DRAFT Creating Financial Instruments

DRAFT

Creating Financial Instruments

with the Prometheus Financial Core

Developer Guide

Capital One Confidential

List of Figures and Tables

---

1. About the Prometheus Program

Prometheus is a Capital-One-wide program to modernize our banking systems and operations.

Our objective is to build a new, improved Enterprise Banking Platform backed by a Modern Financial Core so that the Lines of Business can "cross the canyon": migrate account processing from the current Legacy Cores. This includes peeling away business functionality from the Legacy Cores into new surround systems, such as Credit Bureau Reporting.

For details about the program, see the Prometheus Program Site.

1.1. About this guide

This guide describes how financial instruments are designed, programmed, tested, and deployed for use in production with the Prometheus Financial Core (PFC).

1.1.1. Acknowledgements

This guide represents the excellent work of members of the Prometheus Engineering Group (PEG-SF), not limited to the following persons:

- Annette Chen
- Matt Fellows
- Leandra Irvine
- Erik Jacobsen

1.1.2. Intended audiences and skills

This guide is for the following audiences:

1. Product managers who define requirements for new or existing Capital One financial products.

2. Computer programmers in the Capital One LOBs who are implementing financial instruments based on product requirements for use with the Prometheus systems.

We recommend the following skills and knowledge:

- Familiarity with basic accounting principles and standard practices, including:
  - Double-entry bookkeeping
  - Debits and credits
  - General Journal
  - General Ledger
  - Posting transactions
  - Accrued interest

    For background information, see these reference materials on basics of accounting.

- Deep knowledge of your organization's accounting needs.
- Comfort with computer programming.
- Familiarity with Scala programming language is a plus.
- Comfort with basic system administration on Linux or macOS.
- Experience with Capital One's infrastructure and tools, including GitHub.

1.1.3. Notational conventions

This guide uses the following notational conventions:

- Programming code, commands, and filenames are `monospace`.
- Lines omitted from program examples are indicated with `. . .` .
- Definition of terms and variables in text (but not in code excerpts) are *italic.*

1.1.4. Code snippets

Only code snippets

- This guide's examples include only snippets of code, not the complete code you might need.
- Some snippets might be directly usable in your instrument. Others might not.

1.1.5. Simplification in this guide

Much information in this guide is intentionally oversimplified to not cloud its focus: the financial instrument.

- Most diagrams are conceptual. They do not show the precise internal system architectures, physical implementations, or connections among components of all Prometheus systems.
- Exact movement of data through the system is not the subject of this guide.

1.1.6. Related essential documentation

- Prometheus Program Site: overall structure and management of the Prometheus program.
- *Prometheus Financial Core Theory of Operation.*
- Prometheus Financial Core engineering design and implementation documentation.
- Reference materials on basics of accounting.

1.2. Contacting us

Send us a message in Slack channel #peg-dsl. We are here to help you and to answer your questions.

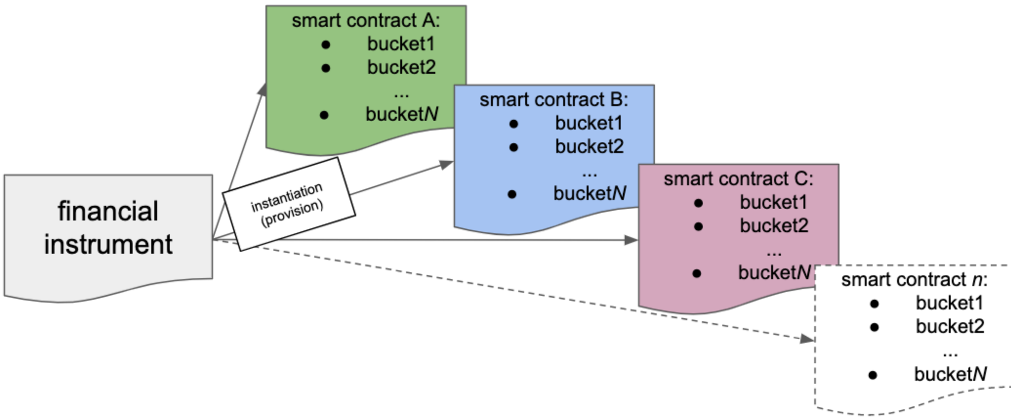
2. Overview to financial instruments

Financial instruments, sometimes called *instruments* for short, are a key part of the Prometheus Financial Core (PFC).

An *instrument* is a computer program constructed along the guidelines and practices described in this guide. It defines the accounting characteristics of a Capital One product. It is a template for customer-specific *smart contracts to* process incoming transactions according to the fields and program defined in the instrument.

Below is a simplified conceptual view of the relation of an instrument to its smart contracts.

Figure: Instrument to smart contract to bucket



The descriptions above are further refined below.

Table: instrument and smart contract defined

Function	Description
Financial instrument	A *financial instrument* encapsulates the structure and rules of a financial product, such as a checking account or installment loan. • The instrument acts as a template for a specific customer's *smart contract*. • Given a sequence of transactions, an instrument's definitions and program govern the evolution of the state of the customer's smart contract.
Smart contract	A *smart contract* is a unique instantiation of an instrument with terms tied to specific, unique accounting *buckets*. Bucket s are "holding spots for calculations and other interpretative functions of the instrument. • The creation of a smart contract based on an instrument is called *provisioning*. • One customer's smart contract is independent of other smart contracts based on the same instrument. • The Prometheus Financial Core's domain-specific language (DSL) determines the effect of the transactions against buckets defined by the instrument from which the smart contract was instantiated.

See also Account vs bucket.

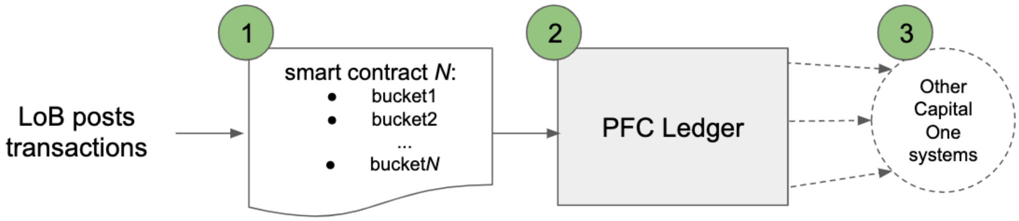
2.1. Instruments in day-to-day business

In the normal course of business:

1. Using the Enterprise Banking Platform (EBP) REST API, the LOBs post transactions to a smart contract, such as disbursement of funds, payments, fees, or refunds.

2. These posts are processed by the PFC Ledger system based on the financial instrument from which the smart contract was created. This results in bank accounting values.
3. These data are then made available for use in other Capital One systems, which is a topic outside the scope of this guide.

Figure: LOB transactions to smart contracts to PFC Ledger to other Capital One systems



2.2. Some important concepts and terms

Some of the terms used in this guide and in the PFC system are detailed here.

2.2.1. Financial instruments, allocations, and amortization schedules

The table below describes some important concepts for working with instruments and further refines the descriptions in the high-level overview.

Table: Definitions of financial instrument, allocations, and amortization schedules

Title	Description	Author
Financial instruments: configuration, buckets, and allocation	A financial instrument is a computer program written in the Scala programming language that instructs the PFC Ledger system how to execute the business logic of a financial product, such as an installment loan. An instrument consists of the following: • A set of configurations that express the accounting relationships of the instrument's financial buckets. Configurations can define double entry interactions, roll-ups, payment allocations, and more complex financial relationships. • A domain-specific language (DSL) program that defines how transactions are processed according to those configurations. An instrument acts as a template on which individual customer smart contracts are based.	Anna Chang and Matt Fellows, PFC Engineering
Payment allocations overview	Payment allocations are their own configuration element in an instrument definition, similar to roll-ups and double entries. Various types of allocations can be accommodated: simple fractional slices with remainder pennies spread among buckets, slices defined by a fixed sequence of amounts with an open-ended remainder, slices defined by an orderable sequence, and slices partially defined by the consumer.	Anna Chang, PFC Engineering
Overview to amortization design and implementation	An amortization schedule is not tied to a specific smart contract. This independence allows us to view “what if” variations without affecting an actual contract terms; that is, how the schedule changes with differing initial values. Possible contract terms can be varied to generate different schedules for a potential loan product. Equal loan payment amounts are based on the DayCount algorithm 30/360 International Swaps and Derivatives Association (ISDA), which simplifies calculations by "flattening" the number of days of the month to 30 and the days of the year to 360, with special rules for the 31st day-of-month.	Serena Chan, PFC Engineering

2.2.2. Account vs bucket

The word "account" in bank accounting can be ambiguous. This guide uses the word "account" in the traditional accounting meaning, like a "customer checking account" or a "General Ledger (GL) account".

Instruments avoid this ambiguity by using the idea of a *bucket*. A bucket is a "holding spot" for the results of transactions as computed by the PFC Ledger and the instrument itself. A smart contract has multiple buckets. Buckets are further described in Instrument fields: financial relationships of product and defined programmatically in Summary of instrument helper functions.

2.2.3. DSL, or domain-specific language

A domain-specific language (DSL) is a computer programming language specialized to make writing programs easier for a specific knowledge domain.

The Capital One financial instrument DSL is tailored to the bank accounting domain, is a primary part of every instrument, and is one of the main subjects of this guide.

2.2.4. PFC Ledger, or "Luca"

A key component of the PFC system is the *Ledger*, which processes transactions by running instrument DSL programs and other important functions. (For historical reasons, this component is also sometimes referred to as "Luca".)

> ⚠ **Not the General Ledger**
>
> The PFC Ledger must not be confused with Capital One's General Ledger accounting systems. The PFC Ledger is strictly concerned with processing incoming transactions based on smart contracts instantiated from instruments, with relaying the resulting data to other Capital One systems, and other functions of the Prometheus systems.

- For a description of the Ledger's place in the PFC systems, see How transactions are processed by the PFC Ledger.
- In developing instruments, you run a local copy of the PFC Ledger service to validate or debug your instrument. See One-time setup of your local system for instrument programming.
- In production, you never interact directly with the PFC Ledger system.

2.2.5. Instrument developer repository, or "Herschel"

The git repo Herschel includes all the Scala packages that you need to develop your instrument, including helper functions.

- To program your instrument, you make a local copy of this repo. See One-time setup of your local system for instrument programming.
- In addition, when you are satisfied with your instrument, to publish it, you `git push` it to Herschel. For more information, see Deploying.

2.2.6. End-of-Day (EOD) processing

End-of-Day (EOD) processing is a function of the PFC systems. At the scheduled end of a day, the system itself sends a transaction that signals the end of day. This signals instruments to emit transactions such as interest accrual or account closing to pass through the PFC Ledger's interpreters.

EOD processing always takes place at the end of the "day", which in your instrument you can base on the local timezone.

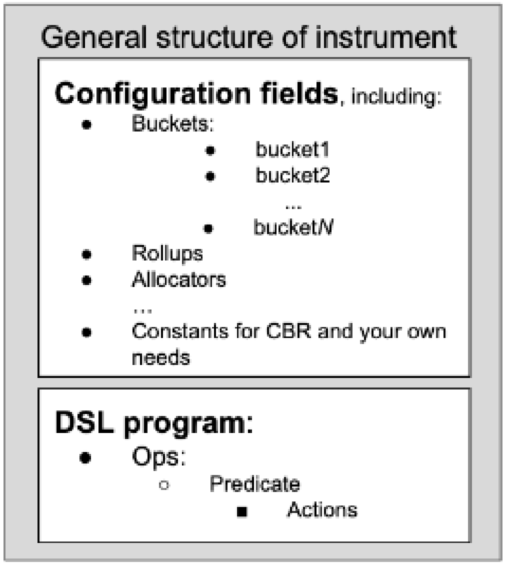
In addition, your instrument might need to accommodate intervals other than days, like weeks, months, or years, when the instrument must do extra processing, such as generating end-of-cycle bills.

2.3. Deeper dive: structures of instruments

An instrument has the following basic parts:

- *Configuration fields* that define accounting relationships and data types. Also included are constants, some of which might be required for Credit Bureau Reporting (CBR).
- A *domain-specific language (DSL) program* with Ops that operate on those fields.

Figure: general structure of an instrument



2.3.1. Instrument fields: financial relationships of product

The summary below is extracted from the full definitions in Financial instruments: configuration, buckets, and allocation.

Table: summary of instrument field names

Field name and link to engineering design docs	**Description/Comment**
SimpleConfig **fields**	SimpleConfig fields are essentially "data types". You use them to define the characteristics of fields: • Amount • BasisPoint • BigDecimal: Decimal floating-point numbers of arbitrary precision. By default, the precision approximately matches that of IEEE 128-bit floating point numbers (34 decimal digits, HALF_EVEN rounding mode). • DayOfMonth • Boolean • Currency • Int • LocalDate • LocalTime • NaturalAmount • Period: A date-based amount of time based on ISO-8601, such as "2 years, 3 months and 4 days" or Universal Time Coordinates (UTC). • String • ZoneId: A time-zone ID in the form "America/New_York", "America/Los_Angeles". See the IANA Time Zone Database for valid values. Programmatic definitions of the other types are in Scala programmer's reference for financial instruments.
Buckets	A bucket is a "container" that holds: • The amount of the most recent transaction recorded in the bucket. • The bucket's balance.

Double Entries	Double entries always come in pairs: the "sides" of posting to a traditional general journal: • debit (DR). • credit (CR).
Allocators	Definitions of how payments should be distributed among various buckets, such as interest paid, principal paid, or overpayment.
Rollups	Aggregations of buckets.

2.3.2. DSL program operators: Ops on fields

A DSL program in Scala is a specification of how the *state* of a smart contract should evolve.

1. Input to the program is the contract's current state and a transaction.
2. Output from the program is one or more transactions for further interpretation by the PFC Ledger's interpreters.

The specification has the programming structure of operators, or Ops, in the form of an Abstract Syntax Tree (AST):

- Sequence of Ops
  - Op
    - Predicate
      - expressions
    - Sequence of Actions
      - expressions

An Op consists of a single *Predicate* and one or more *Actions*:

- Predicates and Actions are composed of expressions.
  - An *expression* is a tree of Scala case classes defined in `com.capitalone.fsor.core.containers.Expressions`. The syntax for these expression case classes is documented in the Scala programmer's reference for financial instruments.
  - Expressions are interpreted by the PFC Ledger. See How transactions are processed by the PFC Ledger.
- Actions are evaluated only if the Predicate is true.
  - Actions specify transactions. They never act directly on the contract state.

2.3.3. How transactions are processed by the PFC Ledger

Transactions posted to a smart contract are processed by the PFC Ledger based on the fields and DSL program defined by the instrument from which the smart contract was instantiated.

Figure: How transactions are processed by the PFC Ledger

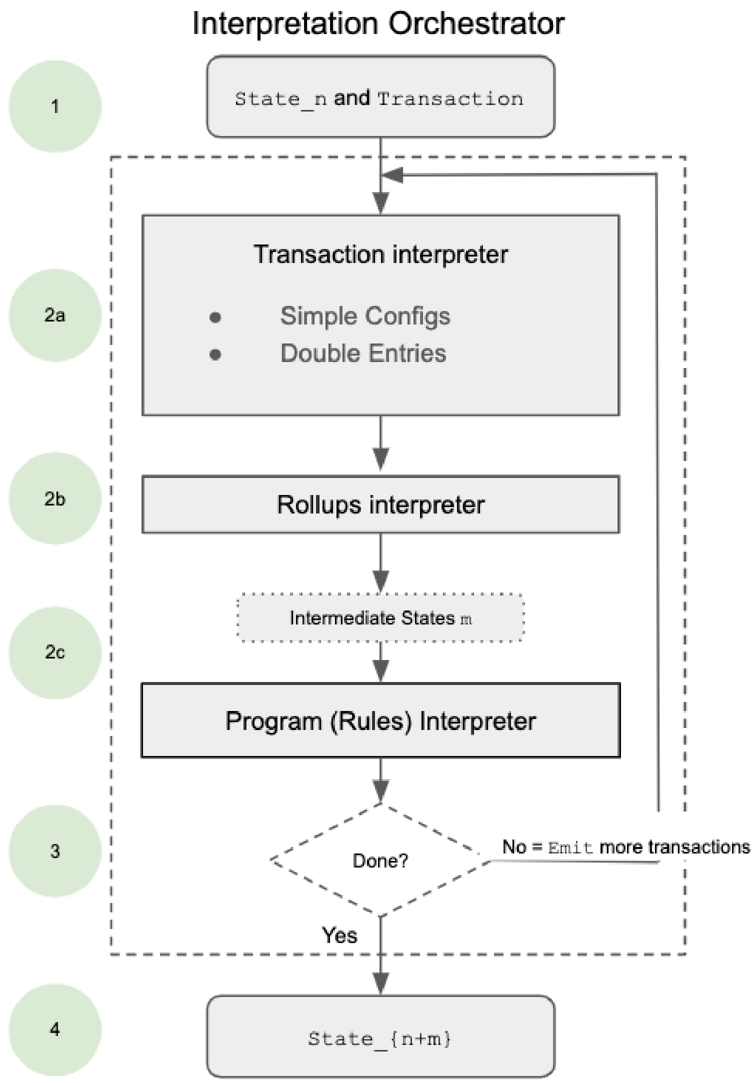
<table>
<tr>
<td>
Structure of instrument
Fields
Program
Functions in the PFC Ledger
Transaction Interpreter.
Rollups Interpreter.
Program (rules) Interpreter.
Program and process
Input:
The state of the contract, <code>State_n</code>
</td>
<td></td>
</tr>
</table>

- A `Transaction`.

1. Interpreters:
   a. Transaction interpreter.
   b. Rollups interpreter.
   c. Rules are interpreted based on intermediate state.

      An Op never modifies a contract state directly. Instead, in the instrument Scala program, you use the `Emit` class in an Op's action to issue more transactions and so repeat the loop through the interpreters. See Emit more transactions for interpretation.

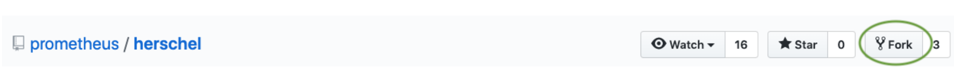
2. “Done” = no more `True` predicates.
3. Output: `State_{n+m}`.



3. One-time setup of your local system for instrument programming

1. Configure your system for working with the PFC Ledger. See this project setup, with the following guidance:
   a. Focus on the Docker, sbt, and IntelliJ sections.
   b. Ignore the Kafka/Zookeeper, Postgres, and Flyway sections.

2. Git-fork the Herschel git repo at `https://github.cloud.capitalone.com/prometheus/herschel`.



   a. Clone your fork:

```
git clone https://github.cloud.capitalone.com/prometheus/herschel
```

   b. Make a local development branch in your fork of the Herschel git repo:

```
cd herschel
```

```
git checkout -b myInstrumentBranch
```

3. The following optional read-eval-print loop (repl) tools are helpful in testing:
    a. Scala `repl` Ammonite: started from Herschel with `sbt test:run`
    b. Clojure `repl`: started with `lein repl` in a Clojure project folder. A Capital-One-developed Clojure client is also available.

4. Designing

Stated simply, designing your instrument is mapping your accounting requirements to the instrument fields and program.

One approach to creating an instrument involves the following general steps.

1. Set up your local system for instrument programming.
2. Learn about the structures of instruments.
3. With your deep knowledge of the product's accounting requirements, sketch out the rough design of your instrument. Rely on a product requirements document (PRD).
4. Program the instrument.
5. Test it.
6. Deploy it.

4.1. Know your own requirements

Here are some questions to ask.

- Do you have a product requirements document (PRD) that defines the product?
- Can you express your instrument's behavior in English?
- Can you write a mathematical formula for each accounting function of the product?
- Do you have an existing similar product you can use as a model?
- What kinds of transactions will you process?
- What kinds of buckets do you need?
- How will you allocate transaction amounts to various buckets?
- Do you need rollups?
- What other requirements do you have?

4.2. Rely on your PRD for concrete details

A good starting point for your design is a product requirements document (PRD). The statements in the PRD should give you guidance about what your instrument needs to do.

- The PRD's explicit math formulas are particularly useful.
- If the PRD has no math, try to write concrete formulas based on the PRD's language.

For an example, see this good model of a PRD for the example instrument discussed in this guide.

4.3. Example instrument: short-term POS installment loan

This guide relies on a specific example instrument to help you learn about creating instruments: a short-term Point-of-Sale (POS) installment loan.

As you read through the programming snippets in this guide, you should examine the full Scala code of the example instrument. The example program is located in the Herschel repository as follows:

ShortTermPOSILSept2019Maker.scala

Hardcoded payment periods

The example instrument assumes that the loan it represents must be paid off in six payments, which are hardcoded in the instrument.

4.3.1. Other example instruments

Other example instruments that might be useful in developing your own are in the Herschel repository:

> https://github.cloud.capitalone.com/prometheus/herschel/tree/master/src/main/scala/com/capitalone/fsor/herschel/instrumentmakers

5. Programming

This section augments the helpful comments in the example instrument and highlights constructs of particular interest.

5.1. Overall structure of example instrument

The basic parts of an instrument described in Deeper dive: structures of instruments is followed by the example instrument. This subdivision is one-for-one with the comments in the example instrument, which you should refer to.

5.2. Basic Scala package, imports, and instrument object

As shown in the example instrument, every instrument needs to begin with a Scala package declaration and import statements.

It must also have an object declaration of the instrument name that `extends DSLInstrument`. The object declaration surrounds the entire instrument definition, like so:

```
object yourInstrumentName extends DSLInstrument {
// entire
// instrument
// definition
}
```

5.3. Helper functions

Instrument helper functions can aid you in developing instruments. They are used throughout the example instrument. See this summary.

5.4. Defining the fields

Name your fields so you can tell at a glance their purpose and use according to the logic of your Ops.

The example instrument uses a prefix `is...` to indicate a boolean field in the form of a question that can be set `true` or `false`. For example:

```
is-fee-upper-limit-initialized
```

The corresponding Op that works with this field follows a convention of the word `...Op` as a suffix:

```
initializeFeeUpperLimitOp
```

5.4.1. Defaults

Your instrument should probably set the types of some defaults required fields.

The defaults for a loan should probably include all of the following.

```
val requiredDefaultSettings = Set(
          NaturalAmountField("late-fee-amount"),
          NaturalAmountField("regular-payment-amount"),
          PeriodField("payment-period"),
          IntField("number-of-payments")
 )
```

For description of `NaturalAmountField` and other types, see SimpleConfig fields.

Your instrument might require other defaults.

5.4.2. Constants

Many instruments have constant data that should be declared at the beginning. The example instrument sets constants for the following:

- Billing cycle: `cycleStrategy`.
- Date to assess late fees: `unpaidBillWindowPeriod`.
- A local date `initializeDateConst`.
- Fee limit: `feeLimitPercentageConst`.

5.4.2.1. Credit Bureau Reporting (CBR) fields

Fields related to CBR that your instrument might require are detailed in the following links.

> **WIP**
>
> The documents below are work-in-progress.

- Installment Loan Account Data needed for CBR. The third column is most helpful.
- EBP Canonical Data Model. Of particular interest for installment loans is the YAML file under the heading **Account - Non Revolving Credit Product Sub-Model**.

5.4.2.2. Example of CBR constants

Most instruments must set constants that relate to CBR, as shown in the example instrument.

```
// **** CBR Fields ****
BooleanField("are-cbr-fields-initialized"),
//StringField("Account/AccountId"), // awaiting implementation
StringField("Account/ProductCategory"),
StringField("Account/ProductType"),
LocalDateField("Account/AccountOpenDate"),
NaturalAmountField("Account/NonRevolvingCreditProduct/LoanTerms/OriginalLoanAmount"),
...
ConstrainedStringField("Account/AccountLifeCycleStatus/StatusCode",
Enumeration(List("OPENING", "OPEN", "CLOSING", "CLOSED", "ESCHEAT"))),
...
```

5.4.3. SimpleConfigs

SimpleConfigs fields declare the data type of program variables. For example, the example instrument sets the following:

```
...
LocalTimeField(eodCutoffTimeKey),
ZoneIdField(zoneIdKey),
...
```

- `eodCutoffTimeKey` is the local time that PFC End-of-Day marker generation occurs and is processed by this Op.
- `zoneIdKey` conforms to ISO 8601 and is described in Instrument fields: financial relationships of product.

5.4.4. Buckets

First, define the buckets your instrument needs and their types. Then declare a `Set` of `buckets` that includes all of those buckets. For example:

??HOLDING SPOT NOTE to Reviewers: any decision what to do about the name "ach-in"?

```
// Buckets
val buckets = Set(
   DebitBucketField("ach-in", Asset, Company),
   CreditBucketField("ach-out", ContraAsset, Company),
   CreditBucketField("unallocated-payment", ContraAsset, Customer),
   CreditBucketField("fee-income", Income, Company),
   CreditBucketField("payment-waypoint", ContraAsset, Customer)
CreditBucketField("refund-waypoint", ContraAsset, Customer),
) ++ billPackage.buckets
```

Compare Rollups.

5.4.4.1. CreditBucketField and DebitBucketField

As shown in Buckets, the helper functions `CreditBucketField` and `DebitBucketField` define the function of a field and its accounting characteristics:

- Asset.
- The offset ContraAsset.
- Income.
- The entity involved: either `Customer` or `Company`.

Example:

```
...
DebitBucketField("ach-in", Asset, Company),
...
CreditBucketField("unallocated-payment", ContraAsset, Customer),
CreditBucketField("fee-income", Income, Company),
...
```

5.4.4.2. BucketMaps

A bucketMap is a mechanism for organizing fields for easier use.

5.4.5. Rollups

First, define your individual rollup fields. Then logically group them as a `Set` of `rollups` so you can process that entire set, and not the individual fields or Ops themselves.

- For example, the individual `balanceRollup` shown below combines the fields `overpayment`, `unallocated-payment`, `principal`, and `fee`.
- `balanceRollup` is then combined with `principalBalanceRollup` and other rollup definitions into a `Set` named `rollups`.

See also the progression of the `overpayment` field from `balanceRollup` and `principalBalanceRollup` through the entire instrument in Example tying it together: tracing a field through its Ops.

Figure: rolling up the individual rollups

	Individual rollup definitions

Define the individual rollups	`val balanceRollup = DebitRollupField(`
	` "balance",`
	` Set(`
	`   Constituent("overpayment", Set(bucketMap("overpayment"))),`
	`   Constituent("unallocated-payment", Set(bucketMap("unallocated-payment")))`
	` ) ++`

	`billPackage.rollupMap("principal").constituents ++ billPackage.rollupMap("fee").constituents )`
	`val payoffRollup` `...`
	`val outstandingPaymentRollup` `...`
	`val principalBalanceRollup` `...`

	`val billBalance1to2Rollup = DebitRollupField(`
	
	`...`

	`... // Other billBalance rollups 2 through 5`
	`val billBalance1to6Rollup = DebitRollupField(` `...`

Combine the rollups as a `Set`	`// Rollups`
	`val rollups = Set(`
	`balanceRollup,`
	`payoffRollup,`
	`outstandingPaymentRollup,`
	`principalBalanceRollup,`
	`billBalance1to2Rollup,`
	`...`
	`billBalance1to6Rollup`
	`) ++ billPackage.rollups`

Compare Buckets.

5.4.6. Allocators

Allocators determine how payments should be spread among buckets, such as fees, interest, principal, and overpayment. For the theory of allocators, see the engineering design document Allocators.

The following lines create an `allocatorMap` that spreads a payment via a `rollupMap` to these buckets:

- "principal 1 through 6".
- "paid principal due 1 through 6".
- "overpayment".

```
// Allocators
  val refundAllocator = AllocationField(
  "refund-allocator",
  List(
    Target(rollupMap("principal-unbilled-balance"), bucketMap("principal-unbilled-paid")),
    Target(rollupMap("principal-6"), bucketMap("principal-due-paid-6")),
    ...
    Target(rollupMap("principal-1"), bucketMap("principal-due-paid-1"))
    ),
 bucketMap("overpayment")
)
```

5.4.7. Double entries

This snippet from the example instrument shows the `DoubleEntryField` class to specify the two sides of a transaction.

```
...
DoubleEntryField("payment", bucketMap("ach-in"), bucketMap("payment-waypoint")),
DoubleEntryField("refund-payment", bucketMap("ach-in"), allocatorMap("refund-allocator")),
...
```

1. A "`payment`" transaction will be posted against the `bucketMaps` `ach-in` and `payment-waypoint`. The field `payment-waypoint` is a transitory field used in other calculations.
2. A "`refund-payment`" will be posted against "`ach-in`" and an `allocatorMap` named "`refund-allocator`".

5.5. Writing the Ops

The sequence of the Ops in an instrument is not order-dependent. They can come in any order anywhere. However, you might want to sequence them to follow your own thinking for easier understanding and clarity.

5.6. General structure of an Op and Emits

As shown in the example program, an Op has a general structure that follows a pattern generalized below. Some of this structure and naming of Ops is by convention for easier understanding.

- Initializing the objects
- Setting certain variables, operating on them, and testing for certain conditions, including the end of the Op.
- An `Emit` of more transactions for more interpretation:
  - Recall that an Op never acts on the contract state directly. The `Emit` statement sends the intermediate contract state and its associated transactions through the interpretation orchestration loop for further interpretation.
  - An Op does not necessarily have to `Emit` but usually does.

```
val opNameOp (  // the Op suffix is for clarity
  Not(Get("some-end-of-op-flag")), // Test for end of the op flag as false
    //
    // Test certain fields for conditions
//
    // Other operations on certain fields
    //
),
//
// Emit more transactions until some condition instructs the op
// to set a field to stop processing
Emit("some-end-of-op-flag", BoolConst(true)) // Set the end of Op variable to true.
)
```

Testing for `true` depends on your Op's logic. You could test for `false` but that might be less obvious than testing for `true`.

5.6.1. Object initialization

The example instrument groups all initialization of objects at the top of the program block. This grouping is a good practice for ease of maintenance.

```
// Initialization
object Initialization {
...
}
```

5.6.1.1. Emitting CBR fields

The `initializeCBRFieldsOp` emits values for some standard data that must be reported. These constants are defined earlier in the instrument.

```
val initializeCBRFieldsOp = Op(
"initialize-cbr-fields",
Not(Get("are-cbr-fields-initialized")),
Emit("are-cbr-fields-initialized", BoolConst(true)),
Emit("Account/ProductCategory", StringConst(PRODUCT_CATEGORY)),
Emit("Account/ProductType", StringConst(PRODUCT_TYPE)),
Emit("Account/NonRevolvingCreditProduct/LoanTerms/CyclePaymentAmount", Get("regular-payment-amount")),
...
)
```

5.6.1.2. A list of initialization Ops

As with the examples of buckets and rollups, the various Ops to initialize values are wrapped in a List:

```
val ops = List(
initializeDueDatesOp,
initializeRemainingBillsOp,
initializeLoanAmountOp,
initializeFeeUpperLimitOp,
initializeCBRFieldsOp
)
```

5.6.1.3. Setting up the loan payment due dates

The short term POS installment loan instrument hardcodes the due dates of payments in six periods:

```
val initializeDueDatesOp = Op(
   "initialize-due-date",
    Not(Get("are-due-dates-initialized")),
    Emit("due-date-6", initializeDateConst),
    ...
    Emit("due-date-1", initializeDateConst),
    Emit("are-due-dates-initialized", BoolConst(true))
   )
```

5.6.1.4. An Op to initialize a loan's upper fee limit

Below is an annotated simple Op to get the upper limit on a loan's fees.

```
val initializeFeeUpperLimitOp = Op(
"initialize-fee-upper-limit",
And(
  Not(Get("is-fee-upper-limit-initialized")), // Test if end condition has been met
  Get("is-loan-amount-initialized")           // Make sure the loan amount has been set
),
 // Calculate the upper limit on fees
Emit("fee-upper-limit", ClampNatural(Multiply(ToAmount(Get("loanAmount")), feeLimitPercentageConst))),
// The Op is done. No more interpretation loops.
Emit("is-fee-upper-limit-initialized", BoolConst(true))
)
```

5.6.2. Ops to do the real work

The Ops briefly mentioned here all emit various transactions to do the work that is the heart of a loan. Examine the example instrument for complete code.

5.6.2.1. Provision the loan

After the loan is approved, funds must be disbursed. This is accomplished by the `Activation` object with its `List` of Ops to watch for disbursement of funds.

5.6.2.2. Create bills

`createBillsOp` does the work of setting up bills for each payment period. It distinguishes between the first bill, because of disbursement of funds, and the remaining bills, as explained in the commentary.

See also Example tying it together: tracing a field through its Ops, which examines the `overpayment` field in various Ops.

5.6.2.3. Track late and unpaid bills

Because the example instrument represents a loan with six fixed payment periods, it uses the `DerivedValues` object to track late payments and the `UnpaidBills` object to track unpaid bills.

5.6.2.4. An example Op for End-of-Day processing

This Op's logic handles the effect of PFC End-of-Day processing on the smart contract.

```
...
val runEodPred = And(HasChanged(lastEndOfDayKey), Get("is-disbursed"))
...
val eodDueDateOp = Op(

  "eod-on-due-date",
  multiAnd(
  runEodPred,
  Eq(nextCycleDateExpr, Get(currentDateKey))
),
// Set a variable that triggers the running of a post-bill allocator
Emit("should-run-post-bill-allocator", BoolConst(true))
)
```

Consider whether your instrument should be aware of EOD processing and if so, what it must do at cyclical intervals such as day, week, month, or year.

5.6.3. Example tying it together: tracing a field through its Ops

In this section, the example program's field `overpayment` is traced from its first definition through the Ops that work with it.

This example is more useful if you follow the field through the entire program source. Pay attention to the helpful commentary.

Output on the left is from `grep`:

```
grep overpayment ShortTermPOSILSept2019Maker.scala
```

Table: annotated grep output for overpayment field traced through the instrument

Individual lines grepped from ShortTermPOSILSept2019Maker.scala	Comment
1. `Constituent("overpayment", Set (bucketMap("overpayment"))),` 2. `Constituent("overpayment", Set (bucketMap("overpayment"))),` 3. `bucketMap("overpayment")`	1. Field `overpayment` defined in the `DebitRollupField` named `balanceRollup`. 2. Field `overpayment` also defined in the `DebitRollupField` named **`principal`**`BalanceRollup`. 3. From the `AllocatorField` named `refundAllocator` for possible refund to customer.

4. `DoubleEntryField("shift-overpayment", bucketMap("overpayment"), bucketMap("unallocated-payment")),` 5. `(Emit("shift-overpayment", ClampNatural(Get("overpayment"))) ::` 6. `(Emit("shift-overpayment", ClampNatural(Get("overpayment"))) ::`	4. The `DoubleEntryField` named `shift-overpayment` puts the value of `overpayment` into `unallocated-payment`. 5. From `createBillOp`, emit another transaction for interpretation. The `ClampNatural` class forces (that is, typecasts) a numeric value of an `Amount` (which can be any integer, positive or negative) to a `NaturalAmount`, a positive integer or zero. **Negative Amount = NaturalAmount 0** ClampNatural converts a negative value of `Amount` to zero because a `NaturalAmount` posted as a debit or a credit must *not* be negative, as required by double-entry bookkeeping. 6. From `createFirstBillOp`, emit another transaction for interpretation, as explained by the comments in the program.

5.7. Validating the instrument

5.7.1. Obtaining the instrument in JSON

Use the Ledger's DSL development tools to compile your instrument.

Steps:

1. Setup your program to extend `DSLInstrument`.
2. Run your program with `sbt`.

5.7.2. Setup for using DSLInstrument

The `DSLInstrument.scala` program is located as follows:

https://github.cloud.capitalone.com/prometheus/herschel/blob/master/src/main/scala/com/capitalone/fsor/herschel/instrumentmakers/DSLInstrument.scala

Important highlighted snippets of the `DSLInstrument.scala` program are shown below in comparison with the `example` program that uses them.

Table: DSLInstrument.scala heading compared to ShortTermPOSILSept2019Maker.scala

DSLInstrument	ShortTermPOSILSept2019Maker

`package com.capitalone.fsor.herschel.instrumentmakers`
 	
`import java.security.MessageDigest`
 	
`import com.capitalone.fsor.core.containers.Expressions.Program`
`import com.capitalone.fsor.core.containers._`
`import com.capitalone.fsor.core.parser.circe.InstrumentCodecs._`
`import io.circe.syntax._`
 	
`trait DSLInstrument` `{`
`...`
`}`
	`package com.capitalone.fsor.herschel.instrumentmakers`
	
	`import com.capitalone.fsor.core.common.NaturalAmount`
	`import com.capitalone.fsor.core.containers.Expressions._`
	`import com.capitalone.fsor.core.containers.RollupField.Constituent`
	`import com.capitalone.fsor.core.containers._`
	`import com.capitalone.fsor.herschel.dsl.syntax._`
	`import spire.math.Natural`
	
	`object ShortTermPOSILSept2019Maker extends DSLInstrument`
	`{`
	`...`
	`}`

`object DSLInstrument {`
`...`
`}`	

5.7.2.1. Generating JSON format of your instrument

For details about generating the JSON representation of your instrument, see the instructions towards the end of Develop.md.

5.7.2.2. DSLInstrument commands in sbt

Command	Description
`list`	Lists names of available instruments.
`show` *`instrumentName`*	Displays the internal representation of the instrument identified by *`instrumentName`*.
`print` *`instrumentName`*	Displays the external representation of the instrument.

5.7.3. Debugging

There are several ways you can debug the interpretation of your instrument.

- If the log level is set to `DEBUG`, the Ledger logs the full interpretation lineage.

  Set the following environment variable in your `docker-compose` file:
  `LUCA_SERVER_LOG_LEVEL=DEBUG`

  Then, to view the log output in realtime:
  `docker logs -f fsor_luca_1`

- To see only the interpretation loop logs:

  a. Install `jq` on your MacBook with this command:
     `brew install jq`

  b. Run the following:
     `docker logs -f fsor_luca_1 | grep -o "{\"Audit\".*" --line-buffered | jq`

- You can also inspect the interpretation lineage in the unit tests because you can make the interpreter return an `AuditedSnapshot` with the following code:

AuditedSnapshot code

```
decoratedInterpreter.interpret(snapshot, TypedKeyMap.unsafeApply(transactions.last)) match { case Left(e) =>
... case Right(audited @ AuditedSnapshot(actual, _)) => // You can print out the audit trail here
println(Audited.showAuditedSnapshot.show(audited)) ... }
```

6. Testing

Programs you can imitate and tools for testing are located in the Herschel repo as follows:

https://github.cloud.capitalone.com/prometheus/herschel/tree/master/src/**test**/scala/com/capitalone/fsor/herschel

6.1. Unit tests

Unit tests can quickly verify the correctness of your instrument without your having to start the local PFC Ledger service.

Each instrument should have a corresponding spec in Herschel's `test` subdirectory. By convention for clarity, the filename of your test program should end with the word `Spec`, as shown below:

```
src/test/com/capitalone/fsor/herschel/instrumentmakers/yourInstrumentNameSpec.scala
```

Instrumentation:

- At minimum, unit tests should include the following types of tests, as illustrated in `ShortTermPOSILSept2019MakerSpec.scala`. Highlighted snippets are shown in Instrumenting your program for unit tests.

    1. A test that runs ProgramValidation.
    2. A test that validates types.
    3. A test of the initial state after provisioning.
    4. Testing sequences of transactions to ensure they result in the expected state.

- Additionally, there are various test helpers you can take advantage of:
    - Helpers: General validation helpers.
    - Generators: Test data generators for fields.
    - Fakes: Fakes for dependencies such as database.

6.1.1. Instrumenting your program for unit tests

The following program snippets are from `ShortTermPOSILSept2019MakerSpec.scala`. It highlights lines that you add to the test version of your instrument to exercise the four types of unit tests.

```
package com.capitalone.fsor.herschel.instrumentmakers

import java.time.{LocalDate, LocalTime, Period, ZoneId}

import com.capitalone.fsor.core.api.DataValue.StringDataValue
import com.capitalone.fsor.core.common.Amount
import com.capitalone.fsor.core.containers.{CreditBucket, DebitBucket, Key}
import com.capitalone.fsor.herschel.fakes.CommandHandlerFake
import com.capitalone.fsor.luca.interpreter.ProgramValidatorNec
import org.scalatest.prop.TableDrivenPropertyChecks
import org.scalatest.{FlatSpec, Matchers, Succeeded}
import com.capitalone.fsor.herschel.Helpers.{
multiTransactionChecker,
multipleEODTransactions,
naturalAmount,
validateTypesForExpressions
}

// Replace all instances of ShortTermPOSILSept2019MakerSpec with the name of your instrument
// FYI the non-test version of the instrument extends DSLInstrument
class ShortTermPOSILSept2019MakerSpec extends FlatSpec with TableDrivenPropertyChecks with Matchers {

val testSubject = ShortTermPOSILSept2019Maker

...

it should "pass Program validation" in {
val instrument = shared.instrument
ProgramValidatorNec.validate(instrument).toEither.toOption.get shouldEqual shared.instrument
}

...

it should "type check successfully" in {
val result = CommandHandlerFake.provision(
...
)
```

```
...
it should "result in the expected snapshot upon provision" in {
val result = CommandHandlerFake.provision(
...

it should "respond correctly to a sequence of transactions" in {
val initialSnapshot = CommandHandlerFake.provision(
...
```

6.1.2. Running the unit tests

Run the tests with the following command:

```
sbt "testOnly "yourInstrumentNameSpec.scala"
```

6.2. Running the instrument interactively against the local PFC Ledger

To verify that an instrument is behaving correctly, run it against the local PFC Ledger service.

6.2.1. Serializing the instruments

To write the serialized instruments into a mapping of instrument names to hashes, run the following command:

```
sbt serialize -l herschelVersion
```

Option	Description
`herschel/instrument-name-map`	Default location of output is the generated subdirectory `instrument-name-map` in the base `herschel` directory.
`HERSCHEL_JSON_DIRECTORY`	Environment variable to specify a different location for the serialized output.
`herschelVersion`	Any arbitrary value when serializing locally. The value `snapshot` is often used.

6.2.2. Starting the local PFC Ledger service

To interactively test the instrument, you need a local instance of the Ledger service.

To start a local instance of the Ledger service, follow these steps:

```
cd fsor # Change to the fsor directory
make run # Start the service
```

6.2.3. Running the Clojure interactive client app against the local PFC Ledger

You can use an interactive client in Clojure to test instruments against the local PFC Ledger service.

For installation and usage, see the clj-luca-client documentation.

7. Deploying

Publishing your instrument consists of git-adding, git-committing, git-pushing, and git-merging it into the herschel git repository.

On each merge to the Herschel master branch, the build for Herschel publishes all instruments as JSON to an instrument repository on S3 to make them available to other parts of the PFC system.

The filename for each instrument is calculated from sha256sum of the minified JSON. Each instrument is published with the following file names:

- `yourHashedFileName.pretty.json`
- `yourHashedFileName.min.json`
- `yourHashedFileName.metadata`

7.1. Prerequisites: reviews and sign-off

NOTE to Reviewers: what sort of product management or legal signoffs are needed?

7.2. Steps to publish instrument

These are the mechanics of publishing.

1. Change directory to your local forked copy of the herschel repo.

2. Make sure you are up to date:
   `git pull`

3. Switch to the development branch that you created in One-time setup of your local system for instrument programming. For example:
   `git checkout myInstrumentBranch`

4. Copy your main file *instrumentName*`.scala` file into your local copy of `herschel/src/`**`main`**`/scala/com/capitalone/fsor/herschel/instrumentmakers`

5. Copy your test file *testOfinstrumentName*`.scala` file into your local copy of `herschel/src/`**`test`**`/scala/com/capitalone/fsor herschel/instrumentmakers`

6. Git-add your production and test versions of the instrument. For example:
   `git add herschel/src/`**`main`**`/scala/com/capitalone/fsor/herschel/instrumentmakers`
   `git add herschel/src/`**`test`**`/scala/com/capitalone/fsor/herschel/instrumentmakers`

7. If you created a serialized version of your instrument as described in Serializing your instruments, *do not add it*.

8. Git-commit your additions with an appropriate commit message. For example:
   `git commit -m "Instrument for checking account product"`

9. Git-push your additions to the herschel repo. For example:
   `git push --set-upstream origin myInstrumentBranch`

10. Create a git pull request (PR) for the the instrument reviewing people to be notified of your addition. The easiest way to create a PR is to go to the top of the herschel repo and click **Compare and pull request**.

    a. Fill in the details about your instrument in the PR.

       Be explicit. Note any important points for particular attention or concern.

    b. As reviewers, select one of the following persons:
       1. Matt Fellows
       2. Erik Jacobsen
       3. Annette Chen
       4. Leandra Irvine

    c. Submit your PR.

After you have submitted your PR, the PEG team will review your instrument. They might make suggestions for improvement.

After review, your instrument will be merged into Herschel's master branch to be part of the build.

8. Programmer's syntax reference

8.1. Summary of instrument helper functions

There are various helper functions in Herschel to reduce the tedium of developing instruments. These helpers are located in the following packages, which you import at the top of your instrument:

- The `com.capitalone.fsor.herschel.helpers.`**`instrumenthelpers`** object
- The `com.capitalone.fsor.herschel.helpers.`**`billpackage`** object

Uses of the helper functions are illustrated in the example instrument and in this guide. For full syntax of the helpers, see the scaladoc reference.

8.1.1. Helper functions in instrumenthelpers

```
import com.capitalone.fsor.herschel.helpers.instrumenthelpers._
```

8.1.1.1. Bucket helpers

- `makeBuckets`: Creates a list of buckets with a list of suffixes.
- `makeBucketMap`: Builds a map of buckets in the form of `BucketName -> Bucket`.
- `makeConfigMap`: Builds a map of all fields in the form of `FieldName -> Field`.
- `makeDebitCreditBucketPairs`: Creates a list of debit-credit bucket pairs from a list of tuples.

8.1.1.2. Double-Entry helpers

- `makeShiftDoubleEntries`: Makes a set of shifting double entries.

8.1.1.3. Allocation helpers

- `makeTargets`: Makes a list of targets for an allocation

8.1.1.4. Rollup helpers

- `makeConstituentListFromBuckets`: Makes a list of constituents for a rollup from a list of buckets.
- `makeConstituentListFromRollups`: Makes a list of constituents for a rollup from a list of rollups.
- `makeRollupCollection`: Makes a list of constituents for a given `suffixList`, debit and credit base.

8.1.1.5. Program Expression helpers

- `isPositiveBalance`: Predicate for verifying if a balance is positive.
- `isZeroBalance`: Predicate for verifying if a balance is zero.
- `multiAdd`: Chains multiple `Add` operands.
- `multiAnd`: Chains multiple `And` operands.
- `multiOr`: Chains multiple `Or` operands.

8.1.2. Helpers in billpackage

```
import com.capitalone.fsor.herschel.helpers.billpackage.BillPackage
```

When given the desired number of bills and a list of bill component names, such as `"interest"`, `"fee"`, or `"principal"`, the call:

```
BillPackage(numBills: Int, orderedPrefixList: List\[String\])
```

exposes the following `val`s:

- `simpleConfigs` - a `Set` of SimpleConfigs
  - `due-date-$i` for each bill (`i` from `1` to `numBills`)
- `buckets` - a `Set` of BucketConfigs
  - `$p-due-$i` and `$p-due-paid-$i` for each value `$p` in the `orderedPrefixList`, and `$i` in `1` to `numBills`

    - `$p-unbilled`, `$p-unbilled-paid`, `$p-due-past`, $p-due-paid-past
    - `overpayment` credit bucket
- `doubleEntries` - Double entries for moving amounts between bill buckets
- `rollups` - Rollup Configs for computing balances
    - $p-unbilled-balance - a rollup of $p-unbilled and $p-unbilled-paid, which will result in a positive amount if $p-unbilled > $p-unbilled-paid
    - $p - (e.g., "interest") is a rollup of all bill buckets with prefix $p
    - bill-balance-$i - a rollup of $p-due-$i, and $p-due-paid-$i for a given $i, across all $p
    - among others
- `allocators` - Two allocators for allocating payments to bill due amounts
    - `"normal-bill-allocator"` - allocates to oldest bill (that is, `$p-due-paid-1`) first, to each bill component in the order specified in `orderedPrefixList`, then to bill -2, and so forth. Finally, allocates to `$p-unbilled-paid`. Any excess goes into the `"overpayment"` bucket.
    - `"only-billed-allocator"` - Same as the normal allocator, but does not allocate to $p-unbilled-paid.
- `allocatorMap`: A map for `allocators`.
- `bucketMap`: A map from name to bucket config made from buckets.
- `doubleEntryMap`: A map from name to Double Entry config from `doubleEntries`.
- `rollupMap`: A map for rollups.
- `simpleConfigMap`: a `Map[String, SimpleConfig]` made from `simpleConfigs`, mapping each name to its config.

`makeShiftEmitExpressions(nextCycleDateExpr: Expressions.ValueExpr): List\[Action\]` returns a list of `Emits` for shifting bill values when a new bill is created. See Bill Exploration for explanation of rationale.

8.2. Scala programmer's reference for financial instruments

NOTE to Reviewers: Scaladocs from fsor and herschel repos will be plugged into the nightly builds for serving directly from a separate repo via github-pages. See the Project proposal: dedicated git repo for scaladoc output. The final links should be added to the proper bullet points below.

- BillPackage and instrument helpers reference from herschel repo.
- Account restrictions helpers from herschel repo. NOTE to Reviewers: Include this on ein scaladoc?
- Expression classes reference from fsor repo.

9. Revision history

Creating Financial Instruments with the Prometheus Financial Core Developer Guide

Date	Description
2019-12-09	Draft for review by internal-to-Prometheus team before wider review by LOBs