JFT/API

Programming Manual

Version 4.2.1

Confidential

To contact the helpdesk nearest to you visit our website
www.list−group.com

Table of Contents

JFT/Api Programmers Guide **1**

JFT/Api Introduction **3**

FastTrack Overview **5**
- System Architecture Overview 5
- Access points 5
- JFT/Api Access Point 6

Data Distribution **7**
- Publish 7
- Subscribe 7
- Queries 7
- Transactions 8
- Connections and Contexts 8

JFT/Api Details **9**
- Asynchronous Communication Model 9
- LifeCycle 9
- Data Model 9
- Other Peculiarities 9
- JFT/Api Entry Point 9

Package it.list.jft **11**
- Package it.list.jft Description 12
- Package it.list.jft Data Model 12

Hierarchy For Package it.list.jft **15**
- Interface Hierarchy 15
- Interface EntityClass 15
- Interface Entity 18
- Interface EntityField 21
- Interface EntityKey 25
- Interface LifeCycle 26
- Interface CommunicationLifeCycle 31
- Interface ActivityLifeCycle 34
- Interface EntityClassQuery 36
- Interface Filter 38
- Interface EntityFilter 44
- Interface Query 47
- Interface Subscription 53
- Interface Transaction 60
- Interface Connection 67
- Interface MulticastConnection 72
- Interface Context 74
- Interface JFT 83
- Interface Mask 96
- Interface Param 99
- Interface ConnectionParam 101
- Interface EntityClassQueryParam 121

Table of Contents

Hierarchy For Package it.list.jft
Interface FilterParam....122
Interface MulticastConnectionParam....125
Interface QueryParam....127
Interface SubscriptionParam....129
Interface TransactionParam....139
Interface TimeStamp....146
Interface TransactionID....147
Interface Tracer....150

Package it.list.jft.event....151
Package it.list.jft.event Description....152
Package it.list.jft.event Data Model....152

Hierarchy For Package it.list.jft.event....155
Interface Hierarchy....155
Interface Event....155
Interface ConnectionEvent....157
Interface ConnectionCloseEvent....158
Interface ConnectionLostEvent....159
Interface ConnectionOpenEvent....160
Interface EntityClassQueryEvent....166
Interface FilterEvent....167
Interface FilterCreateEvent....168
Interface FilterDestroyEvent....170
Interface FilterSetEvent....171
Interface MulticastConnectionEvent....173
Interface QueryEvent....174
Interface QueryCreateEvent....175
Interface QueryDestroyEvent....178
Interface QueryNotifyEvent....178
Interface QueryRowsEvent....182
Interface SubscriptionEvent....183
Interface SubscriptionIdleEvent....184
Interface SubscriptionNotifyEvent....185
Interface SubscriptionStartEvent....190
Interface SubscriptionStopEvent....191
Interface TransactionEvent....192
Interface TransactionQueryEvent....195
Interface TransactionSendEvent....196
Interface Listener....197
Interface ConnectionListener....197
Interface EntityClassQueryListener....199
Interface FilterListener....200
Interface MulticastConnectionListener....202
Interface QueryListener....203
Interface SubscriptionListener....205
Interface TransactionListener....207

Table of Contents

JFT/Api Application Examples....211
Example 1....212
Example 2....217
Example 3....235

Contacts....239

JFT/Api Programmers Guide

Confidential

This manual describes JFT/API, the Java Application Program Interface developed by LIST within FastTrack to access electronic markets and other services handled by FastTrack.
See: Description

Packages

Package	Description
it.list.jft	Provides interfaces for dealing with different FastTrack objects.
it.list.jft.event	Provides interfaces for dealing with different types of events and listeners.

This manual describes JFT/API, the Java Application Program Interface developed by LIST within FastTrack to access electronic markets and other services handled by FastTrack.

To start use this library read the JFT/Api Introduction or watch the data models (Package it.list.jft Data Model and Package it.list.jft.event Data Model) or jump to JFT or just watch a few Java example programs.

ListGroup & FastTrack Contacts

Requests for clarifications, comments and suggestions to improve the quality of the product are welcome.
Please contact us:

LIST SpA
Via Pietrasantina, 123
56122 PISA ITALY

Or contact us by email:

Marketing: infodesk@list−group.com
General Support: service−desk@list−group.com
JFT/API Programming Support: ftapi@list−group.com

Or visit us at www.list−group.com

Related Documentation

FTAPI.pdf − FT/API Programmer's Guide − V 3.6.5
The traditional C interface to FastTrack.
FastTrade Platform
Introduction to FastTrade technology.

See Also:
JFT/Api Introduction, JFT, JFT Application Examples,
JFT Exceptions, JFT Implementation Threads, JFT Synchronization

Submit a bug or feature to FT\API Programming Support

JFT/Api Introduction

This manual describes version 4.2.1 of JFT/API, the Java Application Program Interface developed by LIST within FastTrack to access electronic markets and other services handled by FastTrack.

Data structure and functions of JFT/API interface are described, along with the main concepts regarding access to the FastTrack server: connections, data subscriptions, transactions, queries, etc...

The first chapter FastTrack Overview introduces the main basic concepts of FastTrack with a short overview of system architecture.

The second chapter Data Distribution explains how data are distributed, searched and retrieved inside FastTrack.

Chapter three JFT/Api Details describes the main peculiarities of the Java Access Point to FastTrack: lifecycle, communication model, exception handling, threads synchronization, etc...

Following chapters contain the effective and detailed descriptions of the Api subdivided in two different packages: `it.list.jft` and `it.list.jft.event`,

Finally, after all details on specific JFT/Api functions, a few examples of Java application are given in order to exploit almost all major capabilities offered by JFT/Api and to be possibly fruitfully studied and used as starting point for effectives Java programs that links to a true FastTrack server.

FastTrack Overview

This chapter explains the basic concepts of FastTrack that are fundamental in developing Java applications that access FastTrack services using this JFT/Api.

System Architecture Overview

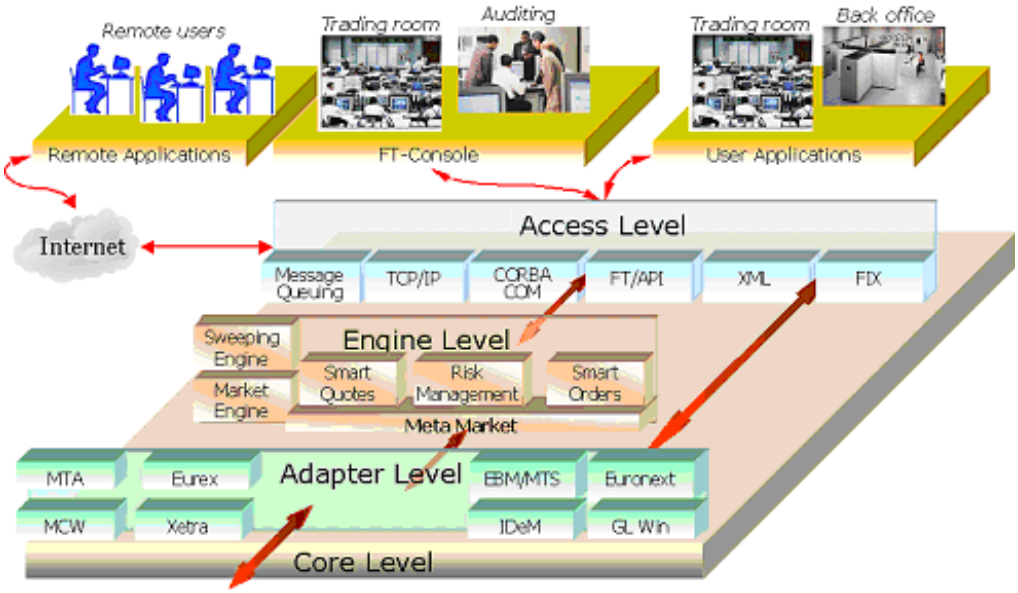
FastTrack's architecture is **modular** and **distributed**.

There are three different architectural levels for the various components, depending on the service they implement:

- Core
- Basic
- Enterprise

The **core level** is the heart of the system, the foundation of FastTrack's modular architecture.

The set of FastTrack modules managing communication with the external world makes up the **basic level**. These are organized into two categories: **Adapters**, through which FastTrack can access other systems or electronic markets, and **Access points**, which give access to FastTrack from the outside. Finally, the **enterprise level** hosts the functional applications in FastTrack (Engine level).



FastTrack's components interact both in a synchronous and asynchronous way − the latter uses a **publish−subscribe** paradigm.

FastTrack is a distributed system. The computing process is subdivided into several steps. It is not performed by one individual component, but by a series of elements (Application Servers or **Services**) each designed to carry out its particular part of the process.

Access points

The Access Point Level is the only interface FastTrack offers to use its services from the outside.

Queries made to the FastTrack platform and answers to these queries both go through the Access Point Level.

The Access Point Level:

- provides access to external applications which use communication protocols which are very different from one another, thus guaranteeing their complete access to FastTrack's functionalities;
- manages connections coming from external applications;
- manages in a centralized way the sessions that have been opened on FastTrack, by users connected via proprietary applications, or the FastTrack Console, or a Web browser, independently of the communication protocol;
- provides all functionalities needed by the external application to utilize the services offered by internal engines, hence:
    - ♦ transactions to send requests;
    - ♦ real−time data distribution mechanisms (for example push on HTML);
    - ♦ mechanisms to perform point to point requests (e.g. Query).

JFT/Api Access Point

This JFT/Api library is just a FastTrack Access Point that offers a Java interface to be used in order to connect FastTrack servers via TCP/IP connections.

Using this library it is possibile to construct Java applications and/or applets that may communicate with one or more FastTrack servers and/or services.

Obviously the effective access to these services is managed/controlled/verified in relation to various credentials (user names, passwords, authorization keys, etc...) that an external application must presents in order to be properly authorized to enter in the system.

Data Distribution

Data are distributed through a publish/subscribe protocol, in which producers and consumers exchange messages to access the data. There are many data structure exchanged within FastTrack. Each data structure is called EntityClass and it is structured in many fields of many different types (numbers, strings, etc...). Each set of values that corresponds to these fields is an instance of the EntityClass. This instance id called Entity. Applications may modify or access an entire Entity (all the fields of an instance of an EntityClass) or a subset of these fields defined using a mask.

From now on we call:

- *client* any Java application that use JFT/Api,
- *server* any FastTrack service that is connected to the *client.*

As we will see below, normally a client is a subscriber, and the server is a producer, of a set of data which are exchanged between them.

Publish

The producer (tipically a fastTrack service) notifies the availability of new data with a publish message. These messages are sent to all connected components (other Faststrack services and/or JFT applications) that expressed interest on these type of data.

> *Example:*
> a FastTrack order−manager publishes all records that describes new received or changed orders.

Only certain EntityClasses of a FastTrack server may be published and then subscribed. The documentation of each FastTrack server clearly says which EntityClasses can be published/subscribed.

Subscribe

A request for data by consumers is done by a subscribe message. This message typically says in which EntityClass the consumer is interested.

> *Example:*
> a customer subscribes to the orders handled by FastTrack.

Subscriptions in JFT/Api are modelled by Subscription. Among other things within subscriptions it's possible to have:

- Incremental Subscriptions, in which the server is only required to send updated contents of an EntityClass, rather than sending all its records;
- Partial Subscriptions, in which the server is requested to send only those entities in a EntityClass that satisfy certain constraints;
- an optional Filter to restrict (at the server level) the set of entities that will be notified;
- an optional Mask to restrict (at the server level) the set of fields of entities that will be notified.

See Subscription Usage to see all specific subscription−related modalities.

Queries

In addition to the publish/subscribe mechanism a client has the possibility to obtain from the server a specific set of entities. This is done with the Query metaphor that mimics the homonymous facility of DBMS. Normally each type of query (identified by a specific number) has an argument that specify the particular request.

> *Example:*
> A client queries for all orders sent by a specific operator.

Only certain queries (each identified by a unique number) are permitted with a FastTrack server. The documentation of each FastTrack server clearly lists which queries (i.e. numbers) are permitted and with which arguments.

Transactions

The client may request a server to make some actions that may result in the update of one or more entities. The server evaluates the request and accepts or refuses it. Accepting the modification often implies application specific check actions by the server with the aim of controlling the access rights and action consistency while interpreting the semantics of the request arrived from the client. This is modeled in JFT/Api with the Transaction metaphor.

> *Example:*
> the client asks the server to issue an order on a specific product.

This operation may take a long period to be completed by the server and so the client have to monitor it until a good or bad (commited vs aborted) final result. This monitoring must be always done by the client, even after a client−restart on previous initiated (past) transactions. Only when the client see the final commited result it may assume that the transaction was succesfully.

See Transaction Usage to see all specific attributes of a transaction and how to monitor past transactions.

Connections and Contexts

All the above described capabilities are communicated between the client Java application (that use these JFT/Api) and the FastTrack server (a specific service of a FastTrack server) using a TCP/IP channel modelled with a Connection. The main attributes of such Connections are the TCP/IP host and port on which a named service reside.

> *Example:*
> an order−manager service may reside on port 1234 of host `myFTserver.myDomain.com` (or something like 194.91.195.33) and beeing named `OrderManager`

See Connection Usage to see all specific attributes that define a connection.

In addition the Context. metaphor has been introduced in order to group together connections (and corresponding subscriptions, transactions, queries, etc..) that refer to the same set of related FastTrack services.

JFT/Api Details

The JFT/Api library may be used starting with Sun JDK 1.7.

Asynchronous Communication Model

The implementation of JFT/API functionalities is based on an asynchronous communication model. A functionality (such as connection opening or a subscription for a set of data) is requested to the library via a method invocation. This request specifies (among other parameters) some notification methods, defined by the user in some Listener, which will be called by the library when data arrive or when other events occur.

See Listener interface for specific details.

LifeCycle

Many objects exposed by the JFT/Api library share the life cycle metaphor: once they are created, their life goes through well defined steps depending on their internal status.

See the LifeCycle interface and its related sub−interfaces to understand how this behaviour is controlled and regulated.

Data Model

The UML data models of the two packages of this JFT/Api library are available.

See `it.list.jft` Data Model and `it.list.jft.event` Data Model to familiarize with the hierarchy and relationship of the various interfaces.

Other Peculiarities

Some other details of the JFT/Api implementation are referenced here in order to use at the best the library:

- JFT Exceptions describes when and how the library throw exceptions,
- JFT Implementation Threads describes how the library use its own threads and how to successfully exit from a Java client,
- JFT Synchronization describes the synchronization needs and requirements that must be obeyed in the execution of some specific Listener methods.

JFT/Api Entry Point

Last but not least:

> *Where is the first initial entry point to use this library?*

It's available in the singleton that implements the JFT interface. Using that singleton, referenced by the THIS constant, every programmer may starts to use the library accessing all its functionalities.

JavaScript is disabled on your browser.

Package it.list.jft

Provides interfaces for dealing with different FastTrack objects.
See: Description

- Interface Summary

Interface	Description
ActivityLifeCycle	Super−interface common to all lifecycles objects of a given `Connection`.
CommunicationLifeCycle	Super−interface common to all lifecycles objects of a given `Context`.
Connection	Logical bidirectional channel with a server.
ConnectionParam	Connection parameter container.
Context	Container and factory of inter−related communication objects.
Entity	Interface that describes a specific instance of a `EntityClass`.
EntityClass	Interface that describes a specific market/service class.
EntityClassQuery	
EntityClassQueryParam	
EntityField	
EntityFilter	Usually, fasttrack serverices implements a default filter to restrict the set of values notified by a Subscription, based on full or partial key values.
EntityKey	An actual (partial or full) key value of a key of an `EntityClass`.
Filter	A manner to restrict the set of values notified by a Subscription.
FilterParam	Filter parameter container.
JFT	Main basic library interface to use within JFT/API.
LifeCycle	Super−interface common to all lifecycles.
Mask	A set of fields of a `EntityClass`.
MulticastConnection	
MulticastConnectionParam	
Param	Super−interface common to all parameter container of `CommunicationLifeCycle` objects.
Query	A client's request to a server to obtain a set of entities (or rows) from its own Data Base.
QueryParam	Query parameter container.
Subscription	An arrangement with the server for receiving a continuing set of interesting entities of the same EntityClass.
SubscriptionParam	Subscription parameter container.
TimeStamp	Interface that allows to represent a temporal indicator.
Tracer	Interface to be implemented in order to handle the library trace.

Transaction	A client's request to the server to add, remove or modify an entity in its own Data Base.
TransactionID	Interface that allows to identify a Transaction.
TransactionParam	Transaction parameter container.

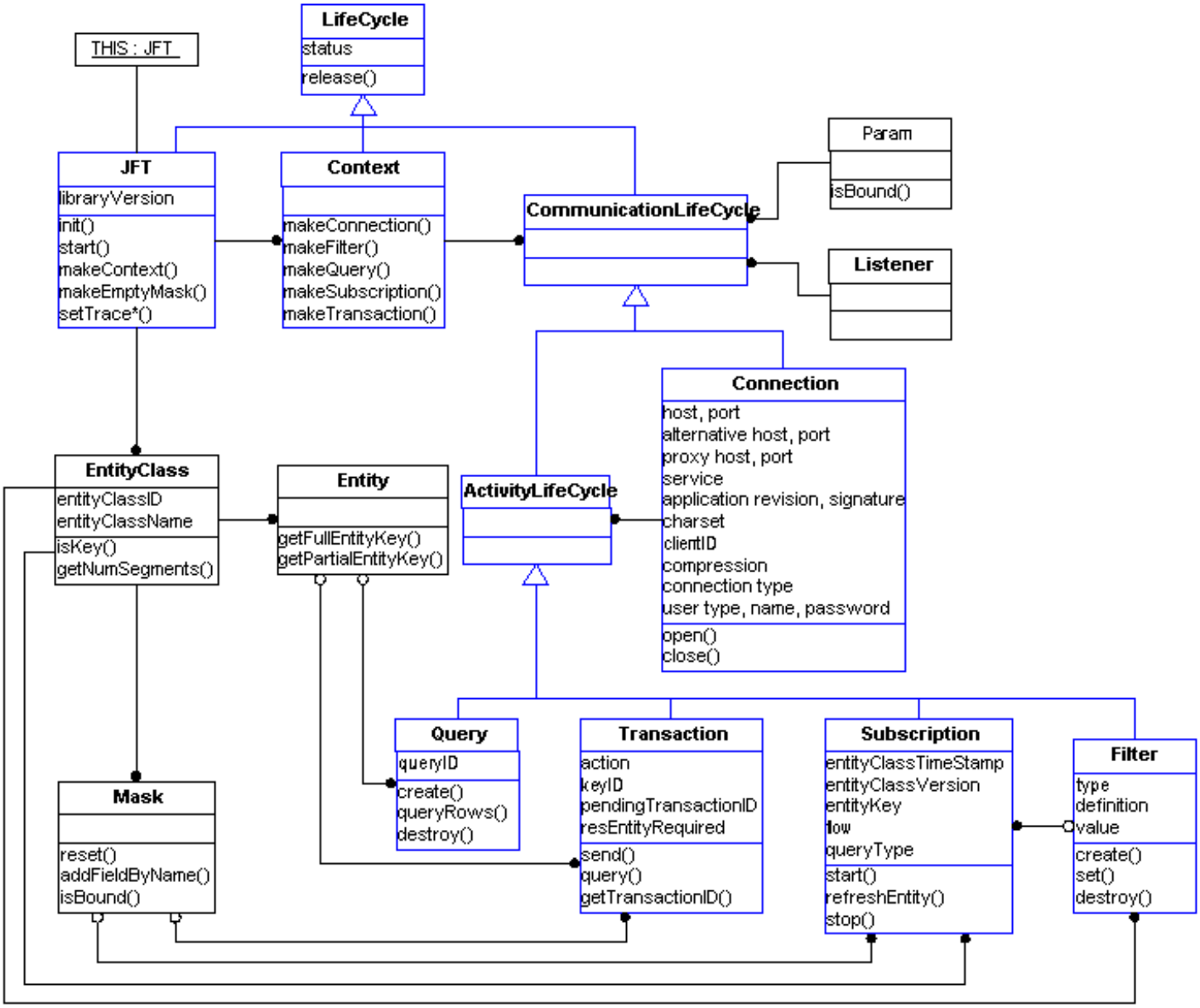
Package it.list.jft Description

Provides interfaces for dealing with different FastTrack objects.

Implementation of `Tracer` must be provided by the JFT application.
Implementation for all other interfaces is already provided by the JFT library.

See the hierarchy of this it.list.jft package and the `JFT` documentation for details.

Package it.list.jft Data Model



The above figure is the UML representation of it.list.jft data model.
In blue all interfaces that implement LifeCycle objects.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

Hierarchy For Package it.list.jft

Package Hierarchies:

- All Packages

Interface Hierarchy

- Cloneable
  - ♦ Entity (also extends EntityClass, Serializable)
- EntityClass
  - ♦ Entity (also extends Cloneable, Serializable)
- EntityField
- EntityKey
- LifeCycle
  - ♦ CommunicationLifeCycle
    - ◊ ActivityLifeCycle
      - · EntityClassQuery
      - · Filter
        - EntityFilter
      - · Query
      - · Subscription
      - · Transaction
    - ◊ Connection
    - ◊ MulticastConnection
  - ♦ Context
  - ♦ JFT
- Mask
- Param
  - ♦ ConnectionParam
  - ♦ EntityClassQueryParam
  - ♦ FilterParam
  - ♦ MulticastConnectionParam
  - ♦ QueryParam
  - ♦ SubscriptionParam
  - ♦ TransactionParam
- Serializable
  - ♦ Entity (also extends Cloneable, EntityClass)
  - ♦ TimeStamp
  - ♦ TransactionID
- Tracer

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface EntityClass

- *All Known Subinterfaces:*
  Entity

```
public interface EntityClass
```

Interface that describes a specific market/service class.

All market/service EntityClasses share a set of common methods to:

- retrieve the `class name`,
- retrieve the `class ID`,
- check if if a given KeyID is a valid key index,
- retrieve the `number of segments of a given KeyID`.

In addition all market/service objects, that implement the `Entity` sub−interface, share, as well, these methods.

If necessary explicitly objects that implement this interface are created and returned by the `JFT.getEntityClass()` method.

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`TYPE_ENTITY`***
`static int`	***`TYPE_ENUM`***

Method Summary

Methods

Modifier and Type	Method and Description
`int`	***`getEntityClassID`***`()` Returns the ID that identifies the EntityClass.
`String`	***`getEntityClassName`***`()` Returns the name that identifies the EntityClass.
`EntityField[]`	***`getEntityFields`***`()`
`int`	***`getNumSegments`***`(int keyID)` Returns the number of segments of the given KeyID of this EntityClass.
`int`	***`getType`***`()`
`boolean`	***`isKey`***`(int keyID)` Check if a given `keyID` is an index of a key for this EntityClass.
`boolean`	***`isKey`***`(int keyID, boolean checkPrimary)` Check if a given `keyID` is an index of a primary or duplicate key for this EntityClass.

`Entity` ***`makeEntity`***`()`

- ♦

Field Detail

◊ **TYPE_ENTITY**

```
static final int TYPE_ENTITY
```

See Also:
Constant Field Values

◊ **TYPE_ENUM**

```
static final int TYPE_ENUM
```

See Also:
Constant Field Values

♦

Method Detail

◊ **getEntityClassName**

```
String getEntityClassName()
```

Returns the name that identifies the EntityClass.

Returns:
the name that identifies the EntityClass.
`null` and empty strings are never returned.

◊ **getEntityClassID**

```
int getEntityClassID()
```

Returns the ID that identifies the EntityClass.

Returns:
the ID that identifies the EntityClass.
zero or negative values are never returned.

◊ **isKey**

```
boolean isKey(int keyID)
```

Check if a given `keyID` is an index of a key for this EntityClass.

Parameters:
`keyID` − index to be checked
Returns:
`true` if `keyID` is an index of a key for this EntityClass,
`false`otherwise.

◊ **isKey**

```
boolean isKey(int keyID,
              boolean checkPrimary)
```

Check if a given `keyID` is an index of a primary or duplicate key for this EntityClass.

Parameters:
`keyID` − index to be checked
`checkPrimary` − check for primary or duplicate key index
Returns:
`true` if `keyID` is an index of a key for this EntityClass and it refers a primary or duplicate key as specified by `checkPrimary` parameter,
`false`otherwise.

◊ **getNumSegments**

```
int getNumSegments(int keyID)
```

Returns the number of segments of the given KeyID of this EntityClass.

Parameters:
`keyID` − the index of a key of this class.
Returns:
the number of segments of the given KeyID of this EntityClass.
0 is returned if the `KeyID` parameter is not a valid index of a key of this class.

◊ **getType**

```
int getType()
```

◊ **getEntityFields**

```
EntityField[] getEntityFields()
```

◊ **makeEntity**

```
Entity makeEntity()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Entity

- *All Superinterfaces:*
  Cloneable, EntityClass, Serializable

```
public interface Entity
extends EntityClass, Cloneable, Serializable
```

Interface that describes a specific instance of a `EntityClass`.

All entities objects share two common methods to retrieve full and partial EntityKeys, in additions to inherithed methods from EntityClass.

All other specific fields of each entity are available as specific fields of the correspinding Java object that implements this interface.

- ♦
Field Summary

◊
Fields inherited from interface EntityClass

`TYPE_ENTITY, TYPE_ENUM`

♦
Method Summary

Methods

Modifier and Type	Method and Description
`Object`	***`clone`***`()` Implement Cloneable interface.
`Object`	***`getField`***`(String fieldName)` Returns a field value of this Entity.
`EntityKey`	***`getFullEntityKey`***`(int keyID)` Returns a given full EntityKey of this Entity.
`EntityKey`	***`getPartialEntityKey`***`(int keyID, int numSegments)` Returns a given partial EntityKey of this Entity.
`void`	***`setField`***`(String fieldName, Object value)` Set a field value of this Entity.

◊
Methods inherited from interface EntityClass

`getEntityClassID, getEntityClassName, getEntityFields, getNumSegments, getType, isKey, isKey, makeEntity`

- ♦
Method Detail

◊ **getFullEntityKey**

```
EntityKey getFullEntityKey(int keyID)
```

Returns a given full EntityKey of this Entity.

Please note:
`getFullEntityKey(keyID) == getPartialEntityKey(keyID, getNumSegments(keyID))`

Parameters:
`keyID` − the index of a key of this class.
Returns:

a given full EntityKey of this Entity.
`null` is returned when the given `keyID` does not refer to a valid key for the `EntityClass` of this Entity.

◊ **getPartialEntityKey**

```
EntityKey getPartialEntityKey(int keyID,
                              int numSegments)
```

Returns a given partial EntityKey of this Entity.

Parameters:
`keyID` − the index of a key of this class.
`numSegments` − number of initial segments that must be present in the partial key.
Returns:
a given partial EntityKey of this Entity.
`null` is returned when the given `keyID` does not refer to a valid key for the `EntityClass` of this Entity,
or when the given `numSegments` parameter is <=0 or > getNumSegments(keyID).

◊ **getField**

```
Object getField(String fieldName)
                throws NullPointerException,
                       IllegalArgumentException
```

Returns a field value of this Entity.

Please note:
− to get array value at index i use "fieldname[i]"
− for nested entity field use "." as separator

For primitive value, it returns the Object corresponding to it (e.g. `int` as returned as `Integer`).

Parameters:
`fieldName` − the name of the field.
Returns:
the Object value of the field.
`null` is returned when the given `fieldName` does not refer to a valid field for the `EntityClass` of this Entity.
Throws:
`IllegalArgumentException` − if the field name is not valid.
`NullPointerException`

◊ **setField**

```
void setField(String fieldName,
              Object value)
                throws NullPointerException,
                       IllegalArgumentException,
                       ClassCastException
```

Set a field value of this Entity.

Parameters:
`fieldName` − the name of the field.

`value` − the Object value of the field.

Throws:

`IllegalArgumentException` − if the field name is not valid.
`ClassCastException` − if the Object value type is not valid.
`NullPointerException`

See Also:

`getField(java.lang.String)`

◊ **clone**

```
Object clone()
            throws CloneNotSupportedException
```

Implement Cloneable interface.

Throws:

`CloneNotSupportedException`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface EntityField

```
public interface EntityField
```

♦

Field Summary

Fields

Modifier and Type	Field and Description
static int	***TYPE_BOOLEAN***
static int	***TYPE_BYTE***
static int	***TYPE_CHAR***
static int	***TYPE_DATE***
static int	***TYPE_DOUBLE***
static int	***TYPE_DTIME***
static int	***TYPE_ENTITY_CLASS***
static int	***TYPE_FLOAT***
static int	***TYPE_INT***
static int	***TYPE_LDATE***
static int	***TYPE_LONG***
static int	***TYPE_LTIME***
static int	***TYPE_MTIME***

static int	***TYPE_SHORT***
static int	***TYPE_STRING***
static int	***TYPE_TDATE***
static int	***TYPE_TIME***
static int	***TYPE_UCHAR***
static int	***TYPE_UINT***
static int	***TYPE_ULONG***
static int	***TYPE_USHORT***

Method Summary

Methods

Modifier and Type	Method and Description
EntityClass	***getEntityClass***()
String	***getName***()
int	***getNumElements***()
int	***getType***()

Field Detail

TYPE_ENTITY_CLASS

```
static final int TYPE_ENTITY_CLASS
```

See Also:
Constant Field Values

TYPE_INT

```
static final int TYPE_INT
```

See Also:
Constant Field Values

TYPE_UINT

```
static final int TYPE_UINT
```

See Also:
Constant Field Values

TYPE_SHORT

```
static final int TYPE_SHORT
```

See Also:

Constant Field Values

◊ **TYPE_USHORT**

```
static final int TYPE_USHORT
```

See Also:

Constant Field Values

◊ **TYPE_LONG**

```
static final int TYPE_LONG
```

See Also:

Constant Field Values

◊ **TYPE_ULONG**

```
static final int TYPE_ULONG
```

See Also:

Constant Field Values

◊ **TYPE_FLOAT**

```
static final int TYPE_FLOAT
```

See Also:

Constant Field Values

◊ **TYPE_DOUBLE**

```
static final int TYPE_DOUBLE
```

See Also:

Constant Field Values

◊ **TYPE_BYTE**

```
static final int TYPE_BYTE
```

See Also:

Constant Field Values

◊ **TYPE_CHAR**

```
static final int TYPE_CHAR
```

See Also:

Constant Field Values

◊ **TYPE_UCHAR**

```
static final int TYPE_UCHAR
```

See Also:

Constant Field Values

◊ **TYPE_STRING**

```
static final int TYPE_STRING
```

See Also:

Constant Field Values

◊ **TYPE_DATE**

```
static final int TYPE_DATE
```

See Also:

Constant Field Values

◊ **TYPE_TIME**

```
static final int TYPE_TIME
```

See Also:

Constant Field Values

◊ **TYPE_TDATE**

```
static final int TYPE_TDATE
```

See Also:

Constant Field Values

◊ **TYPE_LTIME**

```
static final int TYPE_LTIME
```

See Also:

Constant Field Values

◊ **TYPE_LDATE**

```
static final int TYPE_LDATE
```

See Also:

Constant Field Values

◊ **TYPE_BOOLEAN**

```
static final int TYPE_BOOLEAN
```

See Also:

Constant Field Values

◊ **TYPE_MTIME**

```
static final int TYPE_MTIME
```

See Also:

Constant Field Values

◊ **TYPE_DTIME**

```
static final int TYPE_DTIME
```

See Also:

Constant Field Values

♦

Method Detail

◊ **getType**

```
int getType()
```

◊ **getEntityClass**

```
EntityClass getEntityClass()
```

◊ **getName**

```
String getName()
```

◊ **getNumElements**

```
int getNumElements()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface EntityKey

```
public interface EntityKey
```

An actual (partial or full) key value of a key of an `EntityClass`.

An EntityKey is an ordered set of N values corresponding to the ordered set of K (K >= N) segments that describe a key of an `EntityClass`.

If N == K then the EntityKey is full, otherwise (0 < N < K) it's partial.

The type of each segment is a Java primitive type (`boolean, byte, char, short, int, long, float, double`) or it is a `String`.

An Entity Key may be used in subscriptions (`SubscriptionParam.setEntityKey()` and `Subscription.refreshEntity()`) or it may be retrieved from entities (`Entity.getFullEntityKey()` and `Entity.getPartialEntityKey()`) and then re−used.

♦

Method Summary

Methods

Modifier and Type	Method and Description
`int`	***`getEntityClassID`***`()` Returns the ID of the `EntityClass` related to this EntityKey.
`int`	***`getKeyID`***`()` Returns the key ID of this key.
`int`	***`getNumSegments`***`()` Returns N (N>0), the numbers of set segments of this EntityKey.

- ♦ **Method Detail**

◊ **getKeyID**

```
int getKeyID()
```

Returns the key ID of this key.

The returned value is the same `keyID` used as parameter of `Entity.getFullEntityKey()` or `Entity.getPartialEntityKey()` invocations that created this EntityKey.

Returns:
the key ID of this key.

◊ **getEntityClassID**

```
int getEntityClassID()
```

Returns the ID of the `EntityClass` related to this EntityKey.

The returned value is the EntityClassID of the Entity that created (via `Entity.getFullEntityKey()` or `Entity.getPartialEntityKey()` this EntityKey.

Returns:
the ID of the `EntityClass` related to this EntityKey.

◊ **getNumSegments**

```
int getNumSegments()
```

Returns N (N>0), the numbers of set segments of this EntityKey.

The returned value is the number of segments of the EntityClass for a full EntityKey, or it is the same `numSegments` used as parameter of `Entity.getPartialEntityKey()` for a partial EntityKey.

Returns:
N (N>0), the numbers of set segments of this EntityKey.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface LifeCycle

- *All Known Subinterfaces:*
  ActivityLifeCycle, CommunicationLifeCycle, Connection, Context, EntityClassQuery, EntityFilter, Filter, JFT, MulticastConnection, Query, Subscription, Transaction

---

```
public interface LifeCycle
```

Super−interface common to all lifecycles.

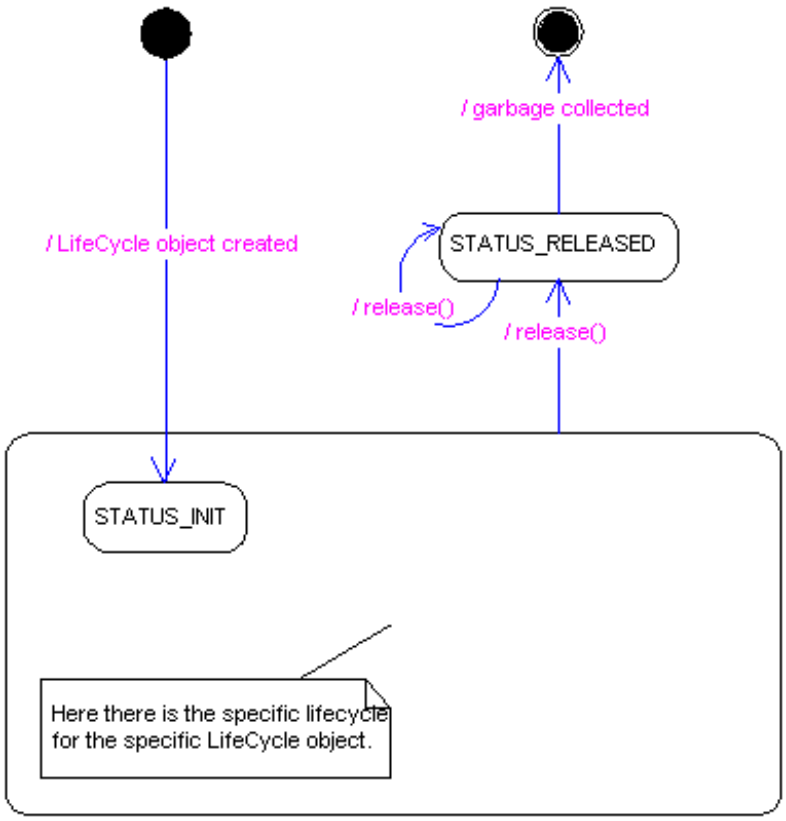
LifeCycle Usage

All JFT LifeCycle objects share:

- an initial `STATUS_INIT` status where every object goes immediately after its creation,
- a final `STATUS_RELEASED` status where every object goes when the `release()` method is explicitly invoked;
- a `getStatus()` method to retrieve the current object status;
- a `release()` method to abruptly and recursively move an object in the final `STATUS_RELEASED` status.
- three result−codes (`RESULT_OK`, `RESULT_INVALID_STATUS` and `RESULT_GENERIC_ERROR`) that may returned by many JFT methods.

The three directs sub−interfaces of LifeCycle (`JFT`, `Context` and `CommunicationLifeCycle`) add many other capabilities to this interface.

Lifecycle



See Also:

JFT LifeCycle, Context LifeCycle, Connection LifeCycle, Filter LifeCycle, Query LifeCycle, Subscription LifeCycle, Transaction LifeCycle

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***RESULT_GENERIC_ERROR*** Generic failure−code returned when a more specific error is not available.
`static int`	***RESULT_INVALID_STATUS*** Failure−code returned when an operation is requested whitin a not correct status.
`static int`	***RESULT_OK*** Positive answer returned when the operation completed successfully.
`static int`	***STATUS_INIT*** Lifecycle status: initial status for every object that implements the `LifeCycle` interface.
`static int`	***STATUS_RELEASED*** Lifecycle status: final status for every object that implements the `LifeCycle` interface.

Method Summary

Methods

Modifier and Type	Method and Description
`Enumeration`	***enumChilds***`()` Returns an enumeration of all non−`STATUS_RELEASED` childs of this LifeCycle.
`int`	***getStatus***`()` Returns the current lifecycle status of this object.
`void`	***release***`()` Abruptly and recursively move an object in the final `STATUS_RELEASED` status.

Field Detail

RESULT_OK

```
static final int RESULT_OK
```

Positive answer returned when the operation completed successfully.
See Also:
Constant Field Values

RESULT_GENERIC_ERROR

```
static final int RESULT_GENERIC_ERROR
```

Generic failure−code returned when a more specific error is not available.
See Also:
Constant Field Values

◊ RESULT_INVALID_STATUS

```
static final int RESULT_INVALID_STATUS
```

Failure−code returned when an operation is requested whitin a not correct status.
See Also:
Constant Field Values

◊ STATUS_INIT

```
static final int STATUS_INIT
```

Lifecycle status: initial status for every object that implements the `LifeCycle` interface. This value may be returned by `getStatus()`.

Status Entry:
object creation `STATUS_INIT`.
Status Activities:
`getStatus()` and other specific activities allowed in this status for the specific subinterfaces of LifeCycle.
Status Exit:
any status `release()` `STATUS_RELEASED`.
See Also:
Constant Field Values

◊ STATUS_RELEASED

```
static final int STATUS_RELEASED
```

Lifecycle status: final status for every object that implements the `LifeCycle` interface. This value may be returned by `getStatus()`.

Status Entry:
any status `release()` `STATUS_RELEASED`.
Status Activities:
`getStatus()` or other activities that does not depend from the status of the object (e.g. `JFT.getLibraryVersion()` in JFT, `CommunicationLifeCycle.getContext()` in CommunicationLifeCycle, etc...).
Status Exit:
none: an object in this status will never change status.
See Also:
Constant Field Values

♦ Method Detail

◊ release

```
void release()
```

Abruptly and recursively move an object in the final `STATUS_RELEASED` status.

This object, and all others objects that depends from this object (see later), are abruptly moved on the final `STATUS_RELEASED` status. For the objects in this final status:

· very few activities are availables (see STATUS_RELEASED description),
· any automatic Listener method invocation is never made.

The recursive moving of an object in the final status obeys to the following tree structure:

- JFT
  - Context
    - Connection
      - Filter
      - Subscription
      - Query
      - Transaction

E.g. if this method is called on a connection then this connection **and** all its childs (filters, subscriptions, queries and transactions) are moved on the final STATUS_RELEASED status. Please note: JFT, all contexts, all other connections and all other childs of others connections are unaffected by this operation in this example.

This method may be invoked at any time, even if this object is already in the final STATUS_RELEASED status.

◊ getStatus

```
int getStatus()
```

Returns the current lifecycle status of this object.

Each object has a status that may be one of the two commons LifeCycle status (STATUS_INIT or STATUS_RELEASED) or a specific status that is described by a constant STATUS_ described in one of the subinterface of LifeCycle.

Returns:

the current lifecycle status of this object.

◊ enumChilds

```
Enumeration enumChilds()
```

Returns an enumeration of all non−STATUS_RELEASED childs of this LifeCycle.

The hierarchy is depicted in the release() description.

This method returns only the first level childs of a given LifeCycle: e.g. for a Context it returns only the Connections of this Context and not their ActivityLifeCycle childs.

This method may be invoked at any time, even if this object is in the final STATUS_RELEASED status.

Returns:

an enumeration of all non−STATUS_RELEASED childs of this LifeCycle. null is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface CommunicationLifeCycle

- *All Superinterfaces:*
  LifeCycle

All Known Subinterfaces:
ActivityLifeCycle, Connection, EntityClassQuery, EntityFilter, Filter, MulticastConnection, Query, Subscription, Transaction

---

```
public interface CommunicationLifeCycle
extends LifeCycle
```

Super−interface common to all lifecycles objects of a given `Context`.

The 5 kind of objects (`Connection`, `Filter`, `Query`, `Subscription` and `Transaction`) that implement this interface:

- are created by `makeSomething` methods of their `Context`,
- shares 3 common methods to get their associated Context, `Listener` and `parameter`.

JFT Implementation Threads

The underlying implementation of JFT use some implementation threads to:

- read and write data from/to the server, and
- call the `Listener` methods in the Java application.

These JFT implementation threads are not daemons threads, so the `main` Java thread may safely terminate and the application continue to run until there are some JFT implementation threads running (see the section 12.8 of the *Java Language Specification*).

There is at least one JFT implementation thread:

- when there are some `Listener` methods eligible to be called (e.g. a `Connection.STATUS_CONNECTING`, or a `Subscription.STATUS_STARTED`, or a `Query.STATUS_DESTROYING`, or ...) because there are some i/o data pending.

There are no JFT implementation threads when all CommunicationLifeCycle objects are `LifeCycle.STATUS_RELEASED` so a sure manner to terminate a Java application that use JFT is:

```
JFT.THIS.release()
```

JFT Synchronization

The JFT library guarantees that any implementation thread never holds any monitor lock on any objects that implements any JFT interfaces.
JFT library internally never synchronize on JFT, EntityClasses, Entities, Connections, Subscriptions, etc... objects.

This is **always guaranteed**, even inside the `Listener` methods called inside the JFT implementation threads. So the programmer that uses the JFT library is free to synchronize on these objects for his needs.

But **Warning**:

Each Listener method must execute its task within the shortest time possible.
Do not synchronize anything or make time−intensive computation inside the Listener methods because this may deadlock the JFT library.
If you have an impelling necessity to do so, you have to start another thread to accomplish your needs.
So, please, do not download a file from FTP and/or do not compute the first one million digits of PI inside a Listener method!

In addition there is no need to synchronize any activity between the programmer and the JFT library. All needed synchronizations are made by JFT library on internal hidden objects, not accessible to the JFT programmer that use this JFT API.
E.g. for internal needs the JFT implementation may synchronize their implementation threads on the subscriptions or on the connections: this is made synchronizing on the private hidden fields *lockObject* that are present in the JFT implementation of Subscription and Connection.

The primitive data returned by the implementation to the JFT application (e.g. the status returned by `LifeCycle.getStatus()` method) are internally implemented as primitive `volatile` data, so they are almost always in synch between the implementation and the application.

So the JFT application does not need to protect itself with something like:

```
if (mySubscription.getStatus() == LyfeCycle.STATUS_INIT)
   mySubscription.start();
```

because, even if the status returned in `mySubscription.getStatus()` is `LyfeCycle.STATUS_INIT`, when the application invoke the `mySubscription.start()` method, the status may already be changed to `STATUS_RELEASED` because another thread, in the meantime, has released the Connection associated to `mySubscription`.

The JFT library implementation instead surely synchronize the access to the various objects and status with something like:

```
public int start() {
   synchronized(lockObject) {
      if (status == STATUS_INIT) {
         ...
         status = STATUS_STARTING;
      }
   }
}
```

E.g. this prevents any status changes by others threads inside the `start()` code.

In brief:

- The JFT application does not care of JFT implementation threads.
- The JFT application does not need to synchronize with these JFT implementation threads.
- The JFT application is free to synchronize on any objects for its own needs.

but

- **WARNING:** The JFT application must not synchronize inside a Listener method.
- **WARNING:** The JFT application must not made time−intensive computation inside a Listener method.

See Also:
`LifeCycle`

- ♦
Field Summary

◊
Fields inherited from interface LifeCycle

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

♦
Method Summary

Methods

Modifier and Type	Method and Description
`Context`	***`getContext`***`()` Returns the associated Context.
`Listener`	***`getListener`***`()` Returns the associated Listener.
`Param`	***`getParam`***`()` Returns the associated Param.

◊
Methods inherited from interface LifeCycle

`enumChilds`, `getStatus`, `release`

- ♦
Method Detail

◊ **getContext**

```
Context getContext()
```

Returns the associated Context.

Each CommunicationLifeCycle object has a Context from which it was created (e.g. for a `Subscription` the associated Context is the object on which the `Context.makeSubscription()` method was invoked).
This method return this Context.

Returns:
the associated Context.
`null` is never returned.

◊ **getParam**

```
Param getParam()
```

Returns the associated Param.

Each CommunicationLifeCycle object has a specialized sub−interface of Param that described the specific creation parameter for that object (e.g. for a `Subscription` the associated Param is the second parameter (`SubscriptionParam`) of `Context.makeSubscription()`).
This method return this parameter casted to the super−interface `Param`.

All parameters returned by this method are `bound`.

Returns:
the associated Param.
`null` is never returned.

◊ **getListener**

```
Listener getListener()
```

Returns the associated Listener.

Each CommunicationLifeCycle object has a specialized sub−interface of Listener that described the specific listener for that object (e.g. for a `Subscription` the associated Listener is the Third parameter (`SubscriptionListener`) of `Context.makeSubscription()`).
This method return this parameter casted to the super−interface `Listener`.

Returns:
the associated Listener.
`null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface ActivityLifeCycle

- *All Superinterfaces:*
  CommunicationLifeCycle, LifeCycle

All Known Subinterfaces:
EntityClassQuery, EntityFilter, Filter, Query, Subscription, Transaction

---

```
public interface ActivityLifeCycle
extends CommunicationLifeCycle
```

Super−interface common to all lifecycles objects of a given `Connection`.

The 4 kind of objects (`Filter`, `Query`, `Subscription` and `Transaction`) that implement this interface:

- are created by `makeSomething` methods of their `Context`,
- share a common method to get the associated Connection,
- share a common failure−code for attempted operations when the associated Connection is not in a good state.

See Also:
`CommunicationLifeCycle`

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`RESULT_INVALID_CONNECTION_STATUS`*** Failure−code returned when an operation is requested whitin a not correct status of the associated Connection.

Fields inherited from interface LifeCycle

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

Method Summary

Methods

Modifier and Type	Method and Description
`Connection`	***`getConnection`***`()` Returns the associated Connection.

Methods inherited from interface CommunicationLifeCycle

`getContext`, `getListener`, `getParam`

Methods inherited from interface LifeCycle

`enumChilds`, `getStatus`, `release`

Field Detail

RESULT_INVALID_CONNECTION_STATUS

```
static final int RESULT_INVALID_CONNECTION_STATUS
```

Failure−code returned when an operation is requested whitin a not correct status of the

associated Connection.

Typically an ActivityLifeCycle objects returns this failure−code when the associated Connection is not in the `Connection.STATUS_CONNECTED` status.

See Also:
Constant Field Values

Method Detail

getConnection

```
Connection getConnection()
```

Returns the associated Connection.

Each ActivityLifeCycle object has a Connection that created it (e.g. for a `Subscription` the associated Connection is the first parameter of `Context.makeSubscription()`). This method return this Connection.

Returns:
the associated Connection.
`null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface EntityClassQuery

- *All Superinterfaces:*
ActivityLifeCycle, CommunicationLifeCycle, LifeCycle

```
public interface EntityClassQuery
extends ActivityLifeCycle
```

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***STATUS_QUERIED_NO***
`static int`	***STATUS_QUERIED_OK***
`static int`	***STATUS_QUERYING***

◊ **Fields inherited from interface ActivityLifeCycle**

RESULT_INVALID_CONNECTION_STATUS

◊ **Fields inherited from interface LifeCycle**

RESULT_GENERIC_ERROR, RESULT_INVALID_STATUS, RESULT_OK, STATUS_INIT, STATUS_RELEASED

♦ **Method Summary**

Methods

Modifier and Type	Method and Description
int	*query*()

◊ **Methods inherited from interface ActivityLifeCycle**

getConnection

◊ **Methods inherited from interface CommunicationLifeCycle**

getContext, getListener, getParam

◊ **Methods inherited from interface LifeCycle**

enumChilds, getStatus, release

- ♦ **Field Detail**

◊ **STATUS_QUERYING**

```
static final int STATUS_QUERYING
```

See Also:
Constant Field Values

◊ **STATUS_QUERIED_OK**

```
static final int STATUS_QUERIED_OK
```

See Also:
Constant Field Values

◊ **STATUS_QUERIED_NO**

```
static final int STATUS_QUERIED_NO
```

See Also:

Constant Field Values

♦

Method Detail

◊ **query**

```
int query()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Filter

- *All Superinterfaces:*
  ActivityLifeCycle, CommunicationLifeCycle, LifeCycle

All Known Subinterfaces:
EntityFilter

```
public interface Filter
extends ActivityLifeCycle
```

A manner to restrict the set of values notified by a Subscription.

A filter may be used in a subscription to restrict (at the server level) the set of entities that will be `notified` to the client application.

Filter Usage

A filter is defined by:

- an associated EntityClass,
- a filter type,
- and a filter definition,
  all used during filter creation,
- a filter value,
  used during filter extension.

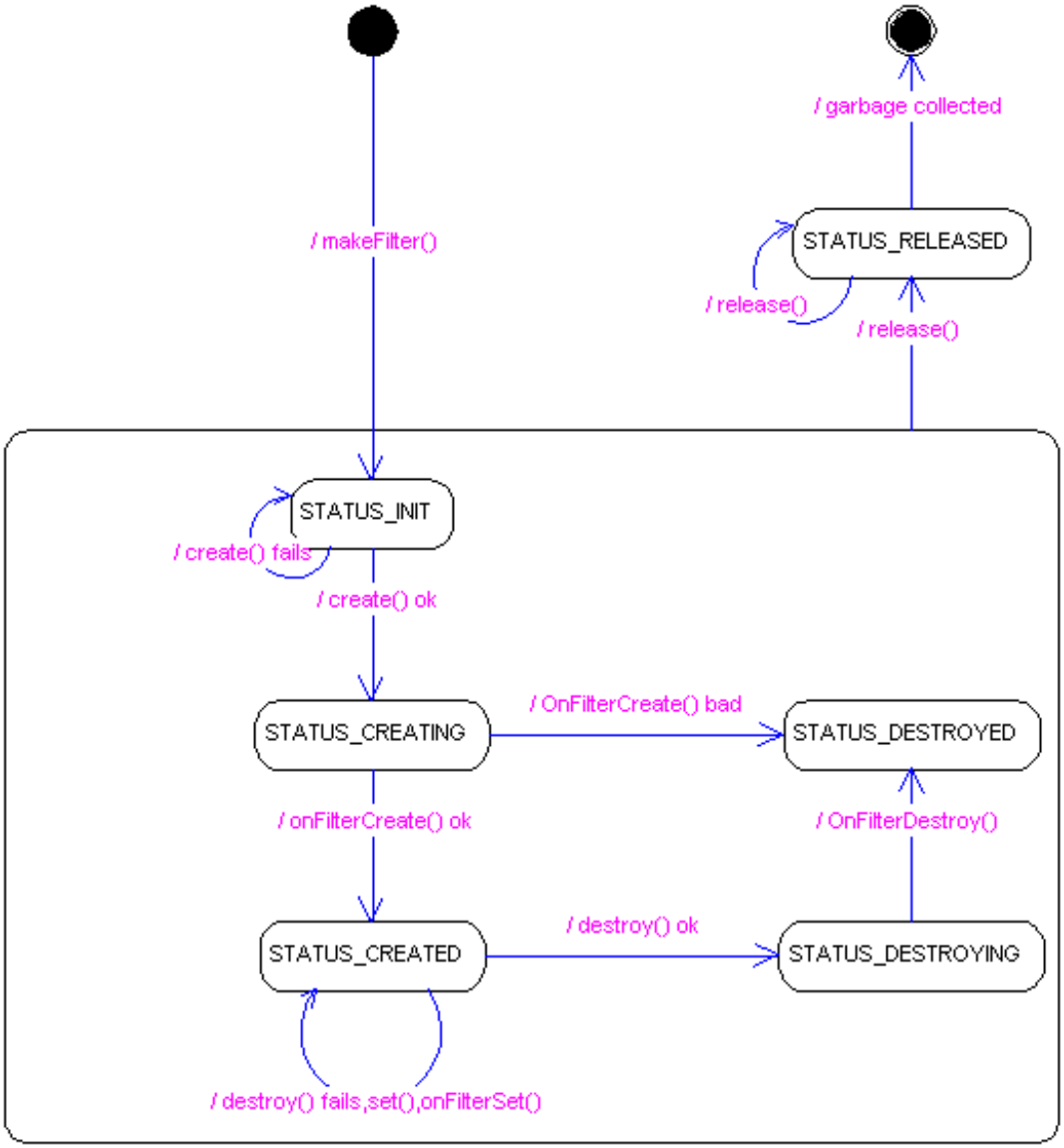
The precise meaning of these 4 things depends from the particular filter and, in general, must be agreed between the client and the server.

In brief:

Filters are locally created by `Context.makeFilter()` in which the filter parameters are described by `FilterParam` and the event listeners are described by `FilterListener`.

Once locally created a filter must be also server created, eventually server extended, used in the Subscription and then server destroyed.

Filter Lifecycle



See Also:

`Context.makeFilter()`, `FilterParam`, `FilterListener`

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`STATUS_CREATED`*** Lifecycle status: Filter created on the server and ready to be used.
`static int`	***`STATUS_CREATING`*** Lifecycle status: Filter waiting the `create()` server−answer.
`static int`	***`STATUS_DESTROYED`***

Lifecycle status: Filter destroyed into the server and ready to be released.

`static int` ***STATUS_DESTROYING***
Lifecycle status: Filter waiting the `destroy()` server−answer.

◊ **Fields inherited from interface ActivityLifeCycle**

`RESULT_INVALID_CONNECTION_STATUS`

◊ **Fields inherited from interface LifeCycle**

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

♦ **Method Summary**

Methods

Modifier and Type	Method and Description
`int`	***create***`()` Try to create this filter on the server.
`int`	***destroy***`()` Try to destroy this filter on the server.
`int`	***set***`(String value)` Try to extend this filter on the server.

◊ **Methods inherited from interface ActivityLifeCycle**

`getConnection`

◊ **Methods inherited from interface CommunicationLifeCycle**

`getContext`, `getListener`, `getParam`

◊ **Methods inherited from interface LifeCycle**

`enumChilds`, `getStatus`, `release`

• ♦ **Field Detail**

◊ **STATUS_CREATING**

```
static final int STATUS_CREATING
```

Lifecycle status: Filter waiting the `create()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
STATUS_INIT create() ok STATUS_CREATING.
Status Activities:
none: waiting an automatic onFilterCreate() call.
Status Exit:
STATUS_CREATING onFilterCreate() ok STATUS_CREATED.
STATUS_CREATING onFilterCreate() bad STATUS_DESTROYED.
See Also:
Filter lifecycle, Constant Field Values

◊ STATUS_CREATED

```
static final int STATUS_CREATED
```

Lifecycle status: Filter created on the server and ready to be used.
This value may be returned by LifeCycle.getStatus().

Status Entry:
STATUS_CREATING onFilterCreate() ok STATUS_CREATED.
Status Activities:
- the filter may be eventually server extended,
- the filter may be used in Subscription.

Status Exit:
STATUS_CREATED destroy() ok STATUS_DESTROYING.
See Also:
Filter lifecycle, Constant Field Values

◊ STATUS_DESTROYING

```
static final int STATUS_DESTROYING
```

Lifecycle status: Filter waiting the destroy() server−answer.
This value may be returned by LifeCycle.getStatus().

Status Entry:
STATUS_CREATED destroy() ok STATUS_DESTROYING.
Status Activities:
none: waiting an automatic onFilterDestroy() call.
Status Exit:
STATUS_DESTROYING onFilterDestroy() STATUS_DESTROYED.
See Also:
Filter lifecycle, Constant Field Values

◊ STATUS_DESTROYED

```
static final int STATUS_DESTROYED
```

Lifecycle status: Filter destroyed into the server and ready to be released.
This value may be returned by LifeCycle.getStatus().

It's always a good practice to release a Filter in this status.

Status Entry:
STATUS_DESTROYING onFilterDestroy() STATUS_DESTROYED.
Status Activities:
LifeCycle.release().

Status Exit:

STATUS_DESTROYED `LifeCycle.release()` `STATUS_RELEASED`.

See Also:

Filter lifecycle, Constant Field Values

♦ Method Detail

◊ create

```
int create()
```

Try to create this filter on the server.

This method must be called only when

- the current status is `STATUS_INIT`,
- the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

- the create request was sent to server,
- the current status changed to `STATUS_CREATING`,
- when the server−answer will be available the `FilterListener.onFilterCreate()` will be automatically called to handle it.

otherwise

- the client has rejected the create,
- the create request was **not** sent to the server,
- automatic call of `FilterListener.onFilterCreate()` will **not** be made,
- the current status remains unchanged.

In the latter case it is a good practice to release this Filter.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_INIT`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

◊ set

```
int set(String value)
```

Try to extend this filter on the server.

The filter extension is used to extend an already created filter.

The precise meaning of this value depends from the particular filter and, in general, it must be agreed between the client and the server.

The server may subsequently returns a `FilterSetEvent.RESULT_SYNTAX_ERROR` or

a `FilterSetEvent.RESULT_INVALID_FILTER_LEN` failure−code if it does not understand this value or if this value is too long.

This method must be called only when

· the current status is `STATUS_CREATED`,
· the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

· the extension request was sent to server,
· when the server−answer will be available the `FilterListener.onFilterSet()` will be automatically called to handle it.

otherwise

· the client has rejected the extension,
· the extension request was **not** sent to the server,
· automatic call of `FilterListener.onFilterSet()` will **not** be made.

In any case the current status remains unchanged.

Parameters:

`value` − filter extension of the new filter.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_CREATED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise (e.g. the given filter value is `null` or empty).

◊ destroy

```
int destroy()
```

Try to destroy this filter on the server.

This method must be called only when

· the current status is `STATUS_CREATED`,
· the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

· the destroy request was sent to server,
· the current status changed to `STATUS_DESTROYING`,
· when the server−answer will be available the `FilterListener.onFilterDestroy()` will be automatically called to handle it.

otherwise

· the client has rejected the destroy,
· the destroy request was **not** sent to the server,

· automatic call of `FilterListener.onFilterDestroy()` will **not** be made,
· the current status remains unchanged.

It's not a bad practice to unconditionally release this Filter immediately after this method invocation without handling the returned value.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_CREATED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface EntityFilter

- *All Superinterfaces:*
  ActivityLifeCycle, CommunicationLifeCycle, Filter, LifeCycle

---

```
public interface EntityFilter
extends Filter
```

Usually, fasttrack serverices implements a default filter to restrict the set of values notified by a Subscription, based on full or partial key values.

To use this you must set the `filter type value` as TYPE_ENTITYFILTER and the `Entity Class ID` used by subscription. The filter returned by `makeFilter` can be safely casted to EntityFilter.
After that, it may be used in a `subcription` setting an empty `EntityKey` whill will be used to set filter key values. Please note, you must create and set up different filters for different subscriptions; a filter is not allowed to be used for many subscription simultaneously.
To change filter value you may call:

- add to add an entity key,
- del to remove an entity key from the filter,
- reset to remove all key values from the filter

Plese note, the `set method` doesn't perform any action for this class.

You should call flush after many filter operations to force any buffered action to be sent to the server.

See Also:
`Filter`, `FilterParam`, `FilterListener`

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
static int	***TYPE_ENTITYFILTER*** The filter type value.

- ### Fields inherited from interface Filter

  STATUS_CREATED, STATUS_CREATING, STATUS_DESTROYED, STATUS_DESTROYING

- ### Fields inherited from interface ActivityLifeCycle

  RESULT_INVALID_CONNECTION_STATUS

- ### Fields inherited from interface LifeCycle

  RESULT_GENERIC_ERROR, RESULT_INVALID_STATUS, RESULT_OK, STATUS_INIT, STATUS_RELEASED

Method Summary

Methods

Modifier and Type	Method and Description
int	***add***(EntityKey entityKey) Add the EntityKey to the filter.
int	***del***(EntityKey entityKey) Remove the EntityKey from the filter.
int	***flush***() Send to the server any buffered filter action.
int	***reset***() Remove all the key values from the filter.

- ### Methods inherited from interface Filter

  create, destroy, set

- ### Methods inherited from interface ActivityLifeCycle

  getConnection

- ### Methods inherited from interface CommunicationLifeCycle

  getContext, getListener, getParam

- ### Methods inherited from interface LifeCycle

  enumChilds, getStatus, release

♦ Field Detail

◊ TYPE_ENTITYFILTER

```
static final int TYPE_ENTITYFILTER
```

The filter type value.
See Also:
Constant Field Values

♦ Method Detail

◊ add

```
int add(EntityKey entityKey)
```

Add the EntityKey to the filter.

Parameters:
`entityKey` − the partial or full EntityKey.
Returns:
- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `Filter.STATUS_CREATED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

◊ del

```
int del(EntityKey entityKey)
```

Remove the EntityKey from the filter.

Parameters:
`entityKey` − the partial or full EntityKey.
Returns:
- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `Filter.STATUS_CREATED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

◊ reset

```
int reset()
```

Remove all the key values from the filter.
Returns:
- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not

- `Filter.STATUS_CREATED`,
  - `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
  - `RESULT_GENERIC_ERROR` otherwise.

◊ **flush**

```
int flush()
```

Send to the server any buffered filter action.
Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `Filter.STATUS_CREATED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Query

- *All Superinterfaces:*
  ActivityLifeCycle, CommunicationLifeCycle, LifeCycle

```
public interface Query
extends ActivityLifeCycle
```

A client's request to a server to obtain a set of entities (or rows) from its own Data Base.

An entire result−set (or a part of it, as eventually requested by `queryRows(int, int)`) is returned to the client application, one row at time.

Query Usage

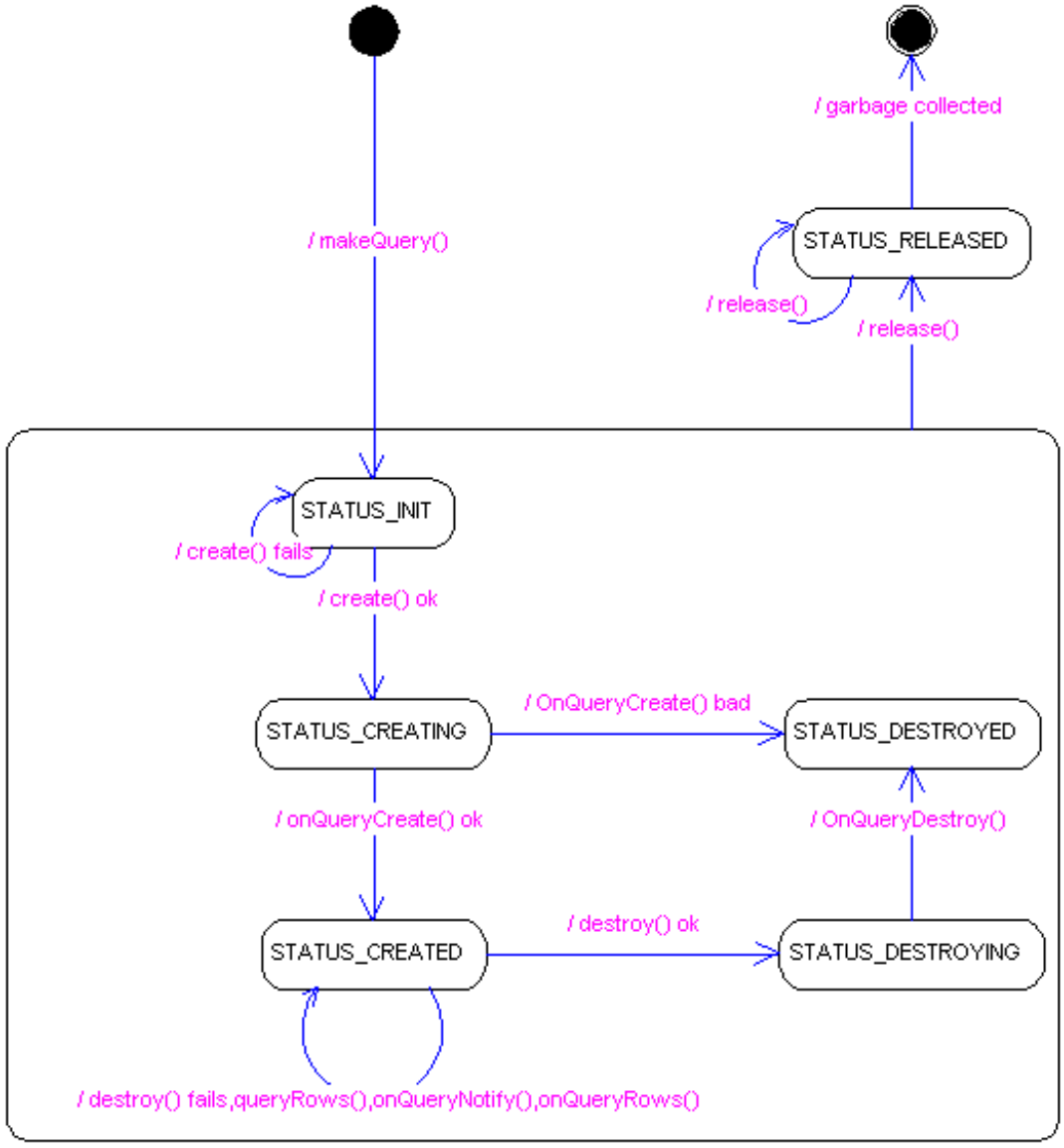
A query is defined by:

- a QueryID identifier that identifies the query in the server,
- an optional Entity that is the optional argument of the query.

The precise meaning of the QueryID identifier must be agreed between the client and the server.
In brief:
Queries are locally created by `Context.makeQuery()` in which the query parameters are described by `QueryParam` and the event listeners are described by `QueryListener`.

Once locally created a query must be also server created, eventually result−set partitioned, used and then server destroyed.

Query Lifecycle



See Also:

`Context.makeQuery()`, `QueryParam`, `QueryListener`

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`STATUS_CREATED`*** Lifecycle status: Query created on the server and ready to be used.
`static int`	***`STATUS_CREATING`*** Lifecycle status: Query waiting the `create()` server−answer.
`static int`	***`STATUS_DESTROYED`***

	Lifecycle status: Query destroyed into the server and ready to be released.
static int	***STATUS_DESTROYING*** Lifecycle status: Query waiting the `destroy()` server−answer.

◊ **Fields inherited from interface ActivityLifeCycle**

`RESULT_INVALID_CONNECTION_STATUS`

◊ **Fields inherited from interface LifeCycle**

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

Method Summary

Methods

Modifier and Type	Method and Description
int	***create***() Try to create this query on the server.
int	***destroy***() Try to destroy this query on the server.
int	***queryRows***(int firstRow, int rowNumber) Try to retrieve a subset of the result−set of this query from the server.

◊ **Methods inherited from interface ActivityLifeCycle**

`getConnection`

◊ **Methods inherited from interface CommunicationLifeCycle**

`getContext`, `getListener`, `getParam`

◊ **Methods inherited from interface LifeCycle**

`enumChilds`, `getStatus`, `release`

Field Detail

◊ **STATUS_CREATING**

```
static final int STATUS_CREATING
```

Lifecycle status: Query waiting the `create()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_INIT` `create()` ok `STATUS_CREATING`.
Status Activities:
none: waiting an automatic `onQueryCreate()` call.
Status Exit:
`STATUS_CREATING` `onQueryCreate()` ok `STATUS_CREATED`.
`STATUS_CREATING` `onQueryCreate()` bad `STATUS_DESTROYED`.
See Also:
Query lifecycle, Constant Field Values

◊ STATUS_CREATED

```
static final int STATUS_CREATED
```

Lifecycle status: Query created on the server and ready to be used.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_CREATING` `onQueryCreate()` ok `STATUS_CREATED`.
Status Activities:
- the query may be eventually result−set partitioned,
- the query is used by `onQueryNotify()` to retrieve the result−set one row at time.

Status Exit:
`STATUS_CREATED` `destroy()` ok `STATUS_DESTROYING`.
See Also:
Query lifecycle, Constant Field Values

◊ STATUS_DESTROYING

```
static final int STATUS_DESTROYING
```

Lifecycle status: Query waiting the `destroy()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_CREATED` `destroy()` ok `STATUS_DESTROYING`.
Status Activities:
none: waiting an automatic `onQueryDestroy()` call.
Status Exit:
`STATUS_DESTROYING` `onQueryDestroy()` `STATUS_DESTROYED`.
See Also:
Query lifecycle, Constant Field Values

◊ STATUS_DESTROYED

```
static final int STATUS_DESTROYED
```

Lifecycle status: Query destroyed into the server and ready to be released.
This value may be returned by `LifeCycle.getStatus()`.

It's always a good practice to release a Query in this status.

Status Entry:
`STATUS_DESTROYING` `onQueryDestroy()` `STATUS_DESTROYED`.
Status Activities:

`LifeCycle.release()`.

Status Exit:

`STATUS_DESTROYED` `LifeCycle.release()` `STATUS_RELEASED`.

See Also:

Query lifecycle, Constant Field Values

Method Detail

create

```
int create()
```

Try to create this query on the server.

This method must be called only when

- the current status is `STATUS_INIT`,
- the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

- the create request was sent to server,
- the current status changed to `STATUS_CREATING`,
- when the server−answer will be available the `QueryListener.onQueryCreate()` will be automatically called to handle it.

otherwise

- the client has rejected the create,
- the create request was **not** sent to the server,
- automatic call of `QueryListener.onQueryCreate()` will **not** be made,
- the current status remains unchanged.

In the latter case it is a good practice to release this Query.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_INIT`,
- `ActivityLifeCycle.RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

queryRows

```
int queryRows(int firstRow,
              int rowNumber)
```

Try to retrieve a subset of the result−set of this query from the server.

The `firstRow` and `rowNumber` parameters describe the subset of result−set to be retrieved.

This method must be called only when

· the current status is STATUS_CREATED,
· the associated Connection current status is Connection.STATUS_CONNECTED.
If this method invocation completed successfully,
then

· the subset request was sent to server,
· when the server−answer will be available the QueryListener.onQueryRows() will be automatically called to handle it.
otherwise

· the client has rejected the subset request,
· the subset request was **not** sent to the server,
· automatic call of QueryListener.onQueryRows() will **not** be made.
In any case the current status remains unchanged.

Parameters:
`firstRow` − index (1−based) of the first row to be retrieved.
`rowNumber` − number of rows to be retrieved.
Returns:
- RESULT_OK if the operation completed successfully,
- RESULT_INVALID_STATUS if the current status is not STATUS_CREATED,
- RESULT_INVALID_CONNECTION_STATUS if the associated Connection current status is not Connection.STATUS_CONNECTED,
- RESULT_GENERIC_ERROR otherwise (e.g. `firstRow` <= 0 or `rowNumber <= 0`).

◊ destroy

```
int destroy()
```

Try to destroy this query on the server.

This method must be called only when

· the current status is STATUS_CREATED,
· the associated Connection current status is Connection.STATUS_CONNECTED.
If this method invocation completed successfully,
then

· the destroy request was sent to server,
· the current status changed to STATUS_DESTROYING,
· when the server−answer will be available the QueryListener.onQueryDestroy() will be automatically called to handle it.
otherwise

· the client has rejected the destroy,
· the destroy request was **not** sent to the server,
· automatic call of QueryListener.onQueryDestroy() will **not** be made,
· the current status remains unchanged.
It's not a bad practice to unconditionally release this Query immediately after this method invocation without handling the returned value.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_CREATED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Subscription

- *All Superinterfaces:*
  ActivityLifeCycle, CommunicationLifeCycle, LifeCycle

```
public interface Subscription
extends ActivityLifeCycle
```

An arrangement with the server for receiving a continuing set of interesting entities of the same EntityClass.

Opening a subscription on a `EntityClass` of a Server entails both the initial acquisition of the `Entity`s of the EntityClass of the Server and the subsequent notification of any additions or cancellations executed by the Server on that EntityClass.

Special subscription modalities enable the acquisition of just a subset of the Entities of a EntityClass of the Server: see the section below.

Subscription Usage

A subscription is described by:

- an EntityClass on which the Subscription is made,
- a couple (EntityClass Version, EntityClass TimeStamp) that refers the last past notification received by the client (see Incremental Subscriptions below),
- a query selection criteria used by the server to chose whose Entities must be received:
  - ◊ all entities, both past and current values,
  - ◊ all entities that match a partial EntityKey, both past and current values,
  - ◊ all entities, but only past values,
  - ◊ all entities, but only current values,
- an optional EntityKey that describe the partial EntityKey to be used in case of the query selection criteria indicates a partial EntityKey (see Partial Subscriptions below),
- an optional filter to restrict (at the server level) the set of entities that will be notified,
- a data transimission policy used by the server to eventually adapt the sending server speed with the reception client speed,
- an optional mask to restrict (at the server level) the set of fields of entities that will be notified.

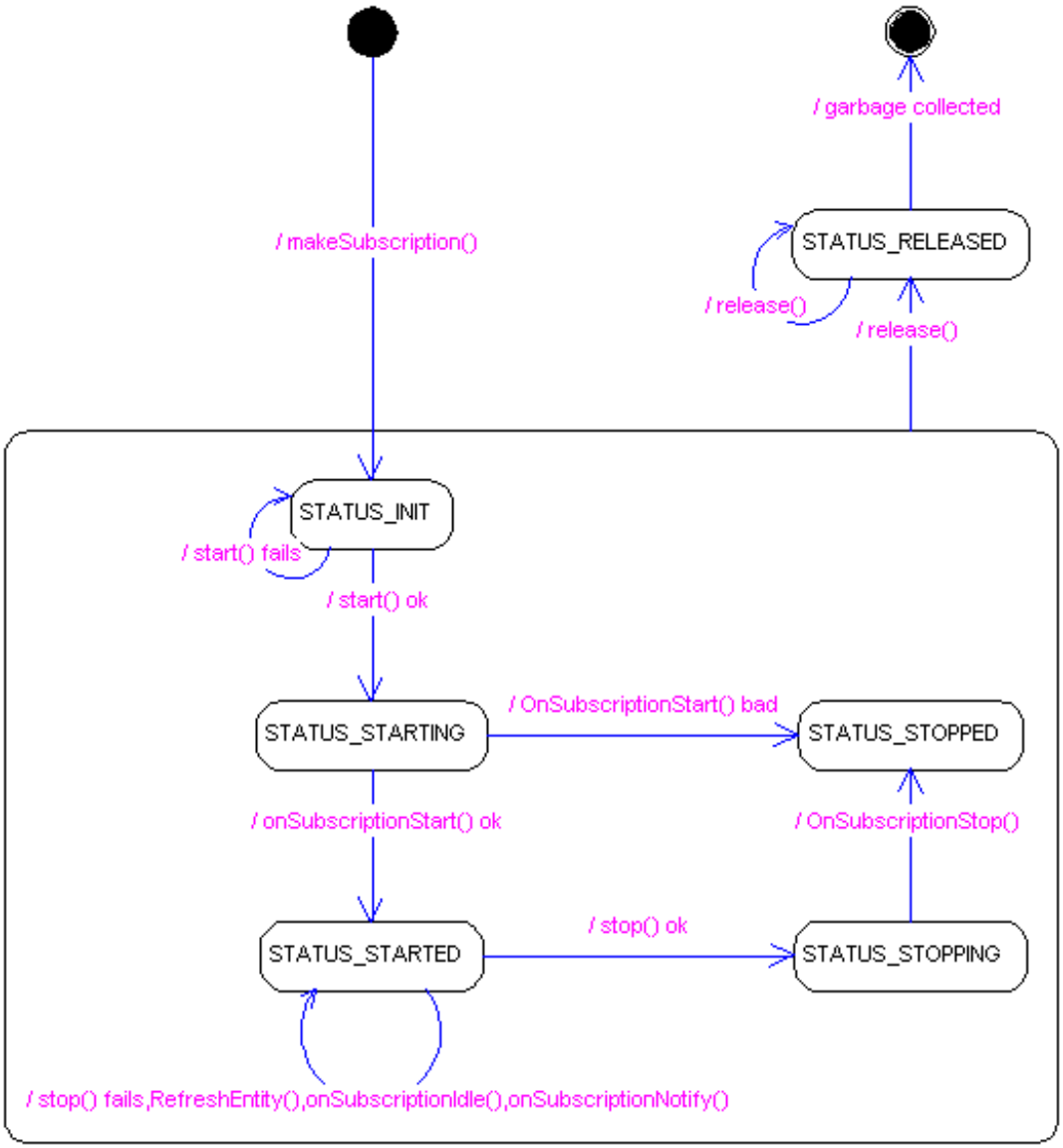
In brief:

Subscriptions are created by `Context.makeSubscription()` in which the subscription parameters are described by `SubscriptionParam` and the event listeners are described by

`SubscriptionListener`.

Once created a subscription must be started with the server, used (eventually via refreshEntity) and then stopped from the server.

Subscription Lifecycle



Incremental Subscriptions

This section outlines how to manage incremental subscriptions, in which the server is only required to send updated contents of an EntityClass, rather than sending all its records.

A client application can consequently keep a local data base aligned with the market server's, or more generally avoid processing data twice, by minimizing, at the same time, the volume of data to transfer and the needed time.

In particular, the subscription allows you to specify the pair of values (EntityClass Version, EntityClass TimeStamp) i.e., respectively, the last version index of the EntityClass and the time−stamp of the last Entity (record) received of that class during the previous subscription.

Thus, if the version of the EntityClass maintained in the server coincides with the one supplied, only those entities with a time stamp that is later than the one supplied will be sent.

On the other hand if the version of the class is earlier than the server's current one, then `SubscriptionStartEvent.isEntityClassReset()` will be `true` . This indicates a general invalidation of any entities from that EntityClass that have been archived until that moment. In the latter case it will not be possible to proceed to an incremental acquisition, but all the entities in the EntityClass will have to be received.

In order to be able to make this data available, the client application has to maintain for each EntityCass the following information:

- Current time stamp,
- Current version.

The current time stamp can be maintained by updating local `TimeStamp` data whenever ACTION_ENTITY_ADD, ACTION_ENTITY_RWT or ACTION_ENTITY_DEL operations are made, on the basis of the `SubscriptionNotifyEvent.getTimeStamp()` returned value.

The current version can be maintained by acquiring the value of the version at the opening of the subscription by `SubscriptionStartEvent.getEntityClassVersionOnServer()`, and updating it on every version variation of the server class, this basically means at each notification of an ACTION_ENTITY_KIL operation, by setting it to the `SubscriptionNotifyEvent.getTimeStamp()`.getDateTime() value.

Remarks

ACTION_ENTITY_DEL and ACTION_ENTITY_KIL operations with `SubscriptionNotifyEvent.getKeyID()` > 0, should be intended as notifications of (logical or physical} cancellation of an individual entity at a server level. In this case the Entity returned by `SubscriptionNotifyEvent.getEntity()` is generally undefined on any fields apart from those associated with SubscriptionNotifyEvent.getKeyID().

Operations of ACTION_ENTITY_KIL with SubscriptionNotifyEvent.getKeyID() `<= 0` should be intended as the physical cancellation of every entity in the specified EntityClass that has been acquired beforehand. In this case: `SubscriptionNotifyEvent.getEntity() == null`.

Partial Subscriptions

This section outlines how to manage partial subscriptions, in which the server is requested to send only those entities in a EntityClass that satisfy certain constraints.

Partial subscriptions can be formulated in two manners:

- by setting the parameter `SubscriptionParam.getQueryType()` to the value `SubscriptionParam.QUERY_TYPE_SET` and setting the parameter `SubscriptionParam.getEntityKey()` to an appropriate EntityKey (an Entitykey where the number of set segments is <= number of segments of the given KeyID),
- or by setting the parameter `SubscriptionParam.getFilter()` to a non-`null` value representing an appropriate Filter in `Filter.STATUS_CREATED` status.

In the first case the server will send only Entities whose complete EntityKey match the given partial EntityKey.

In the latter case the server will send only Entities that satisfy the given filter.

Both manners cannot coexist.

See Also:
`Context.makeSubscription()`, `SubscriptionParam`, `SubscriptionListener`

- ♦
Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`STATUS_STARTED`*** Lifecycle status: Subscription started and ready to be used.
`static int`	***`STATUS_STARTING`*** Lifecycle status: Subscription waiting the `start()` server−answer.
`static int`	***`STATUS_STOPPED`*** Lifecycle status: Subscription stopped with the server and ready to be released.
`static int`	***`STATUS_STOPPING`*** Lifecycle status: Subscription waiting the `stop()` server−answer.

◊
Fields inherited from interface ActivityLifeCycle

`RESULT_INVALID_CONNECTION_STATUS`

◊
Fields inherited from interface LifeCycle

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

♦
Method Summary

Methods

Modifier and Type	Method and Description
`int`	***`refreshEntity`***`(EntityKey entityKey)` Request the server to re−publish a single complete (not masked) Entity.
`int`	***`start`***`()` Try to start this subscription with the server.
`int`	***`stop`***`()` Try to stop this subscription with the server.

◊
Methods inherited from interface ActivityLifeCycle

`getConnection`

◊ Methods inherited from interface CommunicationLifeCycle

`getContext`, `getListener`, `getParam`

◊ Methods inherited from interface LifeCycle

`enumChilds`, `getStatus`, `release`

- ♦

Field Detail

◊ STATUS_STARTING

```
static final int STATUS_STARTING
```

Lifecycle status: Subscription waiting the `start()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_INIT` `start()` ok `STATUS_STARTING`.
Status Activities:
none: waiting an automatic `onSubscriptionStart()` call.
Status Exit:
`STATUS_STARTING` `onSubscriptionStart()` ok `STATUS_STARTED`.
`STATUS_STARTING` `onSubscriptionStart()` bad `STATUS_STOPPED`.
See Also:
Subscription lifecycle, Constant Field Values

◊ STATUS_STARTED

```
static final int STATUS_STARTED
```

Lifecycle status: Subscription started and ready to be used.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_STARTING` `onSubscriptionStart()` ok `STATUS_STARTED`.
Status Activities:
- ♦ the subscription is used by `SubscriptionListener.onSubscriptionNotify()` to get all interesting entities,
- ♦ the `refreshEntity()` may be eventually used.

Status Exit:
`STATUS_STARTED` `stop()` ok `STATUS_STOPPING`.
See Also:
Subscription lifecycle, Constant Field Values

◊ STATUS_STOPPING

```
static final int STATUS_STOPPING
```

Lifecycle status: Subscription waiting the `stop()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_STARTED` `stop()` ok `STATUS_STOPPING`.
Status Activities:
none: waiting an automatic `onSubscriptionStop()` call.
Status Exit:
`STATUS_STOPPING` `onSubscriptionStop()` `STATUS_STOPPED`.
See Also:
Subscription lifecycle, Constant Field Values

◊ STATUS_STOPPED

```
static final int STATUS_STOPPED
```

Lifecycle status: Subscription stopped with the server and ready to be released.
This value may be returned by `LifeCycle.getStatus()`.

It's always a good practice to release a Subscription in this status.

Status Entry:
`STATUS_STOPPING` `onSubscriptionStop()` `STATUS_STOPPED`.
Status Activities:
`LifeCycle.release()`.
Status Exit:
`STATUS_STOPPED` `LifeCycle.release()` `STATUS_RELEASED`.
See Also:
Subscription lifecycle, Constant Field Values

♦ Method Detail

◊ start

```
int start()
```

Try to start this subscription with the server.

This method must be called only when

· the current status is `STATUS_INIT`,
· the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

· the start request was sent to server,
· the current status changed to `STATUS_STARTING`,
· when the server−answer will be available the `SunscriptionListener.onSubscriptionStart()` will be automatically called to handle it.

otherwise

· the client has rejected the start,
· the start request was **not** sent to the server,
· automatic call of `SubscriptionListener.onSubscriptionStart()` will

not be made,
- the current status remains unchanged.

In the latter case it is a good practice to release this Subscription.

Returns:
- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_INIT`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

◊ refreshEntity

```
int refreshEntity(EntityKey entityKey)
```

Request the server to re−publish a single complete (not masked) Entity.

This method is useful with masked subscriptions to retrieve all fields of an interesting entity.

This method must be called only when

- the current status is `STATUS_STARTED`,
- the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

- the re−publish request was sent to server,
- when the server−answer will be available (**please note** that a server−answer will be returned if and only if the requested Entity exists in the server, otherwise there will not be any failure indication of any sort!) the `SubscriptionListener.onSubscriptionNotify()` will be automatically called to handle it and in this case all fields of `SubscriptionNotifyEvent.getEntity()` will be available, even if this subscription is masked: the `SubscriptionNotifyEvent.isMasked()` method may be used to discriminate between maske and not−masked entities.

otherwise

- the client has rejected the re−publish request,
- the re−publish request was **not** sent to the server,
- automatic call of `SubscriptionListener.onSubscriptionNotify()` will **not** be made.

In any case the current status remains unchanged.

Parameters:
`entityKey` − the entitykey that identifies the requested entity.
Returns:
- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_STARTED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise (e.g. the `entityKey` parameter is `null` or it refers an EntityClass different from the subscribed class).

◊ stop

```
int stop()
```

Try to stop this subscription with the server.

This method must be called only when

- the current status is `STATUS_STARTED`,
- the associated Connection current status is `Connection.STATUS_CONNECTED`.

If this method invocation completed successfully,
then

- the stop request was sent to server,
- the current status changed to `STATUS_STOPPING`,
- when the server−answer will be available the `SubscriptionListener.onSubscriptionStop()` will be automatically called to handle it.

otherwise

- the client has rejected the stop,
- the stop request was **not** sent to the server,
- automatic call of `SubscriptionListener.onSubscriptionStop()` will **not** be made,
- the current status remains unchanged.

It's not a bad practice to unconditionally release this Subscription immediately after this method invocation without handling the returned value.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_STARTED`,
- `RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Transaction

- *All Superinterfaces:*
  ActivityLifeCycle, CommunicationLifeCycle, LifeCycle

```
public interface Transaction
extends ActivityLifeCycle
```

A client's request to the server to add, remove or modify an entity in its own Data Base.

Transaction Usage

A new transaction (i.e. a new client's request to the server) is defined by:

- the requested action: add or modify or logically remove or physically remove,
- the entity on which the action will be done,
- a KeyID of the EntityClass of that Entity
  (all the KeyID fields of the Entity must be properly filled),
- an optional mask that describes which fields of the Entity must be filled,
- an eventual request to obtain an Entity inside the result that will be sent from the server.

Once the transaction is properly sent to the server:

- its TransactionID may be retrieved and saved for future use,
- its status may be queried using the `query()` method.

The TransactionID of a sent transaction may be saved, i.e. the 5 `int` values that represents the TransactionID may be properly saved (see `TransactionID` for details). From these saved values it's possibile to recreate the original TransactionID and then create a Transaction object, that represents this past transaction, using the following two values:

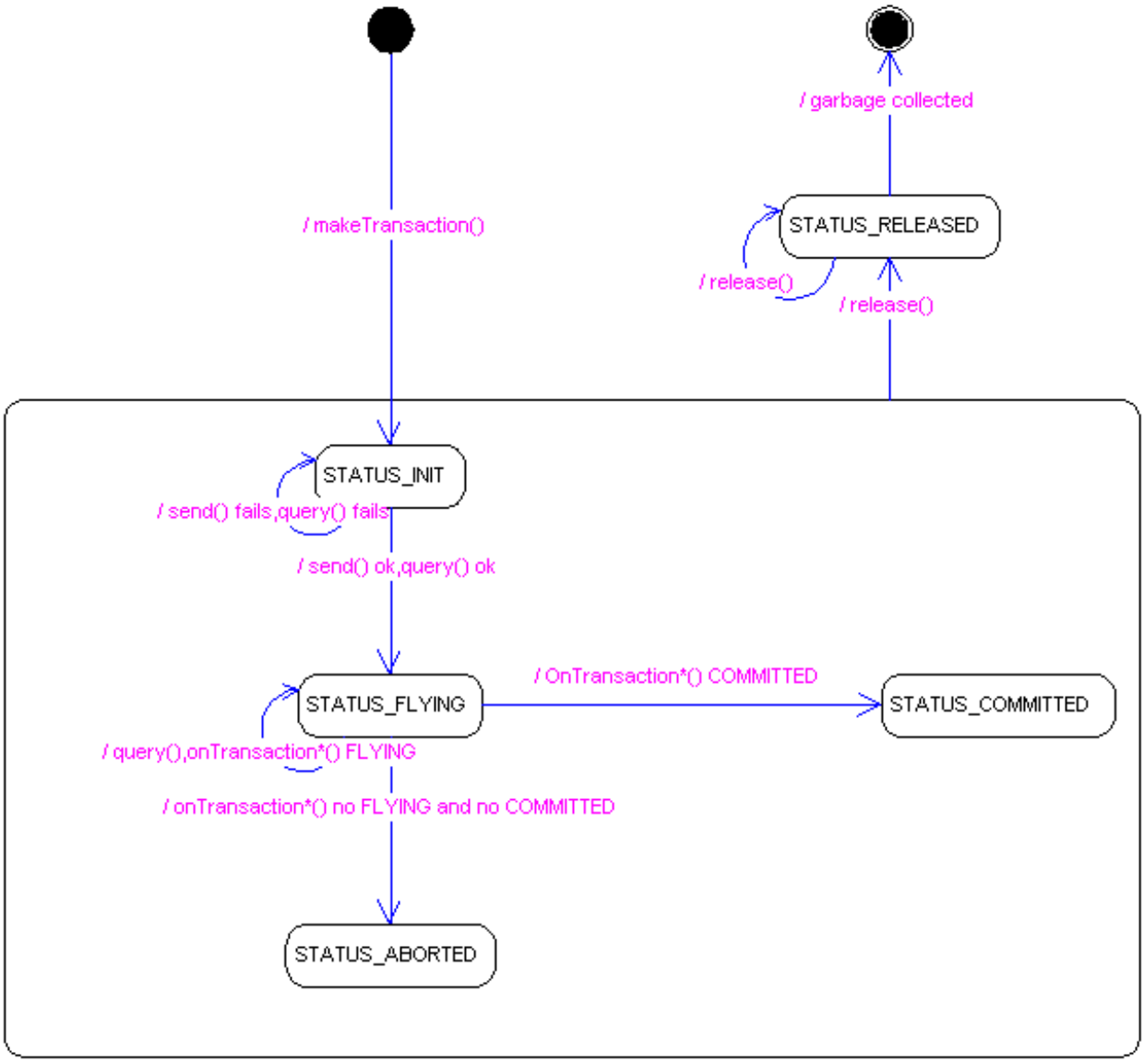
- this past pending TransactionID,
- an eventual request to obtain an Entity inside the query result that will be sent from the server.

In this manner this created Transaction object may be used to query the status of this past Transaction to the server.

In brief:

> Transactions are locally created by `Context.makeTransaction()` in which the transaction parameters are described by `TransactionParam` and the event listeners are described by `TransactionListener`.
>
> Once locally created a transaction may be sent to the server, and/or its status may be queried to the server.
> It is also possibile to query the status of a past pending transaction if the programmer created a new Transaction with a saved TransactionID of a sent transaction.

Transaction Lifecycle



See Also:

`Context.makeTransaction()`, `TransactionParam`, `TransactionListener`

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`STATUS_ABORTED`*** Lifecycle and Transaction status: Transaction completed unsuccessfully and ready to be released.
`static int`	***`STATUS_COMMITTED`*** Lifecycle and Transaction status: Transaction completed successfully and ready to be released.
`static int`	***`STATUS_FLYING`***

Lifecycle and Transaction status: Transaction is flying waiting to become STATUS_COMMITTED or STATUS_ABORTED.

◊ **Fields inherited from interface ActivityLifeCycle**

RESULT_INVALID_CONNECTION_STATUS

◊ **Fields inherited from interface LifeCycle**

RESULT_GENERIC_ERROR, RESULT_INVALID_STATUS, RESULT_OK, STATUS_INIT, STATUS_RELEASED

Method Summary

Methods

Modifier and Type	Method and Description
TransactionID	***getTransactionID***() Returns the TransactionID of this (new or past) transaction.
int	***query***() Try to query the server for the status of this (new or past) transaction.
int	***send***() Try to send this new transaction to the server.

◊ **Methods inherited from interface ActivityLifeCycle**

getConnection

◊ **Methods inherited from interface CommunicationLifeCycle**

getContext, getListener, getParam

◊ **Methods inherited from interface LifeCycle**

enumChilds, getStatus, release

Field Detail

◊ **STATUS_FLYING**

```
static final int STATUS_FLYING
```

Lifecycle and Transaction status: Transaction is flying waiting to become STATUS_COMMITTED or STATUS_ABORTED.
This value may be returned by LifeCycle.getStatus().

Status Entry:

STATUS_INIT send() ok STATUS_FLYING.
STATUS_INIT query() ok STATUS_FLYING.

Status Activities:

query(),
or waiting an automatic onTransactionSend() call,
or waiting an automatic onTransactionQuery() call.

Status Exit:

STATUS_FLYING onTransactionSend() bad STATUS_ABORTED.
STATUS_FLYING onTransactionQuery() RESULT_COMMITTED STATUS_COMMITTED.
STATUS_FLYING onTransactionQuery() RESULT_INVALID_TRANSACTION_ID STATUS_ABORTED.
STATUS_FLYING onTransactionQuery() RESULT_GENERIC_ERROR STATUS_ABORTED.
STATUS_FLYING onTransactionQuery() RESULT_ABORTED STATUS_ABORTED.

See Also:

Transaction lifecycle, Constant Field Values

◊ STATUS_COMMITTED

```
static final int STATUS_COMMITTED
```

Lifecycle and Transaction status: Transaction completed successfully and ready to be released.
This value may be returned by LifeCycle.getStatus().

It's always a good practice to release a Transaction in this status.

Status Entry:

STATUS_FLYING onTransactionQuery() RESULT_COMMITTED STATUS_COMMITTED.

Status Activities:

LifeCycle.release().

Status Exit:

STATUS_COMMITTED LifeCycle.release() STATUS_RELEASED.

See Also:

Transaction lifecycle, Constant Field Values

◊ STATUS_ABORTED

```
static final int STATUS_ABORTED
```

Lifecycle and Transaction status: Transaction completed unsuccessfully and ready to be released.
This value may be returned by LifeCycle.getStatus().

It's always a good practice to release a Transaction in this status.

Status Entry:

STATUS_FLYING onTransactionSend() bad STATUS_ABORTED.
STATUS_FLYING onTransactionQuery() RESULT_INVALID_TRANSACTION_ID STATUS_ABORTED.
STATUS_FLYING onTransactionQuery() RESULT_GENERIC_ERROR STATUS_ABORTED.

STATUS_FLYING onTransactionQuery() RESULT_ABORTED STATUS_ABORTED.

Status Activities:

LifeCycle.release().

Status Exit:

STATUS_ABORTED LifeCycle.release() STATUS_RELEASED.

See Also:

Transaction lifecycle, Constant Field Values

♦ Method Detail

◊ send

```
int send()
```

Try to send this new transaction to the server.

This method must be called only when

- · the current status is STATUS_INIT,
- · the associated Connection current status is Connection.STATUS_CONNECTED,
- · this is a new transaction (i.e. the past TransactionID is null).

If this method invocation completed successfully,
then

- · the transaction was sent to server,
- · the current status changed to STATUS_FLYING,
- · when the server−answer will be available the TransactionListener.onTransactionSend() will be automatically called to handle it.

otherwise

- · the client has rejected the send,
- · the transaction was **not** sent to the server,
- · automatic call of TransactionListener.onTransactionSend() will **not** be made,
- · the current status remains unchanged.

In the latter case it is a good practice to release this Transaction.

Returns:

- ♦ RESULT_OK if the operation completed successfully,
- ♦ RESULT_INVALID_STATUS if the current status is not STATUS_INIT,
- ♦ ActivityLifeCycle.RESULT_INVALID_CONNECTION_STATUS if the associated Connection current status is not Connection.STATUS_CONNECTED,
- ♦ RESULT_GENERIC_ERROR otherwise (e.g. the Entity is null).

◊ query

```
int query()
```

Try to query the server for the status of this (new or past) transaction.

This method must be called only when

- the current status is `STATUS_INIT` or `STATUS_FLYING`,
- the associated Connection current status is `Connection.STATUS_CONNECTED`,
- this transaction has a valid TransactionID (`getTransactionID()` is not `null`).

If this method invocation completed successfully,
then

- the query was sent to server,
- the current status changed to `STATUS_FLYING`,
- when the server−answer will be available the `TransactionListener.onTransactionQuery()` will be automatically called to handle it.

otherwise

- the client has rejected the query,
- the query was **not** sent to the server,
- automatic call of `TransactionListener.onTransactionQuery()` will **not** be made,
- the current status remains unchanged.

In the latter case it is a good practice to release this Transaction.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `RESULT_INVALID_STATUS` if the current status is not `STATUS_INIT` nor `STATUS_FLYING`,
- `ActivityLifeCycle.RESULT_INVALID_CONNECTION_STATUS` if the associated Connection current status is not `Connection.STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise (e.g. `getTransactionID()` is `null`).

◊ getTransactionID

```
TransactionID getTransactionID()
```

Returns the TransactionID of this (new or past) transaction.

If there is a valid `past TransactionID` then this method returns it, otherwise it returns the new TransactionID that will be used by the `send()` method.

Returns:

the TransactionID of this (new or past) transaction.
`null` is returned when the `past TransactionID` exists but it does not `belongs to` the associated connection.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Connection

- *All Superinterfaces:*
  CommunicationLifeCycle, LifeCycle

```
public interface Connection
extends CommunicationLifeCycle
```

Logical bidirectional channel with a server.

Within the same JFT application several Connections can be created and then opened with the same Server or with several Servers.

Connection Usage

Every new connection is described by:

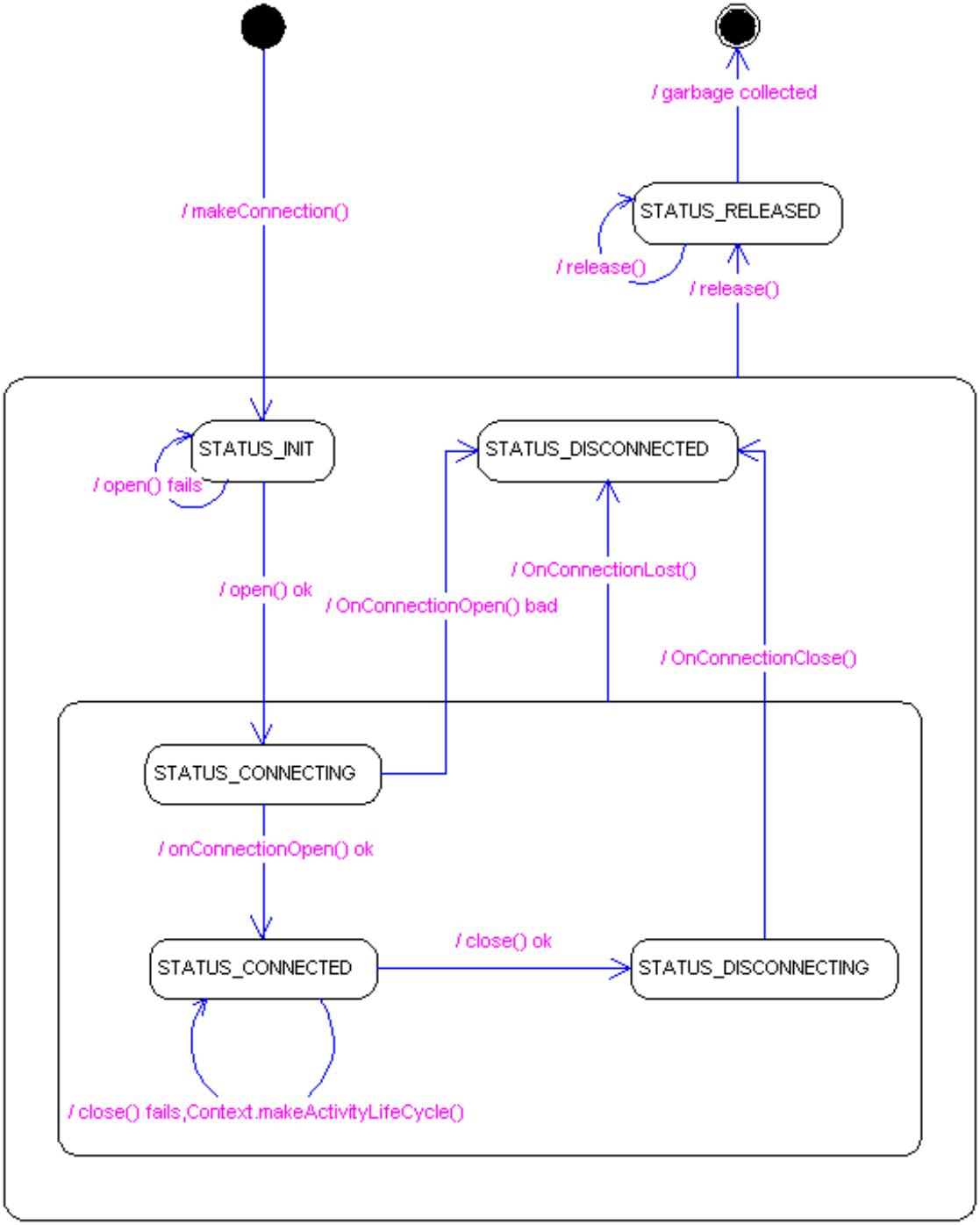
- a pair remote host, remote port) that define the principal server to which the client must talk,
- an optional pair remote alternative host, remote alternative port) that define an optional server server to which the client must talk in the case the connection to the principal server failed,
- the transport−type to be used,
- an optional pair proxy host, proxy port) that describe the proxy to be used when the transport−type is not `TCP/IP`,
- an indication about a compressed transmission between client and server,
- a `charset` used to code/decode the strings over the line,
- an optional market/service name to which the client must talk,
- an indication of the type of activities that will be done on the connection,
- a pair (user−name, password) that describe the user associated to the connection,
- a ClientID that identifies the client,
- a client application revision,
- a client application signature,
- an authorization file or an authorization key that allow a client to open and use a connection with a server.

In brief:

Connections are created by `Context.makeConnection()` in which the connection parameters are described by `ConnectionParam` and the event listeners are described by `ConnectionListener`.

Once created a connection must be opened (attached to the server), used, and then closed (detached from the server).

Connection Lifecycle



See Also:

`Context.makeConnection()`, `ConnectionParam`, `ConnectionListener`

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`STATUS_CONNECTED`***

	Lifecycle status: Connection connected to the server and ready to be used.
static int	***STATUS_CONNECTING*** Lifecycle status: Connection waiting the `open()` server−answer.
static int	***STATUS_DISCONNECTED*** Lifecycle status: Connection no more connected to the server and ready to be released.
static int	***STATUS_DISCONNECTING*** Lifecycle status: Connection waiting the `close()` server−answer.

◊

Fields inherited from interface LifeCycle

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

Method Summary

Methods

Modifier and Type	Method and Description
int	***close***() Try to close this connection with a given server.
int	***open***() Try to open this connection with a given server.

◊

Methods inherited from interface CommunicationLifeCycle

`getContext`, `getListener`, `getParam`

◊

Methods inherited from interface LifeCycle

`enumChilds`, `getStatus`, `release`

Field Detail

◊ **STATUS_CONNECTING**

```
static final int STATUS_CONNECTING
```

Lifecycle status: Connection waiting the `open()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:
`STATUS_INIT` `open()` ok `STATUS_CONNECTING`.

Status Activities:

none: waiting an automatic `onConnectionOpen()` call.

Status Exit:

`STATUS_CONNECTING` `onConnectionOpen()` ok `STATUS_CONNECTED`.
`STATUS_CONNECTING` `onConnectionOpen()` bad `STATUS_DISCONNECTED`.
`STATUS_CONNECTING` `onConnectionLost()` `STATUS_DISCONNECTED`.

See Also:

Connection lifecycle, Constant Field Values

◊ STATUS_CONNECTED

```
static final int STATUS_CONNECTED
```

Lifecycle status: Connection connected to the server and ready to be used.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:

`STATUS_CONNECTING` `onConnectionOpen()` ok `STATUS_CONNECTED`.

Status Activities:

- the connection may be used to create `ActivityLifeCycle` objects (with `Context` methods `makeSubscription()`, `makeQuery()`, `makeTransanction()` and `makeFilter()`),
- the connection may be used in `TransactionID.belongsTo()`,
- `ActivityLifeCycle` objects may be used (e.g. `Subscription.start()`, `Query.create()`, etc...).

Status Exit:

`STATUS_CONNECTED` `close()` ok `STATUS_DISCONNECTING`.
`STATUS_CONNECTED` `onConnectionLost()` `STATUS_DISCONNECTED`.

See Also:

Connection lifecycle, Constant Field Values

◊ STATUS_DISCONNECTING

```
static final int STATUS_DISCONNECTING
```

Lifecycle status: Connection waiting the `close()` server−answer.
This value may be returned by `LifeCycle.getStatus()`.

Status Entry:

`STATUS_CONNECTED` `close()` ok `STATUS_DISCONNECTING`.

Status Activities:

none: waiting an automatic `onConnectionClose()` call.

Status Exit:

`STATUS_DISCONNECTING` `onConnectionClose()` `STATUS_DISCONNECTED`.
`STATUS_DISCONNECTING` `onConnectionLost()` `STATUS_DISCONNECTED`.

See Also:

Connection lifecycle, Constant Field Values

◊ STATUS_DISCONNECTED

```
static final int STATUS_DISCONNECTED
```

Lifecycle status: Connection no more connected to the server and ready to be released.
This value may be returned by `LifeCycle.getStatus()`.

It's always a good practice to release a Connection in this status.

Status Entry:
`STATUS_DISCONNECTING` `onConnectionClose()`
`STATUS_DISCONNECTED`.
any status `onConnectionLost()` `STATUS_DISCONNECTED`.
Status Activities:
`LifeCycle.release()`.
Status Exit:
`STATUS_DISCONNECTED` `LifeCycle.release()` `STATUS_RELEASED`.
See Also:
Connection lifecycle, Constant Field Values

♦ Method Detail

◊ open

```
int open()
```

Try to open this connection with a given server.

This method must be called only when current status is `STATUS_INIT`.

If this method invocation completed successfully,
then

- the open request was sent to server,
- the current status changed to `STATUS_CONNECTING`,
- when the server−answer will be available the `ConnectionListener.onConnectionOpen()` will be automatically called to handle it.

otherwise

- the client has rejected the open,
- the open request was **not** sent to the server,
- automatic call of `ConnectionListener.onConnectionOpen()` will **not** be made,
- the current status remains unchanged.

In the latter case it is a good practice to release this Connection.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_INVALID_STATUS` if the current status is not `STATUS_INIT`,
- `RESULT_GENERIC_ERROR` otherwise
  (e.g. some server name (as host or the optionals alternative host and proxy host) is unresolvable).

◊ close

```
int close()
```

Try to close this connection with a given server.

This method must be called only when current status is `STATUS_CONNECTED`.

If this method invocation completed successfully,
then

- the close request was sent to server,
- the current status changed to `STATUS_DISCONNECTING`,
- when the server−answer will be available the `ConnectionListener.onConnectionClose()` will be automatically called to handle it.

otherwise

- the client has rejected the close,
- the close request was **not** sent to the server,
- automatic call of `ConnectionListener.onConnectionClose()` will **not** be made,
- the current status remains unchanged.

It's not a bad practice to unconditionally release this Connection immediately after this method invocation without handling the returned value.

Returns:

- `RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_INVALID_STATUS` if the current status is not `STATUS_CONNECTED`,
- `RESULT_GENERIC_ERROR` otherwise.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface MulticastConnection

- *All Superinterfaces:*
  CommunicationLifeCycle, LifeCycle

```
public interface MulticastConnection
extends CommunicationLifeCycle
```

- ♦

Field Summary

Fields

Modifier and Type	Field and Description

static int **STATUS_CONNECTED**

static int **STATUS_DISCONNECTED**

- ◊ **Fields inherited from interface LifeCycle**

  RESULT_GENERIC_ERROR, RESULT_INVALID_STATUS, RESULT_OK, STATUS_INIT, STATUS_RELEASED

♦ Method Summary

Methods

Modifier and Type	Method and Description
int	**close**()
int	**enableNotify**(int entityClassID)
int	**open**()

- ◊ **Methods inherited from interface CommunicationLifeCycle**

  getContext, getListener, getParam

- ◊ **Methods inherited from interface LifeCycle**

  enumChilds, getStatus, release

- ♦ **Field Detail**

  - ◊ **STATUS_CONNECTED**

    static final int STATUS_CONNECTED

    See Also:
    Constant Field Values

  - ◊ **STATUS_DISCONNECTED**

    static final int STATUS_DISCONNECTED

    See Also:
    Constant Field Values

♦ Method Detail

◊ **open**

```
int open()
```

◊ **close**

```
int close()
```

◊ **enableNotify**

```
int enableNotify(int entityClassID)
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Context

- *All Superinterfaces:*
  LifeCycle

```
public interface Context
extends LifeCycle
```

Container and factory of inter−related communication objects.

Within the same JFT application several Contexts can be created and then used to interact with one or more FastTrack servers.

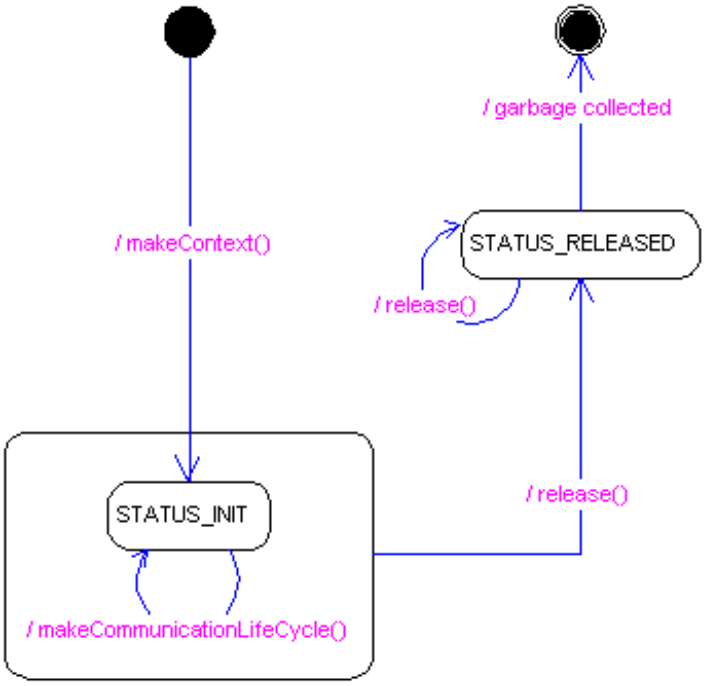
This interface contains methods to create CommunicationLifeCycle objects: i.e. Connections and their related ActivityLifeCycle objects:

- `Filter`,
- `Query`,
- `Subscription`,
- `Transaction`.

Methods to create their parameters are provided as well.

Please note that each Connection and its related ActivityLifeCycle objects are enforced to exist in the same Context: e.g. it's not possible to create a subscription in a Context that does not contains the associated Connection.

Context Lifecycle



See Also:

`JFT.makeContext()`, Acceptable Values

- ♦

Field Summary

- ◊

Fields inherited from interface LifeCycle

`RESULT_GENERIC_ERROR`, `RESULT_INVALID_STATUS`, `RESULT_OK`, `STATUS_INIT`, `STATUS_RELEASED`

- ♦

Method Summary

Methods

Modifier and Type	Method and Description
`EntityClass`	***`getEntityClass`***`(int EntityClassID)` Returns the registered EntityClass corresponding to the given EntityClassID, if registered in this context.
`EntityClass`	***`getEntityClass`***`(String EntityClassName)` Returns the registered EntityClass in this context corresponding to the given EntityClass Name.
`boolean`	***`isRegistered`***`(int EntityClassID)` Returns the indication that a given EntityClass (identified by an `EntityClassID`) has been registered in this context.

Connection	***makeConnection***(ConnectionParam param, ConnectionListener listener) Create and returns a new connection.
ConnectionParam	***makeConnectionParam***() Create and returns a new connection parameter container.
Mask	***makeEmptyMask***(int entityClassID) Create and returns a new empty mask for a given Entity Class registered in this context.
EntityClassQuery	***makeEntityClassQuery***(Connection connection, EntityClassQueryParam param, EntityClassQueryListener listener)
EntityClassQueryParam	***makeEntityClassQueryParam***()
Filter	***makeFilter***(Connection connection, FilterParam param, FilterListener listener) Create and returns a new filter.
FilterParam	***makeFilterParam***() Create and returns a new filter parameter container.
MulticastConnection	***makeMulticastConnection***(MulticastConnectionParam MulticastConnectionListener listener)
MulticastConnectionParam	***makeMulticastConnectionParam***()
Query	***makeQuery***(Connection connection, QueryParam param, QueryListener listener) Create and returns a new query.
QueryParam	***makeQueryParam***() Create and returns a new query parameter container.
Subscription	***makeSubscription***(Connection connection, SubscriptionParam param, SubscriptionListener listener) Create and returns a new subscription.
SubscriptionParam	***makeSubscriptionParam***() Create and returns a new subscription parameter container.
Transaction	***makeTransaction***(Connection connection, TransactionParam param, TransactionListener listener) Create and returns a new transaction.
TransactionParam	***makeTransactionParam***() Create and returns a new transaction parameter container.
int	***register***(EntityClass entityClass) Like JFT.register, this method registers another EntityClass at the Context level that increases the number of the classes managed by the library.
int	***register***(EntityClass[] entityClass) Like register(it.list.jft.EntityClass), this

method register an array of EntityClass.

◊

Methods inherited from interface LifeCycle

`enumChilds`, `getStatus`, `release`

- ♦

Method Detail

◊ **register**

```
int register(EntityClass entityClass)
```

Like `JFT.register`, this method registers another EntityClass at the Context level that increases the number of the classes managed by the library.

The `EntityClass` registered at the Context level are visible only through the connections created from this Context. This could be useful when you need to create a connection and subscribe entities from another version of the same service. With this method you can override the entities registered with the `JFT.register` as it enables you to connect and receive different versions of the same entity. When an Entity is received, the API looks for a registered entity at the Context level and, if not found, looks for it in the entities registered with the `JFT.register`.

Parameters:

`entityClass` − EntityClass to be registered.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. some `entityClass` does not refer a valid EntityClass).

◊ **register**

```
int register(EntityClass[] entityClass)
```

Like `register(it.list.jft.EntityClass)`, this method register an array of EntityClass.

Parameters:

`entityClass` − Array of EntityClass to be registered.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. some `entityClass` does not refer a valid EntityClass).

◊ **isRegistered**

```
boolean isRegistered(int EntityClassID)
```

Returns the indication that a given EntityClass (identified by an `EntityClassID`) has been registered in this context.

Parameters:

`EntityClassID` − ID of the EntityClass to be checked.

Returns:
the indication that a given EntityClass (identified by an `EntityClassID`) has been registered.
`false` is returned when the current status is `LifeCycle.STATUS_INIT`.

◊ **getEntityClass**

```
EntityClass getEntityClass(int EntityClassID)
```

Returns the registered EntityClass corresponding to the given EntityClassID, if registered in this context.

Parameters:
`EntityClassID` − ID of the EntityClass to be retrieved.
Returns:
the registered EntityClass corresponding to the given EntityClassID.
`null` is returned when the current status is `LifeCycle.STATUS_INIT`, or when the given `EntityClassID` is not registered.

◊ **getEntityClass**

```
EntityClass getEntityClass(String EntityClassName)
```

Returns the registered EntityClass in this context corresponding to the given EntityClass Name.

Parameters:
`EntityClassName` − The name of the EntityClass to be retrieved.
Returns:
the registered EntityClass corresponding to the given EntityClass Name.
`null` is returned when the current status is `LifeCycle.STATUS_INIT`, or when the given `EntityClassID` is not registered.

◊ **makeEmptyMask**

```
Mask makeEmptyMask(int entityClassID)
```

Create and returns a new empty mask for a given Entity Class registered in this context.

A mask may be used in subscriptions (`SubscriptionParam.setMask()`) or transactions (`TransactionParam.setMask()`).

Parameters:
`entityClassID` − Entity Class ID of the market class.
Returns:
the new empty mask.
`null` is returned when the parameter `entityClassID` is wrong.

◊ **makeConnection**

```
Connection makeConnection(ConnectionParam param,
                          ConnectionListener listener)
  throws NullPointerException,
         IllegalArgumentException,
         IllegalStateException
```

Create and returns a new connection.

The current status of the returned connection is `STATUS_INIT`.

At the return of this method the given `ConnectionParam` parameter container is bound.

Parameters:
`param` − connection parameter container.
`listener` − connection listener.
Returns:
the new connection.
`null` is never returned.
Throws:
`NullPointerException` − if some parameter is `null`.
`IllegalArgumentException` − if `Param` contains un−acceptable values.
`IllegalStateException` − if the current status is not `STATUS_INIT`.
See Also:
Acceptable Values

◊ **makeConnectionParam**

```
ConnectionParam makeConnectionParam()
```

Create and returns a new connection parameter container.

Each parameter of the returned container has its value equal to default−value as described in the corresponding `ConnectionParam.getSomething` description.

Returns:
the new connection parameter container.
`null` is returned when the current status is not `STATUS_INIT`.

◊ **makeFilter**

```
Filter makeFilter(Connection connection,
                  FilterParam param,
                  FilterListener listener)
                    throws NullPointerException,
                           IllegalArgumentException,
                           IllegalStateException
```

Create and returns a new filter.

The current status of the returned filter is `STATUS_INIT`.

At the return of this method the given `FilterParam` parameter container is bound.

Parameters:
`connection` − associated connection.
`param` − filter parameter container.
`listener` − filter listener.
Returns:
the new filter.
`null` is never returned.
Throws:
`NullPointerException` − if some parameter is `null`.
`IllegalArgumentException` − if `Param` contains un−acceptable values,

or the given `connection` is not associated to this context.
`IllegalStateException` − if the current status is not `STATUS_INIT`,
or the status of the given `connection` is not
`Connection.STATUS_CONNECTED`.

See Also:

Acceptable Values

◊ makeFilterParam

```
FilterParam makeFilterParam()
```

Create and returns a new filter parameter container.

Each parameter of the returned container has its value equal to default−value as described in the corresponding `FilterParam.getSomething` description.

Returns:

the new filter parameter container.
`null` is returned when the current status is not `STATUS_INIT`.

◊ makeQuery

```
Query makeQuery(Connection connection,
                QueryParam param,
                QueryListener listener)
                  throws NullPointerException,
                         IllegalArgumentException,
                         IllegalStateException
```

Create and returns a new query.

The current status of the returned query is `STATUS_INIT`.

At the return of this method the given `QueryParam` parameter container is bound.

Parameters:

`connection` − associated connection.
`param` − query parameter container.
`listener` − query listener.

Returns:

the new query.
`null` is never returned.

Throws:

`NullPointerException` − if some parameter is `null`.
`IllegalArgumentException` − if `Param` contains un−acceptable values,
or the given `connection` is not associated to this context.
`IllegalStateException` − if the current status is not `STATUS_INIT`,
or the status of the given `connection` is not
`Connection.STATUS_CONNECTED`.

See Also:

Acceptable Values

◊ makeQueryParam

```
QueryParam makeQueryParam()
```

Create and returns a new query parameter container.

Each parameter of the returned container has its value equal to default−value as described in the corresponding `QueryParam.getSomething` description.

Returns:
the new query parameter container.
`null` is returned when the current status is not `STATUS_INIT`.

◊ makeSubscription

```
Subscription makeSubscription(Connection connection,
                              SubscriptionParam param,
                              SubscriptionListener listener)
                                throws NullPointerException,
                                       IllegalArgumentException,
                                       IllegalStateException
```

Create and returns a new subscription.

The current status of the returned subscription is `STATUS_INIT`.

At the return of this method the given `SubscriptionParam` parameter container is bound.

Parameters:
`connection` − associated connection.
`param` − subscription parameter container.
`listener` − subscription listener.
Returns:
the new subscription.
`null` is never returned.
Throws:
`NullPointerException` − if some parameter is `null`.
`IllegalArgumentException` − if `Param` contains un−acceptable values, or the given `connection` is not associated to this context.
`IllegalStateException` − if the current status is not `STATUS_INIT`, or the status of the given `connection` is not `Connection.STATUS_CONNECTED`.
See Also:
Acceptable Values

◊ makeSubscriptionParam

```
SubscriptionParam makeSubscriptionParam()
```

Create and returns a new subscription parameter container.

Each parameter of the returned container has its value equal to default−value as described in the corresponding `SubscriptionParam.getSomething` description.

Returns:

the new subscription parameter container.
`null` is returned when the current status is not `STATUS_INIT`.

◊ **makeTransaction**

```
Transaction makeTransaction(Connection connection,
                            TransactionParam param,
                            TransactionListener listener)
                              throws NullPointerException,
                                     IllegalArgumentException,
                                     IllegalStateException
```

Create and returns a new transaction.

The current status of the returned transaction is `STATUS_INIT`.

At the return of this method the given `TransactionParam` parameter container is bound.

Parameters:
`connection` − associated connection.
`param` − transaction parameter container.
`listener` − transaction listener.
Returns:
the new transaction.
`null` is never returned.
Throws:
`NullPointerException` − if some parameter is `null`.
`IllegalArgumentException` − if `Param` contains un−acceptable values, or the given `connection` is not associated to this context.
`IllegalStateException` − if the current status is not `STATUS_INIT`, or the status of the given `connection` is not `Connection.STATUS_CONNECTED`.
See Also:
Acceptable Values

◊ **makeTransactionParam**

```
TransactionParam makeTransactionParam()
```

Create and returns a new transaction parameter container.

Each parameter of the returned container has its value equal to default−value as described in the corresponding `TransactionParam.getSomething` description.

Returns:
the new transaction parameter container.
`null` is returned when the current status is not `STATUS_INIT`.

◊ **makeEntityClassQuery**

```
EntityClassQuery makeEntityClassQuery(Connection connection,
                                      EntityClassQueryParam param,
                                      EntityClassQueryListener listener)
                                        throws NullPointerException,
                                               IllegalArgumentException,
                                               IllegalStateException
```

Throws:
`NullPointerException`
`IllegalArgumentException`
`IllegalStateException`

◊ **makeEntityClassQueryParam**

```
EntityClassQueryParam makeEntityClassQueryParam()
```

◊ **makeMulticastConnection**

```
MulticastConnection makeMulticastConnection(MulticastConnectionParam param,
                                            MulticastConnectionListener listener)
                                              throws NullPointerException,
                                                     IllegalArgumentException,
                                                     IllegalStateException
```

Throws:
`NullPointerException`
`IllegalArgumentException`
`IllegalStateException`

◊ **makeMulticastConnectionParam**

```
MulticastConnectionParam makeMulticastConnectionParam()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface JFT

- *All Superinterfaces:*
  LifeCycle

```
public interface JFT
extends LifeCycle
```

Main basic library interface to use within JFT/API.
Use this interface to initialize, configure and start to use the library.

A singleton of this interface is available in the `THIS` constant.
Using this constant it's possible to access any functionality exposed by this library.

To start use this library read the JFT/Api Introduction or watch the data models (Package it.list.jft Data Model and Package it.list.jft.event Data Model) or just watch a few Java example programs.

JFT Exceptions

Almost all the methods of this library does not throw any exceptions of any sort: they instead return appropriate values to indicate any error condition found. There are very few exceptions to this policy:

- the alone method `addFieldByName()` in the `Mask` interface,
- all `setSomething` methods in `Param` sub−interfaces,

- all `makeSomething` methods in `Context`.

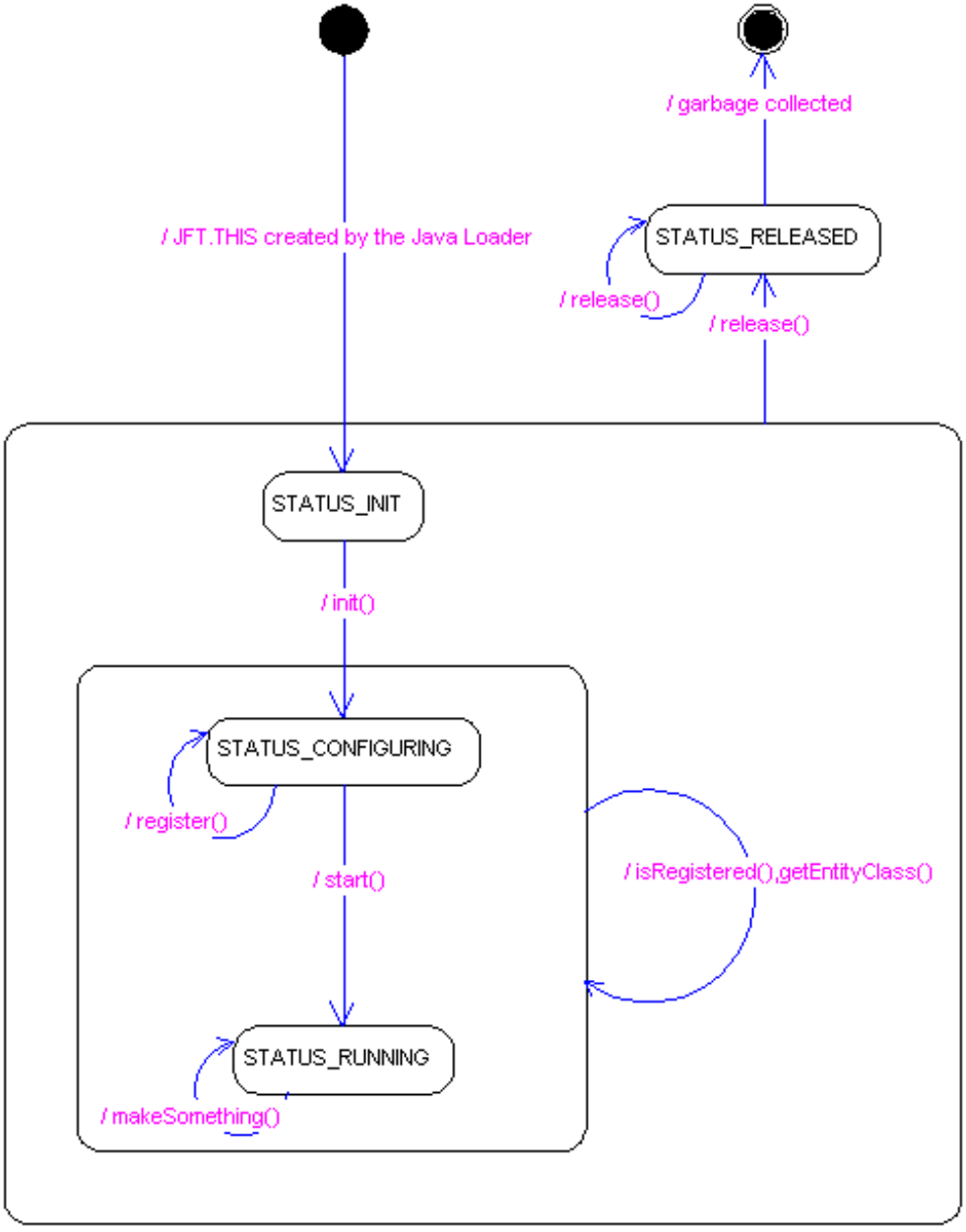
Even in this case the totality of thrown exceptions are unchecked (subclasses of `RuntimeException`), so they do not need to be catched or declared in the `throws` clause of method signature.

The JFT library does not throws any checked exception.

JFT Other Details

See JFT Implementation Threads and JFT Synchronization for details on implementation threads and synchronization.

JFT lifecycle



See Also:

JFT/Api Introduction, JFT Application Examples, `LifeCycle`, JFT Implementation Threads, JFT

Synchronization

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
static int	***MODE_MULTI_THREAD*** Threading mode: multi−thread.
static int	***MODE_NO_ENTITY_CLONING*** Disable entity cloning in the library.
static int	***STATUS_CONFIGURING*** Lifecycle status: JFT initialized: ready to be configured and then started.
static int	***STATUS_RUNNING*** Lifecycle status: JFT started: ready to be used and then released.
static JFT	***THIS*** Reference to the JFT singleton.
static int	***TRACE_LEVEL_DEBUG*** Trace level: messages in **bold** are traced: **DEBUG, TEST, INFO, WARN, ERROR, FATAL**.
static int	***TRACE_LEVEL_ERROR*** Trace level: messages in **bold** are traced: DEBUG, TEST, INFO, WARN, **ERROR, FATAL**.
static int	***TRACE_LEVEL_FATAL*** Trace level: messages in **bold** are traced: DEBUG, TEST, INFO, WARN, ERROR, **FATAL**.
static int	***TRACE_LEVEL_INFO*** Trace level: messages in **bold** are traced: DEBUG, TEST, **INFO, WARN, ERROR, FATAL**.
static int	***TRACE_LEVEL_TEST*** Trace level: messages in **bold** are traced: DEBUG, **TEST, INFO, WARN, ERROR, FATAL**.
static int	***TRACE_LEVEL_WARN*** Trace level: messages in **bold** are traced: DEBUG, TEST, INFO, **WARN, ERROR, FATAL**.

◊

Fields inherited from interface LifeCycle

RESULT_GENERIC_ERROR, RESULT_INVALID_STATUS, RESULT_OK, STATUS_INIT, STATUS_RELEASED

Method Summary

Methods

Modifier and Type	Method and Description
EntityClass	***getEntityClass***(int EntityClassID) Returns the registered EntityClass corresponding to the given EntityClassID.
EntityClass	***getEntityClass***(String EntityClassName) Returns the registered EntityClass corresponding to the given EntityClass Name.
String	***getLibraryVersion***() Returns the version of this library.
int	***init***(int mode) Initialize the library (with a specific threading model) and start the configuration phase.
boolean	***isRegistered***(int EntityClassID) Returns the indication that a given EntityClass (identified by an EntityClassID) has been registered.
Context	***makeContext***() Create and returns a new context.
Mask	***makeEmptyMask***(int entityClassID) Create and returns a new empty mask for a given Entity Class.
TimeStamp	***makeTimeStamp***(int dateTime, int prog) Create and returns a new TimeStamp.
TransactionID	***makeTransactionID***(int clientID, int clientServiceID, int businessServiceID, TimeStamp timeStamp) Create and returns a new TransactionID.
int	***register***(EntityClass entityClass) Register another EntityClass augmenting the number of the classes that can be manipulated by the library.
int	***register***(EntityClass[] entityClass) Like register(), this method register an array of EntityClass.
void	***setExitOnListenerException***(boolean enable) Enable/Disable the automatically termination of the JVM when an exception is thrown and not catch inside a Listener method.
void	***setTrace***(boolean enable) Enable/Disable the library trace.
int	***setTraceLevel***(int traceLevel) Set the mimun displayable level of the library trace.
int	***setTraceMode***(boolean autoFlush, File file) Set a file tracer.
int	***setTraceMode***(boolean autoFlush, PrintWriter writer) Set a printwriter (file, standard output/error, socketd, etc...) tracer.

int	***setTraceMode***(Tracer tracer) Set a customer tracer.
int	***start***() End the configuration phase and start to use the library.
void	***trace***(String module, int traceLevel, String message) Trace a given message.

Methods inherited from interface LifeCycle

enumChilds, getStatus, release

Field Detail

THIS

```
static final JFT THIS
```

Reference to the JFT singleton.

STATUS_CONFIGURING

```
static final int STATUS_CONFIGURING
```

Lifecycle status: JFT initialized: ready to be configured and then started.
This value may be returned by LifeCycle.getStatus().

Status Entry:
LifeCycle.STATUS_INIT init() ok STATUS_CONFIGURING.
Status Activities:
the JFT libray may be configured via the register() method.
Status Exit:
STATUS_CONFIGURING start() ok STATUS_RUNNING.
See Also:
JFT lifecycle, Constant Field Values

STATUS_RUNNING

```
static final int STATUS_RUNNING
```

Lifecycle status: JFT started: ready to be used and then released.
This value may be returned by LifeCycle.getStatus().

Status Entry:
STATUS_CONFIGURING start() ok STATUS_RUNNING.
Status Activities:
the JFT library may be used via the makeContext() or makeTimeStamp() or makeTransactionID() or makeEmptyMask() methods.
Status Exit:
STATUS_RUNNING LifeCycle.release() LifeCycle.STATUS_RELEASED.
See Also:

JFT lifecycle, Constant Field Values

◊ TRACE_LEVEL_DEBUG

```
static final int TRACE_LEVEL_DEBUG
```

Trace level: messages in **bold** are traced: **DEBUG, TEST, INFO, WARN, ERROR, FATAL**.

This value may be used as argument of `setTraceLevel(int)`.

See Also:

Constant Field Values

◊ TRACE_LEVEL_TEST

```
static final int TRACE_LEVEL_TEST
```

Trace level: messages in **bold** are traced: DEBUG, **TEST, INFO, WARN, ERROR, FATAL**.

This value may be used as argument of `setTraceLevel(int)`.

See Also:

Constant Field Values

◊ TRACE_LEVEL_INFO

```
static final int TRACE_LEVEL_INFO
```

Trace level: messages in **bold** are traced: DEBUG, TEST, **INFO, WARN, ERROR, FATAL**.

This value may be used as argument of `setTraceLevel(int)`.

See Also:

Constant Field Values

◊ TRACE_LEVEL_WARN

```
static final int TRACE_LEVEL_WARN
```

Trace level: messages in **bold** are traced: DEBUG, TEST, INFO, **WARN, ERROR, FATAL**.

This value may be used as argument of `setTraceLevel(int)`.

See Also:

Constant Field Values

◊ TRACE_LEVEL_ERROR

```
static final int TRACE_LEVEL_ERROR
```

Trace level: messages in **bold** are traced: DEBUG, TEST, INFO, WARN, **ERROR, FATAL**.

This value may be used as argument of `setTraceLevel(int)`.

See Also:

Constant Field Values

◊ **TRACE_LEVEL_FATAL**

```
static final int TRACE_LEVEL_FATAL
```

Trace level: messages in **bold** are traced: DEBUG, TEST, INFO, WARN, ERROR, **FATAL**.

This value may be used as argument of `setTraceLevel(int)`.

See Also:
Constant Field Values

◊ **MODE_MULTI_THREAD**

```
static final int MODE_MULTI_THREAD
```

Threading mode: multi−thread.

Details on threads and synchronization are available in JFT Implementation Threads

This value may be used as argument of `init(int)`.

See Also:
JFT Implementation Threads, Constant Field Values

◊ **MODE_NO_ENTITY_CLONING**

```
static final int MODE_NO_ENTITY_CLONING
```

Disable entity cloning in the library.

Usually library functions return a safe copy of an entity in its method (especially in the callback events). For this reason users can without any problems safely modify them or store their reference if needed. In case of heavy subscription load, it is possibile to improve the library performance using this flag; in this case entities coming from callback events are valid inside the callback event context, whereas out of that context their value can be changed using library calls (however you can clone them in the event if you want to save their value)

This value may be used as argument of `init(int)`.

See Also:
Constant Field Values

♦ Method Detail

◊ **getLibraryVersion**

```
String getLibraryVersion()
```

Returns the version of this library.

Returns:
the version of this library.
`null` is never returned.

◊ setExitOnListenerException

```
void setExitOnListenerException(boolean enable)
```

Enable/Disable the automatically termination of the JVM when an exception is thrown and not catch inside a `Listener` method.

By default (if this method is never invoked) the automatically invocation of `System.exit(0)` is enabled.

Parameters:

`enable` – `true` or `false` to enable or disable this switch.

◊ setTrace

```
void setTrace(boolean enable)
```

Enable/Disable the library trace.

By default (if this method is never invoked) the trace is disabled.

Parameters:

`enable` – `true` or `false` to enable or disable the trace.

◊ setTraceLevel

```
int setTraceLevel(int traceLevel)
```

Set the mimun displayable level of the library trace.

Available trace level are: `TRACE_LEVEL_DEBUG`, `TRACE_LEVEL_TEST`, `TRACE_LEVEL_INFO`, `TRACE_LEVEL_WARN`, `TRACE_LEVEL_ERROR` and `TRACE_LEVEL_FATAL`.

By default (if this method is never invoked) the trace level is `TRACE_LEVEL_WARN`.

Parameters:

`traceLevel` – one of the `TRACE_LEVEL_` constants.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. the `traceLevel` parameter is bad).

◊ setTraceMode

```
int setTraceMode(Tracer tracer)
```

Set a customer tracer.

The current customer tracer (the last set by this method or none if this method was never invoked) is replaced with the given customer tracer.

A customer tracer is described by the `Tracer` interface in which the `Tracer.onTrace()` method is automatically invoked whenever the trace is enabled and the current trace−message has a level greater or equal than the current trace level.

Parameters:

`tracer` − customer tracer (it may be `null`)

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise. is `null`).

◊ **setTraceMode**

```
int setTraceMode(boolean autoFlush,
                 File file)
```

Set a file tracer.

The current file tracer (the last set by this method or none if this method was never invoked) is replaced with the given file tracer.

If the trace is enabled a file tracer allow to trace in a file all trace−messages that have a level greater or equal than the current trace level.

Parameters:

`autoFlush` − `true/false` to enable/disable flush after every trace message.
`file` − file (it may be `null`) on which the trace messages are appended.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. the `file` parameter refers an unexisting/unaccessible file).

◊ **setTraceMode**

```
int setTraceMode(boolean autoFlush,
                 PrintWriter writer)
```

Set a printwriter (file, standard output/error, socketd, etc...) tracer.

The current printwriter tracer (the last set by this method or none if this method was never invoked) is replaced with the given printwriter tracer.

If the trace is enabled a printwriter tracer allow to trace in a PrintWriter all trace−messages that have a level greater or equal than the current trace level.

Parameters:

`autoFlush` − `true/false` to enable/disable flush after every trace message.
`writer` − PrintWriter (it may be `null`) on which the trace messages are appended.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. the `writer` parameter refers an unexisting/unaccessible writer).

◊ **trace**

```
void trace(String module,
           int traceLevel,
           String message)
```

Trace a given message.

The message will appear on the requested trace, depending on setting controlled by `setTrace(boolean)`, `setTraceLevel()` and the required `setTraceMode()`.

Parameters:

`module` − caller module name.
`traceLevel` − one of the `TRACE_LEVEL_` constants.
`message` − **not** newline−terminated message to be traced.

◊ init

```
int init(int mode)
```

Initialize the library (with a specific threading model) and start the configuration phase.

This method must be called only when current status is `LifeCycle.STATUS_INIT`, i.e. this method must be called before any other methods invocations (except for `getLibraryVersion()` and all trace methods that can be called at every time).

If this method invocation completed successfully, the current status changed to `STATUS_CONFIGURING`, otherwise it remains unchanged.

Parameters:

`mode` − threading mode (only `MODE_MULTI_THREAD` currently allowed).

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_INVALID_STATUS` if the current status is not `LifeCycle.STATUS_INIT`,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. the `mode` parameter is not `MODE_MULTI_THREAD`).

◊ register

```
int register(EntityClass entityClass)
```

Register another `EntityClass` augmenting the number of the classes that can be manipulated by the library.

This method must be called only when current status is `STATUS_CONFIGURING`, i.e. in the configuration phase between the `init(int)` and `start()` invocations.

This method must be called for each market or service EntityClass which is referenced or used in the rest of the application. To facilitate this the FastTrack libraries are equipped with several market/service libraries each containing the Java `EntityClasses` of the market/service structures.

Parameters:

`entityClass` − EntityClass to be registered.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_INVALID_STATUS` if the current status is not `STATUS_CONFIGURING`,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. the `entityClass` parameter does not refer a valid EntityClass).

◊ **register**

```
int register(EntityClass[] entityClass)
```

Like `register()`, this method register an array of EntityClass.
Parameters:
`entityClass` − Array of EntityClass to be registered.
Returns:
- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_INVALID_STATUS` if the current status is not `STATUS_CONFIGURING`,
- `LifeCycle.RESULT_GENERIC_ERROR` otherwise (e.g. some `entityClass` does not refer a valid EntityClass).

◊ **isRegistered**

```
boolean isRegistered(int EntityClassID)
```

Returns the indication that a given EntityClass (identified by an `EntityClassID`) has been registered.

Parameters:
`EntityClassID` − ID of the EntityClass to be checked.
Returns:
the indication that a given EntityClass (identified by an `EntityClassID`) has been registered.
`false` is returned when the current status is `LifeCycle.STATUS_INIT`.

◊ **getEntityClass**

```
EntityClass getEntityClass(int EntityClassID)
```

Returns the registered EntityClass corresponding to the given EntityClassID.

Parameters:
`EntityClassID` − ID of the EntityClass to be retrieved.
Returns:
the registered EntityClass corresponding to the given EntityClassID.
`null` is returned when the current status is `LifeCycle.STATUS_INIT`, or when the given `EntityClassID` is not registered.

◊ **getEntityClass**

```
EntityClass getEntityClass(String EntityClassName)
```

Returns the registered EntityClass corresponding to the given EntityClass Name.

Parameters:
`EntityClassName` − The name of the EntityClass to be retrieved.
Returns:
the registered EntityClass corresponding to the given EntityClass Name.
`null` is returned when the current status is `LifeCycle.STATUS_INIT`, or when the given `EntityClassID` is not registered.

◊ start

```
int start()
```

End the configuration phase and start to use the library.

This method must be called only when current status is `STATUS_CONFIGURING` after all the `register(it.list.jft.EntityClass)` invocations.

If this method invocation completed successfully, the current status changed to `STATUS_RUNNING`. otherwise it remains unchanged.

Returns:

- `LifeCycle.RESULT_OK` if the operation completed successfully,
- `LifeCycle.RESULT_INVALID_STATUS` if the current status is not `STATUS_CONFIGURING`.

◊ makeEmptyMask

```
Mask makeEmptyMask(int entityClassID)
```

Create and returns a new empty mask for a given Entity Class.

A mask may be used in subscriptions (`SubscriptionParam.setMask()`) or transactions (`TransactionParam.setMask()`).

This method must be called only when current status is `STATUS_RUNNING`, i.e. after the `start()` invocation.

Parameters:

`entityClassID` − Entity Class ID of the market class.

Returns:

the new empty mask.
`null` is returned when the current status is not `STATUS_RUNNING`, or when the parameter `entityClassID` is wrong.

◊ makeContext

```
Context makeContext()
```

Create and returns a new context.

A context is used to interact with one or more FastTrack servers.

This method must be called only when current status is `STATUS_RUNNING`, i.e. after the `start()` invocation.

Returns:

the new context.
`null` is returned when the current status is not `STATUS_RUNNING`.

◊ makeTimeStamp

```
TimeStamp makeTimeStamp(int dateTime,
                        int prog)
```

Create and returns a new TimeStamp.

This convenience method may be used to re−create a timestamp previously saved as 2 ints returned by invocation of `TimeStamp.getDateTime()` and `TimeStamp.getProg()`.

This method must be called only when current status is `STATUS_RUNNING`, i.e. after the `start()` invocation.

Parameters:

`dateTime` − saved value returned by a `TimeStamp.getDateTime()` invocation.

`prog` − saved value returned by a `TimeStamp.getProg()` invocation.

Returns:

the new TimeStamp.

`null` is returned when the current status is not `STATUS_RUNNING`, or when some parameter is < 0.

◊ makeTransactionID

```
TransactionID makeTransactionID(int clientID,
                                int clientServiceID,
                                int businessServiceID,
                                TimeStamp timeStamp)
```

Create and returns a new TransactionID.

This convenience method may be used to re−create a TransactionID previously saved as 5 ints returned by invocations of `TransactionID.getClientID()`, `TransactionID.getClientServiceID()`, `TransactionID.getBusinessServiceID()` and `TransactionID.getTimeStamp()`.

This method must be called only when current status is `STATUS_RUNNING`, i.e. after the `start()` invocation.

Parameters:

`clientID` − saved value returned by a `TransactionID.getClientID()` invocation.

`clientServiceID` − saved value returned by a `TransactionID.getClientServiceID()` invocation.

`businessServiceID` − saved value returned by a `TransactionID.getBusinessServiceID()` invocation.

`timeStamp` − saved value returned by a `TransactionID.getTimeStamp()` invocation.

Returns:

the new TransactionID.

`null` is returned when the current status is not `STATUS_RUNNING`, or when the `timeStamp` parameter is `null`.

See Also:

`makeTimeStamp(int, int)`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Mask

```
public interface Mask
```

A set of fields of a `EntityClass`.

A mask may be used in subscriptions (`SubscriptionParam.setMask()`) or transactions (`TransactionParam.setMask()`).

Empty masks are created by `JFT.makeEmptyMask()` and then filled with `addFieldByName()`.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	***`addFieldByName`***`(String fieldName)` Add a specific field to the mask.
`int`	***`getEntityClassID`***`()` Returns the ID of the `EntityClass` related to this mask.
`byte[]`	***`getMask`***`()`
`boolean`	***`isBound`***`()` Returns the bound−indication of this mask.
`void`	***`reset`***`()` Reset this mask to an emtpy mask.
`void`	***`setMask`***`(byte[] mask)`

Method Detail

◊ reset

```
void reset()
           throws IllegalStateException
```

Reset this mask to an emtpy mask.

Throws:

`IllegalStateException` − if this mask is bound.

addFieldByName

```
void addFieldByName(String fieldName)
                    throws IllegalArgumentException,
                           IllegalStateException
```

Add a specific field to the mask.

A masked field of this `EntityClass` related to this mask may be:

- 1. a `String` field of this EntityClass.
- 2. a primitive (i.e. boolean or numeric types) field of this EntityClass.
- 3. a component of an array, of primitive types (but here byte, boolean and char are **not** allowed!), field of this EntityClass.
  In this case the component is identified by an unsigned integer between square barckets [].
- 4. a masked field (recursive definition!) of a EntityClass reference field of this EntityClass.
  In this case the masked field is identified by the dot notation "field.field".
- 5. a masked field (recursive definition!) of an array, of EntityClass references, field of this EntityClass.
  In this case the component is identified by an unsigned integer between square barckets [].

Spaces (blank, tab, carriage return, etc...) are not allowed inside a masked field representation.

As an example please consider the following two EntityClasses:

```
class TypeA implements EntityClass {
  int n;
  int v[10];
  String s;
}
class TypeB implements EntityClass {
  TypeA a[10];
  TypeA x;
  double d[10];
  char c[10];
  byte b;
  String ss;
}
```

Here a list of **valid** masked fields of `TypeB`:

- Rule 1: `"ss"`
- Rule 2: `"b"`
- Rule 3: `"d[2]"`
- Rule 4: `"x.n", "x.v[3]", "x.s"`
- Rule 5: `"a[8].n", "a[0].v[9]", "a[5].s"`

Here a list of **invalid** masked fields of `TypeB`:

- Rule 1: `"s s", "t"`
- Rule 2: `"x"`
- Rule 3: `"c[2]", "d[+2]", "d[20]", "d[2", "d[ 2 ]", "ss[3]"`
- Rule 4: `"a.n", "x.y"`

◊ · Rule 5: `"a[8]. n", "a[8]", "a[0].v", "a[5].s[3]"`

Parameters:
`fieldName` − field of the `related EntityClass`,
Throws:
`IllegalArgumentException` − if `fieldName` is `null` or it does not exists in the `related EntityClass`.
`IllegalStateException` − if this mask is bound.

◊ **getEntityClassID**

```
int getEntityClassID()
```

Returns the ID of the `EntityClass` related to this mask.

The returned value is the same `entityClassID` used as parameter of `JFT.makeEmptyMask(int)` invocation that created this mask.

Returns:
the ID of the `EntityClass` related to this mask.

◊ **isBound**

```
boolean isBound()
```

Returns the bound−indication of this mask.

A mask is bound if it was used as creation parameter in a `SubscriptionParam.setMask()` or in a `TransactionParam.setMask()`.

`reset()` and `addFieldByName()` methods invocation on bound masks will throw a `IllegalStateException`.

Returns:
the bound−indication of this mask.

◊ **setMask**

```
void setMask(byte[] mask)
             throws IllegalArgumentException,
                    IllegalStateException
```

Throws:
`IllegalArgumentException`
`IllegalStateException`

◊ **getMask**

```
byte[] getMask()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Param

- *All Known Subinterfaces:*
  ConnectionParam, EntityClassQueryParam, FilterParam, MulticastConnectionParam, QueryParam, SubscriptionParam, TransactionParam

---

```
public interface Param
```

Super−interface common to all parameter container of `CommunicationLifeCycle` objects.

Bound

The 5 classes (`ConnectionParam`, `FilterParam`, `QueryParam`, `SubscriptionParam` and `TransactionParam`) that implement this `Param` interfaces share the concept of being `bound`.

Every time a parameter container (i.e. an object of the previous 5 classes) is created it is not `bound`.

When such parameter container is not bound then a specific parameter may be set via a `setSomething` method (e.g. a `setEntityClassID()` may be issued on a `SubscriptionParam`).

Once a parameter container is given as creation parameter of a `makeSomething` method of Context (e.g. `Context.makeSubscription()`) it becomes `bound`:

- it can be shared with others `CommunicationLifeCycle` object,
  i.e. it can be re−used in another `makeSomething` method of Context;
- but it cannot never change:
  all `setSomething` method will throw a `IllegalStateException` in this case.

Acceptable Values

Each single parameter (of each class that implements this `Param` interface) has the concept of being acceptable.

> E.g. the EntityClassID of a SubscriptionParam is acceptable only if it has been registered.

In this documentation each description of each single parameter has a section, titled "**Acceptable values:**", that describes the acceptable values for the corresponding parameter using a Java `boolean` expression that must be satisfied, i.e. its computed value at run−time must be `true`.

> E.g. a value for the EntityClassID of a SubscriptionParam is acceptable only if:
> ```
> JFT.THIS.isRegistered(getEntityClassID())
> // i.e. acceptable only if registered
> ```

These run−time checks (to see if a parameter has an acceptable value) are **not** executed when the parameter is set (the various `setSomething` methods of sub−interface of Param), but instead when it's given to a specific `CommunicationLifeCycle` object creation (the various `makeSomething` methods of `Context`).

> E.g. the check to see if the value of the EntityClassID of a SubscriptionParam is acceptable is not made inside:

```
   mySubscrParam.setEntityClassID(myEntityClassID);
   // no check here
```
instead it's made inside:
```
   myContext.makeSubscription(myConnection, mySubscrParam,
mySubscrListener);
   // here all mySubscrParam parameters are checked !
```

Please note that when a check is made it regards **all** the parameters used by the specific operation and not only the parameters explicitly set by the programmer.

E.g. inside a Subscription creation all the following parameters are checked to see if they have acceptable values:
EntityClassID, EntityClassVersion, EntityClassTimeStamp, QueryType, EntityKey, Filter, Flow and Mask.
If some has an unacceptable value (i.e. if some corresponding `boolean` expression valuated `false`) then the operation fails and a IllegalArgumentException is thrown.

- ♦

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	***`isBound`***`()` Returns the bound−indication of this parameter container.

- ♦

Method Detail

◊ isBound

```
boolean isBound()
```

Returns the bound−indication of this parameter container.

A parameter container is bound if it was used as creation parameter in some `CommunicationLifeCycle` object.

`setSomething` method invocation on bound objects will throw a `IllegalStateException`.

Returns:
the bound−indication of this parameter container.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface ConnectionParam

- *All Superinterfaces:*
  Param

```
public interface ConnectionParam
extends Param
```

Connection parameter container.

This container of connection parameters is created by `Context.makeConnectionParam()` and it is used by `Context.makeConnection()`.

See Also:
Connection Usage, Acceptable Values

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
static int	***CONN_TYPE_HTTP_TUNNEL*** Connection−transport type−code: to use an HTTP tunnel, over TCP/IP, as transport.
static int	***CONN_TYPE_SOCKS4A*** Connection−transport type−code: to use an Socks4A proxy.
static int	***CONN_TYPE_SOCKS5*** Connection−transport type−code: to use an Socks5 proxy.
static int	***CONN_TYPE_TCP*** Connection−transport type−code: to use TCP/IP as transport.
static int	***USER_TYPE_AUTOTRADER*** User−type code: to create a connection on which send data variations (transaction).
static int	***USER_TYPE_CONTROLLER*** User−type code: to create a specialized (no more here described) connection.
static int	***USER_TYPE_MASTERSLAVE*** User−type code: to create a specialized (no more here described) connection.
static int	***USER_TYPE_MONITOR*** User−type code: to create a connection on which request data (subscriptions and/or queries).
static int	***USER_TYPE_TRADER***

	User−type code: to create a connection on which send data variations (transaction).
`static int`	***USER_TYPE_VIEW*** User−type code: to create a connection on which request data (subscriptions and queries).

Method Summary

Methods

Modifier and Type	Method and Description
`String`	***getAlternativeHost*`()`** Returns the optional alternative requested server−host of a new connection.
`int`	***getAlternativePort*`()`** Returns the optional alternative requested server−port of a new connection.
`int[]`	***getApplRevision*`()`** Returns the client version for the new connection.
`int`	***getApplSignature*`()`** Returns the client signature for the new connection.
`File`	***getAuthFile*`()`** Returns the File that contains an authorization key for the new connection.
`String`	***getAuthKey*`()`** Returns the authorization key for the new connection.
`String`	***getCharSet*`()`** Returns the String that represents the charset used to code/decode the strings on the new connection.
`int`	***getClientID*`()`** Returns the ClientID for the new connection.
`boolean`	***getCompression*`()`** Returns the requested indication about a compressed transmission for a new connection.
`int[]`	***getConnType*`()`** Returns the array of connection−transport code of a new connection.
`String`	***getHost*`()`** Returns the requested server−host of a new connection.
`String`	***getPassword*`()`** Returns the requested user−password of a new connection.
`int`	***getPort*`()`** Returns the requested server−port of a new connection.
`String`	***getProxyHost*`()`** Returns the optional requested proxy−host of a new connection.
`int`	***getProxyPort*`()`** Returns the optional requested proxy−port of a new connection.

String	***getProxyUserName***() Returns the optional requested proxy−username of a new connection.
String	***getService***() Returns the optional market/service name to which the new connection must talk.
boolean	***getTcpNoDelay***() Tests if TCP_NODELAY is enabled.
String	***getUserName***() Returns the requested user−name of a new connection.
int	***getUserType***() Returns the user−type code of a new connection.
void	***setAlternativeHost***(String alternativeHost) Set/replace the optional alternative requested server−host of a new connection.
void	***setAlternativePort***(int alternativePort) Set/replace the optional alternative requested server−port of a new connection.
void	***setApplRevision***(int[] applRevision) Set/replace the client version for the new connection.
void	***setApplSignature***(int applSignature) Set/replace the client signature for the new connection.
void	***setAuthFile***(File file) Set/replace the File that contains an authorization key for the new connection.
void	***setAuthKey***(String key) Set/replace the authorization key for the new connection.
void	***setCharSet***(String charSet) Set/replace the String that represents the charset used to code/decode the strings on the new connection.
void	***setClientID***(int clientID) Set/replace the ClientID for the new connection.
void	***setCompression***(boolean enable) Set/replace the requested indication about a compressed transmission for a new connection.
void	***setConnType***(int connType) Set/replace the connection−transport code of a new connection.
void	***setConnType***(int[] connType) Set/replace the array of connection−transport code to try for a new connection.
void	***setHost***(String host) Set/replace the requested server−host of a new connection.
void	***setPassword***(String password) Set/replace the requested user−password of a new connection.
void	***setPort***(int port) Set/replace the requested server−port of a new connection.
void	***setProxyHost***(String proxyHost) Set/replace the optional requested proxy−host of a new connection.

void	***setProxyPassword***(String proxyPassword) Set/replace the optional requested proxy−password of a new connection.
void	***setProxyPort***(int proxyPort) Set/replace the optional requested proxy−port of a new connection.
void	***setProxyUserName***(String proxyUserName) Set/replace the optional requested proxy−username of a new connection.
void	***setService***(String service) Set/replace the optional market/service name to which the new connection must talk.
void	***setTcpNoDelay***(boolean noDelay) Set the TCP_NODELAY setting.
void	***setUserName***(String userName) Set/replace the requested user−name of a new connection.
void	***setUserType***(int userType) Set/replace the user−type code of a new connection.

◊

Methods inherited from interface Param

isBound

- ♦

Field Detail

◊ **USER_TYPE_TRADER**

```
static final int USER_TYPE_TRADER
```

User−type code: to create a connection on which send data variations (transaction).

This value may be set with setUserType(int) and retrieved by getUserType().

This and USER_TYPE_VIEW values are the most commonly used values.

See Also:
Constant Field Values

◊ **USER_TYPE_AUTOTRADER**

```
static final int USER_TYPE_AUTOTRADER
```

User−type code: to create a connection on which send data variations (transaction).

This value may be set with setUserType(int) and retrieved by getUserType().

USER_TYPE_TRADER and USER_TYPE_VIEW values are the most commonly used values.

See Also:
Constant Field Values

◊ USER_TYPE_MONITOR

```
static final int USER_TYPE_MONITOR
```

User−type code: to create a connection on which request data (subscriptions and/or queries).

This value may be set with `setUserType(int)` and retrieved by `getUserType()`.

`USER_TYPE_TRADER` and `USER_TYPE_VIEW` values are the most commonly used values.

See Also:
Constant Field Values

◊ USER_TYPE_VIEW

```
static final int USER_TYPE_VIEW
```

User−type code: to create a connection on which request data (subscriptions and queries).

This value may be set with `setUserType(int)` and retrieved by `getUserType()`.

This and `USER_TYPE_TRADER` values are the most commonly used values.

See Also:
Constant Field Values

◊ USER_TYPE_CONTROLLER

```
static final int USER_TYPE_CONTROLLER
```

User−type code: to create a specialized (no more here described) connection.

This value may be set with `setUserType(int)` and retrieved by `getUserType()`.

`USER_TYPE_TRADER` and `USER_TYPE_VIEW` values are the most commonly used values.

See Also:
Constant Field Values

◊ USER_TYPE_MASTERSLAVE

```
static final int USER_TYPE_MASTERSLAVE
```

User−type code: to create a specialized (no more here described) connection.

This value may be set with `setUserType(int)` and retrieved by `getUserType()`.

`USER_TYPE_TRADER` and `USER_TYPE_VIEW` values are the most commonly used values.

See Also:
Constant Field Values

◊ CONN_TYPE_TCP

```
static final int CONN_TYPE_TCP
```

Connection−transport type−code: to use TCP/IP as transport.

This value may be set with `setConnType(int)` and retrieved by `getConnType()`.

See Also:
Constant Field Values

◊ CONN_TYPE_HTTP_TUNNEL

```
static final int CONN_TYPE_HTTP_TUNNEL
```

Connection−transport type−code: to use an HTTP tunnel, over TCP/IP, as transport.

This value may be set with `setConnType(int)` and retrieved by `getConnType()`.

See Also:
Constant Field Values

◊ CONN_TYPE_SOCKS4A

```
static final int CONN_TYPE_SOCKS4A
```

Connection−transport type−code: to use an Socks4A proxy.

This value may be set with `setConnType(int)` and retrieved by `getConnType()`.

See Also:
Constant Field Values

◊ CONN_TYPE_SOCKS5

```
static final int CONN_TYPE_SOCKS5
```

Connection−transport type−code: to use an Socks5 proxy.

This value may be set with `setConnType(int)` and retrieved by `getConnType()`.

See Also:
Constant Field Values

♦ Method Detail

◊ getHost

```
String getHost()
```

Returns the requested server−host of a new connection.

The pair given by this value together with `getPort()` describe the server to which the client must talk.

Used by:
`Connection.open()`
Default value:
`null`
Acceptable values:
`getHost() != null &getHost().length > 0`

Returns:
the requested server−host of a new connection.
See Also:
`who sets this value`, `if you want to use YAS service to find the less loaded service`

◊ getPort

```
int getPort()
```

Returns the requested server−port of a new connection.

The pair given by `getHost()` together with this value describe the server to which the client must talk.

Used by:
`Connection.open()`
Default value:
0
Acceptable values:
`getPort() > 0`
Returns:
the requested server−port of a new connection.
See Also:
`who sets this value`

◊ getAlternativeHost

```
String getAlternativeHost()
```

Returns the optional alternative requested server−host of a new connection.

The optional pair given by this value together with `getAlternativePort()` describe another server to which the client must talk in the case the first attempt to the principal server (pair `getHost()` together with `getPort()`) failed.

Used by:
`Connection.open()`
Default value:
`null`
Acceptable values:
`getAlternativeHost() == null || getAlternativeHost().length > 0 &getAlternativePort() > 0`
Returns:
the optional alternative requested server−host of a new connection.
See Also:
`who sets this value`

◊ getAlternativePort

```
int getAlternativePort()
```

Returns the optional alternative requested server−port of a new connection.

The optional pair given by `getAlternativeHost()` together with this value describe another server to which the client must talk in the case the first attempt to the principal server (pair `getHost()` together with `getPort()`) failed.

Used by:
`Connection.open()`
Default value:
0
Acceptable values:
```
getAlternativePort == 0
|| getAlternativePort() > 0 &getAlternativeHost() != null
&getAlternativeHost().length > 0
```
Returns:
the optional alternative requested server−port of a new connection.
See Also:
`who sets this value`

◊ getConnType

```
int[] getConnType()
```

Returns the array of connection−transport code of a new connection.

The connection−transport code of a connection describe the transport used on that connection. Possible values are:

- `CONN_TYPE_TCP` to use the classic TCP/IP transport,
- `CONN_TYPE_HTTP_TUNNEL` to use a tunnel over HTTP transport, e.g. used in applet inside HTML pages, or else used to bypass firewall,
- `CONN_TYPE_SOCKS4A` to use a socks version 4a proxy,
- `CONN_TYPE_SOCKS5` to use a socks version 5 proxy,

In the two latter cases a pair `getProxyHost()` together with `getProxyPort()` must be defined.

Used by:
`Connection.open()`
Default value:
`CONN_TYPE_TCP`
Acceptable values:
```
   getConnType() == CONN_TYPE_TCP
|| getConnType() == CONN_TYPE_HTTP_TUNNEL
|| getConnType() == CONN_TYPE_SOCKS4A
|| getConnType() == CONN_TYPE_SOCKS5
```
Returns:
the array of connection−transport code of a new connection.
See Also:
`who sets this value`

◊ getProxyHost

```
String getProxyHost()
```

Returns the optional requested proxy−host of a new connection.

The optional pair given by this value together with `getProxyPort()` describe the proxy used when a non−`CONN_TYPE_TCP` connection−transport is used.

Used by:
`Connection.open()`
Default value:
`null`
Acceptable values:
```
getProxyHost() == null
|| getConnType() != CONN_TYPE_TCP &getProxyHost().length
> 0 &getProxyPort() > 0
```
Returns:
the optional requested proxy−host of a new connection.
See Also:
`who sets this value`

◊ getProxyPort

```
int getProxyPort()
```

Returns the optional requested proxy−port of a new connection.

The optional pair given by `getProxyHost()` together with this value describe the proxy used when a non−`CONN_TYPE_TCP` connection−transport is used.

Used by:
`Connection.open()`
Default value:
0
Acceptable values:
```
getProxyPort == 0
|| getConnType() != CONN_TYPE_TCP &getProxyPort() > 0
&getProxyHost() != null &getProxyHost().length > 0
```
Returns:
the requested optional proxy−port of a new connection.
See Also:
`who sets this value`

◊ getProxyUserName

```
String getProxyUserName()
```

Returns the optional requested proxy−username of a new connection.

Used by:
`Connection.open()`
Default value:
0
Acceptable values:
```
getProxyPort == 0
|| getConnType() != CONN_TYPE_TCP &getProxyPort() > 0
&getProxyHost() != null &getProxyHost().length > 0
```
Returns:
the requested optional proxy−username of a new connection.
See Also:

`who sets this value`, `who sets proxy-password`

◊ getCompression

```
boolean getCompression()
```

Returns the requested indication about a compressed transmission for a new connection.

A `true` values indicates that the transmission between the client and the server for this connection must be compressed to save bandwidth.

Used by:
`Connection.open()`
Default value:
`false`
Acceptable values:
`true // any value`
Returns:
the requested indication about a compressed transmission for a new connection.
See Also:
`who sets this value`

◊ getCharSet

```
String getCharSet()
```

Returns the String that represents the charset used to code/decode the strings on the new connection.

Every time a `String` goes from the client to server it's coded as a bytes-sequence using the given charset. The behavior of this transformation when the string cannot be encoded in the given charset is unspecified.

Every time a bytes-sequence goes from the server to the client it's decoded into a String using the given charset. The behavior of this transformation when the given bytes are not valid in the given charset is unspecified.

Please note that the default-value is determined at run-time as "ISO-8859-15" or else "ISO-8859-1" depending on the Java platform on which the clients run:

- "ISO-8859-15" (ISO-LATIN-9) is the first-choice (because it contains the EURO-sign),
- "ISO-8859-1" (ISO-LATIN-1) is the second choice (it does not contain the EURO-sign, but it contains the "broken-bar", cedilla, acute-accent, "1/4", "1/2" and "3/4" characters).

Please see ISO Latin 9 as compared with ISO Latin 1 to check all differences between the two charsets.

Used by:
`Connection.open()`
Default value:
`"ISO-8859-15"` ISO-LATIN-9, if supported by the Java platform on which the client runs,
`"ISO-8859-1"` ISO-LATIN-1, otherwise.
Acceptable values:

```
getCharSet() != null &getCharSet().length > 0
&new String("Hello World").getBytes(getCharSet()) != null
// i.e. no exception thrown
```

Returns:

the String that represents the Charset used to code/decode the strings on the new connection.

See Also:

`who sets this value`

◊ getService

```
String getService()
```

Returns the optional market/service name to which the new connection must talk.

Used by:

`Connection.open()`

Default value:

`null`

Acceptable values:

```
getService() == null || getService.length > 0
```

Returns:

the optional market/service name to which the new connection must talk.

See Also:

`who sets this value`

◊ getUserType

```
int getUserType()
```

Returns the user−type code of a new connection.

The user−type code of a connection describe the activities that can be made on that connection. Possible values are:

- `USER_TYPE_TRADER` to send transactions on the connection,
- `USER_TYPE_AUTOTRADER` to send transactions on the connection,
- `USER_TYPE_MONITOR` to request data (subscription and/or queries) on the connection,
- `USER_TYPE_VIEW` to request data (subscription and/or queries) on the connection,
- `USER_TYPE_CONTROLLER` to use in a specially way (not described here) the connection,
- `USER_TYPE_MASTERSLAVE` to use in a specially way (not described here) the connection.

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_USERTYPE` failure−code if it does not comply with this value.

Used by:

`Connection.open()`

Default value:

`USER_TYPE_VIEW`

Acceptable values:

```
   getUserType() == USER_TYPE_TRADER
|| getUserType() == USER_TYPE_AUTOTRADER
```

```
|| getUserType() == USER_TYPE_MONITOR
|| getUserType() == USER_TYPE_VIEW
|| getUserType() == USER_TYPE_CONTROLLER
|| getUserType() == USER_TYPE_MASTERSLAVE
```

Returns:

the user−type code of a new connection.

See Also:

`who sets this value`

◊ getUserName

```
String getUserName()
```

Returns the requested user−name of a new connection.

The pair given by this value together with `getPassword()` describe the user associated to the new connection.

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_USERNAME` failure−code if it does not comply with this value.

Used by:

`Connection.open()`

Default value:

`null`

Acceptable values:

`getUserName() != null &getUserName().length > 0`

Returns:

the requested user−name of a new connection.

See Also:

`who sets this value`

◊ getPassword

```
String getPassword()
```

Returns the requested user−password of a new connection.

The pair given by `getUserName()` together with this value describe the user associated to the new connection.

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_PASSWORD` failure−code if it does not comply with this value.

Used by:

`Connection.open()`

Default value:

`null`

Acceptable values:

`getPassword() != null &getPassword().length > 0`

Returns:

the requested user−password of a new connection.

See Also:

who sets this value

◊ getClientID

```
int getClientID()
```

Returns the ClientID for the new connection.

The ClientID is a positive number that uniquely identifies the client.

This number is automatically used on `TransactionID`s of all `Transaction`s sent on this connection.

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_CLIENTID` failure−code if it does not comply with this value.

Used by:

`Connection.open()`

Default value:

0

Acceptable values:

`getClientID() > 0`

Returns:

the ClientID for the new connection.

See Also:

`who sets this value`

◊ getApplRevision

```
int[] getApplRevision()
```

Returns the client version for the new connection.

A version is always represented by a three−dimensional array; e.g. the version 2.0.3 is represented by:

```
int[] version = {2, 0, 3};
```

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_REVISION` failure−code if it does not comply with this value.

Used by:

`Connection.open()`

Default value:

`{0,0,0}`

Acceptable values:

```
getApplRevision() != null &getApplRevision().length == 3
&getApplRevision()[0] >= 0 &getApplRevision()[0] <= 255
&getApplRevision()[1] >= 0 &getApplRevision()[1] <= 255
&getApplRevision()[2] >= 0 &getApplRevision()[2] <= 255
```

Returns:

the client version for the new connection.

See Also:

`who sets this value`

◊ getApplSignature

```
int getApplSignature()
```

Returns the client signature for the new connection.

A signature is a non−negative number, that may be be required by the service (on the server) that manages the market.

Used by:
`Connection.open()`
Default value:
`0`
Acceptable values:
`getApplSignature() >= 0`
Returns:
the client signature for the new connection.
See Also:
`who sets this value`

◊ getAuthKey

```
String getAuthKey()
```

Returns the authorization key for the new connection.

An authorization key (normally given by List S.p.a. in an authorization file is a key that allow the client to open and use successfully a connection with a given server.

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_AUTH_KEY` failure−code if it does not comply with this value.

Used by:
`Connection.open()`
Default value:
`null`
Acceptable values:
```
getAuthKey == null
|| getAuthFile() == null &getAuthKey().length > 0
```
Returns:
the authorization key for the new connection.
See Also:
`who sets this value`

◊ getAuthFile

```
File getAuthFile()
```

Returns the File that contains an authorization key for the new connection.

An authorization file, given by List S.p.a., contains a key that allow the client to open and use successfully a connection with a given server.

The server may subsequently returns a `ConnectionOpenEvent.RESULT_INVALID_AUTH_KEY` failure−code if it does not comply with the authorization key.

Used by:
`Connection.open()`
Default value:
`null`
Acceptable values:
```
getAuthFile() == null
|| getAuthKey() == null &getAuthFile().canRead()
```
Returns:
the File that contains an authorization key for the new connection.
See Also:
`who sets this value`

◊ getTcpNoDelay

```
boolean getTcpNoDelay()
```

Tests if TCP_NODELAY is enabled.

A `true` values indicates that Nagle's algorithm is disabled.

Used by:
`Connection.open()`
Default value:
`true`
Acceptable values:
```
true // any value
```
Returns:
a boolean indicating whether or not TCP_NODELAY is enabled.
See Also:
`who sets this value`

◊ setHost

```
void setHost(String host)
             throws IllegalStateException
```

Set/replace the requested server−host of a new connection.

Parameters:
`host` − the requested server−host of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setPort

```
void setPort(int port)
             throws IllegalStateException
```

Set/replace the requested server−port of a new connection.

Parameters:
`port` − the requested server−port of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setAlternativeHost

```
void setAlternativeHost(String alternativeHost)
                        throws IllegalStateException
```

Set/replace the optional alternative requested server−host of a new connection.

Parameters:
`alternativeHost` − the optional alternative requested server−host of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setAlternativePort

```
void setAlternativePort(int alternativePort)
                        throws IllegalStateException
```

Set/replace the optional alternative requested server−port of a new connection.

Parameters:
`alternativePort` − the optional alternative requested server−port of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setConnType

```
void setConnType(int connType)
                 throws IllegalStateException
```

Set/replace the connection−transport code of a new connection.

Parameters:
`connType` − the connection−transport code of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setConnType

```
void setConnType(int[] connType)
                 throws IllegalStateException
```

Set/replace the array of connection−transport code to try for a new connection.

Parameters:
`connType` − array of the connection−transport code of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setProxyHost

```
void setProxyHost(String proxyHost)
                  throws IllegalStateException
```

Set/replace the optional requested proxy−host of a new connection.

Parameters:
`proxyHost` − the optional requested proxy−host of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setProxyPort

```
void setProxyPort(int proxyPort)
                  throws IllegalStateException
```

Set/replace the optional requested proxy−port of a new connection.

Parameters:
`proxyPort` − the optional requested proxy−port of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setProxyUserName

```
void setProxyUserName(String proxyUserName)
```

Set/replace the optional requested proxy−username of a new connection.

Parameters:
`proxyUserName` − the optional requested proxy−username of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setProxyPassword

```
void setProxyPassword(String proxyPassword)
```

Set/replace the optional requested proxy−password of a new connection.

Parameters:
`proxyPassword` − the optional requested proxy−password of a new connection.
Throws:

`IllegalStateException` − if this container is already bound.

◊ setCompression

```
void setCompression(boolean enable)
                    throws IllegalStateException
```

Set/replace the requested indication about a compressed transmission for a new connection.

Parameters:

`enable` − the requested indication about a compressed transmission for a new connection.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

◊ setCharSet

```
void setCharSet(String charSet)
                throws IllegalStateException
```

Set/replace the String that represents the charset used to code/decode the strings on the new connection.

Parameters:

`charSet` − the String that represents the charset used to code/decode the strings on the new connection.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

◊ setService

```
void setService(String service)
                throws IllegalStateException
```

Set/replace the optional market/service name to which the new connection must talk.

If you set the service, you ask to YAS service (identified by host and port parameters) to establish a connection with the less−loaded service.

You can specifies the double services connection required by ASIA platform using "|" (pipe) character as the separator of the public and private service name, for example: "PUBLMETAMARKET|PRIVMETAMARKET"

Parameters:

`service` − the optional market/service name to which the new connection must talk.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

◊ **setUserType**

```
void setUserType(int userType)
                 throws IllegalStateException
```

Set/replace the user−type code of a new connection.

Parameters:
`userType` − the user−type code of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ **setUserName**

```
void setUserName(String userName)
                 throws IllegalStateException
```

Set/replace the requested user−name of a new connection.

Parameters:
`userName` − the requested user−name of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ **setPassword**

```
void setPassword(String password)
                 throws IllegalStateException
```

Set/replace the requested user−password of a new connection.

Parameters:
`password` − the requested user−password of a new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ **setClientID**

```
void setClientID(int clientID)
                 throws IllegalStateException
```

Set/replace the ClientID for the new connection.

Parameters:
`clientID` − the ClientID for the new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setApplRevision

```
void setApplRevision(int[] applRevision)
                     throws IllegalStateException
```

Set/replace the client version for the new connection.

Parameters:
`applRevision` − the client version for the new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setApplSignature

```
void setApplSignature(int applSignature)
                      throws IllegalStateException
```

Set/replace the client signature for the new connection.

Parameters:
`applSignature` − the client signature for the new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setAuthKey

```
void setAuthKey(String key)
                throws IllegalStateException
```

Set/replace the authorization key for the new connection.

Parameters:
`key` − the authorization key for the new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setAuthFile

```
void setAuthFile(File file)
                 throws IllegalStateException
```

Set/replace the File that contains an authorization key for the new connection.

Parameters:
`file` − the File that contains an authorization key for the new connection.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setTcpNoDelay

```
void setTcpNoDelay(boolean noDelay)
                   throws IllegalStateException
```

Set the TCP_NODELAY setting. Setting this to true will disable Nagle's algorithm for TCP. The default is true.

Parameters:

`noDelay` − false to disable TCP_NODELAY (enable Nagle's algorithm), true to enable.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface EntityClassQueryParam

- *All Superinterfaces:*
  Param

```
public interface EntityClassQueryParam
extends Param
```

- ♦
  ## Method Summary

  Methods

Modifier and Type	Method and Description
int	***getEntityClassID***()
String	***getEntityClassName***()
void	***setEntityClassID***(int entityClassID)
void	***setEntityClassName***(String entityClassName)

  ◊
  ## Methods inherited from interface Param

  isBound

- ♦
  ## Method Detail

◊ **getEntityClassID**

```
int getEntityClassID()
```

◊ **getEntityClassName**

```
String getEntityClassName()
```

◊ **setEntityClassID**

```
void setEntityClassID(int entityClassID)
                      throws IllegalStateException
```

Throws:
`IllegalStateException`

◊ **setEntityClassName**

```
void setEntityClassName(String entityClassName)
                        throws IllegalStateException
```

Throws:
`IllegalStateException`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface FilterParam

- *All Superinterfaces:*
  Param

```
public interface FilterParam
extends Param
```

Filter parameter container.

This container of filter parameters is created by `Context.makeFilterParam()` and it is used by `Context.makeFilter()`.

See Also:
Filter Usage, Acceptable Values

- ♦

Method Summary

Methods

Modifier and Type	Method and Description
`String`	***getDefinition***`()` Returns the filter definition of the new filter.
`int`	***getEntityClassID***`()`

	Returns the EntityClassID of the associated `EntityClass` of the new filter.
`int`	***`getType`***`()` Returns the filter type of the new filter.
`void`	***`setDefinition`***`(String definition)` Set/replace the filter definition of the new filter.
`void`	***`setEntityClassID`***`(int entityClassID)` Set/replace the associated `EntityClass` of the new filter.
`void`	***`setType`***`(int type)` Set/replace the filter type of the new filter.

◊

Methods inherited from interface Param

`isBound`

- ♦

Method Detail

◊ **getEntityClassID**

```
int getEntityClassID()
```

Returns the EntityClassID of the associated `EntityClass` of the new filter.

The associated `EntityClass` is one of the 3 things that define a filter: (associated EntityClass, filter type and the optional filter definition).

The precise meaning of these 3 things depends from the particular filter and, in general, it must be agreed between the client and the server.

The server may subsequently returns a `FilterCreateEvent.RESULT_INVALID_ENTITY_CLASS_ID` failure−code if it does not understand this value.

Used by:
`Filter.create()`
Default value:
0
Acceptable values:
`JFT.THIS.isRegistered(getEntityClassID())`
Returns:
the EntityClassID of the associated `EntityClass` of the new filter.
See Also:
`who sets this value`

◊ **getType**

```
int getType()
```

Returns the filter type of the new filter.

The filter type is one of the 3 things that define a filter: (associated EntityClass, filter type and the optional filter definition).

The precise meaning of these 3 things depends from the particular filter and, in general, it must be agreed between the client and the server.

The server may subsequently returns a `FilterCreateEvent.RESULT_INVALID_FILTER_TYPE` failure−code if it does not understand this value.

Used by:
`Filter.create()`
Default value:
0
Acceptable values:
`getType() >= 0`
Returns:
the filter type of the new filter.
See Also:
`who sets this value`

◊ getDefinition

```
String getDefinition()
```

Returns the filter definition of the new filter.

The filter definition is one of the 3 things that define a filter: (associated EntityClass, filter type and the optional filter definition).

The precise meaning of these 3 things depends from the particular filter and, in general, it must be agreed between the client and the server.

The server may subsequently returns a `FilterCreateEvent.RESULT_SYNTAX_ERROR` or a `FilterCreateEvent.RESULT_INVALID_FILTER_LEN` failure−code if it does not understand this value or if this value is too long.

This value is optional: `null` and empty strings are acceptable values and, in this case, no definition is given to the server.

Used by:
`Filter.create()`
Default value:
`null`.
Acceptable values:
`true // any value`
Returns:
the filter definition of the new filter.
See Also:
`who sets this value`

◊ **setEntityClassID**

```
void setEntityClassID(int entityClassID)
                      throws IllegalStateException
```

Set/replace the associated `EntityClass` of the new filter.

Parameters:
`entityClassID` − EntityClassID of the associated `EntityClass` of the new filter.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ **setType**

```
void setType(int type)
             throws IllegalStateException
```

Set/replace the filter type of the new filter.

Parameters:
`type` − type of the new filter.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ **setDefinition**

```
void setDefinition(String definition)
                   throws IllegalStateException
```

Set/replace the filter definition of the new filter.

Parameters:
`definition` − filter definition of the new filter.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface MulticastConnectionParam

- *All Superinterfaces:*
  Param

```
public interface MulticastConnectionParam
extends Param
```

- ♦ **Method Summary**

Methods

Modifier and Type	Method and Description
String	***getAddress***()
String	***getCharSet***()
int	***getPort***()
void	***setAdress***(String address)
void	***setCharSet***(String charSet)
void	***setPort***(int port)

◊ **Methods inherited from interface Param**

isBound

- ♦ **Method Detail**

◊ **getAddress**

```
String getAddress()
```

◊ **getPort**

```
int getPort()
```

◊ **getCharSet**

```
String getCharSet()
```

◊ **setAdress**

```
void setAdress(String address)
               throws IllegalStateException
```

Throws:
IllegalStateException

◊ **setPort**

```
void setPort(int port)
             throws IllegalStateException
```

Throws:
IllegalStateException

◊ **setCharSet**

```
void setCharSet(String charSet)
                throws IllegalStateException
```

Throws:
IllegalStateException

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface QueryParam

- *All Superinterfaces:*
  Param

```
public interface QueryParam
extends Param
```

Query parameter container.

This container of query parameters is created by `Context.makeQueryParam()` and it is used by `Context.makeQuery()`.

See Also:
Query Usage, Acceptable Values

- ♦
  ## Method Summary

  Methods

Modifier and Type	Method and Description
`int`	***getQueryID***`()` Returns the QueryID of the new query.
`Entity`	***getQueryParameterEntity***`()` Returns the Query Parameter Entity of the new query.
`void`	***setQueryID***`(int queryID)` Set/replace the QueryID of the new query.
`void`	***setQueryParameterEntity***`(Entity queryParameter)` Set/replace the Query Parameter Entity of the new query.

  ◊
  ### Methods inherited from interface Param

  `isBound`

- ♦
  ## Method Detail

◊ getQueryID

```
int getQueryID()
```

Returns the QueryID of the new query.

The QueryID identifies a given query into the server and so its value must be agreed between the client and the server.

The server may subsequently returns `QueryCreateEvent.RESULT_WRONG_QUERY_ID` failure−code if it does not understand this value.

Used by:
`Query.create()`
Default value:
0
Acceptable values:
getQueryID() > 0
Returns:
the QueryID of the new query.
See Also:
`who sets this value`

◊ getQueryParameterEntity

```
Entity getQueryParameterEntity()
```

Returns the Query Parameter Entity of the new query.

The Query Parameter Entity is the argument of the new query and it is given to the server and so its meaning must be agreed between the client and the server.

The server may subsequently returns `QueryCreateEvent.RESULT_BAD_PARAMETERS` failure−code if it does not understand this value.

This value is optional: `null` is an acceptable value and, in this case, no argument is given to the server.

Used by:
`Query.create()`
Default value:
`null`
Acceptable values:
`true // any value`
Returns:
the QueryID of the new query.
See Also:
`who sets this value`

◊ **setQueryID**

```
void setQueryID(int queryID)
                throws IllegalStateException
```

Set/replace the QueryID of the new query.

Parameters:
`queryID` − the QueryID of the new query.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ **setQueryParameterEntity**

```
void setQueryParameterEntity(Entity queryParameter)
                             throws IllegalStateException
```

Set/replace the Query Parameter Entity of the new query.

Parameters:
`queryParameter` − the Query Parameter Entity of the new query.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface SubscriptionParam

- *All Superinterfaces:*
  Param

```
public interface SubscriptionParam
extends Param
```

Subscription parameter container.

This container of subscription parameters is created by `Context.makeSubscriptionParam()` and it is used by `Context.makeSubscription()`.

See Also:
Subscription Usage, Incremental Subscriptions, Partial Subscriptions, Acceptable Values

Field Summary

Fields

Modifier and Type	Field and Description
static int	***QUERY_TYPE_ALL*** Query selection code: all entities.
static int	***QUERY_TYPE_ON_TIME*** Query selection code: only current entities values.
static int	***QUERY_TYPE_PAST*** Query selection code: only past entities values and then idle event.
static int	***QUERY_TYPE_SET*** Query selection code: all entities that match a partial EntityKey.
static int	***SUBSCRIBE_FLOW_ALL*** Data transmission policy code: all variations sent by the server.
static int	***SUBSCRIBE_FLOW_LAST*** Data transmission policy code: only the most recent snapshot of each entity sent by the server.
static int	***SUBSCRIBE_MASKED_FLOW_ALL*** Data transmission policy code: all variations sent by the server.
static int	***SUBSCRIBE_MASKED_FLOW_LAST*** Data transmission policy code: only the most recent snapshot of each entity sent by the server.

Method Summary

Methods

Modifier and Type	Method and Description
int	***getEntityClassID***() Returns the requested EntityClassID of the EntityClass for the new Subscription.
TimeStamp	***getEntityClassTimeStamp***() Returns the EntityClass TimeStamp of the last past notification received by the client.
int	***getEntityClassVersion***() Returns the EntityClass version of the last past notification received by the client.
EntityKey	***getEntityKey***() Returns the partial EntityKey for the new subscription.

Filter	***getFilter***() Returns the optional filter for the new subscription.
int	***getFlow***() Returns the requested data transimission policy for the new subscription.
Mask	***getMask***() Returns the optional mask for the new subscription.
int	***getQueryType***() Returns the requested query selection criteria for the new subscription.
void	***setEntityClassID***(int entityClassID) Set/replace the requested EntityClassID of the EntityClass for the new Subscription.
void	***setEntityClassTimeStamp***(TimeStamp entityClassTimeStamp) Set/replace the EntityClass TimeStamp of the last past notification received by the client.
void	***setEntityClassVersion***(int entityClassVersion) Set/replace the EntityClass version of the last past notification received by the client.
void	***setEntityKey***(EntityKey entityKey) Set/replace the partial EntityKey for the new subscription.
void	***setFilter***(Filter filter) Set/replace the optional filter for the new subscription.
void	***setFlow***(int subscribeFlow) Set/replace the requested data transimission policy for the new subscription.
void	***setMask***(Mask mask) Set/replace the optional mask for the new subscription.
void	***setQueryType***(int queryType) Set/replace the requested query selection criteria for the new subscription.

◊ **Methods inherited from interface Param**

isBound

- ♦ **Field Detail**

◊ **SUBSCRIBE_FLOW_ALL**

```
static final int SUBSCRIBE_FLOW_ALL
```

Data transmission policy code: all variations sent by the server.

This value may be set with setFlow(int) and retrieved by getFlow().

See Also:
Constant Field Values

◊ SUBSCRIBE_FLOW_LAST

```
static final int SUBSCRIBE_FLOW_LAST
```

Data transmission policy code: only the most recent snapshot of each entity sent by the server.

This value may be set with `setFlow(int)` and retrieved by `getFlow()`.

See Also:
Constant Field Values

◊ SUBSCRIBE_MASKED_FLOW_ALL

```
static final int SUBSCRIBE_MASKED_FLOW_ALL
```

Data transmission policy code: all variations sent by the server.

The transmission is optimized in a transparent way in a manner that only non−zero values are sent. **Please note** that this is only an optimization hint and so it does not change the behaviour of `SubscriptionNotifyEvent.isMasked()`.

This value may be set with `setFlow(int)` and retrieved by `getFlow()`.

See Also:
Constant Field Values

◊ SUBSCRIBE_MASKED_FLOW_LAST

```
static final int SUBSCRIBE_MASKED_FLOW_LAST
```

Data transmission policy code: only the most recent snapshot of each entity sent by the server.

The transmission is optimized in a transparent way in a manner that only non−zero values are sent. **Please note** that this is only an optimization hint and so it does not change the behaviour of `SubscriptionNotifyEvent.isMasked()`.

This value may be set with `setFlow(int)` and retrieved by `getFlow()`.

See Also:
Constant Field Values

◊ QUERY_TYPE_ALL

```
static final int QUERY_TYPE_ALL
```

Query selection code: all entities.

This value may be set with `setQueryType(int)` and retrieved by `getQueryType()`.

See Also:
Constant Field Values

◊ QUERY_TYPE_SET

```
static final int QUERY_TYPE_SET
```

Query selection code: all entities that match a partial EntityKey.

This value may be set with `setQueryType(int)` and retrieved by `getQueryType()`.

See Also:
Constant Field Values

◊ QUERY_TYPE_PAST

```
static final int QUERY_TYPE_PAST
```

Query selection code: only past entities values and then idle event.

This value may be set with `setQueryType(int)` and retrieved by `getQueryType()`.

See Also:
Constant Field Values

◊ QUERY_TYPE_ON_TIME

```
static final int QUERY_TYPE_ON_TIME
```

Query selection code: only current entities values.

This value may be set with `setQueryType(int)` and retrieved by `getQueryType()`.

See Also:
Constant Field Values

♦ Method Detail

◊ getEntityClassID

```
int getEntityClassID()
```

Returns the requested EntityClassID of the EntityClass for the new Subscription.

Used by:
`Subscription.start()`
Default value:
0
Acceptable values:
`JFT.THIS.isRegistered(getEntityClassID())`
Returns:
the requested EntityClassID of the EntityClass for the new Subscription.
See Also:
`who sets this value`

◊ getEntityClassVersion

```
int getEntityClassVersion()
```

Returns the EntityClass version of the last past notification received by the client.

See Incremental Subscriptions to understand how to use this value.

Used by:

`Subscription.start()`

Default value:

0

Acceptable values:

`getEntityClassTimeStamp() == null ? value == 0 : value > 0`

Returns:

the EntityClassID of the EntityClass on which the new Subscription is made.

See Also:

`who sets this value`, Incremental Subscriptions

◊ getEntityClassTimeStamp

```
TimeStamp getEntityClassTimeStamp()
```

Returns the EntityClass TimeStamp of the last past notification received by the client.

See Incremental Subscriptions to understand how to use this value.

Used by:

`Subscription.start()`

Default value:

`null`

Acceptable values:

```
getEntityClassTimeStamp() == null
? getEntityClassVersion() == 0
: getEntityClassVersion() > 0
```

Returns:

the EntityClass TimeStamp of the last past notification received by the client.

See Also:

`who sets this value`, Incremental Subscriptions

◊ getQueryType

```
int getQueryType()
```

Returns the requested query selection criteria for the new subscription.

The requested query selection criteria may be one of:

- `QUERY_TYPE_ALL` to request notifications for all entities:
  past entities values,
  and then idle event,
  and then current values.
- `QUERY_TYPE_SET` to request notifications for entities that match a partial EntityKey:
  past entities values,
  and then idle event,
  and then current values.
- `QUERY_TYPE_PAST` to request notifications for only past entities values:
  past entities values,
  and then idle event.
- `QUERY_TYPE_ON_TIME` to request notifications for only current entities values:
  current values.

See Partial Subscriptions to understand how to use this value with partial subscriptions.

Used by:

`Subscription.start()`

Default value:

`QUERY_TYPE_ALL`

Acceptable values:

```
   getQueryType == QUERY_TYPE_ALL
|| getQueryType == QUERY_TYPE_SET
|| getQueryType == QUERY_TYPE_PAST
|| getQueryType == QUERY_TYPE_ON_TIME
```

Returns:

the requested query selection criteria for the new subscription.

See Also:

`who sets this value`, Partial Subscriptions

◊ getEntityKey

```
EntityKey getEntityKey()
```

Returns the partial EntityKey for the new subscription.

See Partial Subscriptions to understand how to use this value.

Used by:

`Subscription.start()`

Default value:

`null`

Acceptable values:

```
getQueryType() == QUERY_TYPE_SET
? (getEntityKey() != null &
getEntityKey().getEntityClassID() == getEntityClassID())
: getEntityKey() == null
```

Returns:

the partial EntityKey for the new subscription.

See Also:

`who sets this value`, Partial Subscriptions

◊ getFilter

```
Filter getFilter()
```

Returns the optional filter for the new subscription.

The filter is used to restrict (at the server level) the set of entities that will be notified.

Used by:

`Subscription.start()`

Default value:

`null`

Acceptable values:

```
getFilter() == null || getFilter().getStatus() ==
Filter.STATUS_CREATED &getQueryType() != QUERY_TYPE_SET
```

Returns:

the optional filter for the new subscription.

See Also:

`who sets this value`

◊ getFlow

```
int getFlow()
```

Returns the requested data transimission policy for the new subscription.

The requested data transimission policy may be one of:

- `SUBSCRIBE_FLOW_LAST` or `SUBSCRIBE_MASKED_FLOW_LAST` to request only the most recent snapshot of each entity sent by the server,
- `SUBSCRIBE_FLOW_ALL` or `SUBSCRIBE_MASKED_FLOW_ALL` to request all variations sent by the server.

With the first two values the quantity of the data sent by the server is adapted to the reception speed of the client. In practice, to each send operation, the server only sends the most recent image of the entity, respect the previous send.

The latter two values require the server to send all the variations of the entities of the subscribed class.

Basically `SUBSCRIBE_*FLOW_LAST` adapts the transmission resolution of the Server to the reception band of the Client. `SUBSCRIBE_*FLOW_ALL`, on the other hand, requires each intermediate variation of the Server DataBase to be submitted. Thus delays in the acquisition by the Client may occur due to the increase in load of the buffer nside the communication channel.

The `SUBSCRIBE_MASKED_FLOW_LAST` and `SUBSCRIBE_MASKED_FLOW_ALL` are equivalent to the non−`MASKED` version, apart the fact the transmission is optimized in a manner that only all non−zero values are sent from server to client. This optimization is transparent to the client application, e.g. the behavior of `SubscriptionNotifyEvent.isMasked()` does not change for `MASKED` or non−`MASKED` flow values.

Used by:

`Subscription.start()`

Default value:

`SUBSCRIBE_FLOW_ALL`

Acceptable values:

`getFlow() == SUBSCRIBE_FLOW_ALL || getFlow() == SUBSCRIBE_FLOW_LAST ||`
`getFlow() == SUBSCRIBE_MASKED_FLOW_ALL || getFlow() == SUBSCRIBE_MASKED_FLOW_LAST`

Returns:

the requested data transimission policy for the new subscription.

See Also:

`who sets this value`

◊ getMask

```
Mask getMask()
```

Returns the optional mask for the new subscription.

The mask is used to restrict (at the server level) the set of fields of entities that will be notified.

If this value is not-`null` then the Entitity returned by `SubscriptionNotifyEvent.getEntity()` will contain only the fields specified by this mask.

Used by:
`Subscription.start()`
Default value:
`null`
Acceptable values:
`getMask() == null || getMask().getEntityClassID() == getEntityClassID()`
Returns:
the optional mask for the new subscription.
See Also:
`who sets this value`

◊ setEntityClassID

```
void setEntityClassID(int entityClassID)
                      throws IllegalStateException
```

Set/replace the requested EntityClassID of the EntityClass for the new Subscription.

Parameters:
`entityClassID` − the requested EntityClassID of the EntityClass for the new Subscription.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setEntityClassVersion

```
void setEntityClassVersion(int entityClassVersion)
                           throws IllegalStateException
```

Set/replace the EntityClass version of the last past notification received by the client.

Parameters:
`entityClassVersion` − the requested EntityClassID of the EntityClass for the new Subscription.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setEntityClassTimeStamp

```
void setEntityClassTimeStamp(TimeStamp entityClassTimeStamp)
                             throws IllegalStateException
```

Set/replace the EntityClass TimeStamp of the last past notification received by the client.

Parameters:
`entityClassTimeStamp` − the EntityClass TimeStamp of the last past notification received by the client.

Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setQueryType

```
void setQueryType(int queryType)
                  throws IllegalStateException
```

Set/replace the requested query selection criteria for the new subscription.

Parameters:
`queryType` − the requested query selection criteria for the new subscription.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setEntityKey

```
void setEntityKey(EntityKey entityKey)
                  throws IllegalStateException
```

Set/replace the partial EntityKey for the new subscription.

Parameters:
`entityKey` − the partial EntityKey for the new subscription.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setFilter

```
void setFilter(Filter filter)
               throws IllegalStateException
```

Set/replace the optional filter for the new subscription.

Parameters:
`filter` − the optional filter for the new subscription.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setFlow

```
void setFlow(int subscribeFlow)
             throws IllegalStateException
```

Set/replace the requested data transimission policy for the new subscription.

Parameters:
`subscribeFlow` − the requested data transimission policy for the new subscription.
Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

◊ setMask

```
void setMask(Mask mask)
             throws IllegalStateException
```

Set/replace the optional mask for the new subscription.

Parameters:

`mask` − the optional mask for the new subscription.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface TransactionParam

- *All Superinterfaces:*
  Param

```
public interface TransactionParam
extends Param
```

Transaction parameter container.

This container of transaction parameters is created by `Context.makeTransactionParam()` and it is used by `Context.makeTransaction()`.

See Also:

Transaction Usage, Acceptable Values

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`ACTION_ENTITY_ADD`*** Action−code: request to add an entity on the server.
`static int`	***`ACTION_ENTITY_DEL`*** Action−code: request to logically remove an entity from the server.
`static int`	***`ACTION_ENTITY_KIL`*** Action−code: request to physically remove an entity from the server.

static int	***ACTION_ENTITY_RWT*** Action−code: request to replace an entity on the server.

Method Summary

Methods

Modifier and Type	Method and Description
int	***getAction***() Returns the requested action of a new transaction.
Entity	***getEntity***() Returns the Entity of a new transaction.
int	***getKeyID***() Returns the KeyID of a new transaction.
Mask	***getMask***() Returns the optional mask of a new transaction.
TransactionID	***getPendingTransactionID***() Returns the TransactionID of a past transaction.
boolean	***getResEntityRequired***() Returns the indication that client want an Entity come back from the server.
void	***setAction***(int action) Set/replace the requested action of a new transaction.
void	***setEntity***(Entity entity) Set/replace the Entity of a new transaction.
void	***setKeyID***(int keyID) Set/replace the KeyID of a new transaction.
void	***setMask***(Mask mask) Set/replace the optional mask of a new transaction.
void	***setPendingTransactionID***(TransactionID transactionID) Set/replace the TransactionID of a past transaction.
void	***setResEntityRequired***(boolean required) Set/replace the indication that client want an Entity come back from the server.

Methods inherited from interface Param

isBound

Field Detail

◊ **ACTION_ENTITY_ADD**

```
static final int ACTION_ENTITY_ADD
```

Action−code: request to add an entity on the server.

This value may be set with `setAction(int)` and retrieved by `getAction()`.

See Also:
Constant Field Values

◊ **ACTION_ENTITY_DEL**

```
static final int ACTION_ENTITY_DEL
```

Action−code: request to logically remove an entity from the server.

This value may be set with `setAction(int)` and retrieved by `getAction()`.

See Also:
Constant Field Values

◊ **ACTION_ENTITY_RWT**

```
static final int ACTION_ENTITY_RWT
```

Action−code: request to replace an entity on the server.

This value may be set with `setAction(int)` and retrieved by `getAction()`.

See Also:
Constant Field Values

◊ **ACTION_ENTITY_KIL**

```
static final int ACTION_ENTITY_KIL
```

Action−code: request to physically remove an entity from the server.

This value may be set with `setAction(int)` and retrieved by `getAction()`.

See Also:
Constant Field Values

♦ Method Detail

◊ **getAction**

```
int getAction()
```

Returns the requested action of a new transaction.

The requested action may be one of:

· `ACTION_ENTITY_ADD` to request to add an entity,

· `ACTION_ENTITY_DEL` to request to logically remove an entity,
· `ACTION_ENTITY_RWT` to request to rewrite an entity,
· `ACTION_ENTITY_KIL` to request to physically remove an entity.

Used by:
`Transaction.send()`
Default value:
`ACTION_ENTITY_ADD`
Acceptable values:

```
   getAction() == ACTION_ENTITY_ADD
|| getAction() == ACTION_ENTITY_DEL
|| getAction() == ACTION_ENTITY_RWT
|| getAction() == ACTION_ENTITY_KIL
```

Returns:
the requested action of a new transaction.
See Also:
`who sets this value`

◊ getKeyID

```
int getKeyID()
```

Returns the KeyID of a new transaction.

KeyID may be a primary key of the `EntityClass` of `getEntity()`, otherwise it is zero. In the first case all the KeyID fields of the Entity must be properly filled.

Used by:
`Transaction.send()`
Default value:
0
Acceptable values:

```
getKeyID() == 0
|| getEntity() != null &getEntity().iskey(getKeyID(),
true)
```

Returns:
the KeyID of a new transaction.
See Also:
`who sets this value`

◊ getEntity

```
Entity getEntity()
```

Returns the Entity of a new transaction.

This is the Entity on which the action will be done.

The fields of this Entity that **must** be properly filled are:

· all fields of the primary key described by `getKeyID()` (if `getKeyID()` is not zero),
· all fields of the mask described by `getMask()` (if `getMask()` is not `null`).
All others fields of this Entity **may** be properly filled.

Used by:
`Transaction.send()`
Default value:
`null`
Acceptable values:
`(getEntity() == null) != (getPendingTransactionID() == null)`
Returns:
the Entity of a new transaction.
See Also:
`who sets this value`

◊ getMask

```
Mask getMask()
```

Returns the optional mask of a new transaction.

When this value is not `null it describes which fields of the Entity must be properly filled, because they will be sent to the server.`

This mask and the Entity must refer to the same `EntityClass`.

Used by:
`Transaction.send()`
Default value:
`null`
Acceptable values:
`getMask() == null || getEntity() != null &getMask().getEntityClassID() == getEntity().getEntityClassID()`
Returns:
the optional mask of a new transaction.
See Also:
`who sets this value`

◊ getResEntityRequired

```
boolean getResEntityRequired()
```

Returns the indication that client want an Entity come back from the server.

When this value is `false` the server will not send back an Entity to the client inside the `TransactionSendEvent` and `TransactionQueryEvent` answers to the `Transaction.send()` and `Transaction.query()` requests.

Otherwise, when this value is `true` the server can choose to put an Entity in the answer. In this case the client may find a not−null returned value of `TransactionEvent.getEntity()`.

Used by:
`Transaction.send()` and `Transaction.query()`
Default value:
`false`

Acceptable values:

`true // any value`

Returns:

the indication that client want an Entity come back from the server.

See Also:

`who sets this value`

◊ getPendingTransactionID

```
TransactionID getPendingTransactionID()
```

Returns the TransactionID of a past transaction.

If this value is not `null` then `Context.makeTransaction()` will attempt to create a Transaction that refers a past transaction with this TransactionID. In this case this TransactionID must belong to the same Connection on which the Transaction will be created.

Used by:

`Transaction.getTransactionID()` and `Transaction.query()`

Default value:

`null`

Acceptable values:

`(getPendingTransactionID() == null) != (getEntity() == null)`

Returns:

the TransactionID of a past transaction.

See Also:

`who sets this value`

◊ setAction

```
void setAction(int action)
               throws IllegalStateException
```

Set/replace the requested action of a new transaction.

Parameters:

`action` − the requested action of a new transaction.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

◊ setKeyID

```
void setKeyID(int keyID)
              throws IllegalStateException
```

Set/replace the KeyID of a new transaction.

Parameters:

`keyID` − the KeyID of a new transaction.

Throws:

`IllegalStateException` − if this container is already bound.

See Also:

`default/current/acceptable values and their meaning`

◊ setEntity

```
void setEntity(Entity entity)
               throws IllegalStateException
```

Set/replace the Entity of a new transaction.

Parameters:
`entity` − the Entity of a new transaction.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setMask

```
void setMask(Mask mask)
             throws IllegalStateException
```

Set/replace the optional mask of a new transaction.

Parameters:
`mask` − the optional mask of a new transaction.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setResEntityRequired

```
void setResEntityRequired(boolean required)
                          throws IllegalStateException
```

Set/replace the indication that client want an Entity come back from the server.

Parameters:
`required` − the indication that client want an Entity come back from the server.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

◊ setPendingTransactionID

```
void setPendingTransactionID(TransactionID transactionID)
                             throws IllegalStateException
```

Set/replace the TransactionID of a past transaction.

Parameters:
`transactionID` − the TransactionID of a past transaction.
Throws:
`IllegalStateException` − if this container is already bound.
See Also:
`default/current/acceptable values and their meaning`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface TimeStamp

- *All Superinterfaces:*
  Serializable

```
public interface TimeStamp
extends Serializable
```

Interface that allows to represent a temporal indicator.

Each timestamp is represented by a couple of int:

- the number of seconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.
- an incremental counter used to make the time stamps univocal when it was generated within the same time unit.

This interface extend the `Serializable` interface in order to save and then re−create TimeStamp objects.

In alternative a programmer may save the two ints returned by `getDateTime()` and `getProg()` and then re−create the same TimeStamp object using the `JFT.makeTimeStamp()` with the 2 saved ints as parameters.

- ♦

Method Summary

Methods

Modifier and Type	Method and Description
`int`	***compareTo***`(TimeStamp timeStamp)` Compare two TimeStamps.
`int`	***getDateTime***`()` Returns the number of seconds since January 1, 1970, 00:00:00 GMT.
`int`	***getProg***`()` Returns the associated incremental counter.

- ♦

Method Detail

◊ getDateTime

```
int getDateTime()
```

Returns the number of seconds since January 1, 1970, 00:00:00 GMT.

To obtain a `Date` object use:
```
new Date(getDateTime()*1000L)
```

Returns:
the number of seconds since January 1, 1970, 00:00:00 GMT.

◊ **getProg**

```
int getProg()
```

Returns the associated incremental counter.

The incremental counter is used to make the time stamp univocal when it was generated within the same time unit.

Returns:
the associated incremental counter.

◊ **compareTo**

```
int compareTo(TimeStamp timeStamp)
```

Compare two TimeStamps.
Parameters:
`timeStamp` − the TimeStamp to be compared.
Returns:
a negative integer, zero, or a positive integer as this TimeStamp is less than, equal to, or greater than the specified TimeStamp.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface TransactionID

- *All Superinterfaces:*
  Serializable

```
public interface TransactionID
extends Serializable
```

Interface that allows to identify a Transaction.

Each Transaction is identified by a TransactionID made of:

- the client from which the transaction was sent,
  normally automatically generated by `Transaction.send()` with the value `ConnectionParam.getClientID()` of the Connection on which the transaction was sent.
- the couple ClientServiceID and BusinessServiceID to which the transaction was sent,
  normally automatically generated by `Transaction.send()` with the couple `ConnectionOpenEvent.getClientServiceID()` and `ConnectionOpenEvent.getBusinessServiceID()` returned on the Connection on which the transaction was sent.

- the client TimeStamp of when the transaction was sent,
  normally automatically generated by `Transaction.send()` with the sent time and an incremental counter.

All TransactionIDs share a `belongsTo()` method to check their compatibility with a given Connection.

This interface extend the `Serializable` interface in order to save and then re−create TransactionID objects.

In alternative a programmer may save the five ints returned by `getClientID()`, `getClientServiceID()`, `getBusinessServiceID()`, `getTimeStamp()` and then re−create the same TransactionID object using the `JFT.makeTransactionID()` with the 5 saved ints as parameters.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	***belongsTo***`(Connection connection)` Returns the compatibility of this TransactionID with a given Connection.
`int`	***getBusinessServiceID***`()` Returns the BusinessServiceID to which the transaction was sent.
`int`	***getClientID***`()` Returns the ClientID from which the transaction was sent.
`int`	***getClientServiceID***`()` Returns the ClientServiceID to which the transaction was sent.
`TimeStamp`	***getTimeStamp***`()` Returns the TimeStamp of when the transaction was sent.

Method Detail

◊ belongsTo

```
boolean belongsTo(Connection connection)
```

Returns the compatibility of this TransactionID with a given Connection.

Only a compatible transactionID can be successfully queried using a `TransactionParam.setPendingTransactionID()`.

A TransactionID is compatible with a Connection if:

- the TransactionID clientID is equals to the `Connection ClientID`,
- and the TransactionID ClientServiceID is equals to the `Connection ClientServiceID`,
- and the TransactionID BusinessServiceID is equals to the `Connection BusinessServiceID`.

Parameters:
`connection` − Connection to be checked for compatibility
Returns:
the compatibility of this TransactionID with a given Connection.
`false` is returned when the `connection` parameter is `null`,
or when the connection status is not `Connection.STATUS_CONNECTED`.
See Also:
`Transaction.query()`,
`TransactionParam.getPendingTransactionID()`

◊ getClientID

```
int getClientID()
```

Returns the ClientID from which the transaction was sent.

Returns:
the ClientID from which the transaction was sent.
See Also:
`ConnectionParam.getClientID()`

◊ getClientServiceID

```
int getClientServiceID()
```

Returns the ClientServiceID to which the transaction was sent.

Returns:
the ClientServiceID to which the transaction was sent.
See Also:
`ConnectionOpenEvent.getClientServiceID()`

◊ getBusinessServiceID

```
int getBusinessServiceID()
```

Returns the BusinessServiceID to which the transaction was sent.

Returns:
the BusinessServiceID to which the transaction was sent.
See Also:
`ConnectionOpenEvent.getBusinessServiceID()`

◊ getTimeStamp

```
TimeStamp getTimeStamp()
```

Returns the TimeStamp of when the transaction was sent.

Returns:
the TimeStamp of when the transaction was sent.
`null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft

Interface Tracer

```
public interface Tracer
```

Interface to be implemented in order to handle the library trace.

This interface may be bound to the library with the `JFT.setTraceMode(Tracer)` invocation.

♦ Method Summary

Methods

Modifier and Type	Method and Description
void	***onTrace***`(Date timeStamp, String module, int traceLevel, String message)` Called whenever a trace−message is availbale.

♦ Method Detail

◊ onTrace

```
void onTrace(Date timeStamp,
             String module,
             int traceLevel,
             String message)
```

Called whenever a trace−message is availbale.

This method is automatically invoked when the trace is enabled and the trace−message level `traceLevel` >= current trace level.

Parameters:

`timeStamp` − date.
`module` − JFT library module name.
`traceLevel` − trace−message level: one of availables levels described in `JFT.setTraceLevel()`.
`message` − trace−message.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

Package it.list.jft.event

Provides interfaces for dealing with different types of events and listeners.
See: Description

- Interface Summary

Interface	Description
ConnectionCloseEvent	Server−answer to `Connection.close()`.
ConnectionEvent	Generic event related to the Connection Lifecycle.
ConnectionListener	Interface to be implemented in order to handle the Connection Lifecycle.
ConnectionLostEvent	Event generated when the connection with the server crashed or when the server choose to terminate the connection.
ConnectionOpenEvent	Server−answer to `Connection.open()`.
EntityClassQueryEvent	
EntityClassQueryListener	
Event	Super−interface common to all events.
FilterCreateEvent	Server−answer to `Filter.create()`.
FilterDestroyEvent	Server−answer to `Filter.destroy()`.
FilterEvent	Generic event related to the Filter Lifecycle.
FilterListener	Interface to be implemented in order to handle the Filter Lifecycle.
FilterSetEvent	Server−answer to `Filter.set(java.lang.String)`.
Listener	Super−interface common to all listener interfaces.
MulticastConnectionEvent	
MulticastConnectionListener	
QueryCreateEvent	Server−answer to `Query.create()`.
QueryDestroyEvent	Server−answer to `Query.destroy()`.
QueryEvent	Generic event related to the Query Lifecycle.
QueryListener	Interface to be implemented in order to handle the Query Lifecycle.
QueryNotifyEvent	Event generated when a single entity (or the EOQ indication) of a query result−set is available.
QueryRowsEvent	Server−answer to `Query.queryRows()`.
SubscriptionEvent	Generic event related to the Subscription Lifecycle.
SubscriptionIdleEvent	Event generated when the flow of historical data is finished and the start of actual data is starting.

SubscriptionListener	Interface to be implemented in order to handle the Subscription Lifecycle.
SubscriptionNotifyEvent	Event generated when an actual or historical data or a server−answer to `Subscription.refreshEntity()` is available.
SubscriptionStartEvent	Server−answer to `Subscription.start()`.
SubscriptionStopEvent	Server−answer to `Subscription.stop()`.
TransactionEvent	Generic event related to the Transaction Lifecycle.
TransactionListener	Interface to be implemented in order to handle the Transaction Lifecycle.
TransactionQueryEvent	Server−answer to `Transaction.query()`.
TransactionSendEvent	Server−answer to `Transaction.send()`.

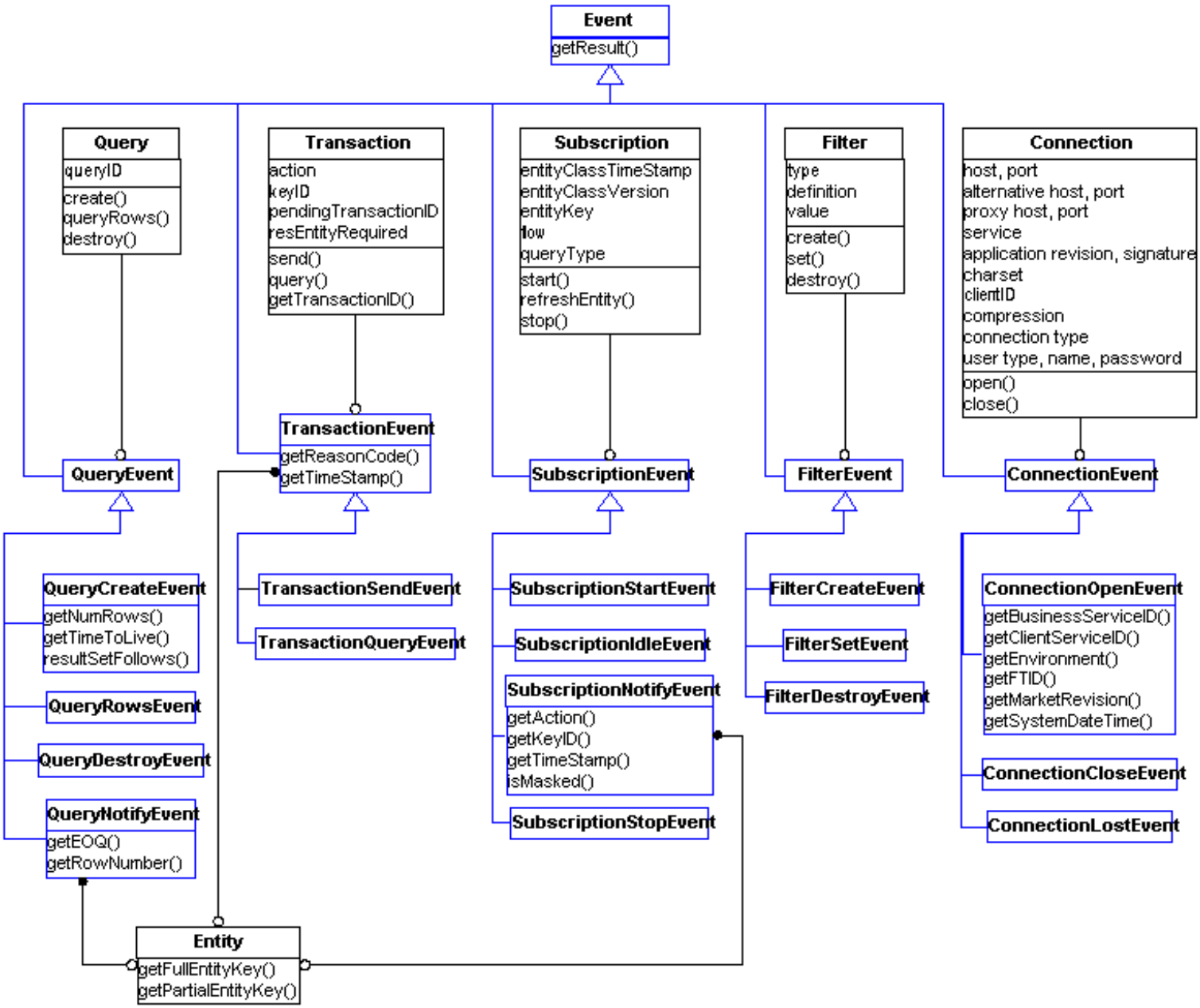
Package it.list.jft.event Description

Provides interfaces for dealing with different types of events and listeners.

Implementation of `Listener` sub−interfaces must be provided by the JFT application.
Implementation of `Listener` sub−interfaces are already provided by the JFT library.

See the hierarchy of this it.list.jft.event package and the `Event` and `Listener` documentation for details.

Package it.list.jft.event Data Model



The above figure is the UML representation of it.list.jft.event data model.
In blue all interfaces that are Events received from FastTrack server and handled by some Listener.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

Hierarchy For Package it.list.jft.event

Package Hierarchies:

- All Packages

Interface Hierarchy

- Event
  - ♦ ConnectionEvent
    - ◊ ConnectionCloseEvent
    - ◊ ConnectionLostEvent
    - ◊ ConnectionOpenEvent
  - ♦ EntityClassQueryEvent
  - ♦ FilterEvent
    - ◊ FilterCreateEvent
    - ◊ FilterDestroyEvent
    - ◊ FilterSetEvent
  - ♦ MulticastConnectionEvent
  - ♦ QueryEvent
    - ◊ QueryCreateEvent
    - ◊ QueryDestroyEvent
    - ◊ QueryNotifyEvent
    - ◊ QueryRowsEvent
  - ♦ SubscriptionEvent
    - ◊ SubscriptionIdleEvent
    - ◊ SubscriptionNotifyEvent
    - ◊ SubscriptionStartEvent
    - ◊ SubscriptionStopEvent
  - ♦ TransactionEvent
    - ◊ TransactionQueryEvent
    - ◊ TransactionSendEvent
- Listener
  - ♦ ConnectionListener
  - ♦ EntityClassQueryListener
  - ♦ FilterListener
  - ♦ MulticastConnectionListener
  - ♦ QueryListener
  - ♦ SubscriptionListener
  - ♦ TransactionListener

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface Event

- *All Known Subinterfaces:*
  ConnectionCloseEvent, ConnectionEvent, ConnectionLostEvent, ConnectionOpenEvent, EntityClassQueryEvent, FilterCreateEvent, FilterDestroyEvent, FilterEvent, FilterSetEvent, MulticastConnectionEvent, QueryCreateEvent, QueryDestroyEvent, QueryEvent, QueryNotifyEvent,

QueryRowsEvent, SubscriptionEvent, SubscriptionIdleEvent, SubscriptionNotifyEvent, SubscriptionStartEvent, SubscriptionStopEvent, TransactionEvent, TransactionQueryEvent, TransactionSendEvent

---

```
public interface Event
```

Super−interface common to all events.

Events related to the various communication objects (`Connection`, `Filter`, `Query`, `Subscription`, `Transaction`) must be handled by the methods of the corresponding `Listener`.

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***`RESULT_GENERIC_ERROR`*** Generic failure−code returned by the server when a more specific error is not available.
`static int`	***`RESULT_OK`*** Positive answer returned by the server when the operation completed successfully.

Method Summary

Methods

Modifier and Type	Method and Description
`int`	***`getResult`***`()` Returns the server−answer associated to this event.

Field Detail

◊ RESULT_OK

```
static final int RESULT_OK
```

Positive answer returned by the server when the operation completed successfully.

This value is returned by `getResult()`.

See Also:
Constant Field Values

◊ RESULT_GENERIC_ERROR

```
static final int RESULT_GENERIC_ERROR
```

Generic failure−code returned by the server when a more specific error is not available.

This value is returned by `getResult()`.

See Also:
Constant Field Values

♦ Method Detail

◊ getResult

```
int getResult()
```

Returns the server−answer associated to this event.

Each event generated by the server transports a server−answer that specifies how the corresponding operation was performed on the server.

E.g.: a `ConnectionOpenEvent` is generated by the server as an answer to the `Connection.open()` sent by the client. If the opening was OK then the server−answer (returned by this method) is `RESULT_OK`, otherwise a generic (`RESULT_GENERIC_ERROR`) or specific (ConnectionOpenEvent result codes) error is returned.

Returns:
the server−answer associated to this event.
It may be `RESULT_OK` or `RESULT_GENERIC_ERROR` or a different value indicating a specific error that is described in the specific event documentation.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface ConnectionEvent

- *All Superinterfaces:*
  Event

All Known Subinterfaces:
ConnectionCloseEvent, ConnectionLostEvent, ConnectionOpenEvent

---

```
public interface ConnectionEvent
extends Event
```

Generic event related to the Connection Lifecycle.

Events related to this super−interface must be handled by the methods of `ConnectionListener`.

- ♦ **Field Summary**

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
`Connection`	***`getConnection`***`()` Returns the connection associated to this event.

  ◊ **Methods inherited from interface Event**

  `getResult`

- ♦ **Method Detail**

  ◊ **getConnection**

  `Connection getConnection()`

  Returns the connection associated to this event.

  Returns:
  the connection associated to this event.
  `null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface ConnectionCloseEvent

- *All Superinterfaces:*
  ConnectionEvent, Event

```
public interface ConnectionCloseEvent
extends ConnectionEvent
```

Server−answer to `Connection.close()`.

This event must be handled by `ConnectionListener.onConnectionClose()`.

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface Event**

    `RESULT_GENERIC_ERROR, RESULT_OK`
  - ♦ **Method Summary**
  - ◊ **Methods inherited from interface ConnectionEvent**

    `getConnection`
  - ◊ **Methods inherited from interface Event**

    `getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface ConnectionLostEvent

- *All Superinterfaces:*
  ConnectionEvent, Event

```
public interface ConnectionLostEvent
extends ConnectionEvent
```

Event generated when the connection with the server crashed or when the server choose to terminate the connection.

This event must be handled by `ConnectionListener.onConnectionLost()`.

With this event the server result is always `Event.RESULT_GENERIC_ERROR`.

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface Event**

    `RESULT_GENERIC_ERROR, RESULT_OK`
  - ♦ **Method Summary**

◊ Methods inherited from interface ConnectionEvent

`getConnection`

◊ Methods inherited from interface Event

`getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface ConnectionOpenEvent

- *All Superinterfaces:*
  ConnectionEvent, Event

```
public interface ConnectionOpenEvent
extends ConnectionEvent
```

Server−answer to `Connection.open()`.

This event must be handled by `ConnectionListener.onConnectionOpen()`.

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***RESULT_ALREADY_LOGGED*** Failure−code: user already logged.
`static int`	***RESULT_EXCEED_SESSION*** Failure−code: too much open sessions with the server.
`static int`	***RESULT_INVALID_AUTH_KEY*** Failure−code: bad configuration key associated to the connection.
`static int`	***RESULT_INVALID_CLIENTID*** Failure−code: bad client ID associated to the connection.
`static int`	***RESULT_INVALID_PASSWORD*** Failure−code: bad password associated to the connection.
`static int`	***RESULT_INVALID_PROFILE*** Failure−code: invalid profile.
`static int`	***RESULT_INVALID_REVISION*** Failure−code: bad application version associated to the connection.
`static int`	***RESULT_INVALID_SERVER_STATUS***

	Failure−code: the server is in a status (e.g. still in a start−up state) in which connections are not allowed.
static int	***RESULT_INVALID_SERVICE*** Failure−code: bad user type associated to the connection.
static int	***RESULT_INVALID_USERNAME*** Failure−code: bad user name associated to the connection.
static int	***RESULT_INVALID_USERTYPE*** Failure−code: bad user type associated to the connection.

◊

Fields inherited from interface Event

RESULT_GENERIC_ERROR, RESULT_OK

◆

Method Summary

Methods

Modifier and Type	Method and Description
int	***getActiveConnectionType***()
int	***getBusinessServiceID***() Returns the business service ID associated to this connection.
int	***getClientServiceID***() Returns the client service ID associated to this connection.
int	***getEnvironment***() Returns an indication of the FastTrack server environment (e.g.: Production, Testing, etc...).
int	***getFTID***() Returns the FastTrack Server ID.
int[]	***getMarketRevision***() Returns the version of the server.
int	***getSystemDate***() Returns the system date of the server.
Date	***getSystemDateTime***() Returns the system date and time of the server.
int	***getSystemTime***() Returns the system time of the server.

◊

Methods inherited from interface ConnectionEvent

getConnection

◊ Methods inherited from interface Event

`getResult`

- ♦

Field Detail

◊ RESULT_INVALID_PASSWORD

```
static final int RESULT_INVALID_PASSWORD
```

Failure−code: bad password associated to the connection.
See Also:
Constant Field Values

◊ RESULT_INVALID_USERNAME

```
static final int RESULT_INVALID_USERNAME
```

Failure−code: bad user name associated to the connection.
See Also:
Constant Field Values

◊ RESULT_INVALID_REVISION

```
static final int RESULT_INVALID_REVISION
```

Failure−code: bad application version associated to the connection.
See Also:
Constant Field Values

◊ RESULT_ALREADY_LOGGED

```
static final int RESULT_ALREADY_LOGGED
```

Failure−code: user already logged.
See Also:
Constant Field Values

◊ RESULT_INVALID_CLIENTID

```
static final int RESULT_INVALID_CLIENTID
```

Failure−code: bad client ID associated to the connection.
See Also:
Constant Field Values

◊ RESULT_INVALID_SERVER_STATUS

```
static final int RESULT_INVALID_SERVER_STATUS
```

Failure−code: the server is in a status (e.g. still in a start−up state) in which connections are not allowed.
See Also:
Constant Field Values

◊ **RESULT_EXCEED_SESSION**

```
static final int RESULT_EXCEED_SESSION
```

Failure−code: too much open sessions with the server.
See Also:
Constant Field Values

◊ **RESULT_INVALID_PROFILE**

```
static final int RESULT_INVALID_PROFILE
```

Failure−code: invalid profile.
See Also:
Constant Field Values

◊ **RESULT_INVALID_AUTH_KEY**

```
static final int RESULT_INVALID_AUTH_KEY
```

Failure−code: bad configuration key associated to the connection.
See Also:
Constant Field Values

◊ **RESULT_INVALID_USERTYPE**

```
static final int RESULT_INVALID_USERTYPE
```

Failure−code: bad user type associated to the connection.
See Also:
Constant Field Values

◊ **RESULT_INVALID_SERVICE**

```
static final int RESULT_INVALID_SERVICE
```

Failure−code: bad user type associated to the connection.
See Also:
Constant Field Values

♦

Method Detail

◊ **getClientServiceID**

```
int getClientServiceID()
```

Returns the client service ID associated to this connection.

The client service ID is one of the elements that identify a `TransactionID`.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:
the client service ID associated to this connection.
−1 is returned when the result is not `Event.RESULT_OK`.
See Also:

`TransactionID.belongsTo(it.list.jft.Connection)`

◊ getBusinessServiceID

```
int getBusinessServiceID()
```

Returns the business service ID associated to this connection.

The business service ID is one of the elements that identify a `TransactionID`.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

the business service ID associated to this connection.
−1 is returned when the result is not `Event.RESULT_OK`.

See Also:

`TransactionID.belongsTo(it.list.jft.Connection)`

◊ getSystemDate

```
int getSystemDate()
```

Returns the system date of the server.

The value returned reflects the date in which the server opened the connection.

The returned value is an `int` whose decimal representation is: YYYYMMDD (i.e.: year*10000 + month*100 + day).

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

the system date of the server.
−1 is returned when the result is not `Event.RESULT_OK`.

See Also:

`getSystemDateTime()`

◊ getSystemTime

```
int getSystemTime()
```

Returns the system time of the server.

The value returned reflects the time in which the server opened the connection.

The returned value is an `int` whose decimal representation is: HHMMSScc (i.e.: hours*1000000 + minutes*10000 + seconds*100 + hundreds).

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

the system time of the server.
−1 is returned when the result is not `Event.RESULT_OK`.

See Also:

`getSystemDateTime()`

◊ getSystemDateTime

```
Date getSystemDateTime()
```

Returns the system date and time of the server.

The value returned reflects the date and time in which the server opened the connection.

This method must be called only when the result is `Event.RESULT_OK`.

This utility method is defined in terms of `getSystemDate()` and `getSystemTime()` as follows:

```
int date = getSystemDate();
int time = getSystemTime();
if(date == -1 || time == -1)
  return -1;
Calendar cal = Calendar.getInstance();
cal.set(date/10000, date%10000/100-1, date%100, time/1000000,
time%1000000/10000, time%1000/100);
return new Date(cal.getTimeInMillis() + time%100*10);
```

Returns:

the system date and time of the server.
−1 is returned when the result is not `Event.RESULT_OK`.

◊ getMarketRevision

```
int[] getMarketRevision()
```

Returns the version of the server.

A version is always represented by a three−dimensional array; e.g. the version 2.0.3 is represented by:

```
int[] version = {2, 0, 3};
```

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

the version of the server.
`null` is returned when the result is not `Event.RESULT_OK`.

◊ getFTID

```
int getFTID()
```

Returns the FastTrack Server ID.

Each FastTrack server in the world is identified by an unique FastTrack Server ID.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

the FastTrack Server ID.
−1 is returned when the result is not `Event.RESULT_OK`.

◊ **getEnvironment**

```
int getEnvironment()
```

Returns an indication of the FastTrack server environment (e.g.: Production, Testing, etc...).

the precise meaning of this value depends on the particular FastTrack server and it is documented in the corresponding manual.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:
an indication of the FastTrack server environment.
−1 is returned when the result is not `Event.RESULT_OK`.

◊ **getActiveConnectionType**

```
int getActiveConnectionType()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface EntityClassQueryEvent

- *All Superinterfaces:*
  Event

```
public interface EntityClassQueryEvent
extends Event
```

- ♦ **Field Summary**

  Fields

Modifier and Type	Field and Description
static int	***RESULT_ENTITY_CLASS_NOT_AVAILABLE***

◊ **Fields inherited from interface Event**

`RESULT_GENERIC_ERROR, RESULT_OK`

♦ **Method Summary**

Methods

Modifier and Type	Method and Description
EntityClass	***getEntityClass***()

EntityClassQuery **getEntityClassQuery**()

boolean **isEnum**()

◊ **Methods inherited from interface Event**

getResult

- ♦ **Field Detail**

◊ **RESULT_ENTITY_CLASS_NOT_AVAILABLE**

```
static final int RESULT_ENTITY_CLASS_NOT_AVAILABLE
```

See Also:
Constant Field Values

♦ **Method Detail**

◊ **getEntityClassQuery**

```
EntityClassQuery getEntityClassQuery()
```

◊ **getEntityClass**

```
EntityClass getEntityClass()
```

◊ **isEnum**

```
boolean isEnum()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface FilterEvent

- *All Superinterfaces:*
Event

All Known Subinterfaces:
FilterCreateEvent, FilterDestroyEvent, FilterSetEvent

```
public interface FilterEvent
extends Event
```

Generic event related to the Filter Lifecycle.

Events related to this super−interface must be handled by the methods of `FilterListener`.

- ♦ **Field Summary**

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
`Filter`	***`getFilter`***`()` Returns the filter associated to this event.

  ◊ **Methods inherited from interface Event**

  `getResult`

- ♦ **Method Detail**

  ◊ **getFilter**

  `Filter getFilter()`

  Returns the filter associated to this event.

  Returns:
  the filter associated to this event.
  `null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface FilterCreateEvent

- *All Superinterfaces:*
  Event, FilterEvent

```
public interface FilterCreateEvent
extends FilterEvent
```

Server−answer to `Filter.create()`.

This event must be handled by `FilterListener.onFilterCreate()`.

- ♦ **Field Summary**

  Fields

Modifier and Type	Field and Description
`static int`	***RESULT_FILTER_NOT_IMPLEMENTED*** Failure−code: Filtering is not implemented by the server.
`static int`	***RESULT_INVALID_ENTITY_CLASS_ID*** Failure−code: Entity Class ID is invalid.
`static int`	***RESULT_INVALID_FILTER_LEN*** Failure−code: filter definition too long.
`static int`	***RESULT_INVALID_FILTER_TYPE*** Failure−code: filter type is invalid.
`static int`	***RESULT_SYNTAX_ERROR*** Failure−code: syntax error in filter definition.

  - ◊ **Fields inherited from interface Event**

    `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
`int`	***getFilterID***`()`

  - ◊ **Methods inherited from interface FilterEvent**

    `getFilter`

  - ◊ **Methods inherited from interface Event**

    `getResult`

- ♦ **Field Detail**

  - ◊ **RESULT_SYNTAX_ERROR**

    ```
    static final int RESULT_SYNTAX_ERROR
    ```

    Failure−code: syntax error in filter definition.

    The server is not able to understand the given filter definition.

See Also:
Constant Field Values

◊ **RESULT_INVALID_FILTER_LEN**

```
static final int RESULT_INVALID_FILTER_LEN
```

Failure−code: filter definition too long.
See Also:
Constant Field Values

◊ **RESULT_INVALID_ENTITY_CLASS_ID**

```
static final int RESULT_INVALID_ENTITY_CLASS_ID
```

Failure−code: Entity Class ID is invalid.
See Also:
Constant Field Values

◊ **RESULT_INVALID_FILTER_TYPE**

```
static final int RESULT_INVALID_FILTER_TYPE
```

Failure−code: filter type is invalid.
See Also:
Constant Field Values

◊ **RESULT_FILTER_NOT_IMPLEMENTED**

```
static final int RESULT_FILTER_NOT_IMPLEMENTED
```

Failure−code: Filtering is not implemented by the server.
See Also:
Constant Field Values

♦

Method Detail

◊ **getFilterID**

```
int getFilterID()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface FilterDestroyEvent

- *All Superinterfaces:*
  Event, FilterEvent

```
public interface FilterDestroyEvent
extends FilterEvent
```

Server−answer to `Filter.destroy()`.

This event must be handled by `FilterListener.onFilterDestroy()`.

- ♦
  ## Field Summary
  - ◊
    ### Fields inherited from interface Event

    `RESULT_GENERIC_ERROR`, `RESULT_OK`
  - ♦
    ## Method Summary
    - ◊
      ### Methods inherited from interface FilterEvent

      `getFilter`
    - ◊
      ### Methods inherited from interface Event

      `getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface FilterSetEvent

- *All Superinterfaces:*
  Event, FilterEvent

```
public interface FilterSetEvent
extends FilterEvent
```

Server−answer to `Filter.set(java.lang.String)`.

This event must be handled by `FilterListener.onFilterSet()`.

- ♦
  ## Field Summary

  Fields

Modifier and Type	Field and Description
`static int`	***`RESULT_ALREADY_SET`*** Failure−code: filter value already set.
`static int`	***`RESULT_INVALID_FILTER_LEN`*** Failure−code: filter value too long.

static int	***RESULT_SYNTAX_ERROR*** Failure−code: syntax error in filter value.

◊

Fields inherited from interface Event

RESULT_GENERIC_ERROR, RESULT_OK

Method Summary

◊

Methods inherited from interface FilterEvent

getFilter

◊

Methods inherited from interface Event

getResult

Field Detail

◊ **RESULT_SYNTAX_ERROR**

```
static final int RESULT_SYNTAX_ERROR
```

Failure−code: syntax error in filter value.

The server is not able to understand the given filter value.

See Also:
Constant Field Values

◊ **RESULT_INVALID_FILTER_LEN**

```
static final int RESULT_INVALID_FILTER_LEN
```

Failure−code: filter value too long.
See Also:
Constant Field Values

◊ **RESULT_ALREADY_SET**

```
static final int RESULT_ALREADY_SET
```

Failure−code: filter value already set.
See Also:
Constant Field Values

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface MulticastConnectionEvent

- *All Superinterfaces:*
  Event

```
public interface MulticastConnectionEvent
extends Event
```

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface Event**

    RESULT_GENERIC_ERROR, RESULT_OK
- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
int	***getAction***()
Entity	***getEntity***()
int	***getKeyID***()
MulticastConnection	***getMulticastConnection***()
TimeStamp	***getTimeStamp***()

  - ◊ **Methods inherited from interface Event**

    getResult
- ♦ **Method Detail**
  - ◊ **getMulticastConnection**

    ```
    MulticastConnection getMulticastConnection()
    ```
  - ◊ **getEntity**

    ```
    Entity getEntity()
    ```
  - ◊ **getAction**

    ```
    int getAction()
    ```

◊ **getKeyID**

```
int getKeyID()
```

◊ **getTimeStamp**

```
TimeStamp getTimeStamp()
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface QueryEvent

- *All Superinterfaces:*
  Event

All Known Subinterfaces:
QueryCreateEvent, QueryDestroyEvent, QueryNotifyEvent, QueryRowsEvent

---

```
public interface QueryEvent
extends Event
```

Generic event related to the Query Lifecycle.

Events related to this super−interface must be handled by the methods of `QueryListener`.

- ♦ **Field Summary**

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

♦ **Method Summary**

Methods

Modifier and Type	Method and Description
`Query`	***`getQuery`***`()` Returns the query associated to this event.

◊ **Methods inherited from interface Event**

`getResult`

- ♦ **Method Detail**

  ◊ **getQuery**

  `Query getQuery()`

  Returns the query associated to this event.

  Returns:
  the query associated to this event.
  `null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface QueryCreateEvent

- *All Superinterfaces:*
  Event, QueryEvent

```
public interface QueryCreateEvent
extends QueryEvent
```

Server−answer to `Query.create()`.

This event must be handled by `QueryListener.onQueryCreate()`.

- ♦ **Field Summary**

  Fields

Modifier and Type	Field and Description
`static int`	***RESULT_BAD_PARAMETERS*** Failure−code: bad parameter associated to the query.
`static int`	***RESULT_WRONG_QUERY_ID*** Failure−code: bad QueryID associated to the query.

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
int	***getNumRows***() Returns the number of rows in the result−set as computed by the server.
int	***getTimeToLive***() Returns the interval time (in seconds) during which the server cache the result−set.
boolean	***resultSetFollows***() Returns the indication that the result−set is immediately available.

◊ **Methods inherited from interface QueryEvent**

getQuery

◊ **Methods inherited from interface Event**

getResult

- ♦ **Field Detail**

◊ **RESULT_BAD_PARAMETERS**

```
static final int RESULT_BAD_PARAMETERS
```

Failure−code: bad parameter associated to the query.

The server is not able to process the query with the given parameter.

See Also:
Constant Field Values

◊ **RESULT_WRONG_QUERY_ID**

```
static final int RESULT_WRONG_QUERY_ID
```

Failure−code: bad QueryID associated to the query.

The server is not able to process the query with the given QueryID.

See Also:
Constant Field Values

- ♦ **Method Detail**

◊ getNumRows

```
int getNumRows()
```

Returns the number of rows in the result−set as computed by the server.

If the server does not known this number then −1 is returned: e.g. when `resultSetFollows()` returns `false`.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

number of rows in the result−set: N >= 0 means that the result−set contains N elements.
−1 is returned when the server is unable to compute this number, or
when the result is not `Event.RESULT_OK`.

◊ getTimeToLive

```
int getTimeToLive()
```

Returns the interval time (in seconds) during which the server cache the result−set.

During this interval the client may issue `Query.queryRows()` invocations to obtains the various parts of the result−set.
The value returned is meaningfull only if `resultSetFollows()` returns `false`.

If the server does not known this interval then zero is returned: e.g. when `resultSetFollows()` returns `true`.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

number of seconds during which the server cache the result−set.
Zero is returned when the interval is not known, or
when the `resultSetFollows()` is `true`, or
when the result is not `Event.RESULT_OK`.

◊ resultSetFollows

```
boolean resultSetFollows()
```

Returns the indication that the result−set is immediately available.

`true` means that `QueryListener.onQueryNotify()` will be automatically called N+1 times:

- N>=0 times (with the `EOQ` indication equals to `false`) for each of the N rows in the result−set;
- + 1 additional time (with the `EOQ` indication equals to `true`) to indicate the end of the result−set.

`false` means that `QueryListener.onQueryNotify()` will not be automatically called (as result of `Query.create()`) and the client must issue a specific `Query.queryRows()` to obtains a subset of the result−set.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

indication regarding the immediate availability of the result−set. `false` is returned when result is not `Event.RESULT_OK`.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface QueryDestroyEvent

- *All Superinterfaces:*
  Event, QueryEvent

```
public interface QueryDestroyEvent
extends QueryEvent
```

Server−answer to `Query.destroy()`.

This event must be handled by `QueryListener.onQueryDestroy()`.

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface Event**

    `RESULT_GENERIC_ERROR, RESULT_OK`
- ♦ **Method Summary**
  - ◊ **Methods inherited from interface QueryEvent**

    `getQuery`
  - ◊ **Methods inherited from interface Event**

    `getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface QueryNotifyEvent

- *All Superinterfaces:*
  Event, QueryEvent

```
public interface QueryNotifyEvent
extends QueryEvent
```

Event generated when a single entity (or the EOQ indication) of a query result−set is available.

This event must be handled by `QueryListener.onQueryNotify()`.

With this event the server result is always `Event.RESULT_OK`.

- ♦
Field Summary

Fields

Modifier and Type	Field and Description
`static int`	***ACTION_ENTITY_ADD*** Action−code: entity is on the server.
`static int`	***ACTION_ENTITY_DEL*** Action−code: entity logically removed on the server.

◊
Fields inherited from interface Event

`RESULT_GENERIC_ERROR, RESULT_OK`

♦
Method Summary

Methods

Modifier and Type	Method and Description
`int`	***getAction***`()` Returns the server action associated with the entity available on this event.
`Entity`	***getEntity***`()` Returns the entity of the current row in the result−set.
`boolean`	***getEOQ***`()` Returns the indication that the result−set is ended.
`int`	***getRowNumber***`()` Returns the index (1−based) of the current row in the result−set.
`TimeStamp`	***getTimeStamp***`()` Returns the entity timestamp.

◊
Methods inherited from interface QueryEvent

`getQuery`

◊ Methods inherited from interface Event

`getResult`

♦ Field Detail

◊ ACTION_ENTITY_ADD

```
static final int ACTION_ENTITY_ADD
```

Action−code: entity is on the server.

This value may be returned by `getAction()`.

See Also:
Constant Field Values

◊ ACTION_ENTITY_DEL

```
static final int ACTION_ENTITY_DEL
```

Action−code: entity logically removed on the server.

This value may be returned by `getAction()`.

See Also:
Constant Field Values

♦ Method Detail

◊ getRowNumber

```
int getRowNumber()
```

Returns the index (1−based) of the current row in the result−set.

The index of the first row of a result−set returned by `Query.create()` is 1.
The index of the first row of a result−set returned by `Query.queryRows()` is `firstRow`.

This method must be called only when the EOQ indication is `false`.

Returns:
the index (1−based) of the current row in the result−set.
Zero is returned when the EOQ indication is `true`
or when the information is not available (some primitives FastTrack services always return zero for every rows of the returned result−set).

◊ getTimeStamp

```
TimeStamp getTimeStamp()
```

Returns the entity timestamp.

This method must be called only when the EOQ indication is `false`.

Returns:
the entity timestamp.
`null` is returned when the EOQ indication is `true` or if entity timestamp is not sent by the server.

◊ getEntity

```
Entity getEntity()
```

Returns the entity of the current row in the result−set.

This method must be called only when the EOQ indication is `false`.

Returns:
the entity of the current row in the result−set.
`null` is returned when the EOQ indication is `true`.

◊ getAction

```
int getAction()
```

Returns the server action associated with the entity available on this event.

The possible returned values are described in the Field Summary section.

Returns:
the server action associated with the entity available on this event if the Entity is not `null`.

◊ getEOQ

```
boolean getEOQ()
```

Returns the indication that the result−set is ended.

If the query result−set computed by the server, as an aswer to a correct `Query.create()` or `Query.queryRows()`, is composed by N entities then the `QueryListener.onQueryNotify()` method (with a `QueryNotifyEvent` as parameter) will be invoked N+1 times: N times with each of the N entities (and this EOQ indication equals to `false`) and one more time with this EOQ indication equals to `true`.

Returns:
the indication that the result−set is ended.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface QueryRowsEvent

- *All Superinterfaces:*
  Event, QueryEvent

```
public interface QueryRowsEvent
extends QueryEvent
```

Server−answer to `Query.queryRows()`.

This event must be handled by `QueryListener.onQueryRows()`.

- ♦
  ## Field Summary

  Fields

Modifier and Type	Field and Description
`static int`	***RESULT_WRONG_FIRST_ROW*** Failure−code: bad `firstRow` parameter of `Query.queryRows()`.
`static int`	***RESULT_WRONG_ROW_NUMBER*** Failure−code: bad `rowNumber` parameter of `Query.queryRows()`.

  ◊
  ### Fields inherited from interface Event

  `RESULT_GENERIC_ERROR, RESULT_OK`

  ♦
  ## Method Summary

  ◊
  ### Methods inherited from interface QueryEvent

  `getQuery`

  ◊
  ### Methods inherited from interface Event

  `getResult`

- ♦
  ## Field Detail

  ◊ **RESULT_WRONG_FIRST_ROW**

  ```
  static final int RESULT_WRONG_FIRST_ROW
  ```

  Failure−code: bad `firstRow` parameter of `Query.queryRows()`.
  See Also:

Constant Field Values

◊ **RESULT_WRONG_ROW_NUMBER**

```
static final int RESULT_WRONG_ROW_NUMBER
```

Failure−code: bad `rowNumber` parameter of `Query.queryRows()`.
See Also:
Constant Field Values

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface SubscriptionEvent

- *All Superinterfaces:*
  Event

All Known Subinterfaces:
SubscriptionIdleEvent, SubscriptionNotifyEvent, SubscriptionStartEvent, SubscriptionStopEvent

---

```
public interface SubscriptionEvent
extends Event
```

Generic event related to the Subscription Lifecycle.

Events related to this super−interface must be handled by the methods of `SubscriptionListener`.

- ♦ **Field Summary**

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
`Subscription`	***`getSubscription`***`()` Returns the subscription associated to this event.

  ◊ **Methods inherited from interface Event**

  `getResult`

- ♦ **Method Detail**

  ◊ **getSubscription**

  `Subscription getSubscription()`

  Returns the subscription associated to this event.

  Returns:
  the subscription associated to this event.
  `null` is never returned.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface SubscriptionIdleEvent

- *All Superinterfaces:*
  Event, SubscriptionEvent

```
public interface SubscriptionIdleEvent
extends SubscriptionEvent
```

Event generated when the flow of historical data is finished and the start of actual data is starting.

This event is generated only if the type of query of the subscription is not `SubscriptionParam.QUERY_TYPE_ON_TIME`.

This event must be handled by `SubscriptionListener.onSubscriptionIdle()`.

- ♦ **Field Summary**

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  ◊ **Methods inherited from interface SubscriptionEvent**

  `getSubscription`

◊ Methods inherited from interface Event

`getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface SubscriptionNotifyEvent

- *All Superinterfaces:*
  Event, SubscriptionEvent

```
public interface SubscriptionNotifyEvent
extends SubscriptionEvent
```

Event generated when an actual or historical data or a server−answer to `Subscription.refreshEntity()` is available.

This event must be handled by `SubscriptionListener.onSubscriptionNotify()`.

With this event the server result is always `Event.RESULT_OK`.

- ♦

Field Summary

Fields

Modifier and Type	Field and Description
static int	***ACTION_ENTITY_ADD*** Action−code: entity added on the server, or just returned as an answer to `Subscription.refreshEntity()`.
static int	***ACTION_ENTITY_DEL*** Action−code: entity logically removed on the server.
static int	***ACTION_ENTITY_KIL*** Action−code: entity physically removed on the server.
static int	***ACTION_ENTITY_RWT*** Action−code: entity rewritten on the server.

◊ Fields inherited from interface Event

`RESULT_GENERIC_ERROR, RESULT_OK`

Method Summary

Methods

Modifier and Type	Method and Description
int	***getAction***() Returns the server action associated with the entity available on this event.
Entity	***getEntity***() Returns the entity available on this event.
int	***getKeyID***() Returns the index of the key on the basis of which the server has carried out getAction().
TimeStamp	***getTimeStamp***() Returns the timestamp associated with the entity available on this event.
boolean	***isMasked***() Returns the indication that some fields of getEntity() may be missings.

Methods inherited from interface SubscriptionEvent

getSubscription

Methods inherited from interface Event

getResult

Field Detail

ACTION_ENTITY_ADD

```
static final int ACTION_ENTITY_ADD
```

Action−code: entity added on the server, or just returned as an answer to Subscription.refreshEntity().

Historical data are always tagged as ACTION_ENTITY_ADD.

This value may be returned by getAction().

See Also:
Constant Field Values

ACTION_ENTITY_DEL

```
static final int ACTION_ENTITY_DEL
```

Action−code: entity logically removed on the server.

In this case the Entity returned by `getEntity()` is generally undefined on any fields apart from those associated with `getKeyID()`.

Historical data are never tagged as `ACTION_ENTITY_DEL`.

This value may be returned by `getAction()`.

See Also:
Constant Field Values

◊ **ACTION_ENTITY_RWT**

```
static final int ACTION_ENTITY_RWT
```

Action−code: entity rewritten on the server.

Historical data are never tagged as `ACTION_ENTITY_RWT`.

This value may be returned by `getAction()`.

See Also:
Constant Field Values

◊ **ACTION_ENTITY_KIL**

```
static final int ACTION_ENTITY_KIL
```

Action−code: entity physically removed on the server.

Every time there is an `ACTION_ENTITY_KIL` the server EntityClass version changed. This new version is available in `getTimeStamp()`.getDateTime() and it's different from both the `initially subscribed EntityClass version` and the initially required EntityClass version.

If `getKeyID()` > 0,
then

- the Entity returned by `getEntity()` is generally undefined on any fields apart from those associated with `getKeyID()`

otherwise (`getKeyID()` <= 0)
- all entities are physically removed from the server, and
- `getEntity()` returns `null`.

Historical data are never tagged as `ACTION_ENTITY_KIL`.

This value may be returned by `getAction()`.

See Also:
Constant Field Values

◆

Method Detail

◊ getAction

```
int getAction()
```

Returns the server action associated with the entity available on this event.

The possible returned values are described in the Field Summary section.

If this event is the server−answer to a `Subscription.refreshEntity()` the value `ACTION_ENTITY_ADD` is returned.

Returns:

the server action associated with the entity available on this event.

◊ getTimeStamp

```
TimeStamp getTimeStamp()
```

Returns the timestamp associated with the entity available on this event.

Returns:

the timestamp associated with the entity available on this event.
`null` is never returned.

◊ getEntity

```
Entity getEntity()
```

Returns the entity available on this event.

If `getKeyID()` <= 0
then

· all entities are physically removed from the server,
· this method returns `null`,

else

· if `getAction()` is `ACTION_ENTITY_DEL` or `ACTION_ENTITY_KIL`

- the Entity returned by this method is generally undefined on any fields apart from those associated with `getKeyID()`,

else

- if this event refers a masked subscriptions and it is **not** an answer to a `Subscription.refreshEntity()`
  then
  - `isMasked()` returns `true`,
  - the Entity returned by this method is generally undefined on any fields apart from those associated with the mask,

  else
  - `isMasked()` returns `false`,
  - all the fields of the Entity returned by this method are meaningful.

Returns:

the entity available on this event.
`null` is returned when `getKeyID()` <= 0.

◊ **getKeyID**

```
int getKeyID()
```

Returns the index of the key on the basis of which the server has carried out `getAction()`.

For `ACTION_ENTITY_DEL` and `ACTION_ENTITY_KIL` this value determines which field are available in `getEntity()`.

Returns:
the index of the key on the basis of which the server has carried out `getAction()`.
See Also:
`getEntity()`

◊ **isMasked**

```
boolean isMasked()
```

Returns the indication that some fields of `getEntity()` may be missings.

For `ACTION_ENTITY_ADD` and `ACTION_ENTITY_RWT` this value determines which field are available in `getEntity()`.

`true` is returned when this event refers a masked subscriptions and it is **not** an answer to a `Subscription.refreshEntity()`.
In this case the Entity returned by `getEntity()` is generally undefined on any fields apart from those associated with the mask.

`false` is returned when this event refers a **not** masked subscriptions or it is an answer to a `Subscription.refreshEntity()`.
In this case all fields of the Entity returned by `getEntity()` are meaningful.

Please note that the behaviour of this method does **not** depends on the values (`SubscriptionParam.SUBSCRIBE_MASKED_FLOW_ALL` and `SubscriptionParam.SUBSCRIBE_MASKED_FLOW_LAST`) given to `SubscriptionParam.setFlow()`.

Returns:
the indication that some fields of `getEntity()` may be missings.
The return value is undefined for `ACTION_ENTITY_DEL` and `ACTION_ENTITY_KIL`.
See Also:
`getEntity()`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface SubscriptionStartEvent

- *All Superinterfaces:*
  Event, SubscriptionEvent

```
public interface SubscriptionStartEvent
extends SubscriptionEvent
```

Server−answer to `Subscription.start()`.

This event must be handled by `SubscriptionListener.onSubscriptionStart()`.

- ♦ **Field Summary**

  ◊ **Fields inherited from interface Event**

  `RESULT_GENERIC_ERROR, RESULT_OK`

- ♦ **Method Summary**

  Methods

Modifier and Type	Method and Description
`int`	***`getEntityClassVersionOnServer`***`()` Returns the version of EntityClass on the server.
`boolean`	***`isEntityClassReset`***`()` Returns a reset−class indication.

  ◊ **Methods inherited from interface SubscriptionEvent**

  `getSubscription`

  ◊ **Methods inherited from interface Event**

  `getResult`

- ♦ **Method Detail**

◊ isEntityClassReset

```
boolean isEntityClassReset()
```

Returns a reset−class indication.

A `true` indicates that the required EntityClass version is different from the `server EntityClass version`. In this case the historical data that will be available into the next `SubscriptionNotifyEvents` are complete and not (as usual) restricted to data following a given timestamp.

This method must be called only when the result is `Event.RESULT_OK`.

Returns:

the indication whether a download is needed on all the entities.
`false` is returned when the result is not `Event.RESULT_OK`.

◊ getEntityClassVersionOnServer

```
int getEntityClassVersionOnServer()
```

Returns the version of EntityClass on the server.

The returned value may be different from the version requested into the Subscription parameter. In this case `isEntityClassReset()` returns `true`.

This method must be called only when the server result is `Event.RESULT_OK`.

Returns:

the version of EntityClass on the server.
−1 is returned when the result is not `Event.RESULT_OK`.

See Also:

`isEntityClassReset()`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface SubscriptionStopEvent

- *All Superinterfaces:*
  Event, SubscriptionEvent

```
public interface SubscriptionStopEvent
extends SubscriptionEvent
```

Server−answer to `Subscription.stop()`.

This event must be handled by `SubscriptionListener.onSubscriptionStop()`.

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface Event**

    `RESULT_GENERIC_ERROR, RESULT_OK`
  - ♦ **Method Summary**
    - ◊ **Methods inherited from interface SubscriptionEvent**

      `getSubscription`
    - ◊ **Methods inherited from interface Event**

      `getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface TransactionEvent

- *All Superinterfaces:*
  Event

  All Known Subinterfaces:
  TransactionQueryEvent, TransactionSendEvent

```
public interface TransactionEvent
extends Event
```

Generic event related to the Transaction Lifecycle.

Events related to this super−interface must be handled by the methods of `TransactionListener`.

With this event the server result is never `Event.RESULT_OK`.

- ♦ **Field Summary**

  Fields

Modifier and Type	Field and Description
`static int`	***RESULT_ABORTED*** Failure−code: transaction was aborted by the server.
`static int`	***RESULT_COMMITTED***

	Failure−code: transaction was commited by the server.
static int	***RESULT_FLYING*** Failure−code: transaction is flying.
static int	***RESULT_INVALID_TRANSACTION_ID*** Failure−code: transaction is not valid because an invalid `Transaction ID` was used.

◊ **Fields inherited from interface Event**

`RESULT_GENERIC_ERROR, RESULT_OK`

♦ **Method Summary**

Methods

Modifier and Type	Method and Description
`Entity`	***getEntity***() Returns the result entity associated to this event.
`int`	***getReasonCode***() Returns the specific market−related reason about the transaction abort.
`TimeStamp`	***getTimeStamp***() Returns the result timestamp.
`Transaction`	***getTransaction***() Returns the transaction associated to this event.

◊ **Methods inherited from interface Event**

`getResult`

• ♦ **Field Detail**

◊ **RESULT_ABORTED**

```
static final int RESULT_ABORTED
```

Failure−code: transaction was aborted by the server.

In this case `getReasonCode()` may be used to understand why the market aborted the transaction.

See Also:

Constant Field Values

◊ **RESULT_COMMITTED**

```
static final int RESULT_COMMITTED
```

Failure−code: transaction was commited by the server.
See Also:
Constant Field Values

◊ **RESULT_FLYING**

```
static final int RESULT_FLYING
```

Failure−code: transaction is flying.
See Also:
Constant Field Values

◊ **RESULT_INVALID_TRANSACTION_ID**

```
static final int RESULT_INVALID_TRANSACTION_ID
```

Failure−code: transaction is not valid because an invalid `Transaction ID` was used.
See Also:
Constant Field Values

♦ Method Detail

◊ **getTransaction**

```
Transaction getTransaction()
```

Returns the transaction associated to this event.

Returns:
the transaction associated to this event.
`null` is never returned.

◊ **getTimeStamp**

```
TimeStamp getTimeStamp()
```

Returns the result timestamp.

This method must be called only when the result is `Event.RESULT_OK` or when this event is instanceof `TransactionQueryEvent`.

Returns:
the result timestamp.
`null` is returned when the result is not `Event.RESULT_OK` and this event is not instanceof `TransactionQueryEvent`.

◊ **getEntity**

```
Entity getEntity()
```

Returns the result entity associated to this event.

This method must be called only when the result is `Event.RESULT_OK` or when this event is instanceof `TransactionQueryEvent`.

Returns:

the result entity associated to this event.
`null` is returned when the result entity was not `required`, or
when the server choose to not send the result, or
the result is not `Event.RESULT_OK` and this event is not instanceof `TransactionQueryEvent`.

◊ **getReasonCode**

```
int getReasonCode()
```

Returns the specific market−related reason about the transaction abort.

This method must be called only when the result is `RESULT_ABORTED`.

The exact meaning of the result depends on the specific market server and it is documented in the correspondig market server manual, apart from the following generic values:

- 10000: internal error
- 10001: not logged
- 10002: inadequate privileges
- 10003: invalid request action
- 10004: invalid Transaction ID

Returns:

the specific market−related reason about the transaction abort.
Zero is returned when the result is not `RESULT_ABORTED`.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface TransactionQueryEvent

- *All Superinterfaces:*
  Event, TransactionEvent

```
public interface TransactionQueryEvent
extends TransactionEvent
```

Server−answer to `Transaction.query()`.

This event must be handled by `TransactionListener.onTransactionQuery()`.

With this event the server result is never `Event.RESULT_OK`.

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface TransactionEvent**

    `RESULT_ABORTED`, `RESULT_COMMITTED`, `RESULT_FLYING`, `RESULT_INVALID_TRANSACTION_ID`
  - ◊ **Fields inherited from interface Event**

    `RESULT_GENERIC_ERROR`, `RESULT_OK`
- ♦ **Method Summary**
  - ◊ **Methods inherited from interface TransactionEvent**

    `getEntity`, `getReasonCode`, `getTimeStamp`, `getTransaction`
  - ◊ **Methods inherited from interface Event**

    `getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface TransactionSendEvent

- *All Superinterfaces:*
  Event, TransactionEvent

```
public interface TransactionSendEvent
extends TransactionEvent
```

Server−answer to `Transaction.send()`.

This event must be handled by `TransactionListener.onTransactionSend()`.

With this event the server result is never `Event.RESULT_OK`.

- ♦ **Field Summary**
  - ◊ **Fields inherited from interface TransactionEvent**

    `RESULT_ABORTED`, `RESULT_COMMITTED`, `RESULT_FLYING`, `RESULT_INVALID_TRANSACTION_ID`

◊ **Fields inherited from interface Event**

`RESULT_GENERIC_ERROR`, `RESULT_OK`

♦ **Method Summary**

◊ **Methods inherited from interface TransactionEvent**

`getEntity`, `getReasonCode`, `getTimeStamp`, `getTransaction`

◊ **Methods inherited from interface Event**

`getResult`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface Listener

- *All Known Subinterfaces:*
  ConnectionListener, EntityClassQueryListener, FilterListener, MulticastConnectionListener, QueryListener, SubscriptionListener, TransactionListener

---

```
public interface Listener
```

Super−interface common to all listener interfaces.

Listener interfaces must be implemented in order to handle the lifecycle of the various CommunicationLifeCycle objects (`Connection`, `Filter`, `Query`, `Subscription`, `Transaction`).

These subinterfaces are bound to the communication objects creation (through the various `makeSomething` methods in `Context`) and then they may be retrieved by `CommunicationLifeCycle.getListener()`.

If an exception is thrown and not catch inside a Listener method then the behavior of the application is controlled by `JFT.setExitOnListenerException(boolean)` method.

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface ConnectionListener

- *All Superinterfaces:*
  Listener

```
public interface ConnectionListener
extends Listener
```

Interface to be implemented in order to handle the Connection Lifecycle.

This interface is bound to connections created by `Context.makeConnection()`.
It may be retrieved by `CommunicationLifeCycle.getListener()`.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	***`onConnectionClose`***`(ConnectionCloseEvent event)` Called when the server−answer to the `Connection.close()` is available.
`void`	***`onConnectionLost`***`(ConnectionLostEvent event)` Called when the connection with server crashed or when the server choose to terminate the connection.
`void`	***`onConnectionOpen`***`(ConnectionOpenEvent event)` Called when the server−answer to the `Connection.open()` is available.

Method Detail

◊ onConnectionOpen

```
void onConnectionOpen(ConnectionOpenEvent event)
```

Called when the server−answer to the `Connection.open()` is available.

If the server result is `Event.RESULT_OK`,
then

- the server has accepted the connection creation.
- the connection status has changed to `Connection.STATUS_CONNECTED`.
- the server may now accept subscriptions, queries, filters or transactions.

otherwise

- the server has rejected the connection (see the various ConnectionOpenEvent result codes to understand why).
- the connection status has changed to `Connection.STATUS_DISCONNECTED`.
- the server does not accept any subscriptions, queries, filters or transactions on this Connection.

In the latter case it is a good practice to `release` the connection associated to the event parameter.

Parameters:
`event` − the server−answer to the `Connection.open()`

◊ **onConnectionClose**

```
void onConnectionClose(ConnectionCloseEvent event)
```

Called when the server−answer to the `Connection.close()` is available.

If the server result is `Event.RESULT_OK`, then the server has closed the connection otherwise some unknow error occured.

In both cases:

· the server does not accept any subscriptions, queries, filters or transactions on this Connection.
· the connection status has changed to `Connection.STATUS_DISCONNECTED`.
· it is a good practice to `release` the connection associated to the event parameter.

It is guaranteed that only one method between `onConnectionClose` and `onConnectionLost` will ever be called on the same Listener.

Parameters:
`event` − the server−answer to the `Connection.close()`

◊ **onConnectionLost**

```
void onConnectionLost(ConnectionLostEvent event)
```

Called when the connection with server crashed or when the server choose to terminate the connection.

In this case:

· the server result is always `Event.RESULT_GENERIC_ERROR`.
· the connection status has changed to `Connection.STATUS_DISCONNECTED`.
· the server does not accept any subscriptions, queries, filters or transactions on this Connection.
· it is a good practice to `release` the connection associated to the event parameter.

It is guaranteed that only one method between `onConnectionClose` and `onConnectionLost` will ever be called on the same Listener.

Parameters:
`event` − the description of this closure

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface EntityClassQueryListener

- *All Superinterfaces:*
  Listener

```
public interface EntityClassQueryListener
extends Listener
```

- ♦
Method Summary

Methods

Modifier and Type	Method and Description
void	***onEntityClassQuery***(EntityClassQueryEvent event)

- ♦
Method Detail

◊ onEntityClassQuery

```
void onEntityClassQuery(EntityClassQueryEvent event)
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface FilterListener

- *All Superinterfaces:*
  Listener

```
public interface FilterListener
extends Listener
```

Interface to be implemented in order to handle the Filter Lifecycle.

This interface is bound to filters created by `Context.makeFilter()`.
It may be retrieved by `CommunicationLifeCycle.getListener()`.

- ♦
Method Summary

Methods

Modifier and Type	Method and Description
void	***onFilterCreate***(FilterCreateEvent event) Called when the server−answer to the `Filter.create()` is available.
void	***onFilterDestroy***(FilterDestroyEvent event) Called when the server−answer to the `Filter.destroy()` is available.

void	***onFilterSet***(FilterSetEvent event) Called when the server−answer to the Filter.set(java.lang.String) is available.

- ♦

Method Detail

◊ onFilterCreate

```
void onFilterCreate(FilterCreateEvent event)
```

Called when the server−answer to the Filter.create() is available.

If the server result is Event.RESULT_OK,
then

- · the server has accepted the filter creation.
- · the filter status has changed to Filter.STATUS_CREATED.
- · the server may now accept Filter.set(java.lang.String) on this filter.
- · the server may now accept subscriptions based on this filter.

otherwise

- · the server has rejected the filter creation (see the various FilterCreateEvent result codes to understand why).
- · the filter status has changed to Filter.STATUS_DESTROYED.
- · the server does not accept any Filter.set(java.lang.String) on this filter.
- · the server does not accept any subscription based on this filter.

In the latter case it is a good practice to release the filter associated to the event parameter.

Parameters:

event − the server−answer to the Filter.create()

◊ onFilterSet

```
void onFilterSet(FilterSetEvent event)
```

Called when the server−answer to the Filter.set(java.lang.String) is available.

If the server result is Event.RESULT_OK then the server has accepted the filter setting otherwise some error occurred (see the various FilterSetEvent result codes to understand why).

In both cases:

◊ the filter status remains unchanged (the most of cases it remains Filter.STATUS_CREATED).

◊ the server may now accept subscriptions based on this filter.

Parameters:

event − the server−answer to the Filter.set(java.lang.String)

- **onFilterDestroy**

```
void onFilterDestroy(FilterDestroyEvent event)
```

Called when the server−answer to the `Filter.destroy()` is available.

If the server result is `Event.RESULT_OK`, then the server has destroyed the filter otherwise some unknow error occured.

In both cases:

- the server does not accept any other operation on this filter.
- the filter status has changed to `Filter.STATUS_DESTROYED`.
- it is a good practice to `release` the filter associated to the event parameter.

Parameters:
`event` − the server−answer to the `Filter.destroy()`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface MulticastConnectionListener

- *All Superinterfaces:*
  Listener

```
public interface MulticastConnectionListener
extends Listener
```

- ♦
  ## Method Summary

  Methods

Modifier and Type	Method and Description
void	***onMulticastConnection***(MulticastConnectionEvent event)

- ♦
  ## Method Detail

  ◊ **onMulticastConnection**

```
void onMulticastConnection(MulticastConnectionEvent event)
```

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.
it.list.jft.event

Interface QueryListener

- *All Superinterfaces:*
  Listener

```
public interface QueryListener
extends Listener
```

Interface to be implemented in order to handle the Query Lifecycle.

This interface is bound to queries created by `Context.makeQuery()`.
It may be retrieved by `CommunicationLifeCycle.getListener()`.

- ♦

Method Summary

Methods

Modifier and Type	Method and Description
`void`	***`onQueryCreate`***`(QueryCreateEvent event)` Called when the server−answer to the `Query.create()` is available.
`void`	***`onQueryDestroy`***`(QueryDestroyEvent event)` Called when the server−answer to the `Query.destroy()` is available.
`void`	***`onQueryNotify`***`(QueryNotifyEvent event)` Called when an entity of a query result−set is available.
`void`	***`onQueryRows`***`(QueryRowsEvent event)` Called when the server−answer to the `Query.queryRows()` is available.

- ♦

Method Detail

◊ onQueryCreate

```
void onQueryCreate(QueryCreateEvent event)
```

Called when the server−answer to the `Query.create()` is available.

If the server result is `Event.RESULT_OK`,
then

- the server has accepted the query creation.
- the query status has changed to `Query.STATUS_CREATED`.
- if `QueryCreateEvent.resultSetFollows()` is `true` then the server starts to send `QueryNotifyEvent events` to notify the query result.
- if `QueryCreateEvent.resultSetFollows()` is `false` then the server may now accept `Query.QueryRows()` on this query.

otherwise

- the server has rejected the query creation (see the various QueryCreateEvent result codes to understand why).
- the query status has changed to `Query.STATUS_DESTROYED`.
- the server does not accept any `Query.QueryRows` on this query.
- the server does not send any `QueryNotifyEvent events` to notify the query result.

In the latter case it is a good practice to `release` the query associated to the event parameter.

Parameters:

`event` − the server−answer to the `Query.create()`

◊ onQueryRows

```
void onQueryRows(QueryRowsEvent event)
```

Called when the server−answer to the `Query.queryRows()` is available.

If the server result is `Event.RESULT_OK`,
then

- the server has accepted the query request.
- the server starts to send `QueryNotifyEvent events` to notify the query result.

otherwise

- the server has rejected the query request (see the various QueryRowsEvent result codes to understand why).
- the server does not send any `QueryNotifyEvent events` to notify the query result.

In both cases the query status remains `Query.STATUS_CREATED`.

Parameters:

`event` − the server−answer to the `Query.queryRows()`

◊ onQueryNotify

```
void onQueryNotify(QueryNotifyEvent event)
```

Called when an entity of a query result−set is available.

If the query result−set computed by the server, as an aswer to a correct `Query.create()` or `Query.queryRows()`, is composed by N entities then this method will be invoked (N+1) times: N times with each of the N entities and one more time with the EOQ indication.

In any case:

- the server result is always `Event.RESULT_OK`.
- the query status remains `Query.STATUS_CREATED`.
- the data is available through the various QueryNotifyEvent methods

Parameters:

`event` − event containing an entity of the query result−set or the EOQ indication.

◊ onQueryDestroy

```
void onQueryDestroy(QueryDestroyEvent event)
```

Called when the server−answer to the `Query.destroy()` is available.

If the server result is `Event.RESULT_OK`, then the server has destroyed the query otherwise some unknow error occured.

In both cases:

- the server does not accept any other operation on this query.
- the query status has changed to `Query.STATUS_DESTROYED`.
- it is a good practice to `release` the query associated to the event parameter.

Parameters:

`event` − the server−answer to the `Query.destroy()`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface SubscriptionListener

- *All Superinterfaces:*
  Listener

```
public interface SubscriptionListener
extends Listener
```

Interface to be implemented in order to handle the Subscription Lifecycle.

This interface is bound to subscriptions created by `Context.makeSubscription()`.
It may be retrieved by `CommunicationLifeCycle.getListener()`.

- ♦

Method Summary

Methods

Modifier and Type	Method and Description
void	***onSubscriptionIdle***(`SubscriptionIdleEvent event`) Called when the flow of historical data is finished and the start of actual data is starting.
void	***onSubscriptionNotify***(`SubscriptionNotifyEvent event`) Called when an actual or historical data is available or when the server−answer of a `Subscription.refreshEntity` is available.

void	***onSubscriptionStart***(SubscriptionStartEvent event) Called when the server−answer to the Subscription.start() is available.
void	***onSubscriptionStop***(SubscriptionStopEvent event) Called when the server−answer to the Subscription.stop() is available.

Method Detail

onSubscriptionStart

```
void onSubscriptionStart(SubscriptionStartEvent event)
```

Called when the server−answer to the Subscription.start() is available.

If the server result is Event.RESULT_OK,
then

- the server has accepted the subscription.
- the subscription status has changed to Subscription.STATUS_STARTED.
- SubscriptionNotifyEvent and/or SubscriptionIdleEvent events will be received.

otherwise

- the server has rejected the subscription.
- the subscription status has changed to Subscription.STATUS_STOPPED.
- any SubscriptionNotifyEvent and/or SubscriptionIdleEvent events will never been received.

In the latter case it is a good practice to release the subscription associated to the event parameter.

Parameters:

event − the server−answer to the Subscription.start()

onSubscriptionIdle

```
void onSubscriptionIdle(SubscriptionIdleEvent event)
```

Called when the flow of historical data is finished and the start of actual data is starting.

In this case:

- the server result is always Event.RESULT_OK.
- the subscription status remains Subscription.STATUS_STARTED.

This method is called only if the type of query of the subscription is not SubscriptionParam.QUERY_TYPE_ON_TIME.

If the type of query of the subscription is SubscriptionParam.QUERY_TYPE_PAST then it is a good practice to stop the subscription.

Parameters:

event − event marking stop/starting of historical/actual data coming from the server

◊ onSubscriptionNotify

```
void onSubscriptionNotify(SubscriptionNotifyEvent event)
```

Called when an actual or historical data is available or when the server−answer of a `Subscription.refreshEntity` is available.

In this case:

- the server result is always `Event.RESULT_OK`.
- the subscription status remains `Subscription.STATUS_STARTED`.
- the data is available through the various SubscriptionNotifyEvent methods

If this is the server−answer of a `refreshEntity` then the data available through `SubscriptionNotifyEvent.getEntity()` is always complete even if the subscription was opened in a masked fashion.

Parameters:

`event` − event containing actual/historical data

◊ onSubscriptionStop

```
void onSubscriptionStop(SubscriptionStopEvent event)
```

Called when the server−answer to the `Subscription.stop()` is available.

If the server result is `Event.RESULT_OK`, then the server has closed the subscription otherwise some unknow error occured.

In both cases:

- the subscription status has changed to `Subscription.STATUS_STOPPED`.
- any `SubscriptionNotifyEvent` and/or `SubscriptionIdleEvent` events will never been received.
- it is a good practice to `release` the subscription associated to the event parameter.

Parameters:

`event` − the server−answer to the `Subscription.stop()`

Submit a bug or feature to FT\API Programming Support

JavaScript is disabled on your browser.

it.list.jft.event

Interface TransactionListener

- *All Superinterfaces:*
  Listener

```
public interface TransactionListener
extends Listener
```

Interface to be implemented in order to handle the Transaction Lifecycle.

This interface is bound to transactions created by `Context.makeTransaction()`.
It may be retrieved by `CommunicationLifeCycle.getListener()`.

Method Summary

Methods

Modifier and Type	Method and Description
`void`	***`onTransactionQuery`***`(TransactionQueryEvent event)` Called when the server−answer to the `Transaction.query()` is available.
`void`	***`onTransactionSend`***`(TransactionSendEvent event)` Called when the server−answer to the `Transaction.send()` is available.

Method Detail

onTransactionSend

```
void onTransactionSend(TransactionSendEvent event)
```

Called when the server−answer to the `Transaction.send()` is available.

Depending on the server result (that cannot never be equal to `Event.RESULT_OK`) the transaction status changes to:

- `TransactionEvent.RESULT_FLYING` ’ `Transaction.STATUS_FLYING`
- `TransactionEvent.RESULT_COMMITTED` ’ `Transaction.STATUS_COMMITTED`
- `TransactionEvent.RESULT_ABORTED` ’ `Transaction.STATUS_ABORTED`
  (in this case it is possible to get the reason code that caused the transaction to fail)
- `TransactionEvent.RESULT_INVALID_TRANSACTION_ID` ’ `Transaction.STATUS_ABORTED`
- `Event.RESULT_GENERIC_ERROR` ’ `Transaction.STATUS_ABORTED`

If the current transaction status is `TransactionEvent.RESULT_FLYING` then the server can accept calls to `Transaction.query()` otherwise it is a good practice to `release` the transaction associated to the event parameter.

Parameters:

`event` − the server−answer to the `Transaction.send()`

onTransactionQuery

```
void onTransactionQuery(TransactionQueryEvent event)
```

Called when the server−answer to the `Transaction.query()` is available.

Depending on the server result (that cannot never be equal to `Event.RESULT_OK`) the transaction status changes to:

- `TransactionEvent.RESULT_FLYING` ’ `Transaction.STATUS_FLYING`
- `TransactionEvent.RESULT_COMMITTED` ’ `Transaction.STATUS_COMMITTED`
- `TransactionEvent.RESULT_ABORTED` ’ `Transaction.STATUS_ABORTED`
  (in this case it is possible to get the reason code that caused the transaction to fail)
- `TransactionEvent.RESULT_INVALID_TRANSACTION_ID` ’ `Transaction.STATUS_ABORTED`
- `Event.RESULT_GENERIC_ERROR` ’ `Transaction.STATUS_ABORTED`

If the current transaction status is `TransactionEvent.RESULT_FLYING` then the server can accept other calls to `Transaction.query()` otherwise it is a good practice to `release` the transaction associated to the event parameter.

Parameters:

`event` − the server−answer to the `Transaction.query()`

Submit a bug or feature to FT\API Programming Support

JFT/Api Application Examples

In this chapter we present 3 applications examples that use JFT/Api:

- **Example 1**
  The first example is the simplest. It show how to create one single connection to a FastTrack MetaMarket service and one single subscription to a given EntityClass. All received entities on this subscriptions are written on standard output and then the application terminates.

  The application is very simple and small: it does not handle any errors and/or exceptions and it does not make any trace.
- **Example 2**
  The second example show how to create two distinct connections at the same FastTrack MetaMarket service. Two subscriptions are opened on the first connection: these two subscriptions are a few complex because they use Mask, partial keys and refreshEntity. Instead on the second connection many transactions will be started and/or queried automatically.
- **Example 3**
  This very simple application example open a connection with a MetaMarket service using YAS service manager, and then it subscribe the first position of the depth of some securities using EntityFilter.

See Also:

JFT/Api Introduction, `JFT`, JFT Exceptions, JFT Implementation Threads, JFT Synchronization

Example 1

The first example is the simplest. It show how to create one single connection to a FastTrack MetaMarket service and one single subscription to a given EntityClass. All received entities on this subscriptions are written on standard output and then the application terminates.

The application is very simple and small: it does not handle any errors and/or exceptions and it does not make any trace.

```
/*

Description
===========
  This application example open a connection with a MetaMarket service
  of a given FastTrack server and then it starts a subscription
  on FT_C_TRADING_STATE EntityClass.
  Every received Entity is written on standard output.
  The application terminates when a SubscriptionIdleEvent is received.


Example Usage
=============

  To compile this example remember to put in the classpath:
     - The path of JDK 1.1.x (or following)
     - The path of your library JFTApi.jar
     - The path of the directory where the metamarket package reside

  To launch this example type:
     java Example1

  To obtain something like:
     eID: HDAT mID: GR sID: Primary p: NOP s: Active pd: NOP t: 73049379
     eID: HDAT mID: GR sID: Repos p: NEG s: Active pd: NEG t: 80000057
     eID: HDAT mID: GR sID: Retail p: NEG s: Active pd: NEG t: 73049404
     eID: HDAT mID: GR sID: Secondary p: NEG s: Active pd: NEG t: 80000074
     eID: GAM mID: MOT sID: 1 p: NEG s: Active pd:  t: 0
     eID: GAM mID: TON sID: 1 p: NEG s: Active pd:  t: 0
     eID: GAM mID: MOT sID: 2 p: NEG s: Active pd:  t: 0
     eID: GAM mID: TON sID: 2 p: NEG s: Active pd:  t: 0
     eID: GAM mID: MOT sID: 3 p: NEG s: Active pd:  t: 0
     eID: GAM mID: TON sID: 3 p: NEG s: Active pd:  t: 0
     eID: GAM mID: MOT sID: 4 p: NEG s: Active pd:  t: 0
     eID: GAM mID: TON sID: 4 p: NEG s: Active pd:  t: 0

Additional Classes (in metamarket package)
==========================================

  In order to profitably use this example there is need for some additional
  Java classes in the metamarket package. These classes are:
    - MetaMarket         It contains global constants (entityClassIDs, keyIDs, etc...)
                         for all data structures of FastTrack MetaMarket service.
    - FT_C_TRADING_STATE Specific EntityClass for the handled trading states.
                         You can see below a skeleton of this class.
    - FT_C_TRADING_PHASE The various enumeration values for fields of FT_C_TRADING_STATE

*/
```

```
/*

MetaMarket
==========
  MetaMarket is the Java class that contains global constants
  for all data structures of FastTrack MetaMarket service.

  This class contains
    - a lot of constants (for all entytClassIDs, keyIDs, etc...)
      that may be used to access all data handled by MetaMarket service.
    - a method registerAll() that may be used to register all
      the EntityClasses of MetaMarket.

  In this example we use only 3 EntityClasses:
  FT_C_TRADING_STATE, FT_C_ORDER and FT_C_ERROR_INFO
  so the only used members of the class MetaMarket are:

  public static final int FT_C_TRADING_STATE_ID = 30010; // FT_C_TRADING_STATE id
  public static final int FT_C_TRADING_STATEKey = 1;     // FT_C_TRADING_STATE prim. key
  public static void registerAll();       // to register all EntityClasses of MetaMarket


The FT_C_TRADING_STATE EntityClass
==================================
  Its entityClassID is FT_C_TRADING_STATE_ID = 30010.
  Its primary keyID is FT_C_TRADING_STATEKey = 1
      and it includes ExchangeID, MarketID and SectionID fields.
  Its structure is something like:

class FT_C_TRADING_STATE {
   String ExchangeID; // ID of the market place
   String MarketID;   // ID of the market
   String SectionID;  // ID of the section
   int    Phase;      // Phase of the security:
                      // 0 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_UNDEF
                      // 1 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_CLOSURE
                      // 2 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_PRE_ISSUE
                      // 3 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_ISSUE
                      // 4 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_PRE_AUCTION
                      // 5 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_AUCTION
                      // 6 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_POST_AUCTION
                      // 7 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_PRE_TRADING
                      // 8 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_TRADING
                      // 9 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_POST_TRADING
                      //10 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_TRADING_AT_LAST
                      //11 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_TRADING_AFTER_HOUR
                      //12 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_FAST_MARKET
                      //13 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_MANAGEMENT
                      //14 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_NO_OPERATION
   int    Status;     // Status of the section
                      // 0 or FT_C_TRADING_STATUS.FT_C_TRADING_STATUS_Active
                      // 1 or FT_C_TRADING_STATUS.FT_C_TRADING_STATUS_Suspended
                      // 2 or FT_C_TRADING_STATUS.FT_C_TRADING_STATUS_Frozen
   String PhaseDescription; // Description of phase
   int    Time;       // Time (format: HHMMSSmmm) of last change
}

*/
```

```
/*

The FT_C_TRADING_PHASE class
============================
  This java class is not an EntityClass.
  This class contains
    - all enumeration value for the trading phase,
    - a method enumAsString that returns a displayable String for a given value.
  Its structure is something like:

class FT_C_TRADING_PHASE {
  public static final int FT_C_TRADING_PHASE_UNDEF             =  0;
  public static final int FT_C_TRADING_PHASE_CLOSURE           =  1;
  public static final int FT_C_TRADING_PHASE_PRE_ISSUE         =  2;
  public static final int FT_C_TRADING_PHASE_ISSUE             =  3;
  public static final int FT_C_TRADING_PHASE_PRE_AUCTION       =  4;
  public static final int FT_C_TRADING_PHASE_AUCTION           =  5;
  public static final int FT_C_TRADING_PHASE_POST_AUCTION      =  6;
  public static final int FT_C_TRADING_PHASE_PRE_TRADING       =  7;
  public static final int FT_C_TRADING_PHASE_TRADING           =  8;
  public static final int FT_C_TRADING_PHASE_POST_TRADING      =  9;
  public static final int FT_C_TRADING_PHASE_TRADING_AT_LAST   = 10;
  public static final int FT_C_TRADING_PHASE_TRADING_AFTER_HOUR = 11;
  public static final int FT_C_TRADING_PHASE_FAST_MARKET       = 12;
  public static final int FT_C_TRADING_PHASE_MANAGEMENT        = 13;
  public static final int FT_C_TRADING_PHASE_NO_OPERATION      = 14;

  public static final String enumAsString(int value) { return "an appropriate value"; }
}

*/

// Effective source-code Example1 starts here.

import it.list.jft.*;       // to use the JFT/Api library
import it.list.jft.event.*; // to use the JFT/Api library
import metamarket.*;        // to use MetaMarket, FT_C_TRADING_STATE, etc...

abstract class Example1 {
  static final int        clientID       = 67899;
  static final String     dirPath        = ".";
  static final String     myOperatorID   = "dario";
  static final String     myOperatorPass = "*";
  static final String     host           = "194.91.195.234";
  static final int        port           = 41005;
  static       Context    context;

  public static void main(String[]args) {
    JFT.THIS.init(JFT.MODE_MULTI_THREAD);
    if(true)
      JFT.THIS.register(new FT_C_TRADING_STATE());
    else // as an expensive alternative
      MetaMarket.registerAll();
    JFT.THIS.start();
    context = JFT.THIS.makeContext();
    new ConnectionEx();
  }
}
```

```
// Class to handle the connection.

class ConnectionEx implements ConnectionListener {

  final Connection connection;

  ConnectionEx() {
    ConnectionParam cp = Example1.context.makeConnectionParam();
    cp.setHost(Example1.host);
    cp.setPort(Example1.port);
    cp.setApplRevision(new int[]{0,0,0});
    cp.setApplSignature(12345);
    cp.setClientID(Example1.clientID);
    cp.setConnType(ConnectionParam.CONN_TYPE_TCP);
    cp.setUserName(Example1.myOperatorID);
    cp.setPassword(Example1.myOperatorPass);
    cp.setUserType(ConnectionParam.USER_TYPE_VIEW);
    connection = Example1.context.makeConnection(cp, this);
    if(connection.open() != Connection.RESULT_OK)
      connection.release(); // good practice
  }

  public void onConnectionOpen(ConnectionOpenEvent ev) {
    if(ev.getResult() == ev.RESULT_OK)
      new SubscriptionEx(connection);
    else
      connection.release(); // good practice
  }

  public void onConnectionClose(ConnectionCloseEvent ev) {
    connection.release(); // good practice
  }

  public void onConnectionLost(ConnectionLostEvent ev) {
    connection.release(); // good practice
  }
}
```

```
// Class to handle the subscription.

class SubscriptionEx implements SubscriptionListener {

  final Subscription subscription;
  final Connection   connection;

  SubscriptionEx(Connection conn) {
    connection = conn;
    SubscriptionParam sp = Example1.context.makeSubscriptionParam();
    sp.setEntityClassID(MetaMarket.FT_C_TRADING_STATE_ID);
    subscription = Example1.context.makeSubscription(connection, sp, this);
    if(subscription.start() != Subscription.RESULT_OK)
      subscription.release(); // good practice
  }

  public void onSubscriptionStart(SubscriptionStartEvent ev){
    if(ev.getResult() != ev.RESULT_OK)
      subscription.release(); // good practice
  }

  public void onSubscriptionIdle(SubscriptionIdleEvent ev){
    if(true)
      JFT.THIS.release();
    else { // as an expensive alternative
      subscription.stop();
      subscription.release(); // good practice
      connection.close();
      connection.release(); // good practice
      JFT.THIS.release();
    }
  }

  public void onSubscriptionNotify(SubscriptionNotifyEvent ev){
    switch(ev.getAction()) {
    case SubscriptionNotifyEvent.ACTION_ENTITY_ADD:
      System.out.println(entityAsString(ev.getEntity()));
      break;
    case SubscriptionNotifyEvent.ACTION_ENTITY_RWT:
    case SubscriptionNotifyEvent.ACTION_ENTITY_DEL:
    case SubscriptionNotifyEvent.ACTION_ENTITY_KIL:
    default:
      break;
    }
  }

  public void onSubscriptionStop(SubscriptionStopEvent ev){
    subscription.release(); // good practice
  }

  String entityAsString(Entity e) {
    FT_C_TRADING_STATE ts = (FT_C_TRADING_STATE) e;
    return "eID: " + ts.ExchangeID
        + " mID: " + ts.MarketID
        + " sID: " + ts.SectionID
        + " p: "   + FT_C_TRADING_PHASE.enumAsString(ts.Phase)
        + " s: "   + FT_C_TRADING_STATUS.enumAsString(ts.Status)
        + " pd: "  + ts.PhaseDescription
        + " t: "   + ts.Time;
  }
}
```

Example 2

The second example show how to create two distinct connections at the same FastTrack MetaMarket service. Two subscriptions are opened on the first connection: these two subscriptions are a few complex because they use Mask, partial keys and refreshEntity. Instead on the second connection many transactions will be started and/or queried automatically.

This example is more complex even because it handles and trace errors and exceptions in a sophisticated way. In addition all LifeCycle activities are handled at two levels: at the higher level all errors and common actions are handled, at the lower level the specific actions for the specific activity is taken.

```
/*

Description
===========
  This application example open two connections with a MetaMarket service
  of a given FastTrack server, and then:

    - Two subscriptions will be started on the first connection:
        - The first one is a non masked subscription
          on FT_C_ORDER EntityClass:
            - All received entities will be written on standard output.
            - All received entities matching a given operator will be written on files.
          This subscription is never stopped.
        - The second one is a partial and masked subscription
          on FT_C_TRADING_STATE EntityClass:
            - Only the ExchangeID and Phase fields
              are subscribed and written on standard output when received.
            - In addition many refreshEntity will be requested for each received
              entity with Phase field that matches a given phase.
              This is done in order to discover and print all fields of these entities.
          This subscription is close when all received non masked entities
          match the corresponding requested refreshEntity.

    - All files *.order and *.order.pending of a given directory are listed.
      Each *.order file contains all 14 mandatory fields of FT_C_ORDER EntityClass.
      Foreach *.order file (if there is not a corresponding *.order.pending file)
        a transaction is sent to the server in order to add a FT_C_ORDER.
        Once the transaction is sent its TransactionID is stored
        on a corresponding *.order.pending file and this file remains untouched
        until the transaction is committed or aborted.
        Once the transaction is sent and it's still flying the command-line option "-x"
        controls if the transaction must be immediately queried for its status
        or if this query must be postponed to the next run of this application.
      Foreach *.order file that is couple with a corresponding *.order.pending file
        a query for the pending transaction is sent to the server in order
        to discover if it is still pending or else it is committed or aborted.
        In the first case the application retry the query after a while.
        In the two latter cases the file *.order.pending is renamed *.order.done.blabla
        because the corresponding transaction is finished.
      There are convenient command-line options to control the delay between
      two subsequents queries and two subsequents scan of *.order files.

  The application terminates when all *.order files are been analyzed.
  Please note that this may happens before the two subscriptions terminate
  the print of their historical data: i.e. not all data may been received.

*/
```

```
/*

Example Usage
=============

  To compile this example remember to put in the classpath:
     - The path of JDK 1.4.x (or following)
     - The path of your library JFTApi.jar
     - The path of the directory where the metamarket package reside

  To launch this example type:
     java Example2 options...
  where options are: [-h host]             # FastTrack server TCP/IP host
                     [-p port]             # FastTrack server TCP/IP port
                     [-n serviceName]      # FastTrack service name
                     [-o opName]           # operator's name
                     [-w opPassword]       # operator's password
                     [-l licPathName]      # license file pathname
                     [-d dirPathName]      # directory with xxx.order[.pending] files
                     [-t traceLevel]       # 0<= traceLevel <= 5
                     [-v ON/OFF]           # trace verbose
                     [-s scanDelay]        # scan delay (in seconds)
                     [-q queryDelay]       # query delay (in seconds)
                     [-x ON/OFF]           # request to make a query after a send
  E.g.:
     java Example2 -h 194.91.195.1 -p 1234 -o dario -w dario -d c:\tmp -s 5 -q 15 -x ON
  requests
     - to talk with FastTrack server on host 194.91.195.1 and port 1234.
     - without specifying any service name (there is no "-n ..." option).
     - with operator name and password both equals to dario.
     - not using any license file (there is no "-l ..." option).
     - listing the directory C:\tmp.
     - using the default trace level TRACE_LEVEL_FATAL=5 (there is no "-t ..." option).
     - using the default non verbose trace (there is no "-v ..." option).
     - reading a new *.order and/or *.order.pending file every 5 seconds:
       i.e. every 5 seconds a send (if there is not a *.order.pending file)
                         or a query (if there is a *.order.pending file)
       will be sent to the server.
     - re-sending other query for the same transaction 15 seconds
       after a preceding flying (i.e. no commit and no abort) result.
     - requesting to automatically send a query after a send.


Additional Classes (in metamarket package)
==========================================

  In order to profitably use this example there is need for some additional
  Java classes in the metamarket package. These classes are:
    - MetaMarket         It contains global constants (entityClassIDs, keyIDs, etc...)
                         for all data structures of FastTrack MetaMarket service.
    - FT_C_ORDER         Specific EntityClass for orders handled by MetaMarket.
                         You can see below a skeleton of this class.
    - FT_C_TRADING_STATE Specific EntityClass for the handled trading states.
                         You can see below a skeleton of this class.
    - FT_C_ERROR_INFO    Specific EntityClass for the transaction errors communications
                         between the server and the application.
    - FT_C_TRADING_PHASE The various enumeration values for fields of FT_C_TRADING_STATE
    etc...

*/
```

```
/*

MetaMarket
==========
  MetaMarket is the Java class that contains global constants
  for all data structures of FastTrack MetaMarket service.

  This class contains
    - a lot of constants (for all entytClassIDs, keyIDs, etc...)
      that may be used to access all data handled by MetaMarket service.
    - a method registerAll() that may be used to register all
      the EntityClasses of MetaMarket.

  In this example we use only 3 EntityClasses:
  FT_C_TRADING_STATE, FT_C_ORDER and FT_C_ERROR_INFO
  so the only used members of the class MetaMarket are:

  public static final int FT_C_TRADING_STATE_ID = 30010; // FT_C_TRADING_STATE id
  public static final int FT_C_TRADING_STATEKey = 1;     // FT_C_TRADING_STATE prim. key
  public static final int FT_C_ORDER_ID         = 30014; // FT_C_ORDER id
  public static final int FT_C_ORDERKey         = 1;     // FT_C_ORDER primary key
  public static void registerAll();       // to register all EntityClasses of MetaMarket


The FT_C_TRADING_STATE EntityClass
==================================
  Its entityClassID is FT_C_TRADING_STATE_ID = 30010.
  Its primary keyID is FT_C_TRADING_STATEKey = 1
      and it includes ExchangeID, MarketID and SectionID fields.
  Its structure is something like:

class FT_C_TRADING_STATE {
   String ExchangeID; // ID of the market place
   String MarketID;   // ID of the market
   String SectionID;  // ID of the section
   int    Phase;      // Phase of the security:
                      // 0 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_UNDEF
                      // 1 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_CLOSURE
                      // 2 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_PRE_ISSUE
                      // 3 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_ISSUE
                      // 4 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_PRE_AUCTION
                      // 5 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_AUCTION
                      // 6 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_POST_AUCTION
                      // 7 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_PRE_TRADING
                      // 8 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_TRADING
                      // 9 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_POST_TRADING
                      //10 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_TRADING_AT_LAST
                      //11 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_TRADING_AFTER_HOUR
                      //12 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_FAST_MARKET
                      //13 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_MANAGEMENT
                      //14 or FT_C_TRADING_PHASE.FT_C_TRADING_PHASE_NO_OPERATION
   int    Status;     // Status of the section
                      // 0 or FT_C_TRADING_STATUS.FT_C_TRADING_STATUS_Active
                      // 1 or FT_C_TRADING_STATUS.FT_C_TRADING_STATUS_Suspended
                      // 2 or FT_C_TRADING_STATUS.FT_C_TRADING_STATUS_Frozen
   String PhaseDescription; // Description of phase
   int    Time;       // Time (format: HHMMSSmmm) of last change
}

*/
```

```
/*

The FT_C_ORDER EntityClass
==========================
  Its entityClassID is FT_C_ORDER_ID = 30014.
  Its primary keyID is FT_C_ORDERKey = 1
      and it includes FTSecID and OrderID fields.
  Its structure is something like:

class FT_C_ORDER {
  String FTSecID;          // ID of the security
  String OrderID;          // ID of the order given by the market
  String OperatorID;       // Operator ID
  String MrkOperatorID;    // ID of the operator on the destination market
  int    Verb;             // Verb of the order:
                           // 0 or FT_C_VERB.FT_C_VERB_Buy
                           // 1 or FT_C_VERB.FT_C_VERB_Sell
  int    OrderType;        // Type of the order
                           // 0 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_Limit
                           // 1 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_Market
                           // 2 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_Market_to_limit
                           // 3 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_Stop_market
                           // 4 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_OpeningPrice
                           // 5 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_Stop_limit
                           // 6 or FT_C_ORDER_TYPE.FT_C_ORDER_TYPE_Subscription
  int    QtyParameter;     // Parameter to choose if the whole quantity of the
                           // quantity must be matched at the same time
                           // 0 or FT_C_QTY_PARAMETER.FT_C_QTY_PARAMETER_Default
                           // 1 or FT_C_QTY_PARAMETER.FT_C_QTY_PARAMETER_All_or_None
  int    TimeInForce;      // Parameter to determine the life of the order
                           // 0 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Default
                           // 1 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Day
                           // 2 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Good_till_Date
                           // 3 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Good_till_Cancel
                           // 4 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Immediate_or_Cancel
                           // 5 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Good_till_Maturity
                           // 6 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Good_till_Hour
                           // 7 or FT_C_TIMEINFORCE.FT_C_TIMEINFORCE_Cancel_after_Filled
  int    ValidityDate;     // Date (format: AAAAMMDD) of the end of the order's validity
  int    Status;           // Status of the order
                           // 0 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_Active
                           // 1 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_PartFilled
                           // 2 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_CompFilled
                           // 3 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_Cancelled
                           // 4 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_Suspended
                           // 5 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_CancelledByGov
                           // 6 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_Stopped
                           // 7 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_Submitted
                           // 8 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_Rejected
                           // 9 or FT_C_ORDER_STATUS.FT_C_ORDER_STATUS_DeletedByEdit
  double Price;            // Limit price of the order
  double Qty;              // Quantity of the order
  double StopPrice;        // Price that triggers a stop order
  int    TriggerMechanism;// Activaction rule for stop orders
                           // 0 or FT_C_STOP_TRIGGER_MECHANISM
                           //      .FT_C_STOP_TRIGGER_MECHANISM_BestPrice
                           // 1 or FT_C_STOP_TRIGGER_MECHANISM
                           //      .FT_C_STOP_TRIGGER_MECHANISM_LastPrice
}

*/
```

```
/*

The FT_C_ERROR_INFO EntityClass
===============================
  Its entityClassID is FT_C_ERROR_INFO_ID = 30050.
  It does not have any key.
  Its structure is something like:

class FT_C_ERROR_INFO {
  int    ReasonCode;
  String ErrorString;
}

The FT_C_TRADING_PHASE class
============================
  This java class is not an EntityClass.
  This class contains
    - all enumeration value for the trading phase,
    - a method enumAsString that returns a displayable String for a given value.
  Its structure is something like:

class FT_C_TRADING_PHASE {
  public static final int FT_C_TRADING_PHASE_UNDEF             =  0;
  public static final int FT_C_TRADING_PHASE_CLOSURE           =  1;
  public static final int FT_C_TRADING_PHASE_PRE_ISSUE         =  2;
  public static final int FT_C_TRADING_PHASE_ISSUE             =  3;
  public static final int FT_C_TRADING_PHASE_PRE_AUCTION       =  4;
  public static final int FT_C_TRADING_PHASE_AUCTION           =  5;
  public static final int FT_C_TRADING_PHASE_POST_AUCTION      =  6;
  public static final int FT_C_TRADING_PHASE_PRE_TRADING       =  7;
  public static final int FT_C_TRADING_PHASE_TRADING           =  8;
  public static final int FT_C_TRADING_PHASE_POST_TRADING      =  9;
  public static final int FT_C_TRADING_PHASE_TRADING_AT_LAST   = 10;
  public static final int FT_C_TRADING_PHASE_TRADING_AFTER_HOUR = 11;
  public static final int FT_C_TRADING_PHASE_FAST_MARKET       = 12;
  public static final int FT_C_TRADING_PHASE_MANAGEMENT        = 13;
  public static final int FT_C_TRADING_PHASE_NO_OPERATION      = 14;

  public static final String enumAsString(int value) { return "an appropriate value"; }
}

*/
```

```
// Effective source-code Example2 starts here.

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.io.FileOutputStream;
import java.io.FilenameFilter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.FileReader;
import java.io.File;
import it.list.jft.*;       // to use the JFT/Api library
import it.list.jft.event.*; // to use the JFT/Api library
import metamarket.*;        // to use MetaMarket, FT_C_ORDER, FT_C_TRADING_STATE, etc...

class Example2 extends UtilityExample implements Runnable {

  static final String      O_SUFFIX       = ".order";
  static final String      O_SUFFIX_READ  = O_SUFFIX + ".read";
  static final String      T_SUFFIX       = ".pending";
  static final String      T_SUFFIX_DONE  = ".done." + System.currentTimeMillis();
  static final String      EXCHANGE_ID    = "HDAT";
  static final int         SUBS_CLIENT_ID = 67890;
  static final int         TRANS_CLIENT_ID= 67891;
  static final int         PHASE_TRADING  = FT_C_TRADING_PHASE.
                                            FT_C_TRADING_PHASE_TRADING;
  static final ThreadGroup THREAD_GROUP   = new ThreadGroupExample();
  static       String      licPath        = null;
  static       String      dirPath        = "c:\\";
  static       String      myOperatorID   = "dario";
  static       String      myOperatorPass = "*";
  static       String      host           = "metamarket.fasttrack.com";
  static       String      service        = null;
  static       int         port           = 1234;
  static       int         delayScanSecs  = 10;
  static       int         delayQuerySecs = 3;
  static       int         traceLevel     = JFT.TRACE_LEVEL_FATAL;
  static       boolean     queryAfterSend = false;
  static       boolean     verbose        = false;
  static       Context     context        = null;
```

```
public static void main(String[]args) {
  handleArgs(args);
  new Thread(THREAD_GROUP, new Example2(), "main").start();
}


public void run() {
  JFT.THIS.init(JFT.MODE_MULTI_THREAD);
  JFT.THIS.setTrace(true);
  JFT.THIS.setTraceLevel(traceLevel);
  JFT.THIS.setTraceMode(this);
  if(true) {
    JFT.THIS.register(new FT_C_ORDER());
    JFT.THIS.register(new FT_C_TRADING_STATE());
    JFT.THIS.register(new FT_C_ERROR_INFO());
  } else // as an expensive alternative
    MetaMarket.registerAll();
  JFT.THIS.start();
  context = JFT.THIS.makeContext();
  new ConnectionForSubscriptions();
  new ConnectionForTransactions();
}


// Just a remind on usage...

static void usage() {
  final String[]usage=
    {"Usage: java Example options...",
     "options: [-h host]         # FastTrack server TCP/IP host",
     "         [-p port]         # FastTrack server TCP/IP port",
     "         [-n serviceName]  # FastTrack service name",
     "         [-o opName]       # operator's name",
     "         [-w opPassword]   # operator's password",
     "         [-l licPathName]  # license file pathname",
     "         [-d dirPathName]  # directory with xxx.order[.trans] files",
     "         [-t traceLevel]   # 0 <= traceLevel <= 5",
     "         [-v ON/OFF]       # trace verbose",
     "         [-s scanDelay]    # scan delay (in seconds)",
     "         [-q queryDelay]   # query delay (in seconds)",
     "         [-x ON/OFF]       # request to make a query after a send"};
  for(int i=0; i<usage.length; i++)
    System.out.println(usage[i]);
  System.exit(0);
}
```

```
  // Parse command-line options.

  static void handleArgs(String[]args) {
    try {
      for(int i=0; i<args.length; i+=2)
        if(args[i].equals("-h"))
          host = args[i+1];
        else if(args[i].equals("-p"))
          port = Integer.parseInt(args[i+1]);
        else if(args[i].equals("-n"))
          service = args[i+1];
        else if(args[i].equals("-o"))
          myOperatorID = args[i+1];
        else if(args[i].equals("-w"))
          myOperatorPass = args[i+1];
        else if(args[i].equals("-l"))
          licPath = args[i+1];
        else if(args[i].equals("-t"))
          traceLevel = Integer.parseInt(args[i+1]);
        else if(args[i].equals("-d"))
          dirPath = args[i+1];
        else if(args[i].equals("-v"))
          verbose = args[i+1].compareToIgnoreCase("ON") == 0;
        else if(args[i].equals("-s"))
          delayScanSecs = Integer.parseInt(args[i+1]);
        else if(args[i].equals("-q"))
          delayQuerySecs = Integer.parseInt(args[i+1]);
        else if(args[i].equals("-x"))
          queryAfterSend = args[i+1].compareToIgnoreCase("ON") == 0;
        else
          throw new IllegalArgumentException();
      if(   args.length == 0
         || host.length() == 0
         || port <= 0
         || service != null && service.length() == 0
         || myOperatorID.length() == 0
         || myOperatorPass.length() == 0
         || traceLevel < JFT.TRACE_LEVEL_DEBUG
         || traceLevel > JFT.TRACE_LEVEL_FATAL
         || licPath != null && ! new File(licPath).canRead()
         || ! new File(dirPath).isDirectory()
         || delayScanSecs < 0
         || delayQuerySecs < 0)
        throw new IllegalArgumentException();
    } catch(Exception e) {
      usage();
    }
  }
}
```

```
// Common superclass for all connections.

abstract class ConnectionExample extends UtilityExample
                                 implements ConnectionListener {
         final Connection connection;

  ConnectionExample(int connectionUserType, int clientID) {
    ConnectionParam cp = Example2.context.makeConnectionParam();
    cp.setHost(Example2.host);
    cp.setPort(Example2.port);
    cp.setService(Example2.service);
    cp.setApplRevision(new int[]{0,0,0});
    cp.setApplSignature(12345);
    cp.setAuthFile(Example2.licPath == null ? null : new File(Example2.licPath));
    cp.setClientID(clientID);
    cp.setConnType(ConnectionParam.CONN_TYPE_TCP);
    cp.setUserName(Example2.myOperatorID);
    cp.setPassword(Example2.myOperatorPass);
    cp.setUserType(connectionUserType);
    connection = Example2.context.makeConnection(cp, this);
    int res = connection.open();
    trace(res);
    if(res != Connection.RESULT_OK)
      connection.release(); // good practice
  }

  public void onConnectionOpen(ConnectionOpenEvent ev) {
    int res = ev.getResult();
    trace(res);
    if(res == ev.RESULT_OK) {
      int[]mrkRev = ev.getMarketRevision();
      trace("csID: " + ev.getClientServiceID() + " bsID: " + ev.getBusinessServiceID()
            + " date: " + UtilityExample.sdf.format(ev.getSystemDateTime())
            + " FTID: " + ev.getFTID() + " env: " + ev.getEnvironment()
            + " mrkRev: " + mrkRev[0] + "." + mrkRev[1] + "." + mrkRev[2]);
    } else
      connection.release(); // good practice
  }

  public void onConnectionClose(ConnectionCloseEvent ev) {
    trace(ev.getResult());
    connection.release(); // good practice
  }

  public void onConnectionLost(ConnectionLostEvent ev) {
    trace(ev.getResult());
    connection.release(); // good practice
  }
}
```

```
// A specific connection: it handles many subscriptions.

class ConnectionForSubscriptions extends ConnectionExample {

  ConnectionForSubscriptions() {
    super(ConnectionParam.USER_TYPE_VIEW, Example2.SUBS_CLIENT_ID);
  }

  public void onConnectionOpen(ConnectionOpenEvent ev) {
    super.onConnectionOpen(ev); // call the overridden method
    if(ev.getResult() == ev.RESULT_OK) {
      new SubscriptionOrder(connection);
      new SubscriptionTradingState(connection);
    }
  }
}
```

```
// Common superclass for all subscriptions.

abstract class SubscriptionExample extends UtilityExample
                                   implements SubscriptionListener {
  final Subscription subscription;

  SubscriptionExample(Connection connection) {
    subscription = Example2.context.makeSubscription(connection,
                                                     makeSubscriptionParam(), this);
    int res = subscription.start();
    trace(res);
    if(res != Subscription.RESULT_OK)
      subscription.release(); // good practice
  }

  abstract SubscriptionParam makeSubscriptionParam();

  abstract String entityAsString(Entity e);

  public void onSubscriptionStart(SubscriptionStartEvent ev){
    trace("Result=" + ev.getResult() +
          (ev.getResult() == ev.RESULT_OK ?
           " version: " + ev.getEntityClassVersionOnServer() +
           " reset: " + ev.isEntityClassReset() : ""));
    if(ev.getResult() != ev.RESULT_OK)
      subscription.release(); // good practice
  }

  public void onSubscriptionIdle(SubscriptionIdleEvent ev){
    trace(ev.getResult());
  }

  public void onSubscriptionNotify(SubscriptionNotifyEvent ev){
    switch(ev.getAction()) {
    case SubscriptionNotifyEvent.ACTION_ENTITY_ADD:
    case SubscriptionNotifyEvent.ACTION_ENTITY_RWT:
      trace((ev.getAction() == ev.ACTION_ENTITY_ADD ? "ADD" : "RWT")
            + " Masked: " + ev.isMasked() + " " + entityAsString(ev.getEntity()));
      break;
    case SubscriptionNotifyEvent.ACTION_ENTITY_DEL:
      trace("DEL KeyID: " + ev.getKeyID());
      break;
    case SubscriptionNotifyEvent.ACTION_ENTITY_KIL:
      if(ev.getKeyID() <= 0)
        trace("KIL ClassReset - New Version: " + ev.getTimeStamp().getDateTime());
      else
        trace("KIL KeyID: " + ev.getKeyID());
      break;
    }
  }

  public void onSubscriptionStop(SubscriptionStopEvent ev){
    trace(ev.getResult());
    subscription.release(); // good practice
  }
}
```

```
// A specific subscription: it handles FT_C_ORDER.

class SubscriptionOrder extends SubscriptionExample {

  SubscriptionOrder(Connection connection) {
    super(connection);
  }

  SubscriptionParam makeSubscriptionParam() {
    SubscriptionParam sp = Example2.context.makeSubscriptionParam();
    sp.setEntityClassID(MetaMarket.FT_C_ORDER_ID);
    return sp;
  }

  String entityAsString(Entity e) {
    FT_C_ORDER o = (FT_C_ORDER) e;
    return (o.OperatorID.equals(Example2.myOperatorID) ? "" : "NO_OWNER")
      + " OrderID: "      + o.OrderID
      + " FTSecID: "      + o.FTSecID
      + " OperatorID: "   + o.OperatorID
      + " Price: "        + o.Price
      + " Qty: "          + o.Qty
      + " Verb: "         + FT_C_VERB.enumAsString(o.Verb)
      + " ValidityDate: " + o.ValidityDate;
  }

  public void onSubscriptionNotify(SubscriptionNotifyEvent ev){
    super.onSubscriptionNotify(ev);  // call the overridden method
     FT_C_ORDER o = (FT_C_ORDER) ev.getEntity();
     if(! o.OperatorID.equals(Example2.myOperatorID))
       writeOrder(o);
  }

  void writeOrder(FT_C_ORDER o) {
    PrintWriter pw = null;
    try {
      pw = new PrintWriter(new FileOutputStream(new File(
                           Example2.dirPath, o.FTSecID + Example2.O_SUFFIX_READ)));
      pw.println(o.FTSecID);
      pw.println(o.OrderID);
      pw.println(o.OperatorID);
      pw.println(o.MrkOperatorID);
      pw.println(o.Verb);
      pw.println(o.OrderType);
      pw.println(o.QtyParameter);
      pw.println(o.TimeInForce);
      pw.println(o.ValidityDate);
      pw.println(o.Status);
      pw.println(o.Price);
      pw.println(o.Qty);
      pw.println(o.StopPrice);
      pw.println(o.TriggerMechanism);
      trace("order file written.");
    } catch(Exception e) {
      trace(e);
    } finally {
      try {pw.close();} catch(Exception e) {}
    }
  }
}
```

```
// Another specific subscription: it handles FT_C_TRADING_STATE.

class SubscriptionTradingState extends SubscriptionExample {

  int counter; // count the # of refreshEntity requested

  SubscriptionTradingState(Connection connection) {
    super(connection);
  }

  SubscriptionParam makeSubscriptionParam() {
    Mask m = JFT.THIS.makeEmptyMask(MetaMarket.FT_C_TRADING_STATE_ID);
    m.addFieldByName("ExchangeID"); // primary key field
    m.addFieldByName("MarketID");   // primary key field
    m.addFieldByName("SectionID");  // primary key field
    m.addFieldByName("Phase");      // what I'm searching!
    SubscriptionParam sp = Example2.context.makeSubscriptionParam();
    sp.setEntityClassID(MetaMarket.FT_C_TRADING_STATE_ID);
    sp.setMask(m);
    sp.setQueryType(sp.QUERY_TYPE_SET);
    FT_C_TRADING_STATE ts = new FT_C_TRADING_STATE();
    ts.ExchangeID = Example2.EXCHANGE_ID;
    sp.setEntityKey(ts.getPartialEntityKey(MetaMarket.FT_C_TRADING_STATEKey, 1));
    return sp;
  }

  String entityAsString(Entity e) {
    FT_C_TRADING_STATE ts = (FT_C_TRADING_STATE) e;
    return "ExchangeID: " + ts.ExchangeID // in mask and partial key subscribed
        + " MarketID: "   + ts.MarketID                                  // in mask
        + " SectionID: "  + ts.SectionID                                 // in mask
        + " Phase: "      + FT_C_TRADING_PHASE.enumAsString(ts.Phase)// in mask
        + " Status: "     + FT_C_TRADING_STATUS.enumAsString(ts.Status)
        + " PhaseDesc: "  + ts.PhaseDescription
        + " Time: "       + ts.Time;
  }

  public void onSubscriptionIdle(SubscriptionIdleEvent ev){
    super.onSubscriptionIdle(ev); // call the overridden method
    checkStop();
  }

  public void onSubscriptionNotify(SubscriptionNotifyEvent ev){
    super.onSubscriptionNotify(ev); // call the overridden method
    if(ev.isMasked()) {
      FT_C_TRADING_STATE ts = (FT_C_TRADING_STATE) ev.getEntity();
      if(ts.Phase == Example2.PHASE_TRADING) {
        int res = subscription.refreshEntity(ts.getFullEntityKey(ev.getKeyID()));
        trace(res);
        if(res == subscription.RESULT_OK)
          counter++;
      }
    } else {
      counter--;
      checkStop();
    }
  }
```

```
  void checkStop() {
    if(counter <= 0) { // Now I'm no more interested in this subscription
      int res = subscription.stop();
      trace(res);
      subscription.release(); // good practice
    }
  }
}


// Another specific connection: it handles many transactions.

class ConnectionForTransactions extends ConnectionExample
                                implements FilenameFilter, Runnable{
  String[]files;

  ConnectionForTransactions() {
    super(ConnectionParam.USER_TYPE_TRADER, Example2.TRANS_CLIENT_ID);
  }

  public void onConnectionOpen(ConnectionOpenEvent ev) {
    super.onConnectionOpen(ev);  // call the overridden method
    if(ev.getResult() == ev.RESULT_OK) {
      files = new File(Example2.dirPath).list(this);
      int n = (files == null) ? 0 : files.length;
      trace(n + " orders to be sent or monitored");
      if(n > 1)
        Arrays.sort(files);
      new Thread(Example2.THREAD_GROUP, this, "listing").start();
    }
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(Example2.O_SUFFIX) && ! name.startsWith(Example2.O_SUFFIX);
  }

  public void run() {
    sleep(Example2.delayScanSecs);
    if(files != null)
      for(int i=0; i<files.length; i++)
        try {
          String filename = files[i] + Example2.T_SUFFIX;
          File f = new File(Example2.dirPath, filename);
          if(f.canRead()) {
            trace("analyzing file " + filename + ": pending trans. to be monitored");
            new TransactionPending(connection, files[i]);
          } else {
            trace("analyzing file " + files[i] + ": new transaction to be created");
            new TransactionNew(connection, files[i]);
          }
          sleep(Example2.delayScanSecs);
        } catch(Exception e) {
          trace(i + " -> " + e);
        }
    JFT.THIS.release();
    trace("JFT library released -> application going to die");
    // no need to explicitly call System.exit() here !
  }
}
```

```
// Common superclass for all transactions.

abstract class TransactionExample extends UtilityExample
                                  implements TransactionListener, Runnable {
  final Transaction transaction;
  final String      filename;
        int         counter;

  TransactionExample(Connection connection, String f) {
    super(f);
    filename = f;
    transaction = Example2.context.makeTransaction(connection,
                                                   makeTransactionParam(), this);
    tidWrite();
  }

  abstract TransactionParam makeTransactionParam();

  String reasonAsString(TransactionEvent ev) {
    int reason = ev.getReasonCode();
    FT_C_ERROR_INFO ei = (FT_C_ERROR_INFO) ev.getEntity();
    return reason == 0 ? "" :
      (" reason: " + reason + (ei == null ? "" : " -> " + ei.ErrorString));
  }

  void onTransaction(TransactionEvent ev, String whoami) {
    int st = transaction.getStatus();
    trace(whoami
          + ev.getResult()
          + (st == transaction.STATUS_FLYING ? " FLYING"
             : (st == transaction.STATUS_ABORTED ? " ABORTED"
                : (st == transaction.STATUS_COMMITTED ? " COMMITTED" : " " + st)))
          + reasonAsString(ev));
    if(st == transaction.STATUS_FLYING)
       if(Example2.queryAfterSend)
         queryDelayed();
       else
         transaction.release(); // Now I'm no more interested in this transaction
    else
      destroy();
  }

  public void onTransactionSend(TransactionSendEvent ev) {
    onTransaction(ev, "onTransactionSend -> ");
  }

  public void onTransactionQuery(TransactionQueryEvent ev) {
    onTransaction(ev, "onTransactionQuery -> ");
  }

  void destroy() {
    tidRemove();
    transaction.release(); // Now I'm no more interested in this transaction
  }

  void query() {
    int res = transaction.query();
    trace(res);
    if(res != transaction.RESULT_OK)
      destroy();
  }
```

```
  void queryDelayed() {
    new Thread(Example2.THREAD_GROUP, this,
               "query " + (++counter) + " on " + filename).start();
  }

  TransactionID tidRead() {
    BufferedReader in = null;
    try {
      in = new BufferedReader(new FileReader(new File(Example2.dirPath,
                                                      filename + Example2.T_SUFFIX)));
      TransactionID tid = JFT.THIS.makeTransactionID(
                              Integer.parseInt(in.readLine()),
                              Integer.parseInt(in.readLine()),
                              Integer.parseInt(in.readLine()),
                              JFT.THIS.makeTimeStamp(Integer.parseInt(in.readLine()),
                                                     Integer.parseInt(in.readLine())));
      trace("transaction file read.");
      return tid;
    } catch(Exception e) {
      trace(e);
      return null;
    } finally {
      try {in.close();} catch(Exception e) {}
    }
  }

  void tidWrite() {
    TransactionID tid = transaction.getTransactionID();
    PrintWriter   pw  = null;
    try {
      pw = new PrintWriter(new FileOutputStream(new File(Example2.dirPath,
                                                         filename+Example2.T_SUFFIX)));
      pw.println(tid.getClientID());
      pw.println(tid.getClientServiceID());
      pw.println(tid.getBusinessServiceID());
      pw.println(tid.getTimeStamp().getDateTime());
      pw.println(tid.getTimeStamp().getProg());
      trace("transaction file written.");
    } catch(Exception e) {
      trace(e);
    } finally {
      try {pw.close();} catch(Exception e) {}
    }
  }

  void tidRemove() {
      File oldF = new File(Example2.dirPath, filename + Example2.T_SUFFIX);
      File newF = new File(Example2.dirPath, filename + Example2.T_SUFFIX_DONE);
      boolean ok = oldF.renameTo(newF);
      trace("transaction file renamed: " + ok);
  }

  public void run(){
    try {
      sleep(Example2.delayQuerySecs);
      query();
    } catch(Exception e) {
      trace(e);
    }
  }
}
```

```
class TransactionNew extends TransactionExample { // specific new transaction

  TransactionNew(Connection connection, String f) {
    super(connection, f);
    int res = transaction.send();
    trace(res);
    if(res != transaction.RESULT_OK)
      destroy();
  }

  TransactionParam makeTransactionParam() {
    TransactionParam tp = Example2.context.makeTransactionParam();
    tp.setAction(TransactionParam.ACTION_ENTITY_ADD);
    tp.setEntity(readOrder());
    tp.setKeyID(MetaMarket.FT_C_ORDERKey);
    return tp;
  }

  FT_C_ORDER readOrder() {
    BufferedReader in = null;
    try {
      FT_C_ORDER order  = new FT_C_ORDER();
      in = new BufferedReader(new FileReader(new File(Example2.dirPath, filename)));
      order.FTSecID          = in.readLine();
      order.OrderID          = in.readLine();
      order.OperatorID       = in.readLine(); // overwritten below !
      order.MrkOperatorID    = in.readLine();
      order.Verb             = Integer.parseInt(in.readLine());
      order.OrderType        = Integer.parseInt(in.readLine());
      order.QtyParameter     = Integer.parseInt(in.readLine());
      order.TimeInForce      = Integer.parseInt(in.readLine());
      order.ValidityDate     = Integer.parseInt(in.readLine());
      order.Status           = Integer.parseInt(in.readLine());
      order.Price            = Double.parseDouble(in.readLine());
      order.Qty              = Double.parseDouble(in.readLine());
      order.StopPrice        = Double.parseDouble(in.readLine());
      order.TriggerMechanism = Integer.parseInt(in.readLine());
      order.OperatorID = Example2.myOperatorID; // overwritten with the right value !
      trace("file read");
      return order;
    } catch(Exception e) {
      trace(e);
      return null;
    } finally {
      try {in.close();} catch(Exception e) {}
    }
  }
}

class TransactionPending extends TransactionExample { // specific past pending transact.
  TransactionPending(Connection connection, String f) {
    super(connection, f);
    query();
  }

  TransactionParam makeTransactionParam() {
    TransactionParam tp = Example2.context.makeTransactionParam();
    tp.setPendingTransactionID(tidRead());
    return tp;
  }
}
```

```
// Common superclass for all classes of this example.

abstract class UtilityExample implements Tracer {
  static final SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
         final String           specificName;

  UtilityExample() {
    specificName = "";
  }

  UtilityExample(String name) {
    specificName = "<" + name + ">";
  }

  void internalTrace(String line) {
    String m = getClass().getName() + specificName;
    Throwable t = new Throwable();
    StackTraceElement[] ste = t.getStackTrace();
    if(ste != null && ste.length >= 3) {
      m += "." + ste[2].getMethodName();
      if(Example2.verbose) {
        String fn = ste[2].getFileName();
        int ln = ste[2].getLineNumber();
        m += "(" + (fn == null ? "" : fn) + (ln < 0 ? "" : ":" + ln) + ")";
      }
    }
    JFT.THIS.trace(m, JFT.TRACE_LEVEL_FATAL, line);
    // In this simple example the level is always TRACE_LEVEL_FATAL
    // so we will see our application trace
    // whichever "-t ..." command-line option was choosen.
  }

  void trace(String message) { internalTrace(message); }

  void trace(int res)        { internalTrace("Result=" + res); }

  void trace(Exception e)    { internalTrace("Exception: " + e); }

  public void onTrace(Date t, String m, int l, String ms) {
    String v = Example2.verbose ?
      sdf.format(t) + " " + l + " [" + Thread.currentThread().getName() + "] " : "";
    System.out.println(v + "[" + m + "] " + ms);
  }

  void sleep(int intervalSecs) {
    try {
      Thread.sleep(intervalSecs * 1000L);
    } catch(InterruptedException ie) {
      trace(ie);
    }
  }
}
```

```
// Specific ThreadGroup to handle in an uniform way all generated exceptions.

class ThreadGroupExample extends ThreadGroup {

  ThreadGroupExample() {
    super("Example");
  }

  public void uncaughtException(Thread t, Throwable e) {
    if(e instanceof ThreadDeath)
      super.uncaughtException(t, e); // call the overridden method
    else {
      System.out.println("Uncaught Exception in thread " + t.getName());
      e.printStackTrace(System.out);
      System.exit(0);
    }
  }
}
```

Example 3

The third example extend the second one adding a query facility.
To use this example you have to compile both second and third examples.

```
/*

Description
===========
This simple application example open a connection with a MetaMarket service
using YAS service manager, and then it subscribe orders
of some securities using EntityFilter.

*/

// Effective source-code Example3 starts here.

import java.io.*;

import it.list.jft.*; // to use the JFT/Api library

import it.list.jft.event.*; // to use the JFT/Api library

import metamarket.*; // to use MetaMarket, FT_C_ORDER, FT_C_TRADING_STATE, etc...

public class Example3 implements ConnectionListener, SubscriptionListener, FilterListener

{

       Context context;
       Connection connection;
       EntityFilter filter;


       public Example3()

       {
```

```
        JFT.THIS.init(JFT.MODE_MULTI_THREAD);

        System.out.println("Starting...");

        JFT.THIS.setTrace(true);
        JFT.THIS.setTraceLevel(JFT.TRACE_LEVEL_DEBUG);

        // uncomment this line for debug trace on system out

        // JFT.THIS.setTraceMode(true, new PrintWriter(System.out));

        // use this if you want to print trace on a file

        // JFT.THIS.setTraceMode(true, new File("C:\\TRACE_Test.txt"));

        // register MetaMarket order class

        JFT.THIS.register(new FT_C_ORDER());
        JFT.THIS.start();

        context = JFT.THIS.makeContext();
        ConnectionParam p = context.makeConnectionParam();

        // change following parameters to fit your configuration

        p.setHost("194.91.195.36");

        p.setPort(37000);

        p.setUserName("marco");

        p.setPassword("*");

        p.setClientID(12345);
        p.setUserType(ConnectionParam.USER_TYPE_TRADER);

        p.setService("METAMARKET");

        connection = context.makeConnection(p, this);

        System.out.println("Open connection result: " + connection.open());

}

public static void main(String[] args)
{

        Example3 test = new Example3();

}

// connection listener interface

public void onConnectionOpen(ConnectionOpenEvent event)
{
System.out.println(event);
```

```
// crete EntityFilter
FilterParam filterparam = context.makeFilterParam();

filterparam.setEntityClassID(MetaMarket.FT_C_ORDER_ID);
filterparam.setType(EntityFilter.TYPE_ENTITYFILTER);

filter = (EntityFilter)context.makeFilter(connection, filterparam, this);

System.out.println("Filter Create Result: " + filter.create());

}

public void onConnectionClose(ConnectionCloseEvent event)
{
System.out.println(event);
}

public void onConnectionLost(ConnectionLostEvent event)
{
System.out.println(event);
}

// filter listener interface
public void onFilterCreate(FilterCreateEvent event)
{
System.out.println(event);

if (event.getResult() == FilterCreateEvent.RESULT_OK)
{
FT_C_ORDER order = new FT_C_ORDER();

SubscriptionParam param = context.makeSubscriptionParam();
param.setEntityClassID(MetaMarket.FT_C_ORDER_ID);
param.setFilter(filter);
param.setEntityKey(order.getFullEntityKey(MetaMarket.FT_C_ORDERKey));

Subscription sub = context.makeSubscription(connection, param, this);

System.out.println("Subscribing Result FT_C_ORDER: " + sub.start());

}
}

public void onFilterSet(FilterSetEvent event)
{
System.out.println(event);
}

public void onFilterDestroy(FilterDestroyEvent event)
{
System.out.println(event);
}

// subscription listener interface

public void onSubscriptionStart(SubscriptionStartEvent event)
{

       System.out.println(event);
```

```
            if (event.getResult()==SubscriptionStartEvent.RESULT_OK)
            {

                  FT_C_ORDER order = new FT_C_ORDER();

                  order.FTSecID = "BITMTAACE";
                  filter.add(order.getPartialEntityKey(MetaMarket.FT_C_ORDERKey,1));
                  order.FTSecID = "BITMTAF";
                  filter.add(order.getPartialEntityKey(MetaMarket.FT_C_ORDERKey,1));
                  order.FTSecID = "BITMTACSP";
                  filter.add(order.getPartialEntityKey(MetaMarket.FT_C_ORDERKey,1));
                  filter.flush();

                  // now you will receive the order of this three
                  security only

            }

      }

      public void onSubscriptionStop(SubscriptionStopEvent event)
      {

            System.out.println(event);

      }

      public void onSubscriptionIdle(SubscriptionIdleEvent event)
      {

            System.out.println(event);

      }

      public void onSubscriptionNotify(SubscriptionNotifyEvent event)
      {

            System.out.println(event);

      }


}
```

Contacts

Any comments or requests for clarifications are welcome.

Email

Marketing: marketing@list−group.com

General Support: service−desk@list−group.com

FT/API Programming Support: ftapi@list−group.com

Website

www.list−group.com