Powered By

FTL

FTL Programming

Fetal Documentation

Introduction

FTL (aka Fetal) stands for Flexible Transaction language. The way it is used in financial applications is to have the bulk of the application written in a structured language like Java and write all of the financial transactions in FTL.

Why FTL

FTL gives you several advantages:

1. It ensures that if a transaction fails, the whole transaction fails.
2. It encapsulates all of the financial transactions in a single area and makes them easy to troubleshoot.
3. It is easily changed if accounting needs change.
4. FTL scripts are not cached so scripts can be hot swapped.
5. FTL is a C like language and can be easily learned and understood.
6. You can even store the script in a database table and selectively execute the script you want, base on a give criteria.
7. Even though FTL is a script, it is object oriented.

FTL Programming

General functions

Every FTL script starts with '**begin**' and ends with '**end**'. Both of these statements define the beginning and ending of the transaction.

if (<expression>)

If statements simply evaluate an expression and execute a block of code depending on the expression. If statements can be nested and statements can be concatenated together with '&&' (and), '||' (or), '^^' exclusive or. Exclusive or work like this... lets say you have 2 statements 'a' and 'b'. An exclusive or will execute the block of code if 'a' is true or 'b' is true but not both.

print(<object>)

The print statement is used to debug an FTL script. It outputs whatever object to standard out. String can be concatenated using the '+' operator.

printVarList()

The printVarList statement is also used for debugging. It prints a list of variables and their values to standard out.

mapfile: <mapfile>

The mapfile statement is used to alias the account numbers to account names, where <*mapfile*> is an XML file that contains the account number to name definitions. The file is stored in /home/WebsiteFiles/config by default.

getVariableType(<variable>)

The getVariableType statement returns the variable type indicated by <*variable*>. There are types of variables;

1. decimal (same as Double).
2. number (same as Long).
3. string
4. boolean
5. date
6. object
7. dao (dao is an object but it's used to represent a record in a table)

getDescription()

The getDescription statement gets a description of the transaction. It is possible to set the description before the script is loaded using the setDescription method in java. The getDescription state simply reads the description set by setDescription.

import(<path>)

The import statement imports an object into the FTL script to add functionality to the script (see invocation). The <*path*> argument is a path to the object to be imported.

Fetal Documentation

Accounting functions

Accounting statements are the heart of FTL.

getBalance(<account number>)

The getBalance statement returns the balance of the account indicated by <*account number*>.**credit(<amount>, <account>)**

The credit statement credits the amount indicated by <*amount*> to the account indicated by <*account*>. Remember that in accounting, crediting an account does not necessarily make the balance go up. With the cash account, for example, the balance goes down when the amount is credited. This is dependent on the type of account and it is handled in FTL.

debit(<amount>, <account>)

The debit statement debits the amount indicated by <*amount*> to the account indicated by <*account*>. As with credit, FTL handles whether to increase or decrease the balance, based on the account type.

ledger(<D or C>, <amount>, <account>, <description>)

The ledger statement records the debit or credit in General Ledger. It has 4 arguments. The first is <*D or C*>; 'D' is for debit and 'C' is for credit. The 2nd is the amount of the transaction, the 3rd is for the account that it goes to and the 4th is the description of the transaction.

fv(<amount>,<rate>,<periods>)

The fv statement will give you the future value with the amount indicated by <*amount*>*,* the rate indicated by <*rate*> and the periods indicated by <*periods*>.

pv(<amount>,<rate>,<periods>)

The pv statement will give you the present value with the amount indicated by <*amount*>*,* the rate indicated by <*rate*> and the periods indicated by <*periods*>.

depreciation(<cost>, <salvage>, <life>, <age>, <double declining>)

The depreciation statement gives the amount of depreciation for a given capital asset. The cost of the asset is indicated by <*cost*>, the salvage value is indicated by <*salvage*>*,* the lifetime of the asset is indicated by <*life*>, the age of the asset is indicated by <*age*>*.* This statement supports 2 types of depreciation methods. If you want double declining depreciation, set <*double declining*> to true.

Time functions

Time statements allow you to manipulate date and time for various accounting tasks.

today()

The today statement gives you the current date and time.

truncate(<date>)

The truncate statement will take the given <*date*> and truncates it to midnight, 0 minutes and 0 seconds.

getMinutes(<begin>, <end>)

The getMinutes statement gets the number of minutes between the time indicated by <*begin*> and the time indicated by <*end*>.

getHours(<begin>, <end>)

The getMinutes statement gets the number of hours between the time indicated by <*begin*> and the time indicated by <*end*>.

getDays(<begin>, <end>)

The getMinutes statement gets the number of days between the time indicated by <*begin*> and the time indicated by <*end*>.

dayOfTheWeek(<date>)

The dayOfTheWeek statement returns a 3 character string that indicates the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat) from the given <*date*>.

getCalendarDay(<date>)

The getCalendarDay statement returns the calendar day from the given <*date*>.

getMonth(<date>)

The getMonth statement returns the month from the given <*date*>.

getYear(<date>)

The getYear statement returns the year from the given <*date*>.

getDate(<year>, <month>, <day>)

The getDate statement returns a date from the year indicated by <*year*>, the month indicated by <*month*> and the day indicated by <*day*>.

nextDate(<base>, <last>, <schedule>)

The nextDate statement will give you the next date in a schedule. The 3 arguments are <*base*> which is the first date in the sequence, <*last*> which is the last date in the sequence, <*schedule*> which is the given schedule. The valid schedules are as follows;

1. Annually
2. Bi-Annually
3. Quarterly
4. Monthly
5. Bi-Monthly
6. Weekly
7. Daily

The schedules are case sensitive.

SQL functions

The SQL statements are based on the Hibernate SQL statements. In most cases, it has an SQL statement and optional arguments. The arguments are inserted into the statement to make well formatted SQL statement. This is similar to the printf function in C. Let's look at a sample SQL statement;

```
dao oldPayment = lookup("FROM LoanPayments WHERE id = {d}",
                                                   transactions.getPayment_ref());
```

You can see that {d} is a numeric placeholder for transaction.getPayment_ref().

The placeholders are as follows;

1. {d} is a placeholder for a numeric.
2. {f} is a placeholder for a decimal
3. {F} is a placeholder for a date (in the format of yyyy-MM-dd:00:00)
4. {s} is a placeholder for a string.

An SQL statement can have any number of arguments. The lookup, list and update statements all take the same form of *<SQL>, <arguments...>*.

lookup(<SQL>, <arguments...)

The lookup statement returns an object based on the *<SQL>*. If more than one record meets that criteria, it will only return the first one.

list(<SQL>, <arguments...>)

The list statement, on the other hand, will return a list of records in the form of set<Object>.

update(<SQL>, <arguments...>)

The update statement returns no records but will update a record or group of records, based upon the query.

insert(<DAO>)

The insert statement will insert the object or dao that is indicated by *<DAO>* into the database table, as long as it is a valid DAO object. The system will figure out which table to insert, base upon the DAO object.

delete(<DAO>)

The delete statement will delete a record indicated by *<DAO>*. It should be used with care when deleting records that have complex relationships (ie one to one or one to many). You can also use update and delete the records in an orderly fashion.

merge(<DAO>)

The merge statement is used when up wish to update records with complex relationships. Simple assign *<DAO>* to the parent record and merge should handle all of the updates for a set of related tables.

Invocation - <object>.<method>

FTL allows the import of other objects and their associated methods. There are 2 ways to get objects into FTL; 1.) publish them before you load the script. 2.) import them. Importing should only be used when you have methods that add functionality to the script. With publishing, you can publish a fully loaded record, ready to be manipulated. Import will only give you a blank record with no fields populated.

Here are some examples of invocations from an object called newPayment. Notice that on first invocation, there is an invocation within an invocation. This is legal in FTL.

```
newPayment.setReference(loanTerms.getReference());
newPayment.setDue(nextDate(base, last, sched));
newPayment.setAmount_due(payAmount);
newPayment.setPayment_made(0.00);
newPayment.setPenalties(0.00);
newPayment.setLoanTerms(loanTerms);
```

Mathematical functions

round(<amount>)

The round statement will round the number indicated by <*amount*> to the nearest whole number.

roundUp(<amount>)

The roundUp statement will round the number indicated by <*amount*> to the next highest whole number.

roundDown(<amount>)

The roundDown statement will round the number indicated by <*amount*> to the next lowest whole number.

roundTo(<amount>, <digits>)

The roundUp statement will round the number indicated by <*amount*> to the number of digits indicated by <*digits*>.

absDecimal(<amount>)

Will return the decimal indicated by <*amount*> without the minus sign.

absNumber(<amount>)

Will return the number indicated by <*amount*> without the minus sign.

toNumber(<amount>)

Will convert <*amount*> (even if it's a string that represents a valid number) to a number.

toDecimal(<amount>)

Will convert <*amount*> (even if it's a string that represents a valid decimal) to a decimal.

Operators

Valid operators follow the C convention and are as follows

Arithmetic operators

+	Plus
–	Minus
*	Multiply
/	Divide by
%	Modulo
^	Exponent

Unary operators

+=	Plus-Equals
-=	Minus-Equals
*=	Multiply-Equals
/=	Divide-by-Equals
%=	Modulo-Equals
^=	Exponent-Equals

Bitwise operators

&	And
|	Or
!	Not

Comparison operators

==	Equal to
<	Less than
<=	Less than or Equal to
>	Greater than
>=	Greater than or Equal to
!=	Not Equal to

Logical operators

&&	And
||	Or
^^	Exclusive Or

Percentage operator

If you place a '%' to the right of a decimal, it will cause the decimal to be divided by 100.