powered by 4D fourth dimension

Fourth Dimension Lite

User Guide

Table of Contents

Overview 3
Calendar Frequency. 4
Data Storage 6
DataPointCollection 6
DataPoint 6
Series 7
Modifying Series 8
Data Retrieval 8
List Adapters 10
Series Descriptor 11
Global Identifier 11
Examples 12

Overview

Fourth Dimension Lite is an implementation of a time and numerically sequential data model. Data stored in a fourth dimension series is aligned to a specified calendar frequency. Integrity of the data is maintained by not allowing data to be stored for a period which doesn't align to the calendar. For example when dealing with financial instruments, markets only trade Monday to Friday. Financial instruments should therefore have a calendar frequency of Business. On the other hand, for the storage of weather data a calendar frequency or Daily or Hourly would be more appropriate.

Fourth Dimension out of the box provides 13 standard frequencies. These frequencies are merely a starting point as further frequencies can be defined on top of the standard frequencies.

Collecting and storing data is only a small component of this library. Information about the contents of the data can also be stored against the series. This may include information as to the location the data was sourced from or even who collected it, when it was collected or in fact anything that can be serialized (via Java) can be stored in the attribute tags.

Fourth Dimension objects are all serializable. This ensures that data objects can be persisted to a file or even stored as a blob in a relational database. The possibilities are endless.

Fourth Dimension Plus and Fourth Dimension Advance are complimentary products. Both these products provide additional features which are not part of the core Fourth Dimension functionality. Please see alternative literature for more information with regards to these products.

Calendar Frequency.

A Calendar Frequency is a rule based implementation governing the generation sequence of periods – either time or observation base. As there is no intrinsic duration class in Java, a long value is used to represent the starting point of a period. An end index is not maintained, as this is equivalent to the starting index of the following period. Using a long value ensures that a period index values can be easily translated into a `java.util.Date`.

The following calendar frequencies are supported in Fourth Dimension.

`Annual`	Periods of 1 year i.e. 2001, 2002, 2003 …
`Semiannual`	Periods of 6 months i.e. JAN01,JUN01,JAN02,JUN02
`Quarterly`	Periods of 3 months i.e. JAN01, MAR01,JUN01,SEP01,JAN02
`Bimonthly`	Periods of 2 months i.e. JAN01,MAR01,MAY01,JUL01,SEP01,NOV01
`Monthly`	Periods of 1 month
`Biweekly`	Periods of 2 weeks
`Weekly`	Periods of 1 week
`Business`	Periods between Mon to Fri.
`Daily`	Periods between through Mon to Sun.
`Hourly`	Hourly periods
`Minutely`	Minutely periods
`Secondly`	Secondly periods
`Millisecondly`	Millisecondly periods
`Observation`	Sequential numbers starting from 1

All frequencies implemented in Fourth Dimension extend the base class `CalendarFrequency`. All frequencies listed above have a default constructor that can be used to produce a frequency with default values. Selected frequencies support additional constructors which allow a frequency to be aligned to a different period within the calendar.

- To create an annual calendar that is aligned to 01 January.

```
Annual calendar = new Annual();
```

- To create an annual calendar that is aligned to 01 June.

```
Annual calendar = new Annual(Annual.REF_JUNE);
```

The object model for CalendarFrequency and subclasses is shown below.

[insert diagram here]

The `Monthly` frequency and all subclasses of the `Intraday` frequencies support a periodicity setting in the constructor. The periodicity setting provides a means to generate a multiplication of the frequency. For example an `Annual` frequency can be thought as being a `Monthly` frequency with a periodicity of 12. Similarly, a `Quarterly` frequency has a periodicity of 3.

A periodicity with intraday frequencies is useful when creating custom frequencies. For example to create a fifteen minutely frequency:

```
Minutely fifteen = new Minutely(15);
```

The `Daily` frequency unlike other frequencies provides an optional constructor that allows patterns to be defined. A pattern is simply an array of boolean values which match the frequency length. i.e. The `Business` frequency which extends `Daily` specifies a pattern of 7 boolean values representing Monday to Friday as true, and Saturday and Sunday as false. The following demonstrates creating a custom daily frequency which produces a frequency of "every Wednesday".

```
boolean[]  pattern = {false,false,false,true,false,false,false}; //Sun ,Mon ,Tue ,Wed
                                                                 //Thu, Fri, Sat
Daily wednesdays = new Daily(pattern);
```

The following is the most useful methods defined by all subclasses of `CalendarFrequency`.

`Iterator iterator(long startIndex,long endIndex)`	Returns an Iterator of `Long`'s between a start index and an end index. The indexes are inclusive.
`long startsOnBefore(long index)`	Returns an index value which represents the period that starts on or before the specified index.
`Date startsOnBefore(Date index)`	As above. Note the returned object is a java.util.Date
`long startsAfter(long index)`	Returns an index which starts after the specified index.
`Date startsAfter(Date index)`	As above. Note the returned object is a java.util.Date.
`boolean aligns(long index)`	Indicates if a specified index is aligned to the frequency.
`boolean aligns(Date index)`	As above.
`long[] getIndexes(long startIndex,long endIndex)`	Returns an array of index values between a starting and ending index. The start and end index are inclusive.
`long[] getIndexes(Date startIndex,Date endIndex)`	As above.
`long periodsBetween(long startIndex,long endIndex)`	Returns the number of periods between a starting and ending index. Note. The start Index and end Index must be aligned to the frequency. **The end Index is not inclusive**.
`long periodsBetween(Date startIndex,Date endIndex)`	As above.
`String getCalendarFrequenyName()`	Returns the name of the frequency.

All calendars provide time zone support. For example if you create a calendar specifying a timezone of Europe/Amsterdam, all indexes will be aligned to CET and CEST times. The default timezone is GMT.

Data Storage

The following diagram depicts the relationship of the data storage classes in Fourth Dimension.

[insert diagram here]

It is important to realise that `DataPointCollection`, and `DataPoint` objects are created on demand. They are not created and stored within an associated `Series` object.

DataPointCollection

A `DataPointCollection` is implemented as a subclass of an immutable `AbstractList`. As with the relationship between `Series` and `DataPointCollection`, `DataPoint` objects are created only when the getter methods are invoked on `DataPointCollection`.

The most useful methods of the `DataPointCollection` are as follows.

`Object get(int position)`	Required by the List interface. Returns a DataPoint object constructed by the values stored at the specified position.
`Object getData(int position)`	Returns the value stored at the specified position as an Object.
`DataPoint getDataPoint(int position)`	Returns the value, index and status encapsulated as a DataPoint at the specified position.
`long getIndex(int position)`	Returns a starting period index value for the specified position.
`long getMaxIndex()`	The maximum index value of this collection.
`long getMinIndex()`	The minimum index value of this collection.
`DataPointStatus getStatus(int position)`	Returns the status of a value stored at the specified position.
`int size()`	Required by the List interface. Indicates the size of this collection.

DataPoint

The `DataPoint` class encapsulates three pieces of information about the data stored; time or observation index, the status, and the data itself. The `DataPoint` object is instantiated only when either the `get(int position)` or `getDataPoint(int position)` methods of the `DataPointCollection` class are called. The `DataPoint` class provides a number of methods to convert the underlying data into different Object types.

`Boolean getBooleanValue()`	Converts the data to a Boolean value.
`Date getDateValue()`	Converts the data to a Date value.
`Double getDoubleValue()`	Converts the data to a Double value.
`Float getFloatValue()`	Converts the data to a Float value.
`Integer getIntegerValue()`	Converts the data to a Integer value.
`Long getLongValue()`	Converts the data to a Long value.
`Short getShortValue()`	Converts the data to a Short value.
`String getStringValue()`	Converts the data to a String value.

Additional methods to retrieve the index, status and data are as follows.

`long getIndex()`	Returns the index of the DataPoint.
`Date getIndexAsJavaDate`	Returns the index of the DataPoint as a Date.
`byte getStatus()`	Returns the status of the DataPoint. The byte value can be used in conjunction with the DataPointStatus class.
`Object getValue()`	Return the data of the DataPoint.

Series

The minimum amount of information required to create a series is a name and a calendar frequency. Additional information such as any descriptions or data source information can be stored against a series using attributes.

The following creates a series that can store double data at a Monthly frequency, setting an additional attribute called Description.

```
Monthly monthly = new Monthly();  //produces the default monthly calendar

Series series = new DoubleSeries(“My Series”,monthly);  //creates a series with no data
Series.setAttribute(“Description”,”My monthly series for demonstration purposes.”);
```

The name of the series can be changed at any time by calling the `setName()` method. Note however that if these object are persisted in a data store then unless the data store registers an appropriate listener, the name change my not be persisted.
When series are created the `CREATED` attribute is set to the local time of the object creation. Similarly, when the series is modified, either as a result of modifying an attribute or DataPoint, the `MODIFIED` attribute is updated.

Additional common attributes are defined in the `AttributeMapKey` interface, but are not limited to this list.

- `CALENDAR`
- `CREATED`
- `MODIFIED`
- `DESCRIPTION`
- `NAME`
- `DATATYPE`
- `AGGREGATION`
- `DOCUMENTATION`
- `TRUENAME`
- `SERVICE_NAME`
- `DATABASE_NAME`

Multiple calls to `setAttribute(String name,Object value)` overwrite previous values.

Attributes of a series can be retrieved using the `getAttribute(String name)` method supplying an appropriate attribute name. Alternatively, calling the `getAttributeMap()` returns an `AttributeMap` object which allows attributes to be retrieved using enumerations or iterators much the same way as in a `HashMap`.

Modifying Series

Adding DataPoints can be accomplished by using either the `setDataPoint(DataPoint data)` or `setDataPoint(long index,Object data,byte status)` methods. Both methods are equivalent,

however the `setDataPoint(long index,Object data,byte status)` method is marginally faster.

```
Monthly monthly = new Monthly();  //produces the default monthly calendar
Series series = new DoubleSeries("My Series",monthly);  //creates a series with no data


Calendar index = new GregorianCalendar(2005,Calendar.FEBURARY,1);  //creates an index
                                                                   //value of 1 FEB 2005

//create a DataPoint object to represent the data to be stored
DataPoint data = new DataPoint(index,1234d,DataPointStatus.VALID);

//add the data to the series
series.setDataPoint(data);
```

or

```
…
//update the series directly without creating a DataPoint object
series.setDataPoint(index,1234d,DataPointStatus.VALID);
```

Subsequent calls to either of the `setDataPoint` methods results in the last call surviving.

It is not possible to remove a DataPoint. Instead a call to the `setDataPoint` method with a status of `DataPointStatus.MISSING` is an equivalent operation.

Note: An index point must be created using the correct calendar frequency for the series. Failure to do so will result in an `IndexAlignmentException` to be thrown.

Every series maintains a list of indexes at which either a `DataPoint` has be added or removed. In future releases of Fourth Dimension an enhanced series called `AuditSeries` will be available that will retain a full history of any changes to the underlying data.

Data Retrieval

There are a number of ways to retrieve data from a series. The first is using the `getDataPoint(long index)` method of the Series. The following section of code demonstrates retrieving a single DataPoint stored at 1 Febuary 2005.

```
…
Calendar index = new GregorianCalendar(2005,Calendar.FEBRUARY,1);  //creates an index
                                                                   //value of 1 FEB 2005
DataPoint data = series.getDataPoint(index);
System.out.println(data.toString());
…
```

A `DataPointCollection` is a collection of `DataPoints` between two indexes. `DataPointCollection` extends `AbstractList` without providing any optional methods i.e. no methods which may modify the series are implemented.

The follows block of code demonstrates retrieving a range of data between the 1 February 2005 and 28 February 2005.

```
…
long start = new GregorianCalendar(2005,Calendar.FEBRUARY,1);
long end = new GregorianCalendar(2005,Calendar.FEBRUARY,28);

DataPointCollection collection = series.getDataPointCollection(start,end);
```

```
DataPoint data1 = collection.get(0);
System.out.println(data1.toString());

DataPoint data2 = collection.get(1);
System.out.println(data2.toString());

…
```

The entire contents can be retrieved without specifying a start and end index. This is equivalent to specifying `series.getMinIndex()` and `series.getMaxIndex()`.

```
…
DataPointCollection collection = series.getDataPointCollection();
…
```

As a series maintains a status as to whether it contains missing data or has been edited, edited and missing DataPoints can be returned using an `Iterator`.

The following block of code illustrates the call to return all DataPoints in a series that have been edited since the creation of the series or the last edit reset.

```
…
Iterator iterator = series.getDataPointIteratorByEdits();
while (iterator.hasNext()) {
        DataPoint data = (DataPoint)iterator.next();
        System.out.println(data.toString());
}
…
```

The following block of code illustrates returning DataPoints that have a status of VALID.

```
…
Iterator iterator = series.getDataPointIteratorByStatus(DataPointStatus.VALID);
while (iterator.hasNext()) {
        DataPoint data = (DataPoint)iterator.next();
        System.out.println(data.toString());
}
…
```

The underlying data of the series can also be accessed providing a more efficient means of displaying the data in a TableModel.

```
…
long[] indexes = series.getDataPointIndexes();  //the index points of the periods
ListAdapter data = series.getDataPointValues(); //the data stored
byte[] statuses = series.getDataPointStatuses(); //the status of the data

for (int i=0;i<indexes.length;i++) {
        System.out.print(“Index:” + indexes[i]);
        System.out.print(“\t Values:” + data.getObject(i));
        System.out.print(“\t Status:” + statuses[i]);
        System.out.println();
}
…
```

List Adapters

Data stored in a series is in most cases primitive data types. i.e. `long` or `double` rather than their `Object` counterparts `Long`, `Double`. In order to accommodate the varying primitive arrays, the `ListAdapter` class provides common access to the underlying data.

If you attempt to access a data type which is not compatible then the exception `UnsupportedOperationException` is thrown.

To retrieve the underlying primitive data array from the list adapter, then simply call the appropriate `toXXXArray()` method.

The useful methods of the `ListAdapter` class are as follows

`ListAdpater makeAdapter(byte[] values)`	Creates a ListAdapter based on the byte[].
`ListAdapter makeAdapter(short[] values)`	Creates a ListAdapter based on the short[].
`ListAdapter makeAdapter(int[] values)`	Creates a ListAdapter based on the int[].
`ListAdapter makeAdapter(long[] values)`	Creates a ListAdapter based on the long[].
`ListAdapter makeAdapter(float[] values)`	Creates a ListAdapter based on the float[].
`ListAdapter makeAdapter(double[] values)`	Creates a ListAdapter based on the double[].
For each primitive datatype:	
…	
`add(byte value)`	Appends the byte value to the end of the collection
`add(int index,byte value)`	Inserts the byte value at index. Shifts the element currently at that position (if any) and any subsequent elements to the right, increasing their indices.
`getByte(int index)`	Returns a byte value stored at index.
`set(int index,byte value)`	Sets the value at index with the byte value
`toByteArray()`	Returns a byte[] representing the underlying series
…	

Series Descriptor

All series maintain an object descriptor that describes essential information for the series. All attributes of the series are available, along with the min and max indexes, calendar frequency and series name. The descriptor object can be used as a lightweight representation of the series object and is especially useful if you simply need a list of objects stored in a database, rather than a fully populated series object.

```
…
SeriesInfo info = series.getSeriesInfo();
System.out.println(info.getName());
…
```

Useful methods when dealing with the series description object are:

`AttributeMap getAttributes()`	Returns all the available attributes for the series. Individual attributes can be extracted using the AttributeMapKey or the appropriate custom key.
`CalendarFrequency getCalendar()`	Returns the Calendar Frequency for the object.
`long getMinIndex()`	Gets the minimum index for the series
`long getMaxIndex()`	Gets the maximum index for the series
`String getName()`	Returns the name of the series
`String getDescription()`	Returns the description of the series if set.

Global Identifier

The `GlobalIdentifier` is used to identify a series across an enterprise. The default implementation is made up of a `SERVICE_NAME`, `DATABASE_NAME` and `NAME`. The format of the returned string is GUID(`SERVICE_NAME://DATABASE_NAME'NAME`).

If this format is not appropriate then a new subclass of the `GlobalIdentifier` can be used as an alternative.

The following section of code demonstrates printing the identifier to the console.

```
…
GlobalIdentifier identifier = series.getGlobalIdentifier;  //the index points of the periods
System.out.println(identifier.toString());
…
```

Examples

- *Create a double series aligned to a Daily calendar frequency.*

```
Daily daily = new Daily();  //produces the default daily calendar
Series series = new DoubleSeries(“My Series”,daily);  //creates a series with no data
```

- *Create a long series aligned to a Monthly calendar frequency.*

```
Monthly monthly = new Monthly();  //produces the default monthly calendar
Series series = new LongSeries(“My Series”,monthly);  //creates a series with no data
```

- *Create a float series aligned to a business calendar frequency. Insert values for business dates between 03Jan05 and 30Dec05, and then print out the results.*

```
Business business = new Business();  //produces business calendar
Series series = new FloatSeries("My Series",business);  //creates a series with no data

//Create the first business day of the year
long start = new GregorianCalendar(2005,Calendar.JANUARY,3).getTimeInMillis();
//Create the last business day of the year
long end = new GregorianCalendar(2005,Calendar.DECEMBER,30).getTimeInMillis();

//Produce a list of indexes between the start and end dates
long[] indexes = business.getIndexes(start,end);

//Update the series with float values based on the index point
for (int i=0;i<indexes.length;i++) {
    series.setDataPoint(indexes[i],(float)indexes[i],DataPointStatus.VALID);
}

//Create a DataPoint collection to cover this range of data.
DataPointCollection collection = series.getDataPointCollection(start,end);

//Using an iterator, print out the contents of the collection
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
    DataPoint data = (DataPoint)iterator.next();
    System.out.println(data.toString());
}
```

- *Create an integer series aligned to a monthy calendar frequency. Insert values for monthly dates between 01Jan04 and 01Jan06. Produce a DataPointCollection for a subrange between Jun04 and Jun04 and print out the results.*

```
TimeZone timezone = TimeZone.getTimeZone("GMT");

Monthly monthly = new Monthly();  //creates monthly calendar with a GMT timezone
monthly.setTimezone(timezone);

//Create an series which can contain integers aligned to the monthly calendar
Series series = new IntegerSeries("My Series",monthly);

//Create the first month of the year, remember to set the TimeZone of the calendar
Calendar startCal = new GregorianCalendar(2004,Calendar.JANUARY,1);
startCal.setTimeZone(timezone);

//Create the last month of the year, remember to set the TimeZone of the calendar
Calendar endCal = new GregorianCalendar(2006,Calendar.JANUARY,1);
endCal.setTimeZone(timezone);

//Produce a list of indexes between the start and end dates
long[] indexes = monthly.getIndexes(startCal.getTimeInMillis(),endCal.getTimeInMillis());

//Update the series with int values based on the index point
for (int i=0;i<indexes.length;i++) {
                    series.setDataPoint(indexes[i],(int)indexes[i],DataPointStatus.VALID);
}
```

```
//Create a DataPoint collection which covers the range Jun04 -> Jun05
Calendar JUN04 = new GregorianCalendar(2004,Calendar.JUNE,1);
JUN04.setTimeZone(timezone);

Calendar JUN05 = new GregorianCalendar(2005,Calendar.JUNE,1);
JUN05.setTimeZone(timezone);


DataPointCollection collection =
series.getDataPointCollection(JUN04.getTimeInMillis(),JUN05.getTimeInMillis());

//Using an iterator, print out the contents of the collection
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
        DataPoint data = (DataPoint)iterator.next();
        System.out.println(data.toString());
}
```