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 `ServiceProvider` is configured, a default implementation is used that uses the `java.util.ServiceLoader` to load and locate the instances.

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

Class Bootstrap

```
public final class Bootstrap{
  public static ServiceProvider init(ServiceProvider serviceProvider);
  public static <T> Collection<T> getServices(Class<T> serviceType){...}
  public static <T> T getService(Class<T> serviceType) {...}
}
```

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 provider implementation, delegating

to `java.util.ServiceLoader` and with no specific ordering/priorization mechanism.

- if *exact one* implementation is registered, this implementation is used for loading/accessing the services required by the JSR 354 API. The 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.

Service Provider

To use an alternate implementation of `javax.money.spi.ServiceProvider` it 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`.

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

Interface ServiceProvider

```
@FunctionalInterface
public interface ServiceProvider {
  <T> Collection<T> getServices(Class<T> serviceType){...}
}
```

Hereby

- if a required service type can not be satisfied, an empty Collection should be returned.

Support for EE / CDI

We have seen that all the singleton accessors defined by this API can be replaced by customized implementations. This allows also to adapt the behaviour in case your application runs in a EE/CDI context. Given this all singleton backing implementations should delegate to the bootstrap component's `ServiceProvider` to evaluate the right instances of components given a special type. As a consequence integration with Java EE can be done in multiple ways:

- you can override the singleton SPIs (`MonetaryAmountsSingletonSpi`, `MonetarFormatsSingletonSpi`, `MonetaryRoundingsSpi`, `MonetaryConversionsSingletonSpi`, `MonetaryFormatsSingletonSpi` and reimplement the mechanism how the different components are located within the current runtime environment. Additionally you can filter or adapt the components accessible, e.g. based on tenant or other contextual information. Overriding the singletons gives you full control. Nevertheless overriding the SPIs requires more knowledge about the specification. Basically we recommended to execute the TCK to identify locations, where your implementation may not be compliant with this spec, when using this approach.
- Far more easy is to reuse the default singleton implementations, but exchange the `ServiceProvider`

used. This is basically quite easy:

- Implement an alternative instance of `ServiceProvider`.
- Register the instance using the JDK's `ServiceLoader`: add a file with the following content to your (system) classpath under `/META-INF/services/javax.money.spi.ServiceProvider`:

Register alternate ServiceLoader in `/META-INF/services/javax.money.spi.ServiceProvider`

```
foo.bar.MyFooServiceProvider
```

This will delegate all requests for SPIs to your `foo.bar.MyFooServiceProvider` implementation. Within this implementation you must: * fall back on SE mechanism, when EE/CDI is not available. * locate the components as required * enable component precedence by ordering the instances found, e.g. you can base your ordering on `@Priority` annotations on the classes loaded. Hereby the components with higher priority must be returned first. They either have precedence in command chains or are selected as final components to be used, e.g. for backing singleton beans.

An according example is implemented within the [JavaMoney] library, basically it looks similar to the following code:

Outline of a CDI based ServiceLoader

```
@ServicePriority(ServicePriority.NORM_PRIORITY + 1)
public class CDIServices implements ServiceProvider {

  @Override
  public <T> List<T> getServices(Class<T> serviceType) {
    List<T> instances = new ArrayList<T>();
    for(T t: ServiceLoader.load(serviceType)){
      instances.add(t);
    }
    for (T t : CDIContainer  // backed up by CDI.current().getBeanManager();
.getInstances(serviceType)) {
instances.add(t);
}
Collections.sort(instances, PrioritySorter::sort);
return instances;
  }

}
```

As a side effect you may add additional functionality to your setup:

- For example you may write a CDI portable extension to add the service that are registered using the JDK `ServiceLoader` to your CDI runtime context.

- With CDI you can, of course, register your SPI implementations simply by implementing them as CDI managed beans:
- You still can use the `ServiceLoader` to register your beans.

Example writing a CurrencyProvider with CDI

```
@Priority(100)
@Singleton
public class MyCurrencyProvider implements CurrencyProvider {

  private final Map<String, MyCurrency> currencies = new HashMap<>();

  public MyCurrencyProvider(){
    this.currencies.put("MSCU", new MyCurrency());
  }

  [...]
}
```

4.5.5. 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.

5. Meta-Data Contexts and Query Models

5.1. Overview

The JSR uses a unified meta-data model to support more advanced use cases, which are not explicitly specified. The main reason for not specifying these aspects is that they are highly use case and organization dependent. In general there are two flavors of meta-data used throughout the JSR:

1. *Contexts* provide additional information on value types or services, such as currencies, amounts, conversions or formats. Contexts are accessible directly from the corresponding value types, by calling methods named `getContext()`.
2. *Queries* models a generic and flexible way to configure/parametrize services for accessing currencies, amounts, conversions or formats. Queries can be passed to the different accessor singletons, and also are forwarded to the SPI implementations effectively providing the data/services required.

Similarly there are two abstract base classes provided:

1. `AbstractContext` models an abstract base type, which is extended by all context implementations within this JSR, such as `MonetaryContext`, `CurrencyContext`, `RoundingContext`, `ProviderContext`, `ConversionContext`, `AmountFormatContext`.

2. `AbstractQuery` models an abstract base query, which is extended by all query implementations within this JSR, such as `MonetaryAmountFactoryQuery`, `CurrencyQuery`, `RoundingQuery`, `ConversionQuery`, `AmountFormatQuery`.

The following sections give further information on these concepts.

5.2. AbstractContext

The abstract class `AbstractContext` models a base type, which is extended by all context implementations within this JSR, such as `MonetaryContext`, `CurrencyContext`, `RoundingContext`, `ProviderContext`, `ConversionContext`, `AmountFormatContext`. Basically this class models a generic data container, which provides a type safe mechanism for retrieving meta-data:

Class AbstractContext

```
public abstract class AbstractContext
implements Serializable{
  ...
  public String getProviderName();

  public Boolean getBoolean(String key);
  public Integer getInt(String key);
  public Long getLong(String key);
  public Float getFloat(String key);
  public Double getDouble(String key);
  public String getText(String key);
  public <T> T get(Class<T> type);
  public <T> T get(String key, Class<T> type);

  public boolean isEmpty();
  public Class<?> getType(String key);
  public Set<String> getKeys(Class<?> type)
}
```

Hereby

- each context instance is related to a provider, that created the context, accessible from `getProviderName()`.

- additional attributes can be set, which models a type safe interface for adding properties, without

duplicating artifacts or creating non portable dependencies.

  - identified by the attribute's type.
  - identified an arbitrary literal key

- the `getXXX, get` methods only return values of the resulting type is assignment compatible, so no class cast exceptions do occur.

The classes extending this class hereby are thread-safe and immutable:

- `javax.money.CurrencyContext`
- `javax.money.MonetaryContext`
- `javax.money.RoundingContext`
- `javax.money.conversion.ProviderContext`
- `javax.money.conversion.ConversionContext`
- `javax.money.format.AmountFormatContext`

Creation of context instances is encapsulated using corresponding builder instances:

- `javax.money.CurrencyContextBuilder`
- `javax.money.MonetaryContextBuilder`
- `javax.money.RoundingContextBuilder`
- `javax.money.conversion.ProviderContextBuilder`
- `javax.money.conversion.ConversionContextBuilder`
- `javax.money.format.AmountFormatContextBuilder`

The builders hereby extend `AbstractContestBuilder`, discussed in the following section.

5.3. Abstract Class AbstractContextBuilder

The abstract class `AbstractContextBuilder` models a base builder type, which is extended by all context builder implementations within this JSR, such as `MonetaryContextBuilder, CurrencyContextBuilder, RoundingContextBuilder, ProviderContextBuilder, ConversionContextBuilder, AmountFormatContextBuilder`. Basically this class models a generic builder, which provides a type safe mechanism for storing arbitrary meta-data:

Class AbstractContextBuilder

```
public abstract class AbstractContextBuilder<B extends AbstractContextBuilder, C extends
AbstractContext>
implements Serializable{
  public B setProviderName(String provider);

  public B set(Object value);
  public <T> B set(T value, Class<? extends T> type);
  public B set(String key, Object value);
  public <T> B set(String key, T value, Class<? extends T> type)
  public B set(String key, int value);
  public B set(String key, long value);
  public B set(String key, float value);
  public B set(String key, double value);

  public B importContext(AbstractContext context, boolean overwriteDuplicates);
  public B importContext(AbstractContext context);
  public B removeAttributes(String... keys);

  public abstract C build();

}
```

5.4. Abstract Class AbstractQuery

The abstract class `AbstractQuery` models a base query type, which is extended by all query implementations within this JSR, such as `MonetaryAmountFactoryQuery`, `CurrencyQuery`, `RoundingQuery`, `ConversionQuery`, `AmountFormatQuery`. Basically this class models a generic query, which provides a type safe mechanism for storing arbitrary query-data:

Abstract class AbstractQuery

```
public abstract class AbstractQuery extends AbstractContext{

  protected AbstractQuery(AbstractQueryBuilder builder);

  public List<String> getProviderNames();
  public Class<?> getTargetType();

}
```

As seen above a query is basically the same as a context, thus inheriting all attribute container functions. The query provides basic query properties:

- a query contains the provider names that defines the provider chain to be used for answering the

query. If not set the corresponding default provider chain must be used. The providers available are accessible by calling `getProviderNames()` on the corresponding singleton accessors/SPI interfaces, e.g. `Monetary.getCurrencyProviderNames`, `Monetary.getRoundingProviderNames`, `MonetaryConversions.getProviderNames` etc.

- a query may defines a target type, hereby defining the target result type expected.

Hereby `AbstractQuery` inherits most of the functionality from the `AbstractContext` super class.

5.5. Abstract Class AbstractQueryBuilder

The abstract class `AbstractQueryBuilder` models a base builder type, which is extended by all query builder implementations within this JSR, such as `MonetaryAmountFactoryQueryBuilder`, `CurrencyQueryBuilder`, `RoundingQueryBuilder`, `ConversionQueryBuilder`, `AmountFormatQueryBuilder`. Basically this class models a generic builder, which provides a type safe mechanism for storing arbitrary query-data:

Abstract class AbstractQueryBuilder

```
public abstract class AbstractQueryBuilder<B extends javax.money.AbstractQueryBuilder, C
extends AbstractQuery>
        extends AbstractContextBuilder<B,C>{
   ...
   public B setProviderNames(String... providers);
   public B setProviderNames(List<String> providers);
   public B setTargetType(Class<?> type);
}
```

Similarly to `AbstractQuery` also here most of the functionality is inherited by the `Abstract Class AbstractContextBuilder` super class.

6. Implementation Recommendations

6.1. 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.

6.2. 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 `MonetaryAmount`.
- 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.

6.3. 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.

6.3.1. 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 [Therefore the default reference implementation class, `Money`,is based on `BigDecimal` and allows to explicitly configure its `MathContext` used on creation.]. 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 rounding for presentation purposes), so there is no such concept as *THE* rounding for a monetary amount.

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).

6.3.2. 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).

6.3.3. 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`).

7. Examples

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

7.1. 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 `Monetary` 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:

Class Money

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

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

Default MonetaryContext settings

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

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

Example Usage of MonetaryAmountFactory

```
MonetaryAmountFactory<Money> fact = Monetary.getAmountFactory(Money.class);
Money m = fact.withCurrency("USD").with(200.50).create();
```

Also a generic `MonetaryAmount` instance can be accessed using a raw factory (hereby it depends on the configured default amount factory, which effective type instance is returned):

Example Usage MonetaryAmountFactory

```
MonetaryAmount amt = Monetary.getDefaultAmountFactory()
                        .withCurrency("USD").with(200.50).create();
```

Still we can evaluate the effective amount's type effectively:

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

But in general, we do not need to know the exact implementation in most cases, since we can access amount meta-data as a `MonetaryContext`, This meta-data provides information, such as the maximal precision, maximal scale supported by the type's implementation as well as other attributes. Refer to The Monetary Context for more details.

Example Usage MonetaryContext

```
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 arithmetic operations in both above scenarios works similar as it is when using `java.math.BigDecimal`:

Example Usage Monetary Arithmetic

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

Also the sample above illustrates how algorithmic operations can be chained together using a fluent API. As mentioned also external functionality can be chained, e.g. using instances of `MonetaryOperator`:

Example Function Chaining [`MonetaryFunctions` is not part of the JSR, its just for illustration purposes.]

```
MonetaryAmount amt = ...;
amt = amt.multiply(2.12345).with(Monetary.getDefaultRounding())
        .with(MonetaryFunctions.minimal(100)).
        .multiply(2.12345).with(Monetary.getDefaultRounding())
        .with(MonetaryFunctions.percent(23));
```

7.1.1. Numeric Precision and Scale

Since the `Money` implementation class, which is part of the reference implementation, internally uses `java.math.BigDecimal` the numeric capabilities match exact the capabilities of `BigDecimal`. When accessing `MonetaryAmountFactory` instances it is possible to configure the `MathContext` effectively used (by default `Money` uses `MathContext.DECIMAL64`).:

Example Configuring a `MonetaryAmountFactory`, using the RI class `Money` as example.

```
MonetaryAmountFactory<Money> fact = Monetary.getAmountFactory(
   MonetaryAmountFactoryQueryBuilder.of(Money.class)
     .set(new MathContext(250, RoundingMode.HALF_DOWN)).build()
);
// Creates an instance of Money with the given MathContext
MonetaryAmount m1 = fact.setCurrency("CHF").setNumber(250.34).create();
Money m2 = fact.setCurrency("CHF").setNumber(250.34).create();
```

7.1.2. Extending the API

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

- When the basic arithmetic 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);
  }
}
```

Hereby the amount type implicitly will throw an `ArithemticException` if the numeric capabilities are not capable of creating the result required.

- In case where the basic operations are not sufficient anymore, or it is more convenient to do a calculation externally, it is still not necessary to cast to any implementation type, since
  - the numeric capabilities can be evaluated using the `MonetaryContext`. On `[MonetaryAmountFactory]` both the default and the maximal supported `MonetaryContext` can be accessed.
  - 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.

Below is a rather academical example of a `MonetaryOperator` that simply converts any given amount to an amount with the same numeric value, but with XXX (undefined) as currency:

Simple example of a `MonetaryOperstor` *using the* `MonetaryAmountFactory` *provided.*

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

7.2. 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 `FastMoney` with `Money`:

Usage Example - FastMoney

```
MonetaryAmountFactory<FastMoney> fact = Monetary.getAmountFactory(FastMoney.class);
// Creates an instance of Money with the given MathContext
MonetaryAmount m1 = fact.setCurrency("CHF").setNumber(250.34).create();
FastMoney m2 = fact.setCurrency("CHF").setNumber(250.34).create();
```

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

The MonetaryContext of FastMoney

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

7.3. Calculating a Total

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

Usage Example Calculating a Total

```
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:

Example of total/add method

```
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 Monetary.getAmountFactory(amountType).with(total).create();
   }

}
```

IMPORTANT

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 to this example:

- 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:

Example Using Total/add method

```
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.

7.4. 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:

```
C / ((1+r)^n)
```

Hereby

- `n` is the time of the cash flow (in periods)
- `r` is 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.
- `C` is 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 follows:

Example Using Total/add method

```
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`:

Example Implementing a 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 pre-calculation of divisor matrices. Now given the `MonetaryOperator` a present value can be calculated as follows:

Example Using a Financial Function

```
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));
```

7.5. 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:

Example Currency Conversion

```
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 `ConversionQuery` with additional parameters for conversion, e.g.:

```
MonetaryAmount inCHF =...;
CurrencyConversion conv = MonetaryConversions.getConversion(ConversionQueryBuilder.of()
        .setProviders("CS", "EZB", "IMF")
        .setTermCurrency("EUR")
        .set(MonetaryAmount.class, inCHF, MonetaryAmount.class)
        .set(LocalDate.of(2008, 1, 1))
        .setRateType(RateType.HISTORIC)
        .set(StockExchange.NYSE) // custom type
        .set("contractId", "AA-1234.2")
        .build());
MonetaryAmount inEUR = inCHF.with(conv);
```

APPENDIX

Bibliography

[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] http://www.iso20022.org

[JodaMoney] http://www.joda.org/joda-money/

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

[JSR354] http://jcp.org/en/jsr/detail?id=354

[source] Public Source Code Repository on GitHub: http://github.com/JavaMoney, Branch/Tag matching updated PDR is {version}

Links

- JSR 354 on jcp.org
- JavaMoney Project on Java.net
  - JSR 354 API GitHub Repository
  - Moneta RI GitHub Repository
- Java Practices about Representing Money
- Working with Money in Java
- Java currency by Roedy Green, Canadian Mind Products
- UOMo Business, based on ICU4J and concepts by JScience Economics]
- ICU4J Uses Number for all operations and internal storage in its Money type.
- 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.
- 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

- http://timeandmoney.sourceforge.net/ [Eric Evans Time and Money Library]
- http://bitcoinj.github.io/ [Bitcoin Java Client]