JSR 354 Money & Currency TCK User Guide

Anatole Tresch

Table of Contents

JSR 354 *Java Money* defines the Java API for managing monetary amounts and currencies in Java. As other JSRs beside the reference implementations also other third parties can provide their implementations. This document will describe how this technical compatibility kit (TCK) can be configured and used to determine the compatibility of an implementation with the specification and reference implementation.

This document

This is a user guide that describes all relevant aspects of the JSR 354 Java Money TCK, for using this TCK with any reference implementation.

1. Introduction to Java Money

NOTE You may also read the JSR 354 API specification available here.

1.1. Overview

The technical compatibility kit (TCK) of a JSR ensures that implementations of a JSR implement the specification correctly and help to ensure also behavioural interoperability between different implementations. So the TCK for JSR 354 Money and Currency ensures exactly this for implementations of the Money & Currency JSR. The TCK hereby is open sourced under the Apache 2 licence. Nevertheless we would be highly interested to keep track of implementation of this JSR. So if you are are planning or even writing on an implementation of this JSR, drop a mail to the JSR's mailing list on java.net or the JSR's spec lead.

1.2. Downloading and Installing

Basically the JSR 354 API as well as the TCK and its dependencies are available as maven artifacts. Hereby JSR 354 defines a Java 7 based API, which comprises almost all functionality that but also comes with a Java 8 API, which adds Java 8 related additional features (default methods that makes developer's life easier):

- `javax.money:money-api:1.0:jar` contains the full Java 8 based API
- `javax.money:money-api-bp:1.0:jar` contains the Java SE pre8 compatible API.
- `org.javamoney:moneta:<version>:jar` contains the Java 8 compatible reference implementation (RI)
- `org.javamoney:moneta-bp:<version>:jar` contains the Java SE pre8 compatible reference implementation (RI)
- `org.javamoney:money-tck:<version>:jar` contains the technological compatibility kit (TCK), built against the Java 7 API, which is compatible with the Java 8 version.

Summarizing you should add the following dependencies:

Required Dependencies

```
<dependency>
    <groupId>org.javamoney</groupId>
    <artifactId>money-tck</artifactId>
    <version>1.0</version> <!-- or newer, if available. -->
</dependency>
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.8.5</version>
</dependency>
<dependency>
    <groupId>${impl.groupId}</groupId>
    <artifactId>${impl.artifactId}</artifactId>
    <type>${impl.packageType}</type>
    <version>${impl.version}</version>
</dependency>

<!-- Not necessary, if your implementation has this dependency in compile scope... -->
<dependency>
    <groupId>javax.money</groupId>
    <artifactId>money-api</artifactId>
    <!-- Java 7: artifactId>money-api-bp</artifactId -->
    <type>jar</type>
    <version>1.0</version>
</dependency>
```

Also we have added example maven project for Java 8 and example maven project for Java 7 that can be used as a starting point. All you have to do is clone it from GitHub and exchange the implementation under test with your own.

1.3. Main Design Decisions

Basically the TCK is a set of tests executed with TestNG. Basic reason for using TestNG is that this library also is used in OpenJDK. Additionally in case of tests are failing you should see a detailed message and a reference to the according section in the specification. The audit of the test coverage can be found here.

1.4. Component Loading and Bootstrapping

The TCK itself is built as a self-executable TestNG program. To execute the TCK basically two prerequisites are necessary: . you must have all required libraries on the classpath. This includes the JSR's API, your implementation under test, the TCK and TestNG, including all required dependencies. . You have to provide a small class, implementing the `org.javamoney.tck.JSR354TestConfiguration`

interface and register it with the JDK's `java.util.ServiceLoader` mechanism.

This can be achieved in different ways:

- The most easiest way to run the tests is to create a small maven project, where you add the JSR 354 API, your implementation and the TCK as dependency. The only thing to be done is executing the test suite by running `org.javamoney.tck.TCKRunner.main`, e.g. from your IDE.
- Or you can execute the TCK by starting it on the console:

Running the TCK

```
java -cp ... org.javamoney.tck.TCKRunner
```

Hereby the TCK allows to set additional system properties:

- `outputDir` allows to set the output directory used by Test NG.
- `reportFile` allows to define the file, which contains the TCK's test result summary (your TCK report).
- `verbose` allows to enable TestNG to log in verbose mode.

So you can also run the TCK with the options above:

Running the TCK with options

```
java -cp ... -Dverbose=true -DoutputDir=c:/temp -DreportFile=./tck-report.txt
org.javamoney.tck.TCKRunner
```

2. The org.javamoney.tck.JSR354TestConfiguration Interface

As mentioned before it is required that you implement an instance of the TCK's `org.javamoney.tck.JSR354TestConfiguration` interface:

Interface JSR354TestConfiguration

```
/**
 * Libraries that implement this JSR and want to be tested with this TCK must implement
this
 * interface and register it using the {@link ServiceLoader}.
 *
 * @author Anatole Tresch
 */
public interface JSR354TestConfiguration{

    /**
     * Return a collection with all {@link MonetaryAmount} classes that are implemented.
The list
     * must not be empty and should contain <b>every</b> amount class implemented.<br/>
     * This enables the TCK to check in addition to the basic implementation compliance,
if
     * according {@link MonetaryAmountFactoryProviderSpi} are registered/available
correctly.
     *
     * @return a collection with all implemented amount classes, not null.
     */
    Collection<Class> getAmountClasses();

    /**
     * List a collection of {@link CurrencyUnit} implementation.<br/>
     * This enables the TCK to check the basic implementation compliance,
     *
     * @return
     */
    Collection<Class> getCurrencyClasses();


    /**
     * This method allows to let instances of MonetaryOperator to be tested for
requirements and recommendations.
     *
     * @return the list of operators to be checked, not null. It is allowed to return an
empty list here, which will
     * disable TCK tests for MonetaryOperator instances.
     */
    Collection<MonetaryOperator> getMonetaryOperators4Test();

}
```

Following find the example implementation that is used for testing the *moneta* reference implementation:

TestSetup for the Moneta Reference Implementation

```
/**
 * Created by Anatole on 14.06.2014.
 */
public final class MonetaTCKSetup implements JSR354TestConfiguration{

    @Override
    public Collection<Class> getAmountClasses() {
            return Arrays
                    .asList(new Class[]{FastMoney.class,FastMoney.class});
    }

    @Override
    public Collection<Class> getCurrencyClasses() {
        try{
            return Arrays
                    .asList(new Class[] { Class.forName(
"org.javamoney.moneta.internal.JDKCurrencyAdapter")});
        }
        catch(ClassNotFoundException e){
            e.printStackTrace();
            throw new RuntimeException("Currency class not lodable:
org.javamoney.moneta.internal.JDKCurrencyAdapter");
        }
    }

    @Override
    public Collection<MonetaryOperator> getMonetaryOperators4Test(){
        List<MonetaryOperator> ops = new ArrayList<>();
        ops.add(MonetaryFunctions.majorPart());
        ops.add(MonetaryFunctions.minorPart());
        ops.add(MonetaryFunctions.percent(BigDecimal.ONE));
        ops.add(MonetaryFunctions.percent(3.5));
        ops.add(MonetaryFunctions.permil(10.3));
        ops.add(MonetaryFunctions.permil(BigDecimal.ONE));
        ops.add(MonetaryFunctions.permil(10.5, MathContext.DECIMAL32));
        ops.add(MonetaryFunctions.reciprocal());
        ops.add(Monetary.getRounding());
        ops.add(MonetaryConversions.getConversion("EUR"));
        return ops;
    }

}
```

You must register your implementation with the `java.util.ServiceLoader`. This is done by adding the following configuration file to your classpath:

Configuring the TCK Test Setup

```
META-INF
  \_ services
      \_ org.javamoney.tck.JSR354TestConfiguration
```

The *org.javamoney.tck.JSR354TestConfiguration* file should contain exactly one line with the fully qualified class name of your implementation provided:

Contents of the org.javamoney.tck.JSR354TestConfiguration file

```
com.mycomp.mymoney-lib.TCKSetup
```

2.1. Checking the Results

Test execution is logged verbosely on the runtime console. Additionally after TCK execution you should find a folder `tck-results` within your working directory, containing the TestNG result in html (index.html) or xml format. Find following an example output from testing the reference implementation:

Example TCK Console Output

```
... (Basic Trace Output)

================================================
JSR354-TCK - Commons, version 1.0
Total tests run: 219, Failures: 0, Skips: 0
================================================


********************************************************************************************
**** JSR 354 - Money & Currency, Technical Compatibility Kit, version 1.0
********************************************************************************************

Executed on Sun Aug 24 00:38:44 CEST 2014

[SUCCESS] 4.2.1 Ensure registered CurrencyUnit classes are
Comparable.(ModellingCurrenciesTest#testCurrencyClassesComparable)
[SUCCESS] 4.2.1 Ensure registered CurrencyUnit classes implement
hashCode.(ModellingCurrenciesTest#testCurrencyClassesEqualsHashcode)
[SUCCESS] 4.2.1 Test currencies provided have correct ISO 3-letter currency
codes.(ModellingCurrenciesTest#testEnforce3LetterCode4ISO)
[SUCCESS] 4.2.1 Ensure TCK has CurrencyUnit classes
configured.(ModellingCurrenciesTest#testEnsureCurrencyUnit)
[SUCCESS] 4.2.1 Test currencies provided equal at least currencies from
java.util.Currency.(ModellingCurrenciesTest#testEqualISOCurrencies)
[SUCCESS] 4.2.1 Test currencies provided have correct default fraction digits and numeric
code.(ModellingCurrenciesTest#testISOCodes)
[SUCCESS] 4.2.1 Ensure registered CurrencyUnit classes implement
equals.(ModellingCurrenciesTest#testImplementsEquals)
[SUCCESS] 4.2.1 Ensure registered CurrencyUnit classes are
serializable.(ModellingCurrenciesTest#testImplementsSerializable)
[SUCCESS] 4.2.1 Ensure registered CurrencyUnit classes are
immutable.(ModellingCurrenciesTest#testIsImmutable)
[SUCCESS] 4.2.2 For each amount class, test
absolute().(ModellingMonetaryTest#testAbsolute)
[SUCCESS] 4.2.2 For each amount class, check m1.add(m2), m1, m2 = mixed
fractions.(ModellingMonetaryTest#testAddMixedFractions)
[SUCCESS] 4.2.2 For each amount class, check m1.add(m2), m1, m2 = mixed
ints.(ModellingMonetaryTest#testAddMixedIntegers)
[SUCCESS] 4.2.2 For each amount class, check m1.add(m2), m1 <0,
m2<0.(ModellingMonetaryTest#testAddNegativeIntegers)
[SUCCESS] 4.2.2 For each amount class, check m1.add(m2), m2 is
fraction.(ModellingMonetaryTest#testAddPositiveFractions)
[SUCCESS] 4.2.2 For each amount class, check m1.add(m2), m1 >0,
m2>0.(ModellingMonetaryTest#testAddPositiveIntegers)
[SUCCESS] 4.2.2 For each amount class, ensure ArithemticException is thrown when adding
exceeding values.(ModellingMonetaryTest#testAdd_ExceedsCapabilities)
```

```
[SUCCESS] 4.2.2 For each amount class, ensure currency compatibility is
working.(ModellingMonetaryTest#testAdd_IncompatibleCurrencies)
[SUCCESS] 4.2.2 For each amount class, ensure NullPointerException is thrown when calling
m.add(null).(ModellingMonetaryTest#testAdd_Null)
[SUCCESS] 4.2.2 For each amount class, ensure m2 = m1,add(0) ->
m1==m2.(ModellingMonetaryTest#testAdd_Zero)
[SUCCESS] 4.2.2 Ensure amount can be created with all default
currencies.(ModellingMonetaryTest#testCurrencyCode)
[SUCCESS] 4.2.2 For each amount class, ensure correct
division.(ModellingMonetaryTest#testDivide)
[SUCCESS] 4.2.2 For each amount class, ensure correct
divideAndRemainder().(ModellingMonetaryTest#testDivideAndRemainder)
[SUCCESS] 4.2.2 For each amount class, ensure divideAndRemainder(null) throws a
NullPointerException.(ModellingMonetaryTest#testDivideAndRemainderNull)
[SUCCESS] 4.2.2 For each amount class, ensure divideAndRemainder(1) returns same
instance.(ModellingMonetaryTest#testDivideAndRemainderOne)
[SUCCESS] 4.2.2 For each amount class, ensure correct
divideAndRemainderZero().(ModellingMonetaryTest#testDivideAndRemainderZero)
[SUCCESS] 4.2.2 For each amount class, ensure divide by null throws
NullPointerException.(ModellingMonetaryTest#testDivideNull)
[SUCCESS] 4.2.2 For each amount class, ensure divide 1 returns same
instance.(ModellingMonetaryTest#testDivideOne)
[SUCCESS] 4.2.2 For each amount class, ensure correct division with int
values.(ModellingMonetaryTest#testDivideToIntegralValue)
[SUCCESS] 4.2.2 For each amount class, ensure divide(0) throws
ArithmeticException.(ModellingMonetaryTest#testDivideZero)
[SUCCESS] 4.2.2 Ensure Monetary.getAmountTypes() is not null and not
empty.(ModellingMonetaryTest#testEnsureMonetaryAmount)
[SUCCESS] 4.2.2 Ensure amounts created return correct
getMonetaryContext().(ModellingMonetaryTest#testGetMonetaryContext)
[SUCCESS] 4.2.2 Ensure amounts created return correct
getNumber().(ModellingMonetaryTest#testGetNumber)
[SUCCESS] 4.2.2 For each amount class, test iis
immutable.(ModellingMonetaryTest#testImmutable)
[SUCCESS] 4.2.2 For each amount class, test is
Comparable.(ModellingMonetaryTest#testImplementComparable)
[SUCCESS] 4.2.2 For each amount class, test implements
equals().(ModellingMonetaryTest#testImplementsEquals)
[SUCCESS] 4.2.2 For each amount class, test implements
hashCode().(ModellingMonetaryTest#testImplementsHashCode)
[SUCCESS] 4.2.2 For each amount class, test
isNegative().(ModellingMonetaryTest#testIsNegative)
[SUCCESS] 4.2.2 For each amount class, test
isNegativeOrZero().(ModellingMonetaryTest#testIsNegativeOrZero)
[SUCCESS] 4.2.2 For each amount class, test
isPositive().(ModellingMonetaryTest#testIsPositive)
[SUCCESS] 4.2.2 For each amount class, test
isPositiveOrZero().(ModellingMonetaryTest#testIsPositiveOrZero)
```

```
[SUCCESS] 4.2.2 For each amount class, test isZero().(ModellingMonetaryTest#testIsZero)
[SUCCESS] 4.2.2 For each amount class, test isZero(),
advanced.(ModellingMonetaryTest#testIsZeroAdvanced)
[SUCCESS] 4.2.2 For each amount class, access factory and create
amounts.(ModellingMonetaryTest#testMonetaryAmountFactories)
[SUCCESS] 4.2.2 For each amount class, check multiple instances are not
equal.(ModellingMonetaryTest#testMonetaryAmountFactories_CreateWithCurrencies)
[SUCCESS] 4.2.2 For each amount class, check new amounts with explcit
MonetaryContext.(ModellingMonetaryTest#testMonetaryAmountFactories_CreateWithMonetaryCont
ext)
[SUCCESS] 4.2.2 For each amount class, check new amounts are not equal for different
currencies and
contexts.(ModellingMonetaryTest#testMonetaryAmountFactories_CreateWithMonetaryContextNumb
erAndCurrency)
[SUCCESS] 4.2.2 For each amount class, access factory and create amounts, ensure amounts
are equal if
theyshould.(ModellingMonetaryTest#testMonetaryAmountFactories_InstancesMustBeEqual)
[SUCCESS] 4.2.2 For each amount class, check new amounts are not
equal.(ModellingMonetaryTest#testMonetaryAmountFactories_InstantesMustBeNotEqual)
[SUCCESS] 4.2.2 For each amount class, check
isEqualTo().(ModellingMonetaryTest#testMonetaryAmount_isEqualTo)
[SUCCESS] 4.2.2 For each amount class, check isEqualTo(), regardless different
MonetaryContext
instances.(ModellingMonetaryTest#testMonetaryAmount_isEqualToRegardlessMonetaryContext)
[SUCCESS] 4.2.2 For each amount class, check isEqualTo(), regardless implementation
type.(ModellingMonetaryTest#testMonetaryAmount_isEqualToRegardlessType)
[SUCCESS] 4.2.2 For each amount class, check
isGreaterThan().(ModellingMonetaryTest#testMonetaryAmount_isGreaterThan)
[SUCCESS] 4.2.2 For each amount class, check
isGreaterThanOrEquals().(ModellingMonetaryTest#testMonetaryAmount_isGreaterThanOrEquals)
[SUCCESS] 4.2.2 For each amount class, check
isLessThan().(ModellingMonetaryTest#testMonetaryAmount_isLessThan)
[SUCCESS] 4.2.2 For each amount class, check
isLessThanOrEqualTo().(ModellingMonetaryTest#testMonetaryAmount_isLessThanOrEqualTo)
[SUCCESS] 4.2.2 For each amount class, ensure multiplication with exceeding values throws
ArithmeticException.(ModellingMonetaryTest#testMultiplyExceedsCapabilities)
[SUCCESS] 4.2.2 For each amount class, ensure multiplication of null throws
NullPointerException.(ModellingMonetaryTest#testMultiplyNull)
[SUCCESS] 4.2.2 For each amount class, ensure multiplication by one returns same
instance.(ModellingMonetaryTest#testMultiplyOne)
[SUCCESS] 4.2.2 For each amount class, ensure correct multiplication of decimal
values.(ModellingMonetaryTest#testMultiply_Decimals)
[SUCCESS] 4.2.2 For each amount class, ensure correct multiplication of int
values.(ModellingMonetaryTest#testMultiply_Integral)
[SUCCESS] 4.2.2 For each amount class, test negate().(ModellingMonetaryTest#testNegate)
[SUCCESS] 4.2.2 For each amount class, test query().(ModellingMonetaryTest#testQuery)
[SUCCESS] 4.2.2 For each amount class, test query(), MonetaryQuery throws exception,
MonetaryException expected.(ModellingMonetaryTest#testQueryInvalidQuery)
```

```
[SUCCESS] 4.2.2 For each amount class, test query(null), NullPointerException
expected.(ModellingMonetaryTest#testQueryNull)
[SUCCESS] 4.2.2 For each amount class, ensure correct results for
remainder.(ModellingMonetaryTest#testRemainder)
[SUCCESS] 4.2.2 For each amount class, ensure remainder(null), throws
NullPointerException.(ModellingMonetaryTest#testRemainderNull)
[SUCCESS] 4.2.2 For each amount class, ensure remainder(0), double, throws
ArithmeticException.(ModellingMonetaryTest#testRemainderZero_Double)
[SUCCESS] 4.2.2 For each amount class, ensure remainder(0), long, throws
ArithmeticException.(ModellingMonetaryTest#testRemainderZero_Long)
[SUCCESS] 4.2.2 For each amount class, ensure remainder(0), Number, throws
ArithmeticException.(ModellingMonetaryTest#testRemainderZero_Number)
[SUCCESS] 4.2.2 For each amount class, ensure scaleByPowerOfTen(1) returns correct
results.(ModellingMonetaryTest#testScaleByPowerOfTen)
[SUCCESS] 4.2.2 For each amount class, test signum().(ModellingMonetaryTest#testSignum)
[SUCCESS] 4.2.2 For each amount class, ensure correct subtraction of mixed
fractions.(ModellingMonetaryTest#testSubtractMixedFractions)
[SUCCESS] 4.2.2 For each amount class, ensure correct subtraction of mixed
ints.(ModellingMonetaryTest#testSubtractMixedIntegers)
[SUCCESS] 4.2.2 For each amount class, ensure correct subtraction of negative
ints.(ModellingMonetaryTest#testSubtractNegativeIntegers)
[SUCCESS] 4.2.2 For each amount class, ensure correct subtraction of positive
fractions.(ModellingMonetaryTest#testSubtractPositiveFractions)
[SUCCESS] 4.2.2 For each amount class, ensure correct subtraction of positive
ints.(ModellingMonetaryTest#testSubtractPositiveIntegers)
[SUCCESS] 4.2.2 For each amount class, ensure subtraction with exceeding capabilities
throws ArithmeticException.(ModellingMonetaryTest#testSubtract_ExceedsCapabilities)
[SUCCESS] 4.2.2 For each amount class, ensure subtraction with invalid currency throws
MonetaryException.(ModellingMonetaryTest#testSubtract_IncompatibleCurrencies)
[SUCCESS] 4.2.2 For each amount class, ensure subtraction with null throws
NullPointerException.(ModellingMonetaryTest#testSubtract_Null)
[SUCCESS] 4.2.2 For each amount class, ensure subtraction of 0 returns same
instance.(ModellingMonetaryTest#testSubtract_Zero)
[SUCCESS] 4.2.2 For each amount class, test with().(ModellingMonetaryTest#testWith)
[SUCCESS] 4.2.2 For each amount class, test
with().(ModellingMonetaryTest#testWith4ProvidedOperators)
[SUCCESS] 4.2.2 Bad case: For each amount class, test with(), operator throws
exception.(ModellingMonetaryTest#testWithInvalidOperator)
[SUCCESS] 4.2.2 Bad case: For each amount class, test with(null), expected
NullPointerException.(ModellingMonetaryTest#testWithNull)
[SUCCESS] 4.2.2 Bad case: For each amount class, test with(), operator throws
exception.(ModellingMonetaryTest#testWithNull4ProvidedOperators)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances are accessible for all amount
types under test.(CreatingMonetaryTest#testAccessToMonetaryAmountFactory)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create zero amounts from a
factory with an invalid
currency.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateAmountsWithInvalidCurrency)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create zero amounts from a
```

```
factory with an invalid
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateAmountsWithInvalidMo
netaryContext)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts, with
no currency, expect
MonetaryException.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateNegativeInvalidCon
text_BadCase)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts, with
invalid currency, expect
MonetaryException.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateNegativeInvalidCur
rency_BadCase)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts, with
no currency, expect
MonetaryException.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateNegativeNoCurrency
_BadCase)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create positive
amounts.(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveAmountsWitCurrencies
)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create positive amounts with explicit
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveAmountsWithC
ontexts)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create positive amounts using doubles
with explicit MonetaryContext
(precision/scale).(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveAmountsWit
hContexts2)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create positive amounts using BigDecimal
with explicit MonetaryContext
(precision/scale).(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveAmountsWit
hContexts3)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create positive amounts using
invalid numbers, expecting ArithemticException
thrown.(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveAmountsWithInvalidNum
ber)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts with
an invalid currency, expecting MonetaryException
thrown.(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveInvalidContext_BadCas
e)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts with
an invalid currency, expecting MonetaryException
thrown.(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveInvalidCurrency_BadCa
se)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts
without currency, expecting MonetaryException
thrown.(CreatingMonetaryTest#testMonetaryAmountFactoryCreatePositiveNoCurrency_BadCase)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances support creation of 0 amounts,
with explicit
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateZeroAmountsWithDiffC
ontexts)
```

```
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances support creation of 0 amounts,
with different explicit
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateZeroAmountsWithDiffC
ontexts2)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances support creation of 0 amounts,
with different explicit MonetaryContext (precision,
scale).(CreatingMonetaryTest#testMonetaryAmountFactoryCreateZeroAmountsWithDiffContexts3)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances support creation of 0
amounts.(CreatingMonetaryTest#testMonetaryAmountFactoryCreateZeroAmountsWithDiffCurrencie
s)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances accessible for all amount types
under test return correct min/max
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryMinMaxCapabilities)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances accessible for all amount types
under test return correct min/max MonetaryContext (min <=
max).(CreatingMonetaryTest#testMonetaryAmountFactoryMinMaxCapabilities_Compare)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create negative
amounts.(CreatingMonetaryTest#testMonetaryAmountFactoryNegativePositiveAmountsWitCurrenci
es)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create negative amounts, with explicit
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryNegativePositiveAmountsWit
hContexts)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create negative amounts, with explicit
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryNegativePositiveAmountsWit
hContexts2)
[SUCCESS] 4.2.6 For each MonetaryAmount Factory: Create negative amounts, with explicit
MonetaryContext.(CreatingMonetaryTest#testMonetaryAmountFactoryNegativePositiveAmountsWit
hContexts3)
[SUCCESS] 4.2.6 Bad case: For each MonetaryAmount Factory: Create negative amounts, with
invalid numeric value, expect
ArithmeticException.(CreatingMonetaryTest#testMonetaryAmountFactoryNegativePositiveAmount
sWithInvalidNumber)
[SUCCESS] 4.2.6 Ensure MonetaryAmountFactory instances accessible for all amount types
under test return correct amount
type.(CreatingMonetaryTest#testMonetaryAmountFactoryReturnsCorrectType)
[SUCCESS] 4.2.2 Checks if a correct Double value is returned, no truncation is allowed to
be performed.(ExternalizingNumericValueTest#testDoubleNegative)
[SUCCESS] 4.2.3 Check if a correct double value is returned, truncation is allowed to be
performed (but is not
necessary).(ExternalizingNumericValueTest#testDoubleValueWithTruncationZero)
[SUCCESS] 4.2.3 Checks if a correct double value is returned, truncation is allowed to be
performed.(ExternalizingNumericValueTest#testDoubleWithTruncationNegative)
[SUCCESS] 4.2.3 Checks if a correct Integer value is returned, no truncation is allowed
to be performed.(ExternalizingNumericValueTest#testIntegerNegative)
[SUCCESS] 4.2.3 Check if a correct integer value is returned, truncation is allowed to be
performed. Check should be done for every JDK type
supported.(ExternalizingNumericValueTest#testIntegerValueWithTruncationZero)
[SUCCESS] 4.2.3 Check if a correct integer value is returned, truncation is allowed to be
```

```
performed..(ExternalizingNumericValueTest#testIntegerWithTruncationNegative)
[SUCCESS] 4.2.3 Check if a correct integer value is returned, no truncation is  allowed
to be performed.(ExternalizingNumericValueTest#testIntegerZero)
[SUCCESS] 4.2.3 Checks if a correct negative long value is returned, no truncation is
allowed to be performed.(ExternalizingNumericValueTest#testLongNegative)
[SUCCESS] 4.2.3 Check if a correct long value is returned, truncation is allowed to be
performed. Check should be done for every JDK type
supported.(ExternalizingNumericValueTest#testLongValueWithTruncationZero)
[SUCCESS] 4.2.3 Checks if a correct long value is returned, truncation is allowed to be
performed.(ExternalizingNumericValueTest#testLongWithTruncationNegative)
[SUCCESS] 4.2.3 Check if a correct long zero value is returned, no truncation is  allowed
to be performed.(ExternalizingNumericValueTest#testLongZero)
[SUCCESS] 4.2.3 Ensure NumberValue numberValue() works
correnctly.(ExternalizingNumericValueTest#testNumberTypeNegative)
[SUCCESS] 4.2.3 Checks if number type is not null and returning a concrete (no abstract
class or interface).(ExternalizingNumericValueTest#testNumberTypeZero)
[SUCCESS] 4.2.3 Checks if a correct long value is returned, truncation is allowed to be
performed. Check should be done for every JDK
type.(ExternalizingNumericValueTest#testNumberValueWithTruncationNegative)
[SUCCESS] 4.2.3 Checks if a correct double value is returned, truncation is allowed to be
performed. Check should be done for every JDK
type.(ExternalizingNumericValueTest#testNumberValueWithTruncationNegative_Double)
[SUCCESS] 4.2.3 Checks if a correct double value is returned, truncation is allowed to be
performed. Check should be done for every JDK
type.(ExternalizingNumericValueTest#testNumberValueWithTruncationNegative_Float)
[SUCCESS] 4.2.3 Checks if a correct int value is returned, truncation is allowed to be
performed. Check should be done for every JDK
type.(ExternalizingNumericValueTest#testNumberValueWithTruncationNegative_Integer)
[SUCCESS] 4.2.3 Checks if a correct Number value is returned, truncation is allowed to be
performed. Check should be done for every JDK
type.(ExternalizingNumericValueTest#testNumberValueWithTruncationNegative_Long)
[SUCCESS] 4.2.3 Checks if a correct double value is returned, truncation is allowed to be
performed. Check should be done for every JDK
type.(ExternalizingNumericValueTest#testNumberValueWithTruncationNegative_Short)
[SUCCESS] 4.2.3 Check if a correct Number value is returned, truncation is allowed to be
performed. Check should be done for every JDK type
supported.(ExternalizingNumericValueTest#testNumberValueWithTruncationZero)
[SUCCESS] 4.2.3 Check if a correct long zero value is returned, no truncation is  allowed
to be performed.(ExternalizingNumericValueTest#testNumberValueZero)
[SUCCESS] 4.2.3 Check if a correct number value is returned, truncation is  allowed to be
performed. Check should be done for every JDK type
supported.(ExternalizingNumericValueTest#testNumberWithTruncationNegative)
[SUCCESS] 4.2.3 Test correct precision values, including border
cases.(ExternalizingNumericValueTest#testPrecisionNegative)
[SUCCESS] 4.2.3 Ensure NumberValue getPrecision() works
correctly.(ExternalizingNumericValueTest#testPrecisionValues)
[SUCCESS] 4.2.3 Check if a correct precision value is returned. Check should be done for
every JDK type supported.(ExternalizingNumericValueTest#testPrecisionZero)
```

```
[SUCCESS] 4.2.3 Amount types do not return a NumberValue of
null.(ExternalizingNumericValueTest#testReturningNumberValueIsNotNull)
[SUCCESS] 4.2.3 Test correct scale values, including border
cases.(ExternalizingNumericValueTest#testScaleNegative)
[SUCCESS] 4.2.3 Ensure NumberValue getScale() works
correctly.(ExternalizingNumericValueTest#testScaleValues)
[SUCCESS] 4.2.3 Check if a correct scale value is returned. Check should be done for
every JDK type supported.(ExternalizingNumericValueTest#testScaleZero)
[SUCCESS] 4.2.3 Ensure NumberValue doubleValue(), doubleValueExact() provide correct
values.(ExternalizingNumericValueTest#testValidDouble)
[SUCCESS] 4.2.3 Ensure NumberValue doubleValue() is
truncated.(ExternalizingNumericValueTest#testValidDoubleWithTruncation)
[SUCCESS] 4.2.3 Ensure NumberValue intValue(), intValueExact() provide correct
values.(ExternalizingNumericValueTest#testValidInteger)
[SUCCESS] 4.2.3 Ensure NumberValue intValue() is
truncated.(ExternalizingNumericValueTest#testValidIntegerWithTruncation)
[SUCCESS] 4.2.3 Ensure NumberValue longValue(), longValueExact() provide correct
values.(ExternalizingNumericValueTest#testValidLong)
[SUCCESS] 4.2.3 Ensure NumberValue longValue() is
truncated.(ExternalizingNumericValueTest#testValidLongWithTruncation)
[SUCCESS] 4.2.3 Ensure NumberValue asType(BigDecimal.class) provides correct
values.(ExternalizingNumericValueTest#testValidNumberBD)
[SUCCESS] 4.2.3 Ensure NumberValue asType(BigInteger.class) provides correct
values.(ExternalizingNumericValueTest#testValidNumberBI)
[SUCCESS] 4.2.3 Ensure NumberValue byteValue() is
truncated.(ExternalizingNumericValueTest#testValidNumberWithTruncation_Byte)
[SUCCESS] 4.2.3 Ensure NumberValue doubleValue() is
truncated.(ExternalizingNumericValueTest#testValidNumberWithTruncation_Double)
[SUCCESS] 4.2.3 Ensure NumberValue floatValue() is
truncated.(ExternalizingNumericValueTest#testValidNumberWithTruncation_Float)
[SUCCESS] 4.2.3 Ensure NumberValue intValue() is truncated
correctly.(ExternalizingNumericValueTest#testValidNumberWithTruncation_Integer)
[SUCCESS] 4.2.3 Ensure NumberValue shortValue() is
truncated.(ExternalizingNumericValueTest#testValidNumberWithTruncation_Short)
[SUCCESS] 4.2.4 Ensures the result of all operators under test is of the same class as
the input.(FunctionalExtensionPointsTest#testOperatorReturnTypeEqualsParameter)
[SUCCESS] 4.2.7 Access named roundings and ensure TCK named roundings are
registered.(AccessingCurrenciesAmountsRoundingsTest#testAccessCustomRoundings)
[SUCCESS] 4.2.7 Ensure Monetary instances are available, for all registered
currencies.(AccessingCurrenciesAmountsRoundingsTest#testAccessRoundingsForCustomCurrencie
s_Default)
[SUCCESS] 4.2.7 Ensure Monetary instances are available, also for any custom currency
(not
registered).(AccessingCurrenciesAmountsRoundingsTest#testAccessRoundingsForCustomCurrenci
es_Explicit)
[SUCCESS] 4.2.7 Expected NullPointerException accessing a rounding with
'Monetary.getRounding(null)'.(AccessingCurrenciesAmountsRoundingsTest#testAccessRoundings
ForCustomCurrencies_Explicit_Null)
```

```
[SUCCESS] 4.2.7 Ensure NullPointerException is thrown for
'Monetary.getRounding((RoundingContext)
null)'.(AccessingCurrenciesAmountsRoundingsTest#testAccessRoundingsWithMonetaryContext_Nu
ll)
[SUCCESS] 4.2.7 Ensure correct MonetaryRounding returned for a mathematical
RoundingQuery.(AccessingCurrenciesAmountsRoundingsTest#testAccessRoundingsWithRoundingCon
text)
[SUCCESS] 4.2.7 Test if Monetary provides all ISO related entries similar to
java.util.Currency.(AccessingCurrenciesAmountsRoundingsTest#testAllISOCurrenciesAvailable
)
[SUCCESS] 4.2.7 Test if Monetary provides all locale related entries similar to
java.util.Currency.(AccessingCurrenciesAmountsRoundingsTest#testAllLocaleCurrenciesAvaila
ble)
[SUCCESS] 4.2.7 Ensure a default MonetaryAmountFactory is
available.(AccessingCurrenciesAmountsRoundingsTest#testAmountDefaultType)
[SUCCESS] 4.2.7 Ensure correct query function, Monetary.getAmountFactories should return
factoryfor explicit acquired amount
types.(AccessingCurrenciesAmountsRoundingsTest#testAmountQueryType)
[SUCCESS] 4.2.7 Ensure amount factories are accessible for all types available in
Monetary.(AccessingCurrenciesAmountsRoundingsTest#testAmountTypesInstantiatable)
[SUCCESS] 4.2.7 Ensure amount classes to test are setup and registered/available in
Monetary.(AccessingCurrenciesAmountsRoundingsTest#testAmountTypesProvided)
[SUCCESS] 4.2.7 Test if Monetary provides correct ISO related entries similar to
java.util.Currency.(AccessingCurrenciesAmountsRoundingsTest#testCorrectISOCodes)
[SUCCESS] 4.2.7 Test if Monetary provides correct locale related entries similar to
java.util.Currency.(AccessingCurrenciesAmountsRoundingsTest#testCorrectLocales)
[SUCCESS] 4.2.7 Test if Monetary provides customized locale identified
currencies.(AccessingCurrenciesAmountsRoundingsTest#testCustomCurrencies)
[SUCCESS] 4.2.7 Access custom roundings and ensure correct
functionality.(AccessingCurrenciesAmountsRoundingsTest#testCustomRoundings)
[SUCCESS] 4.2.7 Ensure MonetaryException is thrown for accessing invalid named
rounding.(AccessingCurrenciesAmountsRoundingsTest#testCustomRoundings_Foo)
[SUCCESS] 4.2.7 Ensure NullPointerException is thrown for Monetary.getRounding((String)
null).(AccessingCurrenciesAmountsRoundingsTest#testCustomRoundings_Null)
[SUCCESS] 4.3.1 Access Conversion to term currency code XXX for all providers that
support according conversion, ifavailable a non-null CurrencyConversion must be
provided.(MonetaryConversionsTest#testConversionsAreAvailable)
[SUCCESS] 4.3.1 Access Conversion by query to term currency XXX for all providers that
support according conversion, ifavailable a non-null CurrencyConversion must be
provided.(MonetaryConversionsTest#testConversionsAreAvailableWithQuery)
[SUCCESS] 4.3.1 Access and test the default conversion provider
chain.(MonetaryConversionsTest#testDefaultProviderChainIsDefined)
[SUCCESS] 4.3.1 Access and test the default conversion provider chain, by accessing a
defaultCurrencyConversion for term CurrencyUnit
CHF.(MonetaryConversionsTest#testDefaultProviderChainIsDefinedDefault)
[SUCCESS] 4.3.1 Access and test the default conversion provider chain, by accessing a
defaultCurrencyConversion for term currency code
CHF.(MonetaryConversionsTest#testDefaultProviderChainIsDefinedDefault2)
```

```
[SUCCESS] 4.3.1 Access and test the default conversion provider chain, by accessing a
defaultCurrencyConversion for
ConversionQuery.(MonetaryConversionsTest#testDefaultProviderChainIsDefinedDefaultWithCont
ext)
[SUCCESS] 4.3.1 Test if all ExchangeRateProvider instances returns valid
ProviderContext.(MonetaryConversionsTest#testProviderMetadata)
[SUCCESS] 4.3.1 Test if all CurrencyConversion instances returns valid ConversionContext,
accessed by currency code.(MonetaryConversionsTest#testProviderMetadata2)
[SUCCESS] 4.3.1 Test if all CurrencyConversion instances returns valid ConversionContext,
accessed by ConversionQuery/currency
code.(MonetaryConversionsTest#testProviderMetadata2WithContext)
[SUCCESS] 4.3.1 Test if all CurrencyConversion instances returns valid ConversionContext,
accessed by CurrencyUnit.(MonetaryConversionsTest#testProviderMetadata3)
[SUCCESS] 4.3.1 Test if all CurrencyConversion instances returns valid ConversionContext,
accessed by
ConversionQuery/CurrencyUnit.(MonetaryConversionsTest#testProviderMetadata3WithContext)
[SUCCESS] 4.3.1 Ensure at least one conversion provider is available, TestRateProvider
must be present.(MonetaryConversionsTest#testProvidersAvailable)
[SUCCESS] 4.3.1 Bad case: Access invalid ExchangeRateProvider, expect MonetaryException
thrown, using default provider chain.(MonetaryConversionsTest#testUseInvalidProvider)
[SUCCESS] 4.3.1 Bad case: Access invalid ExchangeRateProvider, expect MonetaryException
thrown, using explicit
provider.(MonetaryConversionsTest#testUseInvalidProviderWithinChain)
[SUCCESS] 4.3.3 Test access of Conversion Rates, using TCK provided rate
provider.(ExchangeRatesAndRateProvidersTest#testAccessKnownRates)
[SUCCESS] 4.3.3 Test access to exchange rates from TestRateProvider, using target
CUrrencyUnit.(ExchangeRatesAndRateProvidersTest#testAccessKnownRatesAndContext)
[SUCCESS] 4.3.3 Test access to exchange rates from TestRateProvider, using target
currency code.(ExchangeRatesAndRateProvidersTest#testAccessKnownRatesWithCodes)
[SUCCESS] 4.3.3  Test access to conversion rates, including known factor, using
TestRateProvider.(ExchangeRatesAndRateProvidersTest#testAccessKnownRatesWithCodesAndConte
xt)
[SUCCESS] 4.3.3 Test access to conversion rate for currency codes, using default
provider.(ExchangeRatesAndRateProvidersTest#testAccessRates_IdentityRatesWithCodes)
[SUCCESS] 4.3.3 Test access to identity conversion rate for CurrencyUnits, using default
provider(ExchangeRatesAndRateProvidersTest#testAccessRates_IdentityRatesWithUnits)
[SUCCESS] 4.3.3 Test access to conversion rate for CurrencyQuery, using default
provider.(ExchangeRatesAndRateProvidersTest#testAccessRates_IdentityRatesWithUnitsAndCont
ext)
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with invalid base currency
code.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_InvalidSourceCurrency)
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with null
ConversionQuery.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_InvalidSourceCurrency
AndContext)
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with invalid term currency
code.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_InvalidTargetCurrency)
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with null base currency
code.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_NullSourceCurrency)
```

```
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with null base
CurrencyUnit.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_NullSourceCurrencyUnit)
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with null term currency
code.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_NullTargetCurrency)
[SUCCESS] 4.3.3 Bad case: try accessing exchange rates with null term
CurrencyUnit.(ExchangeRatesAndRateProvidersTest#testInvalidUsage_NullTargetCurrencyUnit)
[SUCCESS] 4.3.3 Ensure additional ConversionQuery data is passed correctly to
SPIs.(ExchangeRatesAndRateProvidersTest#testPassingOverConversionContextToSPIs)
[SUCCESS] 4.3.2 Test successful conversion for CHF -> FOO, using
TestRateProvider.(ConvertingAmountsTest#testConversion)
[SUCCESS] 4.3.2 Test correct ExchangeRate is returned for CHF -> FOO, using
TestRateProvider.(ConvertingAmountsTest#testConversionComparedWithRate)
[SUCCESS] 4.3.2 Bad case: Access CurrencyConversion with a CurrencyUnit==null, ensure
NullPointerException is thrown.(ConvertingAmountsTest#testNullConversion1)
[SUCCESS] 4.3.2 Bad case: Access CurrencyConversion with a currency code==null, ensure
NullPointerException is thrown.(ConvertingAmountsTest#testNullConversion2)
[SUCCESS] 4.3.2 Bad case: Try CurrencyConversion to an inconvertible (custom) currency
(FOOANY), ensure CurrencyConversionException is
thrown.(ConvertingAmountsTest#testUnsupportedConversion)
[SUCCESS] 4.3.4 Test correct rate evaluation for different conversion provider chains,
with historic rates.(ProviderChainsTest#testCorrectRateEvaluationInChainHistoric)
[SUCCESS] 4.3.4 Test correct rate evaluation for different conversion provider
chains.(ProviderChainsTest#testCorrectRateEvaluationInChain_diffProviders)
[SUCCESS] 4.3.4 Test correct rate evaluation for different conversion provider chains,
with duplicate provider
entries.(ProviderChainsTest#testCorrectRateEvaluationInChain_sameProviders)
[SUCCESS] 4.3.4 Test availability of TCK provided
providers.(ProviderChainsTest#testTCKRateChainAvailability)
[SUCCESS] 4.4.1 Ensures for each locale defined by DecimalFormat.getAvailableLocales() a
MonetaryFormats.getAmountFormat(AmountFormatQuery) returns a
formatter.(FormattingMonetaryTest#testAmountStyleOf)
[SUCCESS] 4.4.1 Formats amounts using all available
locales.(FormattingMonetaryTest#testFormattingIsIndependentOfImplementation)
[SUCCESS] 4.4.1 Ensures for each locale defined by DecimalFormat.getAvailableLocales() a
MonetaryAmountFormat instance is provided.(FormattingMonetaryTest#testGetAmountFormat)
[SUCCESS] 4.4.1 Ensures for each locale defined by DecimalFormat.getAvailableLocales() a
MonetaryFormats.isAvailable(Locale) is
true.(FormattingMonetaryTest#testGetAvailableLocales)
[SUCCESS] 4.4.1 Ensures all Locales defined by DecimalFormat.getAvailableLocales() are
available for monetary formatting.(FormattingMonetaryTest#testLocalesSupported)
[SUCCESS] 4.4.1 Ensures the system.s default locale is supported for
MonetaryAmountFormat.(FormattingMonetaryTest#testNoDepOnAmountImplementation)
[SUCCESS] 4.4.1 Test formats and parses (round-trip) any supported amount type for each
supported Locale, using different format
queries.(FormattingMonetaryTest#testParseDifferentStyles)
[SUCCESS] 4.4.1 Test formats and parses (round-trip) any supported amount type for each
supported Locale.(FormattingMonetaryTest#testParseIsIndependentOfImplementation)
[SUCCESS] 4.4.1 Test formats and parses (round-trip) any supported amount type for each
```

```
supported Locale, checks results for different
currencies(FormattingMonetaryTest#testParseWithDifferentCurrencies)

JSR 354 TCK, version 1.0 Summary
-----------------------------------------

TOTAL TESTS EXECUTED : 221
TOTAL TESTS SKIPPED  : 0
TOTAL TESTS SUCCESS  : 221
TOTAL TESTS FAILED   : 0

-- JSR 354 TCK finished --

================================================
Custom suite
Total tests run: 1, Failures: 0, Skips: 0
================================================


Process finished with exit code 0
```

2.2. Contacts

Basically you can use the JSR's public mailing list to get in contact or write an email to the JSR's spec lead.

If you have improvements or fixes, create a pull request on GitHub.