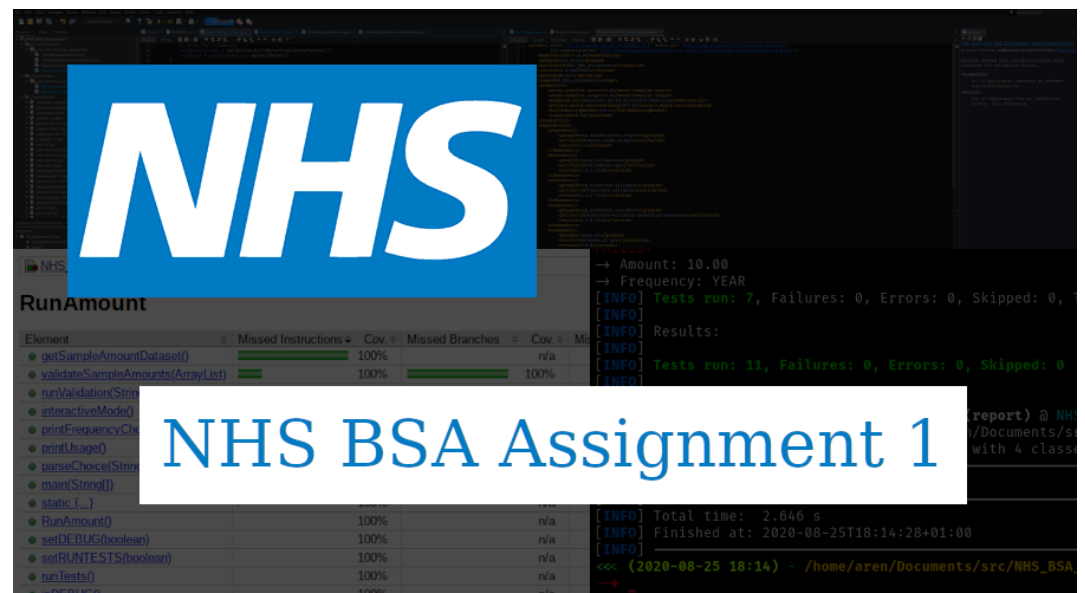

NHS BSA Assignment 1 Developer Documentation

JS-303 Annotation and Constraint Validation

Aren Tyr



August 2020

Contents

1	Metadata	3
2	Synopsis	3
3	Download/Files	3
4	Release	3
5	Requirements Engineering	4
5.1	Terms (Frequency & Amount)	5
5.2	Acceptance Criteria (Defined Parameters in isolation)	5
5.2.1	Invalid Amount	5
5.2.2	Null Frequency	5
5.2.3	Weekly	6
5.2.4	Monthly	6
5.3	Acceptance Criteria (“validation as valid”/“validation as invalid”)	6
5.3.1	Validated as valid	6
5.3.2	Validated as invalid	7
5.4	Missing/Undefined Acceptance Criteria	7
5.4.1	Missing Term definitions	7
5.4.2	Missing/Incomplete Validation criteria (Design Choices)	8
6	Implementation Notes	9
6.1	Currency Verification	9
6.2	Frequency Enumeration Mapping	9
6.3	Exact Pence Calculation	9
6.4	RegularAmount class	10
7	Testing	11
7.1	JUnit Tests	11
7.2	JaCoCo Code Coverage Analysis	12
8	Usage	12
9	Build/Compile	13
10	Issues	13
11	License	15
12	Appendix	15
12.1	Full Source Code Listing	15
12.1.1	CheckRegularAmount.java	15
12.1.2	CheckRegularAmountValidator.java	16
12.1.3	RegularAmount.java	21
12.1.4	RegularAmountTest.java	24
12.1.5	RunAmount.java	32
12.1.6	RunAmountTest.java	42

1 Metadata

- **Document Revision:** 2020-08-25
- **Document Version:** 1.0.0
- **Software Revision:** 2020-08-25
- **Software Version:** 1.0.0

2 Synopsis

This project involved creating a JSR-303 Java Bean annotation and an associated `ConstraintValidator` class, for the purposes of determining currency payments that yield exactly divisible pence amounts (i.e. a scale of two/two decimal points, representing UK currency pence sterling).

3 Download/Files

1. Download an executable “fat” (i.e. self-contained) JAR of the project here: **nhs-assign1-cli-v1.0.jar**.
2. Download the full project documentation here: **nhs-assign1-full-doc-v1.0.zip**.

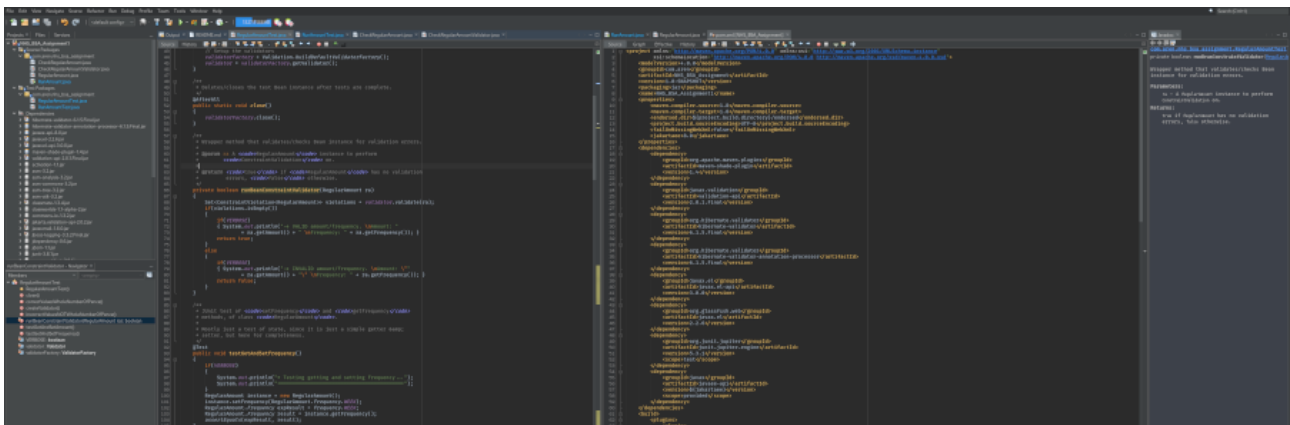
Note that the documentation zip file includes the following:

- Javadoc - for all source files, including test classes
 - JaCaCo - code coverage report
 - Documentation PDF - (this file).
3. `git clone https://github.com/ArenT1981/bsa_constraint_validation_assign1` to get the entire repository. `jar/zip` files are found under the `assets/v1.0` subdirectory.
 4. Alternatively, the next section provides links to the individual components.

4 Release

The following is provided/has been produced:

1. Full source code, with a build based around Maven (`pom.xml`). VCS using `git` with commit history.
Compiled JAR file: **nhs-assign1-cli-v1.0.jar**.



2. Full Javadoc documentation across all classes: **nhs-assign1-javadoc-v1.0.zip**.

PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SEARCH:

Package com.aren.nhs_bsa_assignment

Class Summary

Class	Description
CheckRegularAmountValidator	Java Bean validator class that implements ConstraintValidator.
RegularAmount	Based on the supplied template class with a few modifications/additions.
RunAmount	A driver class used to demonstrate/test/examine the behaviour of CheckRegularAmount/CheckRegularAmountValidator classes/Java Bean ConstraintValidator.

Enum Summary

3. Full unit testing using JUnit.

```

→ Amount: 10.00
→ Frequency: YEAR
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.227 s - in com.aren.nhs_bsa_assignment.RunAmountTest
[INFO] Results:
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- jacoco-maven-plugin:0.8.5:report (report) @ NHS_BSA_Assignment ---
[INFO] Loading execution data file /home/aren/Documents/src/NHS_BSA_Assignment/target/jacoco.exec
[INFO] Analyzed bundle 'NHS_BSA_Assignment1' with 4 classes
[INFO] BUILD SUCCESS
[INFO] Total time: 2.646 s
[INFO] Finished at: 2020-08-25T18:14:28+01:00
[INFO]
<<< (2020-08-25 18:14) - /home/aren/Documents/src/NHS_BSA_Assignment >>>

```

4. Code coverage analysis/report using JaCoCo: nhs-assign1-jacoco-v1.0.zip.

NHS_BSA_Assignment1 > com.aren.nhs_bsa_assignment > RunAmount [Sessions](#)

RunAmount

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
getSampleAmountDataset()		100%		n/a	0	1	0	120	0	1
validateSampleAmounts(ArrayList)		100%		100%	0	7	0	20	0	1
runValidation(String, String)		100%		100%	0	10	0	25	0	1
interactiveMode()		100%		100%	0	2	0	13	0	1
printFrequencyChoiceMenu()		100%		n/a	0	1	0	11	0	1
printUsage()		100%		n/a	0	1	0	11	0	1
parseChoice(String)		100%		100%	0	5	0	16	0	1
main(String[])		100%		100%	0	2	0	4	0	1
static {...}		100%		n/a	0	1	0	2	0	1
RunAmount()		100%		n/a	0	1	0	1	0	1
setDEBUG(boolean)		100%		n/a	0	1	0	2	0	1
setRUNTESTS(boolean)		100%		n/a	0	1	0	2	0	1
runTests()		100%		n/a	0	1	0	2	0	1
isDEBUG()		100%		n/a	0	1	0	1	0	1

5. Development documentation (this file): nhs-assign1-doc-v1.0.pdf.

5 Requirements Engineering

The first remark to be made here is that neither acceptance criteria for the defined/individual parameters/data types, nor the acceptance criteria for the “validation as valid”/“validation as invalid”, represents a complete or unambiguous set of design constraints. Both are incomplete, and therefore requires making particular judgement/interpretations as to how to “interpolate” the undefined/missing criteria. More than one such interpretation is possible, and arguably several equally defensible interpretations exist. The most important consideration is that whichever interpretation is made, that it represents a logically defensible choice, and moreover that it can be implemented in a coherent/consistent manner across the codebase with clearly understood software behaviours.

5.1 Terms (Frequency & Amount)

1. “A regular amount consists of a frequency and amount.”
2. “A frequency defines a regular interval at which a payment is made or income received.”
3. “Frequency may be one of:”
 - WEEK = ?
 - TWO_WEEK = 2
 - FOUR_WEEK = 4
 - MONTH = ?
 - QUARTER = 13
 - YEAR = 52
4. “An amount contains a value of pounds and pence entered as a String with an optional decimal point.”

The actual numerical mappings for these Frequency enumerated types is specified in the “Validated as valid” and “Validated as invalid” definitions later. For clarity, however, we are stating them here. Note that both WEEK and MONTH are undefined, hence why they are indicated with question marks here.

5.2 Acceptance Criteria (Defined Parameters in isolation)

5.2.1 Invalid Amount

Criteria:

GIVEN any Frequency
WHEN a non-numeric or blank Amount is provided
THEN no validation error

Analysis:

Under most circumstances we would typically want to filter our blank or non-numeric values with an annotation over the member variable, since we can immediately identify them as invalid, however (presumably by design), such a straightforward approach is denied here, since we must allow these to get passed onto the validation logic within the validator class.

5.2.2 Null Frequency

Criteria:

GIVEN any Amount
WHEN a null Frequency is provided
THEN no validation error

Analysis:

As per “Invalid Amount” above, we can not simply restrict the acceptable `enum` values with a simple annotation over the member variable.

5.2.3 Weekly

Criteria:

GIVEN a WEEK Frequency
WHEN any Amount is provided
THEN no validation error

Analysis:

Once again, any given Amount (regardless of whether it constitutes a valid numerical currency amount) and the WEEK Frequency will pass through to the next stage of validation.

5.2.4 Monthly

Criteria:

GIVEN a MONTH Frequency
WHEN any Amount is provided
THEN no validation error

Analysis:

As per **Weekly** above. Note, however, that a MONTH is deeply problematical in terms of its relationship to a WEEK value. For precisely what number of weeks does an actual month represent? Averaged throughout the year, it neither constitutes an exact number of days nor, therefore, a whole ratio of weeks. By convention, a “month” is often taken to represent four weeks; however using this value actually only results in month representing the correct number of weeks for the month of February *only*, and only then for non-leap years, where it instead has twenty-nine days, which of course is not divisible into four whole weeks of seven days.

Therefore, as a mathematical/numerical entity, a “month” is a problem, and therefore any usage or non-usage needs to be qualified by whatever particular set of constraints/design decisions we impose on the software. Which decisions we make regarding it are dependent on our intended outcome/usage for the system; i.e. they are particular to the specifics of the usage context.

5.3 Acceptance Criteria (“validation as valid”/“validation as invalid”)

5.3.1 Validated as valid

Criteria:

GIVEN a Frequency is in the set TWO_WEEK, FOUR_WEEK, QUARTER, YEAR
AND an associated Number of Weeks is 2, 4, 13, 52 respectively
WHEN a Amount that divides by the Number of Weeks to a whole number of pence is provided
THEN no validation error

Analysis:

The Frequency enumerated type is mapped to an associated Number that acts as the divisor/denominator. The Amount is the numerator. The result of this expression, if it yields a whole pence amount (i.e. yields an *exact* value within two decimal places), is that it is “validated as valid” and no validation error is produced.

This criteria builds upon the constraints elaborated as “Defined Parameters” above, which as previously alluded, are incomplete. These missing constraints will be discussed below. Here, we can note that for the listed frequencies (TWO_WEEK, FOUR_WEEK, QUARTER, YEAR), they all map unambiguously to a given numerical value and can be implemented without issue as per the Amount requirement of a whole number of pence.

5.3.2 Validated as invalid

Criteria:

GIVEN a Frequency is in the set TWO_WEEK, FOUR_WEEK, QUARTER, YEAR
AND an associated Number of Weeks is 2, 4, 13, 52 respectively
WHEN a Amount that does not divide by the Number of Weeks to a whole number of pence is provided
THEN a validation error is produced

Analysis:

This is effectively the inverse of the above: i.e. given the same input constraints, if the amount does not exactly divide into an exact weekly amount of pence, we therefore validate it as “invalid”.

5.4 Missing/Undefined Acceptance Criteria

Clearly the acceptance criteria are incomplete, as indicated above. This is a combination of undefined/missing terms, and insufficiently specified criteria (edge cases). In order to implement working code, however, specific choices/decisions need to be made regarding these omissions. These shall now be adumbrated, together with their rationale.

5.4.1 Missing Term definitions

Neither WEEK nor MONTH is explicitly defined. The following choices were made:

1. WEEK

A week is not defined; however given that TWO_WEEK and FOUR_WEEK are defined as mapping to the numerical values 2 and 4 respectively, there seems to be no possible logical objection to defining a WEEK as mapping to value 1, as makes intuitive sense.

2. MONTH

A month, as discussed previously, has no possibility of mapping to an exact number of weeks, except in the case of February in non-leap years (28 days = 4 weeks). This makes it eminently unsuitable as a type that maps to a value, for what value do we map it to, or by what process do we reach a numerical determination for it? Several possible mitigations could be implemented: you could take a rounded numerical average of weeks, and use this as the divisor; you could crudely define it as 4; or you could make a determination based on the exact calendar month/year in consideration, meaning that the exact value is specifically determined by the month in question in the particular year. The final option is just to reject it entirely as a weekly divisor.

None of these approaches is particularly satisfactory. Using “4” means that most of the time it is in fact patently wrong, in terms of weeks represented, though it may be an acceptable approximation, and has the virtue of implementation simplicity. Using the average ($365 / 12 = 30.416666\dots$) is not particularly helpful/useful for our particular problem here, since in any case it will almost certainly never result in any exact pence amounts in any case (though in practice, in other systems, such a constraint would not be an issue, since you could simply round up to the next nearest pence, and would therefore provide an extremely usable choice). Finally, implementing based on the exact year/month at current execution time considerably increases the code complexity. This may be justifiable in some applications/contexts; it would depend upon the real world situation that the system is operating in, and the particular behaviour you desire.

Since no specific criteria was stated, the deliberate design choice made here was simply to reject it as a criterion for yielding a valid weekly amount. In the code, therefore, by design, it later maps to -1 and as a result subsequently causes the amount to be “validated as invalid”. Note that a MONTH frequency, as a type, is

accepted *per se*, in accordance with the requirements; it only *subsequently* yields an **invalid amount** (“validated as invalid”).

5.4.2 Missing/Incomplete Validation criteria (Design Choices)

1. **WEEK** is entirely missing as either “validated as valid” or “validated as invalid”, much as it is missing as an explicitly defined term (see above). However, since we have chosen to map it to the value 1 (see above), it seems consistent to also allow its validation determination based on the **result of this division by 1**. The result, naturally, is largely tautological, since any value divided by one is just itself; in practice it will therefore always cause a valid currency value to validate as valid, subject to no violations of any other constraints (see below).
2. The term definition: “An amount contains a value of pounds and pence entered as a String with an optional decimal point” is underspecified/under-determined. A sensible interpretation would accept that both “100” and “100.00” meet this definition. But what of “100.”, or “100.1”? Here, the choice was made to be fairly strict: if a pence amount is specified in the string, it must be fully qualified in accordance to our conventional way of representing currency values in prices/on display. Therefore, “400” is acceptable, since it would universally be interpreted as “£400” (or, being pedantic, “£400.00”). Similarly for a value such as “550.80”. Contrariwise, “550.8”, or “550.” are by design, not accepted. Whilst we could reasonably interpret “550.8” as £550.80, or “550.” as £550.00, the fact is either of these representations would be considered, at best, a sloppy representation of a UK sterling currency value, or, at worst, entirely wrong (after all, suppose “550.8” masks a rounding of “550.83”; therefore it should actually be stated as “550.83”, not simply “550.8”).
3. **MONTH** as discussed above is not an unambiguous weekly divisor. Therefore, in the code, it will always subsequently “validate as invalid”. So, for example, an Amount of £40.00 over a **MONTH** frequency will therefore be “invalid”, even though some might conventionally want to interpret that as equating to a weekly amount of £10.00. This is by design.
4. Technically neither a **QUARTER** nor **YEAR** are strictly either 13 weeks or 52 weeks exactly, since a year itself is not precisely 365 days. Such considerations can be bypassed here, however, by adhering to the specified criteria of 13 and 52. In general usage our conventional calendar is “close enough”, even allowing for some variability due to its inexact division into whole units. Only in the case of a **MONTH** is there a significant deviation, hence why it was deliberately rejected here; 31 days is quite a significant deviation from 28 as a divisor.
5. An Amount of 0 (or 0.00 if you prefer) is an undefined/an edge case. Do we validate it as valid after acceptance? Since it is meaningless (at least in our context here, and computationally, for that matter) to attempt to divide by zero, it is therefore rejected during the validation algorithm and will always “validate as invalid”. Similarly for negative values (though one could make a legitimate argument on the basis of it effectively showing a weekly amount *owed*, rather than an amount to be paid).
6. No constraint is put upon the the maxima (or minima, for that matter) input Amount. Allowing an unconstrained input amount is a potential security risk at worst, or otherwise a bad implementation choice, since there are clear limits on the extent of currency amounts we would legitimately be interested in calculating. The system should not therefore accept an Amount value that has, say, 250,000 *digits*, since it clearly does not represent any sane currency payment. Indeed, the limit was set at 11 characters in total (since the input is a String), which is still exceptionally generous, as it would allow currency payments of (at worst) “99999999.99” or (at best) “99999999999”. So values up to £99,999,999.99 if specified with pence, i.e. 99 million pounds. 11 characters seemed a sensible limit. So this particular limit is enforced by a `@Length` annotation on the Amount member variable.

6 Implementation Notes

6.1 Currency Verification

The verification/acceptance of sane currency values (as per our term definition above; see point 2. in the previous section) is accomplished through the application of a regular expression that pattern matches the Amount String. This method returns a boolean flag that indicates whether the value is a sane/acceptable representation of a currency:

```
private boolean isValidNumericAmount(String inputAmount)
{
    // Regular expression that requires number starts with at least one digit
    // and optionally has a decimal point & two digits
    return inputAmount.matches("[0-9]+(\\.\\d{2})?$");
}
```

Here the optional second pattern matching group, if present, must have *precisely* two further digits after a point “.”. This allows us to accept “300” or “300.30”, but not “300.”, “300.3”, “300.300”, “300.30000”... etc.

6.2 Frequency Enumeration Mapping

The `Frequency` is mapped to a divisor/denominator value through a simple `switch` expression:

```
private BigDecimal validFrequencyDivisor(Frequency inputFreq)
{
    switch(inputFreq)
    {
        case WEEK:
            return new BigDecimal("1.00");
        case TWO_WEEK:
            return new BigDecimal("2.00");
        case FOUR_WEEK:
            return new BigDecimal("4.00");
        case MONTH:
            // We will stipulate as INVALID "MONTH" since !=
            // any specific number of weeks...
            if(DEBUG)
            {
                System.out.println("MONTH value requested, MONTH does not "
                    + "define a WEEKLY amount.");
            }
            return new BigDecimal("-1.00");
        case QUARTER:
            return new BigDecimal("13.00");
        case YEAR:
            return new BigDecimal("52.00");
        default:
            return new BigDecimal("-1.00");
    }
}
```

Note the deliberate mapping of `MONTH` to “-1.00” for reasons previously discussed.

6.3 Exact Pence Calculation

We need to exercise care with our underlying data types when making the actual pence determination. Java’s in-built `double` type, like any floating point data type, is subject to rounding errors which can lead to unin-

tended/unexpected results. For dealing with numbers where we want exact values, or exercise control over whether and how rounding occurs, we should therefore use Java's specialist `BigDecimal` type/object.

Initially, an approach (courtesy of an online StackExchange thread) counting decimal places based on its conversion to a `String`, and then counting the number of characters after "." was used; if this exceeded 2, then we could reject it, as it required greater resolution than one numerical pence sterling:

```
private int getNumberOfDecimalPlaces(BigDecimal bigDecimal)
{
    String string = bigDecimal.stripTrailingZeros().toPlainString();
    int index = string.indexOf(".");
    return index < 0 ? 0 : string.length() - index - 1;
}
```

This solution works fine, however it is rather inelegant, especially as it turns out that `BigDecimal` has various attributes for dealing with rounding/rounding errors, and allows you to explicitly specify the desired scale/precision of the number, so it was replaced with this (excerpt):

```
if(numerator.compareTo(new BigDecimal("0")) > 0
    && divisor.compareTo(new BigDecimal("1")) >= 0)
{
    // Use RoundingMode.UNNECESSARY to throw ArithmeticException if
    //_any_ rounding occurs outside of the scale of two decimal points
    result = numerator.divide(divisor, 2, RoundingMode.UNNECESSARY);
}
```

The `if` statement is here principally to enforce our desired numerator (an amount $> \pounds 0.00$) and denominator (a frequency ≥ 1.00), since we wish to reject combinations that violate either of these two conditions, and then the real work is done by the `RoundingMode.UNNECESSARY` rounding mode in conjunction with a scale (precision) of 2. That is, any number that requires more than two decimal places to *exactly* represent (i.e. less than a pence), throws an `ArithmeticException` which we can catch and therefore use as the basis for rejecting it as an invalid Amount.

6.4 RegularAmount class

Apart from adding Javadoc throughout the class, the only real addition was to use/overload a custom constructor for convenience, and also make sure that we do not have any uninitialised values being passed through to the validator class:

```
@CheckRegularAmount
public class RegularAmount
{
    public RegularAmount()
    {
        this("-1.00", RegularAmount.Frequency.WEEK);
    }

    public RegularAmount(String amt, Frequency frequency)
    {
        this.amount = amt;
        this.frequency = frequency;
    }

    ...
}
```

7 Testing

7.1 JUnit Tests

Believing it to work and being certain it works are two separate things; therefore the requirement for extensive testing was met by writing full JUnit tests for the code, with a large set of test values that try all of the obvious edge cases, awkward values, together, of course, with values that are known to be either good (should validate as valid) or bad (should validate as invalid). `correctValuesWholeNumberOfPence()` tests a large sequence of good values:

```
@Test
public void correctValuesWholeNumberOfPence()
{
    if(VERBOSE)
    {
        System.out.println("* Testing CORRECT input values (-> VALIDATION = \"Is Valid...\");
        System.out.println("=====");
    }
    // 1. Test WEEK - Any valid currency amount (below max @length) should validate
    RegularAmount instance1 = new RegularAmount("0.01", Frequency.WEEK);
    RegularAmount instance2 = new RegularAmount("0.09", Frequency.WEEK);
    RegularAmount instance3 = new RegularAmount("1", Frequency.WEEK);
    RegularAmount instance4 = new RegularAmount("9", Frequency.WEEK);
    ...

    assertTrue(runBeanConstraintValidator(instance1));
    assertTrue(runBeanConstraintValidator(instance2));
    assertTrue(runBeanConstraintValidator(instance2));
    assertTrue(runBeanConstraintValidator(instance3));
    assertTrue(runBeanConstraintValidator(instance4));
    ...

    assertTrue(runBeanConstraintValidator(instance34));
}
```

And similarly for bad values via `incorrectValuesNOTWholeNumberOfPence()`:

```
@Test
public void incorrectValuesNOTWholeNumberOfPence()
{
    if(VERBOSE)
    {
        System.out.println("Testing INCORRECT input values...(-> VALIDATION = \"Is NOT Valid...\");
        System.out.println("=====");
    }

    // 1. Test WEEK - Since division by 1, only other constraint violations are possible
    // Too big/too many characters:
    RegularAmount instance1 = new RegularAmount("1234567891234567890.01", Frequency.WEEK);
    RegularAmount instance2 = new RegularAmount("0.00000000000000000000000000000001", Frequency.WEEK);
    // Dumb inputs (not currency number) tested later...

    // 2. Test TWO_WEEK - Any odd pence values should guarantee to be invalid.
    RegularAmount instance3 = new RegularAmount("0.03", Frequency.TWO_WEEK);
    RegularAmount instance4 = new RegularAmount("3.77", Frequency.TWO_WEEK);
    RegularAmount instance5 = new RegularAmount("303.33", Frequency.TWO_WEEK);
    RegularAmount instance6 = new RegularAmount("444.01", Frequency.TWO_WEEK);
    RegularAmount instance7 = new RegularAmount("4444.43", Frequency.TWO_WEEK);
    //Edge cases, 11 digits
    RegularAmount instance8 = new RegularAmount("100000019.03", Frequency.TWO_WEEK);
}
```

```

RegularAmount instance9 = new RegularAmount("22222222.27", Frequency.TWO_WEEK);

...

assertTrue(!runBeanConstraintValidator(instance1));
assertTrue(!runBeanConstraintValidator(instance2));
assertTrue(!runBeanConstraintValidator(instance3));
assertTrue(!runBeanConstraintValidator(instance4));
assertTrue(!runBeanConstraintValidator(instance5));
assertTrue(!runBeanConstraintValidator(instance6));
assertTrue(!runBeanConstraintValidator(instance7));
assertTrue(!runBeanConstraintValidator(instance8));
assertTrue(!runBeanConstraintValidator(instance9));

...

assertTrue(!runBeanConstraintValidator(instance53));
}

```

In a similar fashion, all of the other methods across the codebase were tested.











7.2 JaCoCo Code Coverage Analysis

Using the JaCaCo code coverage framework in conjunction with JUnit allows us to examine how thoroughly our JUnit tests cover our code. Note that even if you achieve 100% this does **not** mean that your code is automatically guaranteed to either work, or be bug free; it does however mean you have at least subjected it to a range of tests that do indeed exercise the code you have implemented and that it works as you expected it to according to the inputs you subjected it to.

In any case, 99% code coverage was achieved (the missing 1% being an unreachable `default:` switch branch that was necessary for compilation/code safety):

 NHS_BSA_Assignment1 >  com.aren.nhs_bsa_assignment

com.aren.nhs_bsa_assignment

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 CheckRegularAmountValidator		97%		70%	9	25	1	48	0	7	0	1
 RunAmount		100%		100%	0	36	0	231	0	15	0	1
 RegularAmount.Frequency		100%		n/a	0	1	0	7	0	1	0	1
 RegularAmount		100%		n/a	0	6	0	10	0	6	0	1
Total	5 of 1,052	99%	9 of 65	86%	9	68	1	296	0	29	0	4

8 Usage

The Maven shade plugin is used to produce a single “fat” JAR file with all dependencies bundled within it (which can be directly downloaded here, if you would prefer to avoid having to build/compile from source by cloning the repository). Simply run the JAR file:

```
java -jar NHS_BSA_Assignment-1.0.jar
```

This will give you a usage menu, like so:

```
$ java -jar NHS_BSA_Assignment-1.0-SNAPSHOT.jar
```

```
=====
```

Usage:

```
java -jar NHS_BSA_Assigment1.jar ARG
```

Where ARG (without quotes) is one of:

"1" : Run validator in interactive mode.
"2" : Interactive mode, verbose messaging
"3" : Run validator on demo test values.
"4" : Demo test values, verbose messaging.

So then just run it with the desired operation mode by specifying the command line argument. For example, to run it in interactive mode:

```
java -jar NHS_BSA_Assignment-1.0.jar 1
```

This mode allows you to type in a value and specify the desired frequency, and get the validation result. This process is repeated until the user decides to exit, by entering "0" when prompted for the frequency.

Demo mode, as the name suggests, simply runs it across some hard-coded **RegularAmount** amounts to validate.

9 Build/Compile

If you wish to build from source after cloning this repository, do so in usual manner using Apache Maven:

```
mvn clean install
```

The output "fat" JAR file with all dependencies built-in will be at `./target/NHS_BSA_Assignment-1.0-SNAPSHOT.jar`.

10 Issues

Q: What are all the strange symbols on the output text when displaying results?

If you are attempting to run the program under Windows 10, and are seeing something like the following:

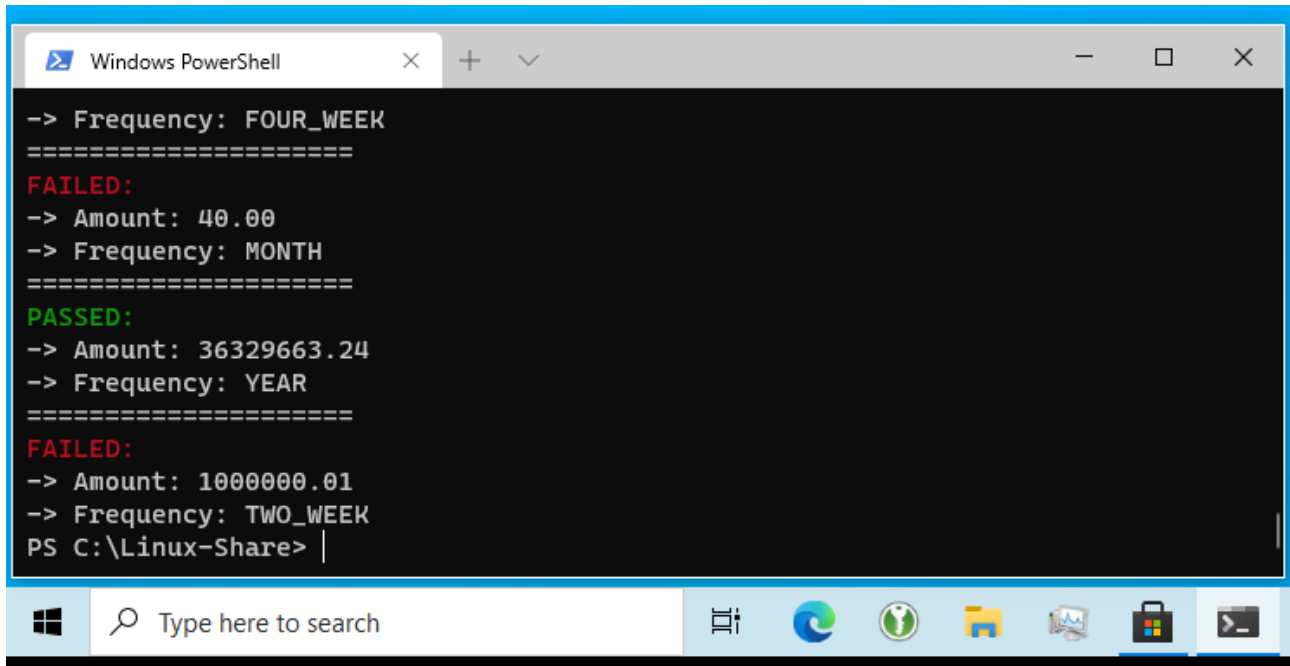
```
Command Prompt

=====
@[ 32mPASSED:@[ 0m
-> Amount: 2.21
-> Frequency: QUARTER
=====
@[ 32mPASSED:@[ 0m
-> Amount: 177929.08
-> Frequency: FOUR_WEEK
=====
@[ 32mPASSED:@[ 0m
-> Amount: 1153407.58
-> Frequency: QUARTER
=====
@[ 31mFAILED:@[ 0m
-> Amount: 8349758937458972398457982374589234759723947374
-> Frequency: FOUR_WEEK
=====
@[ 31mFAILED:@[ 0m
-> Amount: 40.00
-> Frequency: MONTH
=====
@[ 32mPASSED:@[ 0m
-> Amount: 36329663.24
-> Frequency: YEAR
=====
@[ 31mFAILED:@[ 0m
-> Amount: 1000000.01
-> Frequency: TWO_WEEK

Windows PowerShell

-> Amount: 177929.08
-> Frequency: FOUR_WEEK
=====
@[ 32mPASSED:@[ 0m
-> Amount: 1153407.58
-> Frequency: QUARTER
=====
@[ 31mFAILED:@[ 0m
-> Amount: 8349758937458972398457982374589234759723947374
-> Frequency: FOUR_WEEK
=====
@[ 31mFAILED:@[ 0m
-> Amount: 40.00
-> Frequency: MONTH
=====
@[ 32mPASSED:@[ 0m
-> Amount: 36329663.24
-> Frequency: YEAR
=====
@[ 31mFAILED:@[ 0m
-> Amount: 1000000.01
-> Frequency: TWO_WEEK
PS C:\Linux-Share>
```

Please upgrade your Windows terminal to a modern version/terminal that actually supports ANSI colour codes. Microsoft themselves provide a state-of-the-art GPU accelerated terminal, available via the Windows Store here, or via their GitHub project page here. This is the result after installing the new (official Microsoft) Windows Terminal, approximately a 7MB download:

A screenshot of a Windows PowerShell terminal window. The window title is "Windows PowerShell". The terminal output shows a program running with several prompts and responses. The output is color-coded: "FAILED:" is in red, "PASSED:" is in green, and "Amount:" is in blue. The prompts are "-> Frequency:" and "-> Amount:". The responses are "FOUR_WEEK", "40.00", "MONTH", "36329663.24", "YEAR", "1000000.01", and "TWO_WEEK". The terminal ends with the prompt "PS C:\Linux-Share>". The Windows taskbar is visible at the bottom of the window.

Linux users should be unaffected by this as just about all terminals support ANSI colour codes out of the box, and have done so for the last 20+ years. MacOS users should also have no issue, though I do not have access to a recent MacOS system to test this.

11 License

AGPL-3.0. See LICENSE.txt for details.

12 Appendix

12.1 Full Source Code Listing

12.1.1 CheckRegularAmount.java

```
/* CheckRegularAmount.java
 *
 * See LICENSE.txt in project root directory for license details.
 */
package com.aren.nhs_bsa_assignment;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```
import javax.validation.Constraint;
import javax.validation.Payload;

/**
 *
 * @author Aren Tyr
 * @version 1.0 - 2020-08-22
 *
 * <code>ConstraintValidator</code> annotation interface that calls
 * <code>CheckRegularAmountValidator</code>.
 *
 * Primarily targeted at a class level through <code>ElementType.TYPE</code>,
 * since the validation is dependent on the result of both <code>amount</code>
 * and <code>frequency</code> in combination. Body of the interface is standard
 * defaults.
 */
@Documented
@Constraint(validatedBy = CheckRegularAmountValidator.class)
@Target({
    ElementType.TYPE, ElementType.FIELD
})
@Retention(RetentionPolicy.RUNTIME)
public @interface CheckRegularAmount
{
    String message() default "Validation error detected.";

    Class<?>[] groups() default
    {
    };

    Class<? extends Payload>[] payload() default
    {
    };
}
```

12.1.2 CheckRegularAmountValidator.java

```
/* CheckRegularAmountValidator.java
 *
 * See LICENSE.txt in project root directory for license details.
 */
package com.aren.nhs_bsa_assignment;

import com.aren.nhs_bsa_assignment.RegularAmount.Frequency;

import java.math.BigDecimal;
import java.math.RoundingMode;

import javax.validation.ConstraintValidator;
```



```

import javax.validation.ConstraintValidatorContext;

/**
 * Java Bean validator class that implements <code>ConstraintValidator</code>.
 *
 * Responsible for ensuring that all <code>amount</code> values are validated as
 * valid if they divide into an exact number of pence, based on their weekly
 * frequency value, or are validated as invalid otherwise.
 *
 * @author Aren Tyr.
 * @version 1.0 - 2020-08-25
 */
public class CheckRegularAmountValidator implements
    ConstraintValidator<CheckRegularAmount, RegularAmount>
{

    /**
     * Debugging message switch.
     */
    private static final boolean DEBUG = RunAmount.DEBUG;

    /**
     * Internal method that employs a regular expression to only accept valid numerical
     * representations of a currency (in this case, British sterling).
     *
     * Accepts currency amounts that either feature no pence designation (e.g.
     * say, <code>100</code> to represent <code>£100.00</code>) or fully qualified
     * pence designation featuring a decimal point and two digits (e.g. <code>40.50</code>
     * is fine, <code>40.5</code> is not).
     *
     * @param inputAmount An amount representing a currency value, either with no
     *     decimal fraction specified, <b>or</b> with a decimal point and exactly
     *     two digits representing pence.
     *
     * @return <code>true</code> if the regular expression detects a string of the
     *     form "xxx" or "xxx.xx" where <b>x</b> represents a numerical digit
     *     [0-9], or <code>false</code> otherwise.
     */
    private boolean isValidNumericAmount(String inputAmount)
    {
        // Regular expression that requires number starts with at least one digit
        // and optionally has a decimal point & two digits
        return inputAmount.matches("^([0-9]+(\\.\\d{2})?)?$");
    }

    /* DEPRECATED: Replaced with inbuilt BigDecimal object and RoundingMode exception.

    BigDecimal is a more elegant "Java-esque" solution that makes use of inbuilt types.
    -----

```

All correct currency/exact pence values must have ≤ 2 decimal points -
(e.g. 1p is smallest granularity in UK currency)

This function courtesy of StackExchange:

<https://stackoverflow.com/questions/2296110/determine-number-of-decimal-place-using-bigdecimal>

```
private int getNumberOfDecimalPlaces(BigDecimal bigDecimal)
{
    String string = bigDecimal.stripTrailingZeros().toPlainString();
    int index = string.indexOf(".");
    return index < 0 ? 0 : string.length() - index - 1;
}

/**
 * Internal method to convert the enum type <code>Frequency</code> into
 * a valid numerical divisor.
 *
 * <code>MONTH</code> is assigned to -1.00/error condition since a month, by definition,
 * does not precisely specify any exact number of weeks (four weeks merely being a
 * social convention rather than mathematical reality, since months vary from 28-31
 * days).
 *
 * @param inputFreq The given <code>RegularAmount.Frequency</code> enum to convert
 * to a numerical value so that division can be performed.
 *
 * @return <code>BigDecimal</code> value that represents numerical value of the
 * specified frequency (e.g. "FOUR_WEEK" yields 4.00)
 */
private BigDecimal validFrequencyDivisor(Frequency inputFreq)
{
    switch(inputFreq)
    {
        case WEEK:
            return new BigDecimal("1.00");
        case TWO_WEEK:
            return new BigDecimal("2.00");
        case FOUR_WEEK:
            return new BigDecimal("4.00");
        case MONTH:
            // We will stipulate as INVALID "MONTH" since !=
            // any specific number of weeks...
            if(DEBUG)
            {
                System.out.println("MONTH value requested, MONTH does not "
                    + "define a WEEKLY amount.");
            }
            return new BigDecimal("-1.00");
        case QUARTER:
            return new BigDecimal("13.00");
        case YEAR:
            return new BigDecimal("52.00");
    }
}
```

```

        return new BigDecimal("52.00");
    default:
        return new BigDecimal("-1.00");
    }
}

/**
 * Internal method to actually determine whether the amount is exactly divisible to
 * a whole pence amount.
 *
 * Performs the bulk of the work in this class in terms of validating amounts.
 * Employs the <code>BigDecimal</code> object with <code>RoundingMode.UNNECCESARY</code>
 * which will throw an <code>ArithmeticException</code> if the the result of
 * the division does not yield an exact value within the specified scale. Here,
 * scale is <code>2</code> (i.e. two decimal points), since our maximum resolution
 * is a currency pence (UK Sterling).
 *
 * @param amt The given string representing a numerical currency in an acceptable
 *             format (e.g. "400", "495.34", "50", "10.00", "1"). See
 *             <code>isValidNumericAmount</code> method above.
 *
 * @param divisor A <code>BigDecimal</code> value for dividing by.
 *
 * @return <code>true</code> if the amount <code>amt</code> divides into an
 *         exact pence amount as per the <code>divisor</code> value,
 *         <code>false</code> otherwise.
 */
private boolean isValidExactPenceAmount(String amt, BigDecimal divisor)
{
    BigDecimal numerator = new BigDecimal(amt);
    BigDecimal result = new BigDecimal("-9999.9999");

    if(DEBUG)
    {
        System.out.println("Numerator is: " + numerator);
        System.out.println("Divisor is: " + divisor);
    }

    // Only accept positive/non-zero Amount, and reject all frequencies less than
    // 1 (i.e. the negative "default" case)
    try
    {
        if(numerator.compareTo(new BigDecimal("0")) > 0
            && divisor.compareTo(new BigDecimal("1")) >= 0)
        {
            // Use RoundingMode.UNNECESSARY to throw ArithmeticException if
            // _any_ rounding occurs outside of the scale of two decimal points
            result = numerator.divide(divisor, 2, RoundingMode.UNNECESSARY);
        }
        else
        {

```

```
        // Important for edge case of amount = 0 (BigDecimal does not
        // throw an exception dividing by 0)
        return false;
    }
}
catch(ArithmeticException ae)
{
    if(DEBUG)
    {
        System.out.println("Non-exact division within requested scale"
            + "(two decimal places): " + ae);
    }
    return false;
}

if(DEBUG)
{
    System.out.println("BigDecimal division result is: " + result);
}

//DEPRECATED (replaced with BigDecimal)
//BigDecimal checkCurrencyPence = new BigDecimal(val);
//return getNumberOfDecimalPlaces(checkCurrencyPence)<= 2;

// If we reach here then exact division within two decimal points
// must have occurred
return true;
}

/**
 * Default/required <code>initialize</code> function. No customisation.
 *
 * @param constraintAnnotation The <code>CheckRegularAmount</code> constraint
 *        wrapper class.
 */
@Override
public void initialize(CheckRegularAmount constraintAnnotation)
{
}

/**
 *
 * Required <code>isValid</code> function that returns the result of the validation.
 *
 * Calls internal <code>isValidNumericAmount</code>, <code>validFrequencyDivisor</code>,
 * and <code>isValidExactPenceAmount</code> methods to determine validation result.
 *
 * @param value The <code>RegularAmount</code> instance to be validated.
 *
 * @param context The validation context.
 */
```

```
* @return <code>true</code> if the given amount divides into an exactly weekly
*         amount as per the specified frequency, <code>false</code> otherwise.
*/
@Override
public boolean isValid(RegularAmount value, ConstraintValidatorContext context)
{
    String amt = value.getAmount();
    boolean numericAmountCheck = isValidNumericAmount(amt);
    BigDecimal frequencyCheck = validFrequencyDivisor(value.getFrequency());

    if(numericAmountCheck == true)
    {
        if(DEBUG)
        {
            System.out.println("Valid number: " + amt);
        }

        boolean validPenceAmount = isValidExactPenceAmount(amt, frequencyCheck);

        if(validPenceAmount)
        {
            if(DEBUG)
            {
                System.out.println("VALID/EXACT Amount registered");
            }
            return true;
        }
        else
        {
            if(DEBUG)
            {
                System.out.println("INVALID/NON-EXACT pence Amount registered");
            }
            return false;
        }
    }
    else
    {
        if(DEBUG)
        {
            System.out.println("Invalid Number format -> no possibility "
                               + "of exact pence: " + amt);
        }
        return false;
    }
}
```

12.1.3 RegularAmount.java

```
/* RegularAmount.java
```

```
*
* See LICENSE.txt in project root directory for license details.
*/
package com.aren.nhs_bsa_assignment;

import org.hibernate.validator.constraints.Length;

/**
 * Based on the supplied template class with a few modifications/additions.
 *
 * @author Aren Tyr
 * @version 1.0 - 2020-08-24
 */
@CheckRegularAmount
public class RegularAmount
{

    /**
     * Setup default values (that usefully will guarantee to fail validation).
     */
    public RegularAmount()
    {
        this("-1.00", RegularAmount.Frequency.WEEK);
    }

    /**
     * An overloaded constructor for convenience, to setup <code>amount</code>
     * and <code>frequency</code> on object initialisation.
     *
     * @param amt String expression representing a currency value in UK sterling.
     *
     * @param frequency <code>RegularAmount.Frequency</code> enum value; one of <code>
     *     WEEK</code>, <code>TWO_WEEK</code>, <code>FOUR_WEEK</code>,
     *     <code>MONTH</code>, <code>QUARTER</code>, or <code>YEAR</code>.
     */
    public RegularAmount(String amt, Frequency frequency)
    {
        this.amount = amt;
        this.frequency = frequency;
    }

    private Frequency frequency;

    /**
     * Use @Length annotation to restrict maximum input to 11 digits (characters)
     * for security/safety.
     *
     * This allows maximum values up to £99999999999 or £99999999.99 theoretically...!
     */
    @Length(max=11,message="* Bad input. Greater than 11 characters.")
    private String amount;
```

```
/**
 * Standard getter to retrieve the <code>Frequency</code> enum value.
 *
 * @return The currently set <code>Frequency</code> enum.
 */
public Frequency getFrequency() { return frequency; }

/**
 * Standard setter to set the <code>Frequency</code> enum value.
 *
 * @param frequency <code>RegularAmount.Frequency</code> enum value; one of <code>
 *     WEEK</code>, <code>TWO_WEEK</code>, <code>FOUR_WEEK</code>,
 *     <code>MONTH</code>, <code>QUARTER</code>, or <code>YEAR</code>.
 */
public void setFrequency(Frequency frequency) { this.frequency = frequency; }

/**
 * Standard getter to retrieve the <code>amount</code> string representing
 * the currency value (regardless of whether it represents a coherent/valid
 * currency amount).
 *
 * @return The currently set amount string.
 */
public String getAmount() { return amount; }

/**
 * Standard setter to set the <code>amount</code> string representing the
 * currency value.
 *
 * @param amount A string to represent a currency amount; e.g. "9.99".
 */
public void setAmount(String amount) { this.amount = amount; }

/**
 * Enumerated type that defines the set of terms that represents a time
 * series/weekly time amount.
 *
 * Acts as the divisor mapping value for determining whether an exact pence
 * amount is possible or not.
 */
public enum Frequency {

    /**
     * Represents one week = mapped to a divisor of 1.0.
     */
    WEEK,

    /**
     * Represents two weeks = mapped to a divisor of 2.0.
     */
}
```

```
TWO_WEEK,  
  
/**  
 * Represents four weeks = mapped a divisor of 4.0.  
 */  
FOUR_WEEK,  
  
/**  
 * Represents one month, but not accepted as a valid weekly divisor.  
 *  
 * A month does not constitute any exact number of weeks; it is only ever  
 * four weeks for February, and only then for non-leap years. Therefore  
 * specified as mapping to a divisor of -1.0 here by design.  
 */  
MONTH,  
  
/**  
 * Represents one yearly quarter = mapped to a divisor of 13.0.  
 */  
QUARTER,  
  
/**  
 * Represents one entire year = mapped to a divisor of 52.0 by design.  
 *  
 * NOTE: A year technically does not exactly map to 52. However, here we  
 * are deferring to the requirements specification which states to map to  
 * 52.  
 */  
YEAR; }  
}
```

12.1.4 RegularAmountTest.java

```
/* RegularAmountTest.java  
 *  
 * See LICENSE.txt in project root directory for license details.  
 */  
package com.aren.nhs_bsa_assignment;  
  
import static com.aren.nhs_bsa_assignment.RegularAmount.Frequency;  
  
import java.util.Set;  
  
import javax.validation.ConstraintViolation;  
import javax.validation.Validation;  
import javax.validation.Validator;  
import javax.validation.ValidatorFactory;  
  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.BeforeAll;  
import org.junit.jupiter.api.AfterAll;
```



```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

/**
 * JUnit test class for <code>RegularAmount.java</code>.
 *
 * @author Aren Tyr.
 * @version 1.0 2020-08-24
 */
public class RegularAmountTest
{

    private final static boolean VERBOSE = true;

    // Necessary Bean validators
    private static ValidatorFactory validatorFactory;
    private static Validator validator;

    /**
     * Specialist constructor to setup a test Bean instance.
     */
    @BeforeAll
    public static void createValidator()
    {
        // Setup the validators
        validatorFactory = Validation.buildDefaultValidatorFactory();
        validator = validatorFactory.getValidator();
    }

    /**
     * Deletes/closes the test Bean instance after tests are complete.
     */
    @AfterAll
    public static void close()
    {
        validatorFactory.close();
    }

    /**
     * Wrapper method that validates/checks Bean instance for validation errors.
     *
     * @param ra A <code>RegularAmount</code> instance to perform
     *           <code>ConstraintValidation</code> on.
     *
     * @return <code>true</code> if <code>RegularAmount</code> has no validation
     *         errors, <code>false</code> otherwise.
     */
    private boolean runBeanConstraintValidator(RegularAmount ra)
    {
        Set<ConstraintViolation<RegularAmount>> violations = validator.validate(ra);
        if(violations.isEmpty())
    }
```

```
{
    if(VERBOSE)
    { System.out.println("-> VALID amount/frequency. \nAmount: "
        + ra.getAmount() + " \nFrequency: " + ra.getFrequency()); }
    return true;
}
else
{
    if(VERBOSE)
    { System.out.println("-> INVALID amount/frequency. \nAmount: \""
        + ra.getAmount() + "\" \nFrequency: " + ra.getFrequency()); }
    return false;
}
}

/**
 * JUnit test of <code>setFrequency</code> and <code>getFrequency</code>
 * methods, of class <code>RegularAmount</code>.
 *
 * Mostly just a test of state, since it is just a simple getter &
 * setter, but here for completeness.
 */
@Test
public void testGetAndSetFrequency()
{
    if(VERBOSE)
    {
        System.out.println("* Testing getting and setting Frequency...");
        System.out.println("=====");
    }
    RegularAmount instance = new RegularAmount();
    instance.setFrequency(RegularAmount.Frequency.WEEK);
    RegularAmount.Frequency expResult = Frequency.WEEK;
    RegularAmount.Frequency result = instance.getFrequency();
    assertEquals(expResult, result);
}

/**
 * JUnit test of <code>setFrequency</code> and <code>getFrequency</code>
 * methods, of class <code>RegularAmount</code>.
 *
 * Mostly just a test of state, since it is just a simple getter &
 * setter, but here for completeness.
 */
@Test
public void testGetAndSetAmount()
{
    if(VERBOSE)
    {
        System.out.println("* Testing getting and setting Amount...");
        System.out.println("=====");
    }
}
```

```
    }
    RegularAmount instance = new RegularAmount();
    instance.setAmount("100.00");
    String expectedResult = "100.00";
    String result = instance.getAmount();
    assertEquals(expectedResult, result);
}

/**
 * JUnit test that tests the <code>ConstraintValidator</code> class across a
 * series of correct inputs to see that it is validating/accepting as
 * expected.
 */
@Test
public void correctValuesWholeNumberOfPence()
{
    if(VERBOSE)
    {
        System.out.println("* Testing CORRECT input values (-> VALIDATION = \"Is Valid...\");");
        System.out.println("=====");
    }
    // 1. Test WEEK - Any valid currency amount (below max @length) should validate
    RegularAmount instance1 = new RegularAmount("0.01", Frequency.WEEK);
    RegularAmount instance2 = new RegularAmount("0.09", Frequency.WEEK);
    RegularAmount instance3 = new RegularAmount("1", Frequency.WEEK);
    RegularAmount instance4 = new RegularAmount("9", Frequency.WEEK);
    RegularAmount instance5 = new RegularAmount("400.00", Frequency.WEEK);
    RegularAmount instance6 = new RegularAmount("1000.00", Frequency.WEEK);
    RegularAmount instance7 = new RegularAmount("9999.99", Frequency.WEEK);
    // Edge cases, 11 digits:
    RegularAmount instance8 = new RegularAmount("12312312.12", Frequency.WEEK);
    RegularAmount instance9 = new RegularAmount("77777777777", Frequency.WEEK);

    // 2. Test TWO_WEEK - Obviously need to be a modulus 2 zero remainder value
    RegularAmount instance10 = new RegularAmount("0.02", Frequency.TWO_WEEK);
    RegularAmount instance11 = new RegularAmount("2", Frequency.TWO_WEEK);
    RegularAmount instance12 = new RegularAmount("200.00", Frequency.TWO_WEEK);
    RegularAmount instance13 = new RegularAmount("400.00", Frequency.TWO_WEEK);
    RegularAmount instance14 = new RegularAmount("4444.44", Frequency.TWO_WEEK);
    //Edge cases, 11 digits
    RegularAmount instance15 = new RegularAmount("88888888888", Frequency.TWO_WEEK);
    RegularAmount instance16 = new RegularAmount("22222222.22", Frequency.TWO_WEEK);

    // 3. Test FOUR_WEEK - Mod 4 zero remainder
    RegularAmount instance17 = new RegularAmount("0.04", Frequency.FOUR_WEEK);
    RegularAmount instance18 = new RegularAmount("4", Frequency.FOUR_WEEK);
    RegularAmount instance19 = new RegularAmount("400.00", Frequency.FOUR_WEEK);
    RegularAmount instance20 = new RegularAmount("12000.00", Frequency.FOUR_WEEK);
    // Edge cases, 11 digits
    RegularAmount instance21 = new RegularAmount("88888888.88", Frequency.FOUR_WEEK);
    RegularAmount instance22 = new RegularAmount("44444444444", Frequency.FOUR_WEEK);
}
```

```
// 4. Test QUARTER - Mod 13 zero remainder
RegularAmount instance23 = new RegularAmount("0.13", Frequency.QUARTER);
RegularAmount instance24 = new RegularAmount("13", Frequency.QUARTER);
RegularAmount instance25 = new RegularAmount("260.00", Frequency.QUARTER);
RegularAmount instance26 = new RegularAmount("33098.00", Frequency.QUARTER);
// Edge cases, 11 digits
RegularAmount instance27 = new RegularAmount("25384359897", Frequency.QUARTER); // £1,952,643,0
RegularAmount instance28 = new RegularAmount("11554101.00", Frequency.QUARTER); // £887,777

// 5. Test YEAR - Mod 52 zero remainder
RegularAmount instance29 = new RegularAmount("0.52", Frequency.YEAR);
RegularAmount instance30 = new RegularAmount("52", Frequency.YEAR);
RegularAmount instance31 = new RegularAmount("520.00", Frequency.YEAR);
RegularAmount instance32 = new RegularAmount("884.00", Frequency.YEAR);
// Edge cases, 11 digits
RegularAmount instance33 = new RegularAmount("91288860.00", Frequency.YEAR); // £1,755,555
RegularAmount instance34 = new RegularAmount("61711269360", Frequency.YEAR); // £1,186,755,180

assertTrue(runBeanConstraintValidator(instance1));
assertTrue(runBeanConstraintValidator(instance2));
assertTrue(runBeanConstraintValidator(instance2));
assertTrue(runBeanConstraintValidator(instance3));
assertTrue(runBeanConstraintValidator(instance4));
assertTrue(runBeanConstraintValidator(instance5));
assertTrue(runBeanConstraintValidator(instance6));
assertTrue(runBeanConstraintValidator(instance7));
assertTrue(runBeanConstraintValidator(instance8));
assertTrue(runBeanConstraintValidator(instance9));
assertTrue(runBeanConstraintValidator(instance10));
assertTrue(runBeanConstraintValidator(instance11));
assertTrue(runBeanConstraintValidator(instance12));
assertTrue(runBeanConstraintValidator(instance13));
assertTrue(runBeanConstraintValidator(instance14));
assertTrue(runBeanConstraintValidator(instance15));
assertTrue(runBeanConstraintValidator(instance16));
assertTrue(runBeanConstraintValidator(instance17));
assertTrue(runBeanConstraintValidator(instance18));
assertTrue(runBeanConstraintValidator(instance19));
assertTrue(runBeanConstraintValidator(instance20));
assertTrue(runBeanConstraintValidator(instance21));
assertTrue(runBeanConstraintValidator(instance22));
assertTrue(runBeanConstraintValidator(instance23));
assertTrue(runBeanConstraintValidator(instance24));
assertTrue(runBeanConstraintValidator(instance25));
assertTrue(runBeanConstraintValidator(instance26));
assertTrue(runBeanConstraintValidator(instance27));
assertTrue(runBeanConstraintValidator(instance28));
assertTrue(runBeanConstraintValidator(instance29));
assertTrue(runBeanConstraintValidator(instance30));
assertTrue(runBeanConstraintValidator(instance31));
```

```
        assertTrue(runBeanConstraintValidator(instance32));
        assertTrue(runBeanConstraintValidator(instance33));
        assertTrue(runBeanConstraintValidator(instance34));
    }

    /**
     * JUnit test that tests the <code>ConstraintValidator</code> class across a
     * series of incorrect inputs to see that is is rejecting as expected.
     *
     * This function tests a sequence of values all of which should validate as
     * bad/incorrect values (hence the "!" before the return value).
     */
    @Test
    public void incorrectValuesNOTWholeNumberOfPence()
    {
        if(VERBOSE)
        {
            System.out.println("Testing INCORRECT input values...(-> VALIDATION = \"Is NOT Valid...\")");
            System.out.println("=====");
        }

        // 1. Test WEEK - Since division by 1, only other constraint violations are possible
        // Too big/too many characters:
        RegularAmount instance1 = new RegularAmount("1234567891234567890.01", Frequency.WEEK);
        RegularAmount instance2 = new RegularAmount("0.000000000000000000000000000001", Frequency.WEEK);
        // Dumb inputs (not currency number) tested later...

        // 2. Test TWO_WEEK - Any odd pence values should guarantee to be invalid.
        RegularAmount instance3 = new RegularAmount("0.03", Frequency.TWO_WEEK);
        RegularAmount instance4 = new RegularAmount("3.77", Frequency.TWO_WEEK);
        RegularAmount instance5 = new RegularAmount("303.33", Frequency.TWO_WEEK);
        RegularAmount instance6 = new RegularAmount("444.01", Frequency.TWO_WEEK);
        RegularAmount instance7 = new RegularAmount("4444.43", Frequency.TWO_WEEK);
        //Edge cases, 11 digits
        RegularAmount instance8 = new RegularAmount("100000019.03", Frequency.TWO_WEEK);
        RegularAmount instance9 = new RegularAmount("22222222.27", Frequency.TWO_WEEK);

        // 3. Test FOUR_WEEK - Same principle as above
        RegularAmount instance10 = new RegularAmount("0.05", Frequency.FOUR_WEEK);
        RegularAmount instance11 = new RegularAmount("5.11", Frequency.FOUR_WEEK);
        RegularAmount instance12 = new RegularAmount("400.05", Frequency.FOUR_WEEK);
        RegularAmount instance13 = new RegularAmount("12000.63", Frequency.FOUR_WEEK);
        // Edge cases, 11 digits
        RegularAmount instance14 = new RegularAmount("88888888.87", Frequency.FOUR_WEEK);
        RegularAmount instance15 = new RegularAmount("44444444.01", Frequency.FOUR_WEEK);

        // 4. Test QUARTER - Mod 13 zero remainder
        RegularAmount instance16 = new RegularAmount("0.12", Frequency.QUARTER);
        RegularAmount instance17 = new RegularAmount("16", Frequency.QUARTER);
        RegularAmount instance18 = new RegularAmount("261.00", Frequency.QUARTER);
        RegularAmount instance19 = new RegularAmount("33098.03", Frequency.QUARTER);
```

```
// Edge cases, 11 digits
RegularAmount instance20 = new RegularAmount("25384359.98", Frequency.QUARTER);
RegularAmount instance21 = new RegularAmount("11554101.07", Frequency.QUARTER);

// 5. Test YEAR - Mod 52 zero remainder
RegularAmount instance22 = new RegularAmount("0.50", Frequency.YEAR);
RegularAmount instance23 = new RegularAmount("51", Frequency.YEAR);
RegularAmount instance24 = new RegularAmount("521.00", Frequency.YEAR);
RegularAmount instance25 = new RegularAmount("885.00", Frequency.YEAR);
// Edge cases, 11 digits
RegularAmount instance26 = new RegularAmount("91288860.13", Frequency.YEAR);
RegularAmount instance27 = new RegularAmount("61711269.77", Frequency.YEAR);

// Test non-numbers/dumb inputs
RegularAmount instance28 = new RegularAmount("bad value", Frequency.WEEK);
RegularAmount instance29 = new RegularAmount("\\""\\""\\""\\""\\", Frequency.WEEK);
RegularAmount instance30 = new RegularAmount("100f.00", Frequency.TWO_WEEK);
RegularAmount instance31 = new RegularAmount("100.0", Frequency.TWO_WEEK);
RegularAmount instance32 = new RegularAmount("1.00.0", Frequency.FOUR_WEEK);
RegularAmount instance33 = new RegularAmount("00.0", Frequency.FOUR_WEEK);
RegularAmount instance34 = new RegularAmount("!!!", Frequency.FOUR_WEEK);
RegularAmount instance35 = new RegularAmount("*", Frequency.YEAR);
RegularAmount instance36 = new RegularAmount("hello@NaN", Frequency.WEEK);

// Deny/don't allow zero (division by 0) as valid amount
RegularAmount instance37 = new RegularAmount("0", Frequency.WEEK);
RegularAmount instance38 = new RegularAmount("0", Frequency.TWO_WEEK);
RegularAmount instance39 = new RegularAmount("0", Frequency.FOUR_WEEK);
RegularAmount instance40 = new RegularAmount("0", Frequency.MONTH);
RegularAmount instance41 = new RegularAmount("0", Frequency.QUARTER);
RegularAmount instance42 = new RegularAmount("0", Frequency.YEAR);

RegularAmount instance43 = new RegularAmount("0.00", Frequency.WEEK);
RegularAmount instance44 = new RegularAmount("0.00", Frequency.TWO_WEEK);
RegularAmount instance45 = new RegularAmount("0.00", Frequency.FOUR_WEEK);
RegularAmount instance46 = new RegularAmount("0.00", Frequency.MONTH);
RegularAmount instance47 = new RegularAmount("0.00", Frequency.QUARTER);
RegularAmount instance48 = new RegularAmount("0.00", Frequency.YEAR);

// Reject "MONTH" - what is a "month"? No actual defined amount of weeks...!
RegularAmount instance49 = new RegularAmount("10", Frequency.MONTH);
RegularAmount instance50 = new RegularAmount("10.00", Frequency.MONTH);
RegularAmount instance51 = new RegularAmount("200", Frequency.MONTH);
RegularAmount instance52 = new RegularAmount("500000.00", Frequency.MONTH);

// Finally test empty instance (default constructor)
RegularAmount instance53 = new RegularAmount();

assertTrue(!runBeanConstraintValidator(instance1));
assertTrue(!runBeanConstraintValidator(instance2));
assertTrue(!runBeanConstraintValidator(instance3));
```

```
assertTrue(!runBeanConstraintValidator(instance4));
assertTrue(!runBeanConstraintValidator(instance5));
assertTrue(!runBeanConstraintValidator(instance6));
assertTrue(!runBeanConstraintValidator(instance7));
assertTrue(!runBeanConstraintValidator(instance8));
assertTrue(!runBeanConstraintValidator(instance9));
assertTrue(!runBeanConstraintValidator(instance10));
assertTrue(!runBeanConstraintValidator(instance11));
assertTrue(!runBeanConstraintValidator(instance12));
assertTrue(!runBeanConstraintValidator(instance13));
assertTrue(!runBeanConstraintValidator(instance14));
assertTrue(!runBeanConstraintValidator(instance15));
assertTrue(!runBeanConstraintValidator(instance16));
assertTrue(!runBeanConstraintValidator(instance17));
assertTrue(!runBeanConstraintValidator(instance18));
assertTrue(!runBeanConstraintValidator(instance19));
assertTrue(!runBeanConstraintValidator(instance20));
assertTrue(!runBeanConstraintValidator(instance21));
assertTrue(!runBeanConstraintValidator(instance22));
assertTrue(!runBeanConstraintValidator(instance23));
assertTrue(!runBeanConstraintValidator(instance24));
assertTrue(!runBeanConstraintValidator(instance25));
assertTrue(!runBeanConstraintValidator(instance26));
assertTrue(!runBeanConstraintValidator(instance27));
assertTrue(!runBeanConstraintValidator(instance28));
assertTrue(!runBeanConstraintValidator(instance29));
assertTrue(!runBeanConstraintValidator(instance30));
assertTrue(!runBeanConstraintValidator(instance31));
assertTrue(!runBeanConstraintValidator(instance32));
assertTrue(!runBeanConstraintValidator(instance33));
assertTrue(!runBeanConstraintValidator(instance34));
assertTrue(!runBeanConstraintValidator(instance35));
assertTrue(!runBeanConstraintValidator(instance36));
assertTrue(!runBeanConstraintValidator(instance37));
assertTrue(!runBeanConstraintValidator(instance38));
assertTrue(!runBeanConstraintValidator(instance39));
assertTrue(!runBeanConstraintValidator(instance40));
assertTrue(!runBeanConstraintValidator(instance41));
assertTrue(!runBeanConstraintValidator(instance42));
assertTrue(!runBeanConstraintValidator(instance43));
assertTrue(!runBeanConstraintValidator(instance44));
assertTrue(!runBeanConstraintValidator(instance45));
assertTrue(!runBeanConstraintValidator(instance46));
assertTrue(!runBeanConstraintValidator(instance47));
assertTrue(!runBeanConstraintValidator(instance48));
assertTrue(!runBeanConstraintValidator(instance49));
assertTrue(!runBeanConstraintValidator(instance50));
assertTrue(!runBeanConstraintValidator(instance51));
assertTrue(!runBeanConstraintValidator(instance52));

assertTrue(!runBeanConstraintValidator(instance53));
```

```
    }  
}
```

12.1.5 RunAmount.java

```
/* RunAmount.java  
 *  
 * See LICENSE.txt in project root directory for license details.  
 */  
package com.aren.nhs_bsa_assignment;  
  
import java.util.ArrayList;  
import java.util.Scanner;  
import java.util.Set;  
  
import javax.validation.ConstraintViolation;  
import javax.validation.Validation;  
import javax.validation.Validator;  
import javax.validation.ValidatorFactory;  
  
/**  
 * A driver class used to demonstrate/test/examine the behaviour of  
 * CheckRegularAmount/CheckRegularAmountValidator classes/JavaBean  
 * ConstraintValidator.  
 *  
 * @author Aren Tyr  
 * @version 1.0 - 2020-08-25  
 */  
public class RunAmount  
{  
  
    /**  
     * Acts as a global debugging message switch (other classes reference it).  
     */  
    public static boolean DEBUG = true;  
  
    /**  
     * Acts as a switch to run through and display the explicit test/demo values  
     * used as part of the development process.  
     */  
    private static boolean RUNTESTS = false;  
  
    /**  
     * Get the current status of the debugging/verbose output switch.  
     *  
     * @return Boolean value indicating debugging/output status.  
     */  
    public static boolean isDEBUG()  
    {  
        return DEBUG;  
    }  
}
```



```
/**
 * Set the current status of the debugging/verbose output switch.
 *
 * @param DEBUG Boolean value to set debugging status (default value is
 *      <code>true</code>/on.
 */
public static void setDEBUG(boolean DEBUG)
{
    RunAmount.DEBUG = DEBUG;
}

/**
 * Get the current status of the sample test data set/example validation run.
 *
 * @return Boolean value indicating whether to run a series of demonstration tests.
 */
public static boolean isRUNTESTS()
{
    return RUNTESTS;
}

/**
 * Set the current status of the demonstration tests.
 *
 * @param RUNTESTS Boolean value indicating whether to run demonstration tests.
 *      Default value is <code>false</code>.
 */
public static void setRUNTESTS(boolean RUNTESTS)
{
    RunAmount.RUNTESTS = RUNTESTS;
}

/**
 * Some bling/aesthetic touches for console output. Fancy console output
 * colours (works under Unix, should work on a Mac terminal, untested under
 * Windows but should work with recent builds undre Windows 10).
 *
 * @see
 * <a href="https://stackoverflow.com/questions/5762491/how-to-print-color-in-console-using-system-
 * StackExchange thread">StackExchange thread</a>.
 */
public static final String ANSI_RESET = "\u001B[0m";
public static final String ANSI_BLACK = "\u001B[30m";
public static final String ANSI_RED = "\u001B[31m";
public static final String ANSI_GREEN = "\u001B[32m";
public static final String ANSI_YELLOW = "\u001B[33m";
public static final String ANSI_BLUE = "\u001B[34m";
public static final String ANSI_PURPLE = "\u001B[35m";
public static final String ANSI_CYAN = "\u001B[36m";
public static final String ANSI_WHITE = "\u001B[37m";
```

```
/**
 * Internal development/test/demo dataset. Since the validation class is mostly
 * an "invisible" entity that does the back-end work behind the scenes, this
 * at least gives us some output for sanity checking that the code is
 * working as expected, or serve as a demonstration output to show it working.
 *
 * @return An array containing a series of <code>RegularAmount</code>
 * objects with <code>amount</code> and <code>Frequency</code> values to
 * pass to the validator class.
 */
private static ArrayList<RegularAmount> getSampleAmountDataset()
{
    ArrayList<RegularAmount> sampleAmountTestDataset = new ArrayList<>();

    // Demo value 1 - valid number
    RegularAmount amountTestValue1 = new RegularAmount();
    amountTestValue1.setFrequency(RegularAmount.Frequency.WEEK);
    amountTestValue1.setAmount("100.00");
    sampleAmountTestDataset.add(amountTestValue1);

    // Demo value 2 - valid number
    RegularAmount amountTestValue2 = new RegularAmount();
    amountTestValue2.setFrequency(RegularAmount.Frequency.WEEK);
    amountTestValue2.setAmount("100000");
    sampleAmountTestDataset.add(amountTestValue2);

    // Demo value 3 - invalid number (only one digit after decimal point)
    RegularAmount amountTestValue3 = new RegularAmount();
    amountTestValue3.setFrequency(RegularAmount.Frequency.WEEK);
    amountTestValue3.setAmount("100.0");
    sampleAmountTestDataset.add(amountTestValue3);

    // Demo value 4 - invalid number (contains a non-digit character)
    RegularAmount amountTestValue4 = new RegularAmount();
    amountTestValue4.setFrequency(RegularAmount.Frequency.WEEK);
    amountTestValue4.setAmount("10f0.00");
    sampleAmountTestDataset.add(amountTestValue4);

    // Demo value 5 - invalid number (contains a non-digit character)
    RegularAmount amountTestValue5 = new RegularAmount();
    amountTestValue5.setFrequency(RegularAmount.Frequency.WEEK);
    amountTestValue5.setAmount("100.0a");
    sampleAmountTestDataset.add(amountTestValue5);

    // Demo value 6 - invalid number (negative value)
    RegularAmount amountTestValue6 = new RegularAmount();
    amountTestValue6.setFrequency(RegularAmount.Frequency.WEEK);
    amountTestValue6.setAmount("-100.00");
    sampleAmountTestDataset.add(amountTestValue6);
}
```

```
// Demo value 7 - invalid number (all characters)
RegularAmount amountTestValue7 = new RegularAmount();
amountTestValue7.setFrequency(RegularAmount.Frequency.WEEK);
amountTestValue7.setAmount("hello");
sampleAmountTestDataset.add(amountTestValue7);

// Demo value 8 - invalid number (contains character)
RegularAmount amountTestValue8 = new RegularAmount();
amountTestValue8.setFrequency(RegularAmount.Frequency.WEEK);
amountTestValue8.setAmount("10hello");
sampleAmountTestDataset.add(amountTestValue8);

// Demo value 9 - invalid number (empty string)
RegularAmount amountTestValue9 = new RegularAmount();
amountTestValue9.setFrequency(RegularAmount.Frequency.WEEK);
amountTestValue9.setAmount("");
sampleAmountTestDataset.add(amountTestValue9);

// Demo value 10 - invalid number (special character)
RegularAmount amountTestValue10 = new RegularAmount();
amountTestValue10.setFrequency(RegularAmount.Frequency.WEEK);
amountTestValue10.setAmount("*");
sampleAmountTestDataset.add(amountTestValue10);

// Demo value 11 - valid number (TWO_WEEK), correct pence
RegularAmount amountTestValue11 = new RegularAmount();
amountTestValue11.setFrequency(RegularAmount.Frequency.TWO_WEEK);
amountTestValue11.setAmount("100.00");
sampleAmountTestDataset.add(amountTestValue11);

// Demo value 12 - valid number (FOUR_WEEK), correct pence
RegularAmount amountTestValue12 = new RegularAmount();
amountTestValue12.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
amountTestValue12.setAmount("100000");
sampleAmountTestDataset.add(amountTestValue12);

// Demo value 13 - valid number (FOUR_WEEK), no exact pence
RegularAmount amountTestValue13 = new RegularAmount();
amountTestValue13.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
amountTestValue13.setAmount("99.99");
sampleAmountTestDataset.add(amountTestValue13);

// Demo value 14 - valid number (FOUR_WEEK), no exact pence
RegularAmount amountTestValue14 = new RegularAmount();
amountTestValue14.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
amountTestValue14.setAmount("33.33");
sampleAmountTestDataset.add(amountTestValue14);

// Demo value 15 - invalid number (LARGER than 11 char restriction!), and no exact pence
RegularAmount amountTestValue15 = new RegularAmount();
amountTestValue15.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
```

```
amountTestValue15.setAmount("999999999.99");
sampleAmountTestDataset.add(amountTestValue15);

// Demo value 16 - valid number (just within 11 char restriction), correct pence
RegularAmount amountTestValue16 = new RegularAmount();
amountTestValue16.setFrequency(RegularAmount.Frequency.WEEK);
amountTestValue16.setAmount("99999999.99");
sampleAmountTestDataset.add(amountTestValue16);

// Demo value 17 - invalid number, zero not dividable
RegularAmount amountTestValue17 = new RegularAmount();
amountTestValue17.setFrequency(RegularAmount.Frequency.WEEK);
amountTestValue17.setAmount("0");
sampleAmountTestDataset.add(amountTestValue17);

// Demo value 18 - test empty constructor, will fail validation
RegularAmount amountTestValue18 = new RegularAmount();
sampleAmountTestDataset.add(amountTestValue18);

// -----

// Demo value 19 - valid number, correct pence
RegularAmount amountTestValue19 = new RegularAmount();
amountTestValue19.setFrequency(RegularAmount.Frequency.TWO_WEEK);
amountTestValue19.setAmount("888.50");
sampleAmountTestDataset.add(amountTestValue19);

// Demo value 20 - valid number, correct pence
RegularAmount amountTestValue20 = new RegularAmount();
amountTestValue20.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
amountTestValue20.setAmount("0.08");
sampleAmountTestDataset.add(amountTestValue20);

// Demo value 21 - valid number, correct pence
RegularAmount amountTestValue21 = new RegularAmount();
amountTestValue21.setFrequency(RegularAmount.Frequency.TWO_WEEK);
amountTestValue21.setAmount("5000");
sampleAmountTestDataset.add(amountTestValue21);

// Demo value 22 - valid number, correct pence
RegularAmount amountTestValue22 = new RegularAmount();
amountTestValue22.setFrequency(RegularAmount.Frequency.QUARTER);
amountTestValue22.setAmount("130");
sampleAmountTestDataset.add(amountTestValue22);

// Demo value 23 - valid number, correct pence
RegularAmount amountTestValue23 = new RegularAmount();
amountTestValue23.setFrequency(RegularAmount.Frequency.YEAR);
amountTestValue23.setAmount("2958.28");
sampleAmountTestDataset.add(amountTestValue23);
```

```
// Demo value 24 - valid number, correct pence
RegularAmount amountTestValue24 = new RegularAmount();
amountTestValue24.setFrequency(RegularAmount.Frequency.QUARTER);
amountTestValue24.setAmount("2.21");
sampleAmountTestDataset.add(amountTestValue24);

// Demo value 25 - valid number, correct pence
RegularAmount amountTestValue25 = new RegularAmount();
amountTestValue25.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
amountTestValue25.setAmount("177929.08");
sampleAmountTestDataset.add(amountTestValue25);

// Demo value 26 - valid number, correct pence
RegularAmount amountTestValue26 = new RegularAmount();
amountTestValue26.setFrequency(RegularAmount.Frequency.QUARTER);
amountTestValue26.setAmount("1153407.58");
sampleAmountTestDataset.add(amountTestValue26);

// Demo value 27 - invalid number (huge input way beyond 11 char restriction)
RegularAmount amountTestValue27 = new RegularAmount();
amountTestValue27.setFrequency(RegularAmount.Frequency.FOUR_WEEK);
amountTestValue27.setAmount("8349758937458972398457982374589234759723947374");
sampleAmountTestDataset.add(amountTestValue27);

// Demo value 28 - invalid number, "MONTH" does NOT represent a valid weekly divisor
RegularAmount amountTestValue28 = new RegularAmount();
amountTestValue28.setFrequency(RegularAmount.Frequency.MONTH);
amountTestValue28.setAmount("40.00");
sampleAmountTestDataset.add(amountTestValue28);

// Demo value 29 - valid number (just within 11 char restriction), correct pence
RegularAmount amountTestValue29 = new RegularAmount();
amountTestValue29.setFrequency(RegularAmount.Frequency.YEAR);
amountTestValue29.setAmount("36329663.24");
sampleAmountTestDataset.add(amountTestValue29);

// Demo value 30 - invalid number, no correct pence
RegularAmount amountTestValue30 = new RegularAmount();
amountTestValue30.setFrequency(RegularAmount.Frequency.TWO_WEEK);
amountTestValue30.setAmount("1000000.01");
sampleAmountTestDataset.add(amountTestValue30);

return sampleAmountTestDataset;
}

/**
 * Internal method that takes in an array of RegularAmount
 * objects and performs validation on them, displaying any violation
 * messages.
 *
 * @param amountDataset an ArrayList of
```

```
* <code>RegularAmount</code> objects to validate.
*/
private static void validateSampleAmounts(ArrayList<RegularAmount> amountDataset)
{
    for(RegularAmount amt : amountDataset)
    {
        System.out.println("=====");

        if(DEBUG)
        {
            System.out.println("Validating amount... ");
        }

        // Validate the Bean/class
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        Validator validator = factory.getValidator();

        Set<ConstraintViolation<RegularAmount>> violations = validator.validate(amt);

        if(DEBUG)
        {
            System.out.println("Violations detected: " + violations.size());
        }

        if(DEBUG)
        {
            // Display the validation messages
            for(ConstraintViolation<RegularAmount> violation : violations)
            {
                System.out.println(violation.getMessage());
            }
        }

        // Did it pass the ConstraintValidator without any problems? (i.e is it VALID?)
        if(violations.size() == 0)
        {
            System.out.println(ANSI_GREEN + "PASSED:" + ANSI_RESET);
            // NOTE: Aggregate/count valid results to show at the end
        }
        else
        {
            System.out.println(ANSI_RED + "FAILED:" + ANSI_RESET);
        }

        System.out.println("-> Amount: " + amt.getAmount());
        System.out.println("-> Frequency: " + amt.getFrequency());
    }
}
```

```
/**
 * Internal method that displays an information menu of options to choose from.
 *
 * Used for when running the program in interactive command line mode.
 */
private static void printFrequencyChoiceMenu()
{
    System.out.println("Please enter frequency number [1-6], where frequency is one of:");
    System.out.println("[1] - WEEK");
    System.out.println("[2] - TWO_WEEK");
    System.out.println("[3] - FOUR_WEEK");
    System.out.println("[4] - MONTH");
    System.out.println("[5] - QUARTER");
    System.out.println("[6] - YEAR");
    System.out.println("");
    System.out.println("Enter 0 to quit application.");
    System.out.print("> ");

}

/**
 * Internal method (though accessible within the package) for validating a
 * given amount and frequency.
 *
 * @param inputAmount A String representing the currency value (e.g. "75.00").
 *
 * @param frequencyChoice A RegularAmount.Frequency enum value.
 */
protected static void runValidation(String inputAmount, String frequencyChoice)
{
    RegularAmount amount;

    // Protection against any dumb inputs
    if(inputAmount == null)
    {
        inputAmount = "-1.00";
    }

    if(frequencyChoice == null)
    {
        frequencyChoice = "0";
    }

    switch(frequencyChoice)
    {
        case "1":
            amount = new RegularAmount(inputAmount, RegularAmount.Frequency.WEEK);
            break;
        case "2":
            amount = new RegularAmount(inputAmount, RegularAmount.Frequency.TWO_WEEK);
```

```
        break;
    case "3":
        amount = new RegularAmount(inputAmount, RegularAmount.Frequency.FOUR_WEEK);
        break;
    case "4":
        amount = new RegularAmount(inputAmount, RegularAmount.Frequency.MONTH);
        break;
    case "5":
        amount = new RegularAmount(inputAmount, RegularAmount.Frequency.QUARTER);
        break;
    case "6":
        amount = new RegularAmount(inputAmount, RegularAmount.Frequency.YEAR);
        break;
    default:
        amount = new RegularAmount(inputAmount, RegularAmount.Frequency.WEEK);
        frequencyChoice = "0";
        System.out.println("Quitting application.");
        break;
}

if(!frequencyChoice.equals("0"))
{
    ArrayList<RegularAmount> validateAmount = new ArrayList<>();
    validateAmount.add(amount);
    validateSampleAmounts(validateAmount);
}

}

/**
 * Internal method that provides a keyboard prompt/input to interactively run
 * any number of validation tests on a given input of amount and frequency.
 *
 * User triggers program exit by selecting "0" as choice for frequency.
 */
private static void interactiveMode()
{
    String inputAmount;
    String frequencyChoice;

    System.out.println("");
    System.out.println("INTERACTIVE MODE");
    System.out.println("");

    Scanner keyboard = new Scanner(System.in);

    // Keep getting user input until they enter 0 for Frequency to quit application
    do
    {
        System.out.println("Please enter the currency amount: ");
```



```
        System.out.print("> ");
        inputAmount = keyboard.nextLine();
        printFrequencyChoiceMenu();
        frequencyChoice = keyboard.nextLine();
        runValidation(inputAmount, frequencyChoice);
    }
    while(!frequencyChoice.equals("0"));

    // Clean up input stream
    keyboard.close();
}

/**
 * Internal method that simply prompts an instructional usage method to the terminal.
 *
 * Used to indicate the command line option to pass to the program to trigger
 * the desired mode/behaviour.
 */
private static void printUsage()
{
    System.out.println("");
    System.out.println("=====");
    System.out.println("Usage: ");
    System.out.println("java -jar NHS_BSA_Assignment1.jar ARG");
    System.out.println("");
    System.out.println("Where ARG (without quotes) is one of: ");
    System.out.println("\1" : Run validator in interactive mode.");
    System.out.println("\2" : Interactive mode, verbose messaging");
    System.out.println("\3" : Run validator on demo test values.");
    System.out.println("\4" : Demo test values, verbose messaging.");
}

/**
 * Internal wrapper method to run any desired demonstration tests/example dataset.
 */
private static void runTests()
{
    validateSampleAmounts(getSampleAmountDataset());
}

private static void parseChoice(String option)
{
    switch(option)
    {
        case "1":
            setDEBUG(false);
            interactiveMode();
            break;
        case "2":
            interactiveMode();
    }
}
```

```
        break;
    case "3":
        setDEBUG(false);
        setRUNTESTS(true);
        runTests();
        break;
    case "4":
        setRUNTESTS(true);
        System.out.println("Initialising tests...");
        runTests();
        break;
    default:
        printUsage();
        break;
}

//return true;
}

/**
 * Main entry point.
 *
 * @param args Specify the operation mode.
 *
 * Takes a single numerical argument (String) from 1-4. Any other value
 * displays a usage message before exiting.
 */
public static void main(String args[])
{
    if(args.length == 1)
    {
        parseChoice(args[0]);
    }
    else
    {
        printUsage();
    }
}
}
```

12.1.6 RunAmountTest.java

```
/* RunAmountTest.java
 *
 * See LICENSE.txt in project root directory for license details.
 */
package com.aren.nhs_bsa_assignment;

import java.io.ByteArrayInputStream;
```

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

/**
 * JUnit test class for <code>RunAmount.java</code>.
 *
 * @author Aren Tyr.
 * @version 1.0 2020-08-25
 */
public class RunAmountTest
{
    /**
     * Test default constructor.
     *
     * Here for code coverage.
     */
    @Test
    public void testEmptyConstructor()
    {
        // Do nothing but instantiate
        RunAmount newRun = new RunAmount();
    }

    /**
     * JUnit test of <code>isDEBUG</code> and <code>setDEBUG</code> methods, of
     * class <code>RunAmount</code>.
     *
     * Mostly just a test of state, since it is just a simple getter &
     * setter, but here for completeness.
     */
    @Test
    public void testIsDEBUGandSetDEBUG()
    {
        boolean expResult = false;
        boolean result = RunAmount.isDEBUG();
        assertEquals(expResult, result);
        boolean DEBUG = true;
        RunAmount.setDEBUG(DEBUG);
        assertEquals(RunAmount.isDEBUG(), true);
        DEBUG = false;
        RunAmount.setDEBUG(DEBUG);
        assertEquals(RunAmount.isDEBUG(), false);
    }

    /**
     * JUnit test of <code>isRUNTESTS</code> and <code>setRUNTESTS</code>
     * methods, of class <code>RunAmount</code>.
     *

```

```
* Mostly just a test of state, since it is just a simple getter & setter, but here for completeness.
*/
@Test
public void testIsRUNTESTSsandSetRUNTESTS()
{
    boolean expResult = false;
    boolean result = RunAmount.isRUNTESTS();
    assertEquals(expResult, result);
    boolean RUNTESTS = true;
    RunAmount.setRUNTESTS(RUNTESTS);
    assertEquals(RunAmount.isRUNTESTS(), true);
    RUNTESTS = false;
    RunAmount.setRUNTESTS(RUNTESTS);
    assertEquals(RunAmount.isRUNTESTS(), false);
}

/**
 * Test of all of the non-interactive branches from the main method.
 */
@Test
public void testMain()
{
    String[] args = {"4"};
    RunAmount.main(args);
    args[0] = "3";
    RunAmount.main(args);

    // default branch of parseChoice, non-existent option
    args[0] = "5";
    RunAmount.main(args);

    // This will trigger the "default"/printUsage() branch
    String[] wrongArgs = {"1", "2", "3"};
    RunAmount.main(wrongArgs);
}

/**
 * Test interactive branch "1".
 *
 * These are split into separate test methods to avoid issues with
 * pollution/buffering of the input stream cache.
 */
@Test
public void testMainInter1()
{
    System.setIn(new ByteArrayInputStream("10.00\n0\n".getBytes()));
    RunAmount.main(new String[] {"1"});
}
```

```
/**
 * Test interactive branch "2".
 *
 * This one causes the while loop to iterate one more time.
 */
@Test
public void testMainInter2()
{
    System.setIn(new ByteArrayInputStream("10.00\n3\n50.00\n0\n".getBytes()));
    RunAmount.main(new String[] {"2"});
}

/**
 * Test all branches of the validation switch statement.
 */
@Test
public void testrunValidation()
{
    RunAmount.runValidation(null, null);
    RunAmount.runValidation("60.00", "1");
    RunAmount.runValidation("50.00", "2");
    RunAmount.runValidation("40.00", "3");
    RunAmount.runValidation("30.00", "4");
    RunAmount.runValidation("20.00", "5");
    RunAmount.runValidation("10.00", "6");
}
}
```