



# Persisting Java Objects with InterSystems XEP

Version 2023.3  
2024-05-16

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble®, InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Response Center (WRC)**

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: [support@InterSystems.com](mailto:support@InterSystems.com)

# Table of Contents

<b>Using XEP with Java Overview .....</b>	<b>1</b>
<b>1 Introduction to XEP .....</b>	<b>3</b>
1.1 Requirements and Configuration .....	3
1.1.1 Requirements .....	3
1.1.2 Installation .....	3
1.1.3 Configuration .....	4
<b>2 Using XEP Event Persistence .....</b>	<b>5</b>
2.1 Introduction to Event Persistence .....	5
2.1.1 The XepSimple Sample Application .....	6
2.2 Creating and Connecting an EventPersister .....	8
2.3 Importing a Schema .....	9
2.4 Storing and Modifying Events .....	10
2.4.1 Creating and Storing Events .....	10
2.4.2 Accessing Stored Events .....	11
2.4.3 Controlling Index Updating .....	12
2.5 Using Queries .....	13
2.5.1 Creating and Executing a Query .....	13
2.5.2 Processing Query Data .....	14
2.5.3 Defining the Fetch Level .....	16
2.6 Schema Customization and Mapping .....	17
2.6.1 Schema Import Models .....	17
2.6.2 Using Annotations .....	18
2.6.3 Using IdKeys .....	21
2.6.4 Implementing an InterfaceResolver .....	22
2.6.5 Schema Mapping Rules .....	24
<b>3 XEP Quick Reference .....</b>	<b>27</b>
3.1 XEP Quick Reference .....	27
3.1.1 List of XEP Methods .....	27
3.1.2 Class PersisterFactory .....	29
3.1.3 Class EventPersister .....	29
3.1.4 Class Event .....	33
3.1.5 Class EventQuery<T> .....	35
3.1.6 Class EventQueryIterator<T> .....	38
3.1.7 Interface InterfaceResolver .....	38
3.1.8 Class XEPException .....	39



# Using XEP with Java Overview

InterSystems IRIS® provides lightweight Java SDKs for easy database access via relational tables (JDBC and SQL), objects, and multidimensional storage. See [Using Java with InterSystems Software](#) for relational table access with JDBC, and [Using the Native SDK for Java](#) for multidimensional storage access. This book describes how to use XEP for object access.

XEP provides high-performance persistence technology for Java object hierarchies. XEP projects the data in Java objects as *persistent events* (database objects that store a persistent copy of the data fields), accessing the InterSystems IRIS database. XEP is optimized for transaction processing applications that require high speed data persistence and retrieval.

The following topics are discussed in this book:

- [Introduction](#) — provides an overview of the platform architecture, and describes common installation procedures.
- [Using XEP Event Persistence](#) — describes XEP in detail and provides code examples.
- [Quick Reference for XEP Classes](#) — Provides a quick reference for methods of the XEP classes.

There is also a detailed [Table of Contents](#).

## Related Documents

The following documents also contain related material:

- See the [Java XEP online class documentation](#) (javadoc) for the most recent information on XEP classes.
- [Using Java with InterSystems Software](#) describes how to connect to InterSystems IRIS from an external application using the InterSystems JDBC driver, and how to access external JDBC data sources from InterSystems IRIS via SQL.
- [Using the Native SDK for Java](#) describes how to use the Native SDK.
- “Hibernate Support” in the [InterSystems IRIS Implementation Reference for Java Third Party APIs](#) describes the InterSystems IRIS dialect of Hibernate. This dialect implements support for Java Persistence Architecture (JPA), which is the recommended persistence technology for large, complex object hierarchies in Java projects.



# 1

## Introduction to XEP

XEP is a lightweight Java SDK that provides high-performance Java technology for persisting simple to moderately complex object hierarchies. XEP projects the data in Java objects as *persistent events* (database objects that store a persistent copy of the data fields) in an InterSystems IRIS™ database. XEP is optimized for applications that must acquire and persist data at the highest possible speed.

InterSystems IRIS provides Java SDKs for easy relational and object database access. This book discusses XEP object access. See *Using Java with InterSystems Software* for JDBC relational access.

### 1.1 Requirements and Configuration

This section specifies requirements and provides instructions for configuring your environment to use XEP.

#### 1.1.1 Requirements

- A 64-bit Java JDK supported by this release of InterSystems IRIS (see “Supported Java Technologies” in the *InterSystems Supported Platforms* document for this release).
- InterSystems IRIS.

Computers that run your Java client applications do not require InterSystems IRIS, but they must have a connection to the InterSystems IRIS Server and must be running a supported version of the Java JDK.

#### 1.1.2 Installation

- When installing InterSystems IRIS, select the Development environment:
  - In Windows, select the Setup Type: Development option during installation.
  - In UNIX® and related operating systems, select the 1) Development - Install InterSystems IRIS server and all language bindings option during installation (see “[Standard InterSystems IRIS Installation Procedure](#)” in the UNIX® and Linux section of the *Installation Guide*).
- If InterSystems IRIS has been installed with security level 2, open the Management Portal and go to System Administration > Security > Services, select %Service\_CallIn, and make sure the Service Enabled box is checked.

If you installed InterSystems IRIS with security level 1 (minimal) it should already be checked.

## 1.1.3 Configuration

All XEP applications require JAR files `intersystems-jdbc-3.0.0.jar` and `intersystems-xep-3.0.0.jar` (see “[The InterSystems Java Class Packages](#)” in *Using Java with InterSystems Software* for file location and other details). Your `CLASSPATH` environment variable must include the full paths to these files. Alternately, they can be specified in the Java command line `classpath` argument.

**Note:**    **Additional Configuration for Windows**

The default stack size of the Java Virtual Machine on Windows is too small for running XEP applications (running them with the default stack size causes Java to report `EXCEPTION_STACK_OVERFLOW`). To optimize performance, heap size should also be increased. To temporarily modify the stack size and heap size when running an XEP application, add the following command line arguments:

```
-Xss1024k  
-Xms2500m -Xmx2500m
```



# 2

## Using XEP Event Persistence

XEP provides extremely rapid storage and retrieval of Java structured data. It provides ways to control schema generation for optimal mapping of complex data structures, but schemas for simpler data structures can often be generated and used without modification.

The following topics are discussed in this chapter:

- [Introduction to Event Persistence](#) — introduces persistent event concepts and terminology, and provides a simple example of code that uses the XEP API.
- [Creating and Connecting an EventPersister](#) — describes how to create an instance of the EventPersister class and use it to open, test, and close a database connection.
- [Importing a Schema](#) — describes the methods and annotations used to analyze a Java class and generate a schema for the corresponding persistent event.
- [Storing and Modifying Events](#) — describes methods of the Event class used to store, modify, and delete persistent events.
- [Using Queries](#) — describes methods of the XEP classes that create and process query resultsets.
- [Schema Customization and Mapping](#) — provides a detailed description of how Java classes are mapped to database schemas, and how to generate customized schemas for optimal performance.

**Note:**    **Why “Persistent Event”?**

The term *persistent event* originally referred to data acquired from real time events. The current implementation of XEP is designed for much more than simple event processing, so XEP “persistent events” are actually database objects that XEP has persisted from any source.

## 2.1 Introduction to Event Persistence

A *persistent event* is an InterSystems IRIS database object that holds a persistent copy of the data fields in a Java object. By default, XEP stores each event as a standard %Persistent object. Storage is automatically configured so that the data will be accessible to InterSystems IRIS by other means, such as objects, SQL, or direct global access.

Before a persistent event can be created and stored, XEP must analyze the corresponding Java class and *import a schema*, which defines how the data structure of a Java object is projected to a persistent object in the database. A schema can use either of the following two object projection models:

- The default model is the *flat schema*, where all referenced objects are serialized and stored as part of the imported class, and all fields inherited from superclasses are stored as if they were native fields of the imported class. This is the fastest and most efficient model, but does not preserve any information about the original Java class structure.
- If structural information must be preserved, the *full schema* model may be used. This preserves the full Java inheritance structure by creating a one-to-one relationship between Java source classes and InterSystems IRIS projected classes, but may impose a slight performance penalty.

See “[Schema Import Models](#)” for a detailed discussion of both models, and “[Schema Mapping Rules](#)” for detailed information about how various datatypes are projected.

When importing a schema, XEP acquires basic information by analyzing the Java class. You can supply additional information that allows XEP to generate indexes (see “[Using IdKeys](#)”) and override the default rules for importing fields (see “[Using Annotations](#)” and “[Implementing an InterfaceResolver](#)”).

Once a schema has been imported, XEP can be used to store, query, update and delete data at very high rates. Stored events are immediately available for querying, or for full object or global access. The `EventPersister`, `Event`, and `EventQuery<T>` classes provide the main features of the XEP API. They are used in the following sequence:

- The `EventPersister` class provides methods to establish and control a database connection (see “[Creating and Connecting an EventPersister](#)”).
- Once the connection has been established, other `EventPersister` methods can be used to import a schema (see “[Importing a Schema](#)”).
- The `Event` class provides methods to store, update, or delete events, create a query, and control index updating (see “[Storing and Modifying Events](#)”).
- The `EventQuery<T>` class is used to execute simple SQL queries that retrieve sets of events from the database. It provides methods to iterate through the resultset and update or delete individual events (see “[Using Queries](#)”).

The following section describes a very simple application that demonstrates all of these features.

## 2.1.1 The XepSimple Sample Application

The following program demonstrates how to work with XEP and InterSystems IRIS. `XepSimple` is a small program, but it provides examples for most of the key XEP features and will give you an overview of how the XEP API is used.

The program is divided into four sections, each of which demonstrates one of the four main XEP classes, `EventPersister`, `Event`, `EventQuery`, and `EventQueryIterator`. The imported `xep.samples.SingleStringSample` class is listed at the end of this section.

### class XepSimple

```
package xepsimple;
import com.intersystems.xep.*;
import xep.samples.SingleStringSample;

public class XepSimple {
    public static void main(String[] args) throws Exception {
        // Generate 12 SingleStringSample objects for use as test data
        SingleStringSample[] sampleArray = SingleStringSample.generateSampleData(12);

        // EventPersister
        EventPersister xepPersister = PersisterFactory.createPersister();
        xepPersister.connect("127.0.0.1",51774,"User","_SYSTEM","SYS"); // connect to localhost
        xepPersister.deleteExtent("xep.samples.SingleStringSample"); // remove old test data
        xepPersister.importSchema("xep.samples.SingleStringSample"); // import flat schema

        // Event
        Event xepEvent = xepPersister.getEvent("xep.samples.SingleStringSample");
        for (int i=0; i < sampleArray.length; i++) {
            SingleStringSample sample = sampleArray[i]; // array initialized on line 8
            sample.name = "Sample object #" + i;
            xepEvent.store(sample);
        }
    }
}
```

```

        System.out.println("Persisted " + sample.name);
    }

    // EventQuery
    String sqlQuery = "SELECT * FROM xep_samples.SingleStringSample WHERE %ID BETWEEN ? AND ?";

    EventQuery<SingleStringSample> xepQuery = xepEvent.createQuery(sqlQuery);
    xepQuery.setParameter(1,3);    // assign value 3 to first SQL parameter
    xepQuery.setParameter(2,12);   // assign value 12 to second SQL parameter
    xepQuery.execute();            // get resultset for IDs between 3 and 12

    // EventQueryIterator
    EventQueryIterator<SingleStringSample> xepIter = xepQuery.getIterator();
    while (xepIter.hasNext()) {
        SingleStringSample newSample = xepIter.next();
        newSample.name = newSample.name + " has been updated";
        xepIter.set(newSample);
        System.out.println(newSample.name);
    }

    xepQuery.close();
    xepEvent.close();
    xepPersister.close();
} // end main()
} // end class XepSimple

```

XepSimple performs the following tasks:

- First, some sample objects are generated by calling a method of our sample data class, `xep.samples.SingleStringSample`.
- *EventPersister is the main entry point for XEP, and provides the connection to the database.*

It creates an instance named *xepPersister*, which establishes a JDBC connection to the User namespace in the database and deletes any existing sample data. It then calls **importSchema()** to analyze the sample class and send the schema to the database, thus creating an extent to hold persistent `SingleStringSample` objects.

- *Event encapsulates an interface between a Java object and the corresponding database object.*

Once the schema has been generated, *xepPersister* can create an Event object named *xepEvent* for the sample class. In the loop, each instance of `SingleStringSample` is modified and then persisted to the database. The *xepEvent store()* method takes advantage of the connection and schema defined in *xepPersister*.

- *EventQuery is used to prepare and execute SQL queries.*

An `EventQuery<SingleStringSample>` object named *xepQuery* is created by passing a query string to the *xepEvent* object's **createQuery()** method. The string defines an SQL query that accepts two parameters (the ? characters). The parameter values are defined by calls to **setParameter()**, and a call to **execute()** fetches the query resultset.

- *EventQueryIterator is used to read rows from the resultset and update or delete the corresponding persistent objects.*

Now that *xepQuery* contains the query resultset, an iterator named *xepIter* can be created for it by calling **getIterator()**. In the loop, the iterator's **next()** method is used to get each row of data and assign it to a `SingleStringSample` object. The object is then modified, and the iterator's **set()** method updates the corresponding persistent object in the database.

- When processing is done, it cleans up by calling the **close()** methods for the XEP objects.

Here is a listing of the sample class used in the XepSimple application:

## xep.samples.SingleStringSample

### Java

```
public class SingleStringSample {
    public String name;
    public SingleStringSample() {}
    SingleStringSample(String str) {
        name = str;
    }

    public static SingleStringSample[] generateSampleData(int objectCount) {
        SingleStringSample[] data = new SingleStringSample[objectCount];
        for (int i=0;i<objectCount;i++) {
            data[i] = new SingleStringSample("single string test");
        }
        return data;
    }
}
```

## 2.2 Creating and Connecting an EventPersister

The EventPersister class is the main entry point for the XEP API. It provides the methods for connecting to the database, importing schemas, handling transactions, and creating instances of Event to access events in the database.

An instance of EventPersister is created and destroyed by the following methods:

- PersisterFactory.[createPersister\(\)](#) — returns a new instance of EventPersister.
- EventPersister.[close\(\)](#) — closes this EventPersister instance and releases associated resources.

The following method is used to create a connection:

- EventPersister.[connect\(\)](#) — takes arguments for *host*, *port*, *namespace*, *username*, *password*, and establishes a connection to the specified InterSystems IRIS namespace.

The following example establishes a connection:

### Creating and Connecting an EventPersister: Creating a connection

```
// Open a connection
String host = "127.0.0.1";
int port = 51774;
String namespace = "USER";
String username = "_SYSTEM";
String password = "SYS";
EventPersister myPersister = PersisterFactory.createPersister();
myPersister.connect(host, port, namespace, username, password);
// perform event processing here . . .
myPersister.close();
```

The EventPersister.[connect\(\)](#) method establishes a connection to the specified port and namespace of the host machine. If no connection exists in the current process, a new connection is created. If a connection already exists, the method returns a reference to the existing connection object.

If the server address is 127.0.0.1 or localhost, the connection will default to a local *shared memory connection*, which is faster than the standard TCP/IP connection (see “[Shared Memory Connections](#)” in [Using Java with the InterSystems JDBC Driver](#)).

#### Note: Always call close() to release resources

Always call [close\(\)](#) on an instance of EventPersister before it goes out of scope to ensure that all locks, licenses, and other resources associated with the connection are released.

## 2.3 Importing a Schema

Before an instance of a Java class can be stored as a persistent event, a schema must be imported for the class. The schema defines the database structure in which the event will be stored. XEP provides two different schema import models: *flat schema* and *full schema*. The main difference between these models is the way in which Java objects are projected to InterSystems IRIS events. A flat schema is the optimal choice if performance is essential and the event schema is fairly simple. A full schema offers a richer set of features for more complex schemas, but may have an impact on performance. See “[Schema Customization and Mapping](#)” for a detailed discussion of schema models and related subjects.

The following methods are used to analyze a Java class and import a schema of the desired type:

- `EventPersister.importSchema()` — imports a *flat schema*. Takes an argument specifying a .jar file name, a fully qualified class name, or an array of class names, and imports all classes and any dependencies found in the specified locations. Returns a String array containing the names of all successfully imported classes.
- `EventPersister.importSchemaFull()` — imports a *full schema*. Takes the same arguments and returns the same class list as `importSchema()`. A class imported by this method must declare a user-generated IdKey (see “[Using IdKeys](#)”).
- `Event.isEvent()` — a static Event method that takes a Java object or class name of any type as an argument, tests to see if the specified object can be projected as a valid XEP event (see “[Requirements for Imported Classes](#)”), and throws an appropriate error if it is not valid.

The import methods are identical except for the schema model used. The following example imports a simple test class and its dependent class:

### Importing a Schema: Importing a class and its dependencies

The following classes from package test are to be imported:

```
public class MainClass {
    public MainClass() {}
    public String myString;
    public test.Address myAddress;
}

public class Address {
    public String street;
    public String city;
    public String state;
}
```

The following code uses `importSchema()` to import the main class, `test.MainClass`, after calling `isEvent()` to make sure it can be projected. Dependent class `test.Address` is also imported automatically when `test.MainClass` is imported:

```
try {
    Event.isEvent("test.MainClass"); // throw an exception if class is not projectable
    myPersister.importSchema("test.MainClass");
}
catch (XEPEException e) {System.out.println("Import failed:\n" + e);}
```

In this example, instances of dependent class `test.Address` will be serialized and embedded in the same InterSystems IRIS object as other fields of `test.MainClass`. If `importSchemaFull()` had been used instead, stored instances of `test.MainClass` would contain references to instances of `test.Address` stored in a separate InterSystems IRIS class extent.

## 2.4 Storing and Modifying Events

Once the schema for a class has been imported (see “[Importing a Schema](#)”), an instance of `Event` can be created to store and access events of that class. The `Event` class provides methods to store, update, or delete persistent events, create queries on the class extent, and control index updating. This section discusses the following topics:

- [Creating and Storing Events](#) — describes how to create an instance of `Event` and use it to store persistent events of the specified class.
- [Accessing Stored Events](#) — describes `Event` methods for fetching, changing, and deleting persistent events of the specified class.
- [Controlling Index Updating](#) — describes `Event` methods that can increase processing efficiency by controlling when index entries are updated.

### 2.4.1 Creating and Storing Events

Instances of the `Event` class are created and destroyed by the following methods:

- `EventPersister.getEvent\(\)` — takes a *className* String argument and returns an instance of `Event` that can store and access events of the specified class. Optionally takes an *indexMode* argument that specifies the default way to update index entries (see “[Controlling Index Updating](#)” for details).

**Note:**     **Target Class**

An instance of `Event` can only store, access, or query events of the class specified by the *className* argument in the call to `getEvent()`. In this chapter, class *className* is referred to as the *target class*. In the examples, the target class is always `SingleStringSample`.

- `Event.close\(\)` — closes the `Event` instance and releases the native code resources associated with it.

The following `Event` method stores Java objects of the target class as persistent events:

- `store\(\)` — adds one or more instances of the target class to the database. Takes either an event or an array of events as an argument, and returns a long database ID (or 0 if the database id could not be returned) for each stored event.

**Important:**     When an event is stored, it is not tested in any way, and it will never change or overwrite existing data. Each event is appended to the extent at the highest possible speed, or silently ignored if an event with the specified key already exists in the database.

The following example creates an instance of `Event` with `SingleStringSample` as the target class, and uses it to project an array of Java `SingleStringSample` objects as persistent events. The example assumes that *myPersister* has already been created and connected, and that a schema has been imported for the `SingleStringSample` class (listed earlier in “[The Xep-Simple Sample Application](#)”).

## Storing and Modifying Events: Storing an array of objects

```
SingleStringSample[] eventItems = SingleStringSample.generateSampleData(12);
try {
    Event newEvent = myPersister.getEvent("xep.samples.SingleStringSample");
    long[] itemIdList = newEvent.store(eventItems); // store all events
    int itemCount = 0;
    for (int i=0; i < itemIdList.length; i++) {
        if (itemIdList[i]>0) itemCount++;
    }
    System.out.println("Stored " + itemCount + " of " + eventItems.length + " events");
    newEvent.close();
}
catch (XEPException e) {System.out.println("Event storage failed:\n" + e);}
```

- The **generateSampleData()** method of `SingleStringSample` generates twelve `SingleStringSample` objects and stores them in an array named *eventItems*.
- The `EventPersister.getEvent()` method creates an `Event` instance named *newEvent* with `SingleStringSample` as the target class.
- The `Event.store()` method is called to project each object in the *eventItems* array as a persistent event in the database.

The method returns an array named *itemIdList*, which contains a long object ID for each successfully stored event, or 0 for an object that could not be stored. Variable *itemCount* is incremented once for each ID greater than 0 in *itemIdList*, and the total is printed.

- When the loop terminates, the `Event.close()` method is called to release associated resources.

**Note:** **Always call close() to release resources**

Always call **close()** on an instance of `EventPersister` before it goes out of scope to ensure that all locks, licenses, and other resources associated with the connection are released.

## 2.4.2 Accessing Stored Events

Once a persistent event has been stored, an `Event` instance of that target class provides the following methods for reading, updating, deleting the event:

- **deleteObject()** — takes a database object ID or `IdKey` as an argument and deletes the specified event from the database.
- **getObject()** — takes a database object ID or `IdKey` as an argument and returns the specified event.
- **updateObject()** — takes a database object ID or `IdKey` and an `Object` of the target class as arguments, and updates the specified event.

If the target class uses a standard object ID, it is specified as a long value (as returned by the **store()** method described in the previous section). If the target class uses an `IdKey`, it is specified as an array of `Object` where each item in the array is a value for one of the fields that make up the `IdKey` (see “[Using IdKeys](#)”).

In the following example, array *itemIdList* contains a list of object ID values for some previously stored `SingleStringSample` events. The example arbitrarily updates the first six persistent events in the list and deletes the rest.

**Note:** See “[Creating and Storing Events](#)” for the example that created the *itemIdList* array. This example also assumes that an `EventPersister` instance named *myPersister* has already been created and connected to the database.

See “[The XepSimple Sample Application](#)” for a listing of the `SingleStringSample` class.

## Storing and Modifying Events: Fetching, updating, and deleting events

```
// itemIdList is a previously created array of SingleStringSample event IDs
try {
    Event newEvent = myPersister.getEvent("xep.samples.SingleStringSample");
    int itemCount = 0;
    for (int i=0; i < itemIdList.length; i++) {
        try { // arbitrarily update events for first 6 IDs and delete the rest
            SingleStringSample eventObject = (SingleStringSample)newEvent.getObject(itemIdList[i]);

            if (i<6) {
                eventObject.name = eventObject.name + " (id=" + itemIdList[i] + ") " + " updated!";
                newEvent.updateObject(itemIdList[i], eventObject);
                itemCount++;
            } else {
                newEvent.deleteObject(itemIdList[i]);
            }
        }
    }
    catch (XEPException e) {System.out.println("Failed to process event:\n" + e);}
    System.out.println("Updated " + itemCount + " of " + itemIdList.length + " events");
    newEvent.close();
}
catch (XEPException e) {System.out.println("Event processing failed:\n" + e);}
```

- The EventPersister.[getEvent\(\)](#) method creates an Event instance named *newEvent* with SingleStringSample as the target class.
- Array *itemIdList* contains a list of object ID values for some previously stored SingleStringSample events (see “[Creating and Storing Events](#)” for the example that created *itemIdList*).

In the loop, each item in *itemIdList* is processed. The first six items are changed and updated, and the rest of the items are deleted. The following operations are performed:

- The Event.[getObject\(\)](#) method fetches the SingleStringSample object with the object ID specified in *itemIdList[i]*, and assigns it to variable *eventObject*.
- The value of the *eventObject* name field is changed.
- If the *eventObject* is one of the first six items in the list, Event.[updateObject\(\)](#) is called to update it in the database. Otherwise, Event.[deleteObject\(\)](#) is called to delete the object from the database.
- After all of the IDs in *itemIdList* have been processed, the loop terminates and a message displays the number of events updated.
- The Event.[close\(\)](#) method is called to release associated resources.

See “[Using Queries](#)” for a description of how to access and modify persistent events fetched by a simple SQL query.

### Deleting Test Data

When initializing a test database, it is frequently convenient to delete an entire class, or delete all events in a specified extent. The following EventPersister methods delete classes and extents from the InterSystems IRIS database:

- [deleteClass\(\)](#) — takes a *className* string as an argument and deletes the specified InterSystems IRIS class.
- [deleteExtent\(\)](#) — takes a *className* string as an argument and deletes all events in the extent of the specified class.

These methods are intended primarily for testing, and should be avoided in production code. See “[Classes and Extents](#)” in the *Orientation Guide for Server-Side Programming* for a detailed definition of these terms.

## 2.4.3 Controlling Index Updating

By default, indexes are not updated when a call is made to one of the Event methods that act on an event in the database (see “[Accessing Stored Events](#)”). Indexes are updated asynchronously, and updating is only performed after all transactions have been completed and the Event instance is closed. No uniqueness check is performed for unique indexes.



**Note:** This section only applies to classes that use standard object IDs or generated IdKeys (see “[Using IdKeys](#)”). Classes with user-assigned IdKeys can only be updated synchronously.

There are a number of ways to change this default indexing behavior. When an Event instance is created by EventPersister.[getEvent\(\)](#) (see “[Creating and Storing Events](#)”), the optional *indexMode* parameter can be set to specify a default indexing behavior. The following options are available:

- `Event.INDEX_MODE_ASYNC_ON` — enables asynchronous indexing. This is the default when the *indexMode* parameter is not specified.
- `Event.INDEX_MODE_ASYNC_OFF` — no indexing will be performed unless the **`startIndexing()`** method is called.
- `Event.INDEX_MODE_SYNC` — indexing will be performed each time the extent is changed, which can be inefficient for large numbers of transactions. This index mode must be specified if the class has a user-assigned IdKey.

The following Event methods can be used to control asynchronous index updating for the extent of the target class:

- **`startIndexing()`** — starts asynchronous index building for the extent of the target class. Throws an exception if the index mode is `Event.INDEX_MODE_SYNC`.
- **`stopIndexing()`** — stops asynchronous index building for the extent. If you do not want the index to be updated when the Event instance is closed, call this method before calling `Event.close()`.
- **`waitForIndexing()`** — takes an int *timeout* value as an argument and waits for asynchronous indexing to be completed. The *timeout* value specifies the number of seconds to wait (wait forever if -1, return immediately if 0). It returns `true` if indexing has been completed, or `false` if the wait timed out before indexing was completed. Throws an exception if the index mode is `Event.INDEX_MODE_SYNC`.

## 2.5 Using Queries

The Event class provides a way to create an instance of `EventQuery<T>`, which can execute a limited SQL query on the extent of the target class. `EventQuery<T>` methods are used to execute the SQL query, and to retrieve, update, or delete individual items in the query resultset.

The following topics are discussed:

- [Creating and Executing a Query](#) — describes how use methods of the `EventQuery<T>` class to execute queries.
- [Processing Query Data](#) — describes how to access and modify items in an `EventQuery<T>` resultset.
- [Defining the Fetch Level](#) — describes how to control the amount of data returned by a query.

**Note:** The examples in this section assume that EventPersister object *myPersister* has already been created and connected, and that a schema has been imported for the `SingleStringSample` class. See “[The XepSimple Sample Application](#)” for an example of how this is done and a listing of `SingleStringSample`.

### 2.5.1 Creating and Executing a Query

The following methods create and destroy an instance of `EventQuery<T>`:

- `Event.createQuery()` — takes a String argument containing the text of the SQL query and returns an instance of `EventQuery<T>`, where parameter T is the target class of the parent Event.
- `EventQuery<T>.close()` — closes this `EventQuery<T>` instance and releases the native code resources associated with it.

Queries submitted by an instance of `EventQuery<T>` will return Java objects of the specified generic type `T` (the target class of the Event instance that created the query object). Queries supported by the `EventQuery<T>` class are a subset of SQL select statements, as follows:

- Queries must consist of a `SELECT` clause, a `FROM` clause, and (optionally) standard SQL clauses such as `WHERE` and `ORDER BY`.
- The `SELECT` and `FROM` clauses must be syntactically legal, but they are actually ignored during query execution. All fields that have been mapped are always fetched from the extent of target class `T`.
- SQL expressions may not refer to arrays of any type, nor to embedded objects or fields of embedded objects.
- The InterSystems IRIS system-generated object ID may be referred to as `%ID`. Due to the leading `%`, this will not conflict with any field called `id` in a Java class.

The following `EventQuery<T>` methods define and execute the query:

- **`setParameter()`** — binds a parameter for the SQL query associated with this `EventQuery<T>`. Takes int *index* and Object *value* as arguments, where *index* specifies the parameter to be set, and *value* is the value to bind to the specified parameter.
- **`execute()`** — executes the SQL query associated with this `EventQuery<T>`. If the query is successful, this `EventQuery<T>` will contain a resultset that can be accessed by the methods described later (see “[Processing Query Data](#)”).

The following example executes a simple query on events in the `xep.samples.SingleStringSample` extent.

### Using Queries: Creating and executing a query

```
Event newEvent = myPersister.getEvent("xep.samples.SingleStringSample");
EventQuery<SingleStringSample> myQuery = null;
String sql =
    "SELECT * FROM xep_samples.SingleStringSample WHERE %ID BETWEEN ? AND ?";

myQuery = newEvent.createQuery(sql);
myQuery.setParameter(1,3); // assign value 3 to first SQL parameter
myQuery.setParameter(2,12); // assign value 12 to second SQL parameter
myQuery.execute(); // get resultset for IDs between 3 and 12
```

The `EventPersister.getEvent()` method creates an `Event` instance named *newEvent* with `SingleStringSample` as the target class.

The `Event.createQuery()` method creates an instance of `EventQuery<T>` named *myQuery*, which will execute the SQL query and hold the resultset. The *sql* variable contains an SQL statement that selects all events in the target class with IDs between two parameter values.

The `EventQuery<T>.setParameter()` method is called twice to assign values to the two parameters.

When the `EventQuery<T>.execute()` method is called, the specified query is executed for the extent of the target class, and the resultset is stored in *myQuery*.

By default, all data is fetched for each object in the resultset, and each object is fully initialized. See “[Defining the Fetch Level](#)” for options that limit the amount and type of data fetched with each object.

## 2.5.2 Processing Query Data

After a query has been executed, the methods described here can be used to access items in the query resultset, and update or delete the corresponding persistent events in the database. The `EventQueryIterator<T>` class implements `java.util.Iterator<T>` (where `T` is the target class of the parent `EventQuery<T>` instance). The following `EventQuery<T>` method creates an instance of `EventQueryIterator<T>`:

- **`getIterator()`** — returns an `EventQueryIterator<T>` iterator for the current resultset.

`EventQueryIterator<T>` implements `java.util.Iterator<T>` methods [hasNext\(\)](#), [next\(\)](#), and [remove\(\)](#), plus the following method:

- [set\(\)](#) — takes an object of the target class and uses it to update the persistent event most recently fetched by [next\(\)](#).

The following example creates an instance of `EventQueryIterator<T>` and uses it to update each item in the resultset:

### Using Queries: Iteration with `EventQueryIterator<T>`

```
myQuery.execute(); // get resultset
EventQueryIterator<xep.samples.SingleStringSample> myIter = myQuery.getIterator();
while (myIter.hasNext()) {
    currentEvent = myIter.next();
    currentEvent.name = "in process: " + currentEvent.name;
    myIter.set(currentEvent);
}
```

The call to `EventQuery<T>.execute()` runs the query described in the previous example (see “[Creating and Executing a Query](#)”), and the resultset is stored in `myQuery`. Each item in the resultset is a `SingleStringSample` object.

The call to [getIterator\(\)](#) creates iterator `myIter` for the resultset currently stored in `myQuery`.

In the while loop, [hasNext\(\)](#) returns `true` until all items in the resultset have been processed:

- The call to [next\(\)](#) returns the next `SingleStringSample` object from the resultset and assigns it to `currentEvent`.
- The `currentEvent.name` property is changed.
- The [set\(\)](#) method is called, storing the updated `currentEvent` object in the database.

See “[The XepSimple Sample Application](#)” for a listing of the `SingleStringSample` class.

## 2.5.2.1 Alternate Query Iteration Methods

The `EventQuery<T>` class also provides methods that can be used to process a resultset without using `EventQueryIterator<T>`. (This is an alternative for developers who prefer iteration methods similar to those provided by `ObjectScript`). After a query has been executed, the following `EventQuery<T>` methods can be used to access items in the query resultset, and update or delete the corresponding persistent events in the database:

- [getNext\(\)](#) — returns the next object of the target class from the resultset. Returns `null` if there are no more items in the resultset. It requires `null` or an object of the target class as an argument. (In this release, the argument is a placeholder that has no effect on the query).
- [updateCurrent\(\)](#) — takes an object of the target class as an argument and uses it to update the persistent event most recently returned by [getNext\(\)](#).
- [deleteCurrent\(\)](#) — deletes the persistent event most recently returned by [getNext\(\)](#) from the database.
- [getAll\(\)](#) — uses [getNext\(\)](#) to get all items from the resultset, and returns them in a `List`. Cannot be used for updating or deleting. [getAll\(\)](#) and [getNext\(\)](#) cannot access the same resultset — once either method has been called, the other method cannot be used until [execute\(\)](#) is called again.

See “[Accessing Stored Events](#)” for a description of how to access and modify persistent events identified by `Id` or `IdKey`.

### Important: **Never use `EventQuery<T>` and `EventQueryIterator<T>` iteration methods together**

Although query results can be accessed either by direct calls to `EventQuery<T>` methods or by getting an instance of `EventQueryIterator<T>` and using its methods, these access methods must never be used at the same time. Getting an iterator and calling its methods while also making direct calls to the `EventQuery<T>` methods can lead to unpredictable results.

## Using Queries: Updating and Deleting Query Data

```
myQuery.execute(); // get resultset
SingleStringSample currentEvent = myQuery.getNext(null);
while (currentEvent != null) {
    if (currentEvent.name.startsWith("finished")) {
        myQuery.deleteCurrent(); // Delete if already processed
    } else {
        currentEvent.name = "in process: " + currentEvent.name;
        myQuery.updateCurrent(currentEvent); // Update if unprocessed
    }
    currentEvent = myQuery.getNext(currentEvent);
}
myQuery.close();
```

In this example, the call to `EventQuery<T>.execute()` is assumed to execute the query described in the previous example (see “[Creating and Executing a Query](#)”), and the resultset is stored in `myQuery`. Each item in the resultset is a `SingleStringSample` object.

The first call to `getNext()` gets the first item from the resultset and assigns it to `currentEvent`.

In the while loop, the following process is applied to each item in the resultset:

- If `currentEvent.name` starts with the string "finished", `deleteCurrent()` deletes the corresponding persistent event from the database.
- Otherwise, the value of `currentEvent.name` is changed, and `updateCurrent()` is called. It takes `currentEvent` as its argument and uses it to update the persistent event in the database.
- The call to `getNext()` returns the next `SingleStringSample` object from the resultset and assigns it to `currentEvent`.

After the loop terminates, `close()` is called to release the native code resources associated with `myQuery`.

**Note:** **Always call `close()` to release resources**

Always call `close()` on an instance of `EventPersister` before it goes out of scope to ensure that all locks, licenses, and other resources associated with the connection are released.

## 2.5.3 Defining the Fetch Level

The fetch level is an Event property that can be used to control the amount of data returned when running a query. This is particularly useful when the underlying event is complex and only a small subset of event data is required.

The following `EventQuery<T>` methods set and return the current fetch level:

- `getFetchLevel()` — returns an int indicating the current fetch level of the Event.
- `setFetchLevel()` — takes one of the values in the Event fetch level enumeration as an argument and sets the fetch level for the Event.

The following fetch level values are supported:

- `Event.OPTION_FETCH_LEVEL_ALL` — This is the default. All data is fetched, and the returned event is fully initialized.
- `Event.OPTION_FETCH_LEVEL_DATATYPES_ONLY` — Only datatype fields are fetched. This includes all primitive types, all primitive wrappers, `java.lang.String`, `java.math.BigDecimal`, `java.util.Date`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp` and enum types. All other fields are set to null.
- `Event.OPTION_FETCH_LEVEL_NO_ARRAY_TYPES` — All types are fetched except arrays. All fields of array types, regardless of dimension, are set to null. All datatypes, object types (including serialized types) and collections are fetched.
- `Event.OPTION_FETCH_LEVEL_NO_COLLECTIONS` — All types are fetched except implementations of `java.util.List`, `java.util.Map`, and `java.util.Set`.

- `Event.OPTION_FETCH_LEVEL_NO_OBJECT_TYPES` — All types are fetched except object types. Serialized types are also considered object types and are not fetched. All datatypes, array types and collections are fetched.

## 2.6 Schema Customization and Mapping

This section provides details about how a Java class is mapped to an InterSystems IRIS event schema, and how a schema can be customized for optimal performance. In many cases, a schema can be imported for a simple class without providing any meta-information. In other cases, it may be necessary or desirable to customize the way in which the schema is imported. The following sections provide information on customized schemas and how to generate them:

- [Schema Import Models](#) — describes the two schema import models supported by XEP.
- [Using Annotations](#) — XEP annotations can be added to a Java class to specify the indexes that should be created. They can also be added to optimize performance by specifying fields that should not be imported or fields that should be serialized.
- [Using IdKeys](#) — Annotations can be used to specify IdKeys (index values used in place of the default object ID), which are required when importing a full schema.
- [Implementing an InterfaceResolver](#) — By default, a flat schema does not import fields declared as interfaces. Implementations of the `InterfaceResolver` interface can be used during schema import to specify the actual class of a field declared as an interface.
- [Schema Mapping Rules](#) — provides a detailed description of how Java classes are mapped to InterSystems IRIS event schemas.

### 2.6.1 Schema Import Models

XEP provides two different schema import models: flat schema and full schema. The main difference between these models is the way in which Java objects are projected to InterSystems IRIS events.

- [The Embedded Object Projection Model \(Flat Schema\)](#) — imports a *flat schema* where all objects referenced by the imported class are serialized and embedded, and all fields declared in all ancestor classes are collected and projected as if they were declared in the imported class itself. All data for an instance of the class is stored as a single InterSystems IRIS `%Library.Persistent` object, and information about the original Java class structure is not preserved.
- [The Full Object Projection Model \(Full Schema\)](#) — imports a *full schema* where all objects referenced by the imported class are projected as separate InterSystems IRIS `%Persistent` objects. Inherited fields are projected as references to fields in the ancestor classes, which are also imported as InterSystems IRIS `%Persistent` classes. There is a one-to-one correspondence between Java source classes and InterSystems IRIS projected classes, so the Java class inheritance structure is preserved.

Full object projection preserves the inheritance structure of the original Java classes, but may have an impact on performance. Flat object projection is the optimal choice if performance is essential and the event schema is fairly simple. Full object projection can be used for a richer set of features and more complex schemas if the performance penalty is acceptable.

#### 2.6.1.1 The Embedded Object Projection Model (Flat Schema)

By default, XEP imports a schema that projects referenced objects by *flattening*. In other words, all objects referenced by the imported class are serialized and embedded, and all fields declared in all ancestor classes are collected and projected as if they were declared in the imported class itself. The corresponding InterSystems IRIS event extends `%Library.Persistent`, and contains embedded serialized objects where the original Java object contained references to external objects.

This means that a flat schema does not preserve inheritance in the strict sense on the InterSystems IRIS side. For example, consider these three Java classes:

```
class A {
    String a;
}
class B extends class A {
    String b;
}
class C extends class B {
    String c;
}
```

Importing class C results in the following InterSystems IRIS class:

```
Class C Extends %Persistent ... {
    Property a As %String;
    Property b As %String;
    Property c As %String;
}
```

No corresponding InterSystems IRIS events will be generated for the A or B classes unless they are specifically imported. Event C on the InterSystems IRIS side does not extend either A or B. If imported, A and B would have the following structures:

```
Class A Extends %Persistent ... {
    Property a As %String;
}
Class B Extends %Persistent ... {
    Property a As %String;
    Property b As %String;
}
```

All operations will be performed only on the corresponding InterSystems IRIS event. For example, calling `store()` on objects of type C will only store the corresponding C InterSystems IRIS events.

If a Java child class hides a field of the same name that is also declared in its superclass, the XEP engine always uses the value of the child field.

### 2.6.1.2 The Full Object Projection Model (Full Schema)

The full object model imports a schema that preserves the Java inheritance model by creating a matching inheritance structure in InterSystems IRIS. Rather than serializing all object fields and storing all data in a single InterSystems IRIS object, the schema establishes a one-to-one relationship between the Java source classes and InterSystems IRIS projected classes. The full object projection model stores each referenced class separately, and projects fields of a specified class as references to objects of the corresponding InterSystems IRIS class.

Referenced classes must include an annotation that creates a user-defined IdKey (either `@Id` or `@Index` — see “[Using IdKeys](#)”). When an object is stored, all referenced objects are stored first, and the resulting IdKeys are stored in the parent object. As with the rest of XEP, there are no uniqueness checks, and no attempts to change or overwrite existing data. The data is simply appended at the highest possible speed. If an IdKey value references an event that already exists, it will simply be skipped, without any attempt to overwrite the existing event.

The `@Embedded` class level annotation can be used to optimize a full schema by embedding instances of the annotated class as serialized objects rather than storing them separately.

## 2.6.2 Using Annotations

The XEP engine infers XEP event metadata by examining a Java class. Additional information can be specified in the Java class via annotations, which can be found in the `com.intersystems.xep.annotations` package. As long a Java object conforms to the definition of an XEP persistent event (see “[Requirements for Imported Classes](#)”), it is projected as an InterSystems IRIS event, and there is no need to customize it.

Some annotations are applied to individual fields in the class to be projected, while others are applied to the entire class:

- *Field Annotations* — are applied to a field in the class to be imported:
  - `@Id` — specifies that the field will act as an IdKey.
  - `@Serialized` — indicates that the field should be stored and retrieved in its serialized form.
  - `@Transient` — indicates that the field should be excluded from import.
- *Class Annotations* — are applied to the entire class to be imported:
  - `@Embedded` — indicates that a field of this class in a full schema should be embedded (as in a flat schema) rather than referenced.
  - `@Index` — declares an index for the class.
  - `@Indices` — declares multiple indexes for the same class.

### **@Id (field level annotation)**

The value of a field marked with `@Id` will be used as an IdKey that replaces the standard object ID (see “[Using IdKeys](#)”). Only one field per class can use this annotation, and the field must be a String, int, or long (double is permitted but not recommended). To create a compound IdKey, use the `@Index` annotation instead. A class marked with `@Id` cannot also declare a compound primary key with `@Index`. An exception will be thrown if both annotations are used on the same class.

The following parameter must be specified:

- `generated` — a boolean specifying whether or not XEP should generate key values.
  - `generated = true` — (the default setting) key value will be generated by InterSystems IRIS and the field marked as `@Id` must be Long. This field is expected to be null prior to insert/store and will be filled automatically by XEP upon completion of such an operation.
  - `generated=false` — the user-assigned value of the marked field will be used as the IdKey value. Fields can be String, int, Integer, long or Long.

In the following example, the user-assigned value of the `ssn` field will be used as the object ID:

```
import com.intersystems.xep.annotations.Id;
public class Person {
    @Id(generated=false)
    public String ssn;
    public String name;
    public String dob;
}
```

### **@Serialized (field level annotation)**

The `@Serialized` annotation indicates that the field should be stored and retrieved in its serialized form.

This annotation optimizes storage of fields that implement the `java.io.Serializable` interface (including arrays, which are implicitly serializable). The XEP engine will call the relevant read or write method for the serial object, rather than using the default mechanism for storing or retrieving data. An exception will be thrown if the marked field is not serializable. See “[Type Mapping](#)” for more details on the projection of serialized fields.



*Example:*

```
import com.intersystems.xep.annotations.Serialized;
public class MyClass {
    @Serialized
    public xep.samples.Serialized    serialized;
    @Serialized
    public int[][][] quadIntArray;
    @Serialized
    public String[][] doubleStringArray;
}

// xep.samples.Serialized:
public class Serialized implements java.io.Serializable {
    public String name;
    public int value;
}
```

### **@Transient (field level annotation)**

The @Transient annotation indicates that the field should be excluded from import. The annotated field will not be projected to InterSystems IRIS, and will be ignored when events are stored or loaded.

*Example:*

```
import com.intersystems.xep.annotations.Transient;
public class MyClass {
    // this field will NOT be projected:
    @Transient
    public String transientField;

    // this field WILL be projected:
    public String projectedField;
}
```

### **@Embedded (class level annotation)**

The @Embedded annotation can be used when a full schema is to be generated (see “[Schema Import Models](#)”). It indicates that a field of this class should be serialized and embedded (as in a flat schema) rather than referenced when projected to InterSystems IRIS.

*Examples:*

```
import com.intersystems.xep.annotations.Embedded;
@Embedded
public class Address {
    String street;
    String city;
    String zip;
    String state;
}

import com.intersystems.xep.annotations.Embedded;
public class MyOuterClass {
    @Embedded
    public static class MyInnerClass {
        public String innerField;
    }
}
```

### **@Index (class level annotation)**

The @Index annotation can be used to declare an index.

Arguments must be specified for the following parameters:

- **name** — a String containing the name of the composite index
- **fields** — an array of String containing the names of the fields that comprise the composite index
- **type** — the index type. The xep.annotations.IndexType enumeration includes the following possible types:



- `IndexType.none` — default value, indicating that there are no indexes.
- `IndexType.bitmap` — a bitmap index (see “Bitmap Indexes” in *Using InterSystems SQL*).
- `IndexType.bitslice` — a bitslice index (see “Overview” in *Using InterSystems SQL*).
- `IndexType.simple` — a standard index on one or more fields.
- `IndexType.idkey` — an index that will be used in place of the standard ID (see “Using IdKeys”).

*Example:*

```
import com.intersystems.xep.annotations.Index;
import com.intersystems.xep.annotations.IndexType;

@Index(name="indexOne", fields={"ssn", "dob"}, type=IndexType.idkey)
public class Person {
    public String name;
    public Date dob;
    public String ssn;
}
```

### @Indices (class level annotation)

The `@Indices` annotation allows you to specify an array of different indexes for one class. Each element in the array is an `@Index` tag.

*Example:*

```
import com.intersystems.xep.annotations.Index;
import com.intersystems.xep.annotations.IndexType;
import com.intersystems.xep.annotations.Indices;

@Indices({
    @Index(name="indexOne", fields={"myInt", "myString"}, type=IndexType.simple),
    @Index(name="indexTwo", fields={"myShort", "myByte", "myInt"}, type=IndexType.simple)
})
public class MyTwoIndices {
    public int myInt;
    public Byte myByte;
    public short myShort;
    public String myString;
}
```

## 2.6.3 Using IdKeys

IdKeys are index values that are used in place of the default object ID. Both simple and composite IdKeys are supported by XEP, and a user-generated IdKey is required for a Java class that is imported with a full schema (see “Importing a Schema”). IdKeys on a single field can be created with the `@Id` annotation. To create a composite IdKey, add an `@Index` annotation with `IndexType.idkey`. For example, given the following class:

```
class Person {
    String name;
    Integer id;
    Date dob;
}
```

the default storage structure uses the standard object ID as a subscript:

```
^PersonD(1)=$LB("John", 12, "1976-11-11")
```

The following annotation uses the name and id fields to create a composite IdKey named *newIdKey* that will replace the standard object ID:

```
@Index(name="newIdKey", fields={"name", "id"}, type=IndexType.idkey)
```

This would result in the following global structure:

```
^PersonD( "John" , 12 ) = $LB( "1976-11-11" )
```

XEP will also honor IdKeys added by other means, such as SQL commands (see “Using the Unique, PrimaryKey, and IDKey Keywords with Indexes” in *Using InterSystems SQL*). The XEP engine will automatically determine whether the underlying class contains an IdKey, and generate the appropriate global structure.

There are a number of limitations on IdKey usage:

- An IdKey value must be unique. If the IdKey is user-generated, uniqueness is the responsibility of the calling application, and is not enforced by XEP. If the application attempts to add an event with a key value that already exists in the database, the attempt will be silently ignored and the existing event will not be changed.
- A class that declares an IdKey cannot be indexed asynchronously if it also declares other indexes.
- There is no limit of the number of fields in a composite IdKey, but the fields must be String, int, Integer, long or Long. Although double can also be used, it is not recommended.
- There may be a performance penalty in certain rare situations requiring extremely high and sustained insert rates.

See “[Accessing Stored Events](#)” for a discussion of Event methods that allow retrieval, updating and deletion of events based on their IdKeys.

See “[SQL and Object Use of Multidimensional Storage](#)” in *Using Globals* for information on IdKeys and the standard InterSystems IRIS storage model. See “Defining and Building Indexes” in *Using InterSystems SQL* for information on IdKeys in SQL.

## 2.6.4 Implementing an InterfaceResolver

When a flat schema is imported, information on the inheritance hierarchy is not preserved (see “[Schema Import Models](#)”). This creates a problem when processing fields whose types are declared as interfaces, since the XEP engine must know the actual class of the field. By default, such fields are not imported into a flat schema. This behavior can be changed by creating implementations of `com.intersystems.xep.InterfaceResolver` to resolve specific interface types during processing.

**Note:** `InterfaceResolver` is only relevant for the flat schema import model, which does not preserve the Java class inheritance structure. The full schema import model establishes a one-to-one relationship between Java and InterSystems IRIS classes, thus preserving the information needed to resolve an interface.

An implementation of `InterfaceResolver` is passed to `EventPersister` before calling the flat schema import method, `importSchema()` (see “[Importing a Schema](#)”). This provides the XEP engine with a way to resolve interface types during processing. The following `EventPersister` method specifies the implementation that will be used:

- `EventPersister.setInterfaceResolver()` — takes an instance of `InterfaceResolver` as an argument. When `importSchema()` is called, it will use the specified instance to resolve fields declared as interfaces.

The following example imports two different classes, calling a different, customized implementation of `InterfaceResolver` for each class:

### Schema Customization: Applying an InterfaceResolver

```
try {
    myPersister.setInterfaceResolver(new test.MyFirstInterfaceResolver());
    myPersister.importSchema("test.MyMainClass");

    myPersister.setInterfaceResolver(new test.MyOtherInterfaceResolver());
    myPersister.importSchema("test.MyOtherClass");
}
catch (XEPException e) {System.out.println("Import failed:\n" + e);}
```

The first call to `setInterfaceResolver()` sets a new instance of `MyFirstInterfaceResolver` (described in the next example) as the implementation to be used during calls to the import methods. This implementation will be used in all calls to `importSchema()` until `setInterfaceResolver()` is called again to specify a different implementation.

The first call to `importSchema()` imports class `test.MyMainClass`, which contains a field declared as interface `test.MyFirstInterface`. The instance of `MyFirstInterfaceResolver` will be used by the import method to resolve the actual class of this field.

The second call to `setInterfaceResolver()` sets an instance of a different `InterfaceResolver` class as the new implementation to be used when `importSchema()` is called again.

All implementations of `InterfaceResolver` must define the following method:

- `InterfaceResolver.getImplementationClass()` returns the actual type of a field declared as an interface. This method has the following parameters:
  - *interfaceClass* — the interface to be resolved.
  - *declaringClass* — class that contains a field declared as *interfaceClass*.
  - *fieldName* — string containing the name of the field in *declaringClass* that has been declared as an interface.

The following example defines an interface, an implementation of that interface, and an implementation of `InterfaceResolver` that resolves instances of the interface.

### Schema Customization: Implementing an InterfaceResolver

In this example, the interface to be resolved is `test.MyFirstInterface`:

```
package test;
public interface MyFirstInterface{ }
```

The `test.MyFirstImpl` class is the implementation of `test.MyFirstInterface` that should be returned by the `InterfaceResolver`:

```
package test;
public class MyFirstImpl implements MyFirstInterface {
    public MyFirstImpl() {}
    public MyFirstImpl(String s) { fieldOne = s; }
    public String fieldOne;
}
```

The following implementation of `InterfaceResolver` returns class `test.MyFirstImpl` if the interface is `test.MyFirstInterface`, or `null` otherwise:

```
package test;
import com.intersystems.xep.*;
public class MyFirstInterfaceResolver implements InterfaceResolver {
    public MyFirstInterfaceResolver() {}
    public Class<?> getImplementationClass(Class declaringClass,
        String fieldName, Class<?> interfaceClass) {
        if (interfaceClass == xepdemo.MyFirstInterface.class) {
            return xepdemo.MyFirstImpl.class;
        }
        return null;
    }
}
```

When an instance of `MyFirstInterfaceResolver` is specified by `setInterfaceResolver()`, subsequent calls to `importSchema()` will automatically use that instance to resolve any field declared as `test.MyFirstInterface`. For such each field, the `getImplementationClass()` method will be called with parameter *declaringClass* set to the class that contains the field, *fieldName* set to the name of the field, and *interfaceClass* set to `test.MyFirstInterface`. The method will resolve the interface and return either `test.MyFirstImpl` or `null`.

## 2.6.5 Schema Mapping Rules

This section provides details about how an XEP schema is structured. The following topics are discussed:

- [Requirements for Imported Classes](#) — describes the structural rules that a Java class must satisfy to produce objects that can be projected as persistent events.
- [Naming Conventions](#) — describes how Java class and field names are translated to conform to InterSystems IRIS naming rules.
- [Type Mapping](#) — lists the Java data types that can be used, and describes how they are mapped to corresponding InterSystems IRIS types.

### 2.6.5.1 Requirements for Imported Classes

The XEP schema import methods cannot produce a valid schema for a Java class unless it satisfies the following requirements:

- If the imported InterSystems IRIS class or any derived class will be used to execute queries and access stored events, the Java source class must explicitly declare an argumentless public constructor.
- The Java source class cannot contain fields declared as `java.lang.Object`, or arrays, lists, sets or maps that use `java.lang.Object` as part of their declaration. An exception will be thrown if the XEP engine encounters such fields. Use the `@Transient` annotation (see “[Using Annotations](#)”) to prevent them from being imported.

The `Event.isEvent()` method can be used to analyze a Java class or object and determine if it can produce a valid event in the XEP sense. In addition to the conditions described above, this method throws an `XEPException` if any of the following conditions are detected:

- a circular dependency
- an untyped List or Map
- a Map key value that is not a String, primitive, or primitive wrapper

Fields of a persistent event can be primitives and their wrappers, temporal types, objects (projected as embedded/serial objects), enumerations, and types derived from `java.util.List`, `java.util.Set` and `java.util.Map`. These types can also be contained in arrays, nested collections, and collections of arrays.

By default, projected fields may not retain all features of the Java class. Certain fields are changed in the following ways:

- Although the Java class may contain static fields, they are excluded from the projection by default. There will be no corresponding InterSystems IRIS properties. Additional fields can be excluded by using the `@Transient` annotation (see “[Using Annotations](#)”).
- In a flat schema (see “[Schema Import Models](#)”), all object types, including inner (nested) Java classes, are projected as `%SerialObject` classes in InterSystems IRIS. The fields within the objects are not projected as separate InterSystems IRIS properties, and the objects are opaque from the viewpoint of ObjectScript.
- A flat schema projects all inherited fields as if they were declared in the child class.

See “[Type Mapping](#)” for more details on how various datatypes are projected.

### 2.6.5.2 Naming Conventions

Corresponding InterSystems IRIS class and property names are identical to those in Java, with the exception of two special characters allowed in Java but not InterSystems IRIS:

- \$ (dollar sign) is projected as a single "d" character on the InterSystems IRIS side.

- `_` (underscore) is projected as a single "u" character on the InterSystems IRIS side.

Class names are limited to 255 characters, which should be sufficient for most applications. However, the corresponding global names have a limit of 31 characters. Since this is typically not sufficient for a one-to-one mapping, the XEP engine transparently generates unique global names for class names longer than 31 characters. Although the generated global names are not identical to the originals, they should still be easy to recognize. For example, the `xep.samples.SingleStringSample` class (listed in “[The XepSimple Sample Application](#)”) will receive global name `xep.samples.SingleStrinA5BFD`.

### 2.6.5.3 Type Mapping

Fields of a persistent event can be any of the following types:

- primitive types, primitive wrappers and `java.lang.String`
- temporal types (`java.sql.Time`, `java.sql.Date`, `java.sql.Timestamp` and `java.util.Date`)
- object types (projected as embedded/serial objects in a flat schema)
- Java enum types
- any types derived from `java.util.List`, `java.util.Set` and `java.util.Map`.
- nested collections (for example, a list of maps), and collections of arrays
- arrays of any of the above

The following sections list the currently supported Java types, and their corresponding InterSystems IRIS types:

#### Primitives and Primitive Wrappers

The following Java primitives and wrappers are mapped as InterSystems IRIS `%String`:

- `char`, `java.lang.Character`, `java.lang.String`

The following Java primitives and wrappers are mapped as InterSystems IRIS `%Integer`:

- `boolean`, `java.lang.Boolean`
- `byte`, `java.lang.Byte`
- `int`, `java.lang.Integer`
- `long`, `java.lang.Long`
- `short`, `java.lang.Short`

The following Java primitives and wrappers are mapped as InterSystems IRIS `%Double`:

- `double`, `java.lang.Double`
- `float`, `java.lang.Float`

#### Temporal Types

The following Java temporal types are mapped as InterSystems IRIS `%String`:

- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `java.util.Date`

## Object Types

Imported Java classes (the target classes specified in calls to **importSchema()** or **importFullSchema()**) are projected as InterSystems IRIS %Persistent classes. Necessary information is also imported from superclasses and dependent classes, but the schema import model (see “[Schema Import Models](#)”) determines how InterSystems IRIS stores this information:

- In a flat schema, a class that appears as a field type in the imported class is projected as a %SerialObject InterSystems IRIS class, and is embedded in the parent %Persistent class. Superclasses of the imported class are not projected. Instead, all fields inherited from superclasses are projected as if they were native fields of the imported class.
- In a full schema, superclasses and dependent classes are projected as separate %Persistent InterSystems IRIS classes, and the imported class will contain references to those classes.

The java.lang.Object class is not a supported type. An exception will be thrown if the XEP engine encounters fields declared as java.lang.Object, or arrays, lists, sets or maps that use it.

## Serialized

All fields marked with the [@Serialized](#) annotation (see “[Using Annotations](#)”) will be projected in their serialized form as %Binary.

## Arrays

The following rules apply to arrays:

- With the exception of byte and character arrays, all one-dimensional arrays of primitives, primitive wrappers and temporal types are mapped as a list of the underlying base type.
- One-dimensional byte arrays (byte[] and java.lang.Byte[]) are mapped as %Binary.
- One-dimensional character arrays (char[] and java.lang.Character[]) are mapped as %String.
- One-dimensional arrays of objects are mapped as lists of objects.
- All multi-dimensional arrays are mapped as %Binary and are opaque from the viewpoint of ObjectScript.
- Arrays are implicitly serializable, and can be annotated with [@Serialized](#).

## Enumerations

Java enum types are projected as InterSystems IRIS %String, and only the names are stored. When retrieved from InterSystems IRIS, an entire Java enum object will be reconstituted. Arrays, Lists, and other collections of enums are also supported.

## Collections

Classes derived from java.util.List and java.util.Set are projected as InterSystems IRIS lists. Classes derived from java.util.Map are projected as InterSystems IRIS arrays. Untyped Java lists, sets and maps are not supported (type parameters must be used). Nested lists, sets and maps, lists, sets and maps of arrays, as well as arrays of lists, sets or maps are all projected as %Binary and are considered opaque as far as InterSystems IRIS is concerned.

# 3

## XEP Quick Reference

This chapter is a quick reference for the `com.intersystems.xep` package, which contains the public API described in this book.

**Note:** This chapter is intended as a convenience for readers of this book, but it is not the definitive reference for XEP. For the most complete and up-to-date information on these classes, see the [Java XEP API online documentation](#).

### 3.1 XEP Quick Reference

This section is a reference for the XEP API (namespace `com.intersystems.xep`). See [Using XEP Event Persistence](#) for a details on how to use the API. It contains the following classes and interfaces:

- Class [PersisterFactory](#) — provides a factory method to create `EventPersister` objects.
- Class [EventPersister](#) — encapsulates an XEP database connection. It provides methods that set XEP options, establish a connection or get an existing connection object, import schema, produce XEP Event objects, and control transactions.
- Class [Event](#) — encapsulates a reference to an XEP persistent event. It provides methods to store or delete events, create a query, and start or stop index creation.
- Class [EventQuery<T>](#) — encapsulates a query that retrieves individual events of a specific type from the database for update or deletion.
- Class [EventQueryIterator<T>](#) — provides an alternative to `EventQuery<T>` for retrieving, updating and deleting XEP events, using methods similar to those in Java `Iterator`.
- Interface [InterfaceResolver](#) — resolves the actual type of a field during flat schema importation if the field was declared as an interface.
- Class [XEPException](#) — is the exception thrown by most XEP methods.

#### 3.1.1 List of XEP Methods

The following classes and methods of the XEP API are described in this reference:

##### **PersisterFactory**

- [createPersister\(\)](#) — creates a new `EventPersister` object.

## EventPersister

- **close()** — releases all resources held by this instance.
- **commit()** — commits one level of transaction.
- **connect()** — connects to InterSystems IRIS® via TCP/IP using the arguments specified.
- **deleteClass()** — deletes an InterSystems IRIS class.
- **deleteExtent()** — deletes all objects in the given extent.
- **getEvent()** — returns an event object that corresponds to the class name supplied.
- **getInterfaceResolver()** — returns the currently specified instance of InterfaceResolver.
- **getTransactionLevel()** — returns the current transaction level (or 0 if not in a transaction).
- **importSchema()** — imports a flat schema.
- **importSchemaFull()** — imports a full schema.
- **rollback()** — rolls back the specified number of transaction levels, or all levels if no level is specified.
- **setInterfaceResolver()** — specifies the InterfaceResolver object to be used.

## Event

- **close()** — releases all resources held by this instance.
- **createQuery()** — creates a EventQuery<T> instance.
- **deleteObject()** — deletes an event given its database Id or IdKey.
- **getObject()** — returns an event given its database Id or IdKey.
- **isEvent()** — checks whether an object (or class) is an event in the XEP sense.
- **startIndexing()** — starts index building for the underlying class.
- **stopIndexing()** — stops index building for the underlying class.
- **store()** — stores the specified object or array of objects.
- **updateObject()** — updates an event given its database Id or IdKey.
- **waitForIndexing()** — waits for asynchronous indexing to be completed for this class.

## EventQuery<T>

- **close()** — releases all resources held by this instance.
- **deleteCurrent()** — deletes the event most recently fetched by **getNext()**.
- **execute()** — executes this XEP query.
- **getAll()** — fetches all events in the resultset as an array.
- **getFetchLevel()** — returns the current fetch level.
- **getIterator()** — returns an EventQueryIterator<T> that can be used to iterate over query results.
- **getNext()** — fetches the next event in the resultset.
- **setFetchLevel()** — controls the amount of data returned.
- **setParameter()** — binds a parameter for this query.
- **updateCurrent()** — updates the event most recently fetched by **getNext()**



**EventQueryIterator<T>**

- **hasNext()** — returns `true` if the query resultset has more items.
- **next()** — fetches the next event in the resultset.
- **remove()** — deletes the event most recently fetched by **next()**.
- **set()** — assigns a new value to the event most recently fetched by **next()**.

**InterfaceResolver**

- **getImplementationClass()** — if a field was declared as an interface, an implementation of this method can be used to resolve the actual field type during schema importation.

## 3.1.2 Class PersisterFactory

Class `com.intersystems.xep.PersisterFactory` creates a new `EventPersister` object.

**PersisterFactory() Constructor**

Creates a new instance of `PersisterFactory`.

```
PersisterFactory()
```

**createPersister()**

`PersisterFactory.createPersister()` returns an instance of `EventPersister`.

```
static EventPersister createPersister() [inline, static]
```

*see also:*

[Creating and Connecting an EventPersister](#)

## 3.1.3 Class EventPersister

Class `com.intersystems.xep.EventPersister` is the main entry point for the XEP module. It provides methods that can be used to control XEP options, establish a connection, import schema, and produce XEP Event objects. It also provides methods to control transactions and perform other tasks.

In most applications, instances of `EventPersister` should be created by `PersisterFactory.createPersister()`. The constructor should only be used to extend the class.

**EventPersister() Constructor**

Creates a new instance of `EventPersister`.

```
EventPersister()
```

**close()**

`EventPersister.close()` releases all resources held by this instance. Always call **close()** on the `EventPersister` object before it goes out of scope to ensure that all locks, licenses, and other resources associated with the connection are released.

```
void close()
```

**commit()**

EventPersister.**commit()** commits one level of transaction

```
void commit()
```

**connect()**

EventPersister.**connect()** establishes a connection to the specified InterSystems IRIS namespace.

```
void connect(String host, int port, String namespace, String username, String password)
```

*parameters:*

- `host` — host address for TCP/IP connection.
- `port` — port number for TCP/IP connection.
- `namespace` — namespace to be accessed.
- `username` — username for this connection.
- `password` — password for this connection.

If the host address is `127.0.0.1` or `localhost`, the connection will default to a *shared memory connection*, which is faster than the standard TCP/IP connection (see “[Shared Memory Connections](#)” in *Using Java with the InterSystems JDBC Driver*).

*see also:*

[Creating and Connecting an EventPersister](#)

**deleteClass()**

EventPersister.**deleteClass()** deletes an InterSystems IRIS class definition. It does not delete objects associated with the extent (since objects can belong to more than one extent), and does not delete any dependencies (for example, inner or embedded classes).

```
void deleteClass(String className)
```

*parameter:*

- `className` — name of the class to be deleted.

If the specified class does not exist, the call silently fails (no error is thrown).

*see also:*

“Deleting Test Data” in [Accessing Stored Events](#)

**deleteExtent()**

EventPersister.**deleteExtent()** deletes the extent definition associated with a Java event, but does not destroy associated data (since objects can belong to more than one extent). See “Extents” in *Defining and Using Classes* for more information on managing extents.

```
void deleteExtent(String className)
```

- `className` — name of the extent.

Do not confuse this method with the deprecated Event.**deleteExtent()**, which destroys all extent data as well as with the extent definition.

*see also:*

“Deleting Test Data” in [Accessing Stored Events](#)

## getEvent()

`EventPersister.getEvent()` returns an `Event` object that corresponds to the class name supplied, and optionally specifies the indexing mode to be used.

```
Event getEvent(String className)
Event getEvent(String className, int indexMode)
```

*parameter:*

- `className` — class name of the object to be returned.
- `indexMode` — indexing mode to be used.

The following *indexMode* options are available:

- `Event.INDEX_MODE_ASYNC_ON` — enables asynchronous indexing. This is the default when the *indexMode* parameter is not specified.
- `Event.INDEX_MODE_ASYNC_OFF` — no indexing will be performed unless the **startIndexing()** method is called.
- `Event.INDEX_MODE_SYNC` — indexing will be performed each time the extent is changed, which can be inefficient for large numbers of transactions. This index mode must be specified if the class has a user-assigned `IdKey`.

The same instance of `Event` can be used to store or retrieve all instances of a class, so a process should only call the **getEvent()** method once per class. Avoid instantiating multiple `Event` objects for a single class, since this can affect performance and may cause memory leaks.

*see also:*

[Creating Event Instances and Storing Persistent Events, Controlling Index Updating](#)

## getInterfaceResolver()

`EventPersister.getInterfaceResolver()` — returns the currently set instance of `InterfaceResolver` that will be used by **importSchema()** (see “[Implementing an InterfaceResolver](#)”). Returns `null` if no instance has been set.

```
InterfaceResolver getInterfaceResolver()
```

*see also:*

[setInterfaceResolver\(\), importSchema\(\)](#)

## getTransactionLevel()

`EventPersister.getTransactionLevel()` returns the current transaction level (0 if not in a transaction)

```
int getTransactionLevel()
```

## importSchema()

EventPersister.**importSchema()** produces a flat schema (see “[Schema Import Models](#)”) that embeds all referenced objects as serialized objects. The method imports the schema of each event declared in the class or a .jar file specified (including dependencies), and returns an array of class names for the imported events.

```
String[] importSchema(String classOrJarFileName)
String[] importSchema(String[] classes)
```

*parameters:*

- `classes` — an array containing the names of the classes to be imported.
- `classOrJarFileName` — a class name or the name of a .jar file containing the classes to be imported. If a .jar file is specified, all classes in the file will be imported.

If the argument is a class name, the corresponding class and any dependencies will be imported. If the argument is a .jar file, all classes in the file and any dependencies will be imported. If such schema already exists, and it appears to be in sync with the Java schema, import will be skipped. Should a schema already exist, but it appears different, a check will be performed to see if there is any data. If there is no data, a new schema will be generated. If there is existing data, an exception will be thrown.

*see also:*

[Importing a Schema](#)

## importSchemaFull()

EventPersister.**importSchemaFull()** — produces a full schema (see “[Schema Import Models](#)”) that preserves the object hierarchy of the source classes. The method imports the schema of each event declared in the class or .jar file specified (including dependencies), and returns an array of class names for the imported events.

```
String[] importSchemaFull(String classOrJarFileName)
String[] importSchemaFull(String[] classes)
```

*parameters:*

- `classes` — an array containing the names of the classes to be imported.
- `classOrJarFileName` — a class name or the name of a .jar file containing the classes to be imported. If a .jar file is specified, all classes in the file will be imported.

If the argument is a class name, the corresponding class and any dependencies will be imported. If the argument is a .jar file, all classes in the file and any dependencies will be imported. If such schema already exists, and it appears to be in sync with the Java schema, import will be skipped. Should a schema already exist, but it appears different, a check will be performed to see if there is any data. If there is no data, a new schema will be generated. If there is existing data, an exception will be thrown.

*see also:*

[Importing a Schema](#)

## rollback()

EventPersister.**rollback()** rolls back the specified number of levels of transaction, where *level* is a positive integer, or roll back all levels of transaction if no level is specified.

```
void rollback()
void rollback(int level)
```

*parameter:*

- `level` — optional number of levels to roll back.

This method does nothing if *level* is less than 0, and stops rolling back once the transaction level reaches 0 if *level* is greater than the initial transaction level.

### **setInterfaceResolver()**

`EventPersister.setInterfaceResolver()` — sets the instance of `InterfaceResolver` to be used by `importSchema()` (see “[Implementing an InterfaceResolver](#)”). All instances of `Event` created by this `EventPersister` will share the specified `InterfaceResolver` (which defaults to `null` if this method is not called).

```
void setInterfaceResolver(InterfaceResolver interfaceResolver)
```

*parameters:*

- `interfaceResolver` — an implementation of `InterfaceResolver` that will be used by `importSchema()` to determine the actual type of fields declared as interfaces. This argument can be `null`.

*see also:*

[getInterfaceResolver\(\)](#), [importSchema\(\)](#)

## **3.1.4 Class Event**

Class `com.intersystems.xep.Event` provides methods that operate on XEP events (storing events, creating a query, indexing etc.). It is created by the `EventPersister.getEvent()` method.

### **close()**

`Event.close()` releases all resources held by this instance. Always call `close()` on the `Event` object before it goes out of scope to ensure that all locks, licenses, and other resources associated with the connection are released.

```
void close()
```

### **createQuery()**

`Event.createQuery()` takes a `String` argument containing the text of the SQL query and returns an instance of `EventQuery<T>`, where parameter `T` is the target class of the parent `Event`.

```
<T> EventQuery<T> createQuery (String sqlText)
```

*parameter:*

- `sqlText` — text of the SQL query.

*see also:*

[Creating and Executing a Query](#)

### **deleteObject()**

`Event.deleteObject()` deletes an event identified by its database object ID or `IdKey`.

```
void deleteObject(long id)
void deleteObject(Object[] idkeys)
```

*parameter:*

- `id` — database object ID

- `idkeys` — an array of objects that make up the `IdKey` (see “[Using IdKeys](#)”). An `XEPException` will be thrown if the underlying class has no `IdKeys` or if any of the keys supplied is equal to null or of an invalid type.

*see also:*

[Accessing Stored Events](#)

### **getObject()**

`Event.getObject()` fetches an event identified by its database object ID or `IdKey`. Returns `null` if the specified object does not exist.

```
Object getObject(long id)
Object getObject(Object[] idkeys)
```

*parameter:*

- `id` — database object ID
- `idkeys` — an array of objects that make up the `IdKey` (see “[Using IdKeys](#)”). An `XEPException` will be thrown if the underlying class has no `IdKeys` or if any of the keys supplied is equal to null or of an invalid type.

*see also:*

[Accessing Stored Events](#)

### **isEvent()**

`Event.isEvent()` throws an `XEPException` if the object (or class) is not an event in the XEP sense (see “[Requirements for Imported Classes](#)”). The exception message will explain why the object is not an XEP event.

```
static void isEvent(Object objectOrClass)
```

*parameter:*

- `objectOrClass` — the object to be tested.

### **startIndexing()**

`Event.startIndexing()` starts asynchronous index building for the extent of the target class. Throws an exception if the index mode is `Event.INDEX_MODE_SYNC` (see “[Controlling Index Updating](#)”).

```
void startIndexing()
```

### **stopIndexing()**

`Event.stopIndexing()` stops asynchronous index building for the extent. If you do not want the index to be updated when the `Event` instance is closed, call this method before calling `Event.close()`.

```
void stopIndexing()
```

*see also:*

[Controlling Index Updating](#)

**store()**

**Event.store()** stores a Java object or array of objects as persistent events. Returns a long database ID for each newly inserted object, or 0 if the ID could not be returned or the event uses an IdKey.

```
long store(Object object)
long[] store(Object[] objects)
```

*parameters:*

- `object` — Java object to be added to the database.
- `objects` — array of Java objects to be added to the database. All objects must be of the same type.

**updateObject()**

**Event.updateObject()** updates an event identified by its database ID or IdKey.

```
void updateObject(long id, Object object)
void updateObject(Object[] idkeys, Object object)
```

*parameter:*

- `id` — database object ID
- `idkeys` — an array of objects that make up the IdKey (see “[Using IdKeys](#)”). An `XEPException` will be thrown if the underlying class has no IdKeys or if any of the keys supplied is equal to null or of an invalid type.
- `object` — new object that will replace the specified event.

*see also:*

[Accessing Stored Events](#)

**waitForIndexing()**

**Event.waitForIndexing()** waits for asynchronous indexing to be completed, returning `true` if indexing has been completed, or `false` if the wait timed out before indexing was completed. Throws an exception if the index mode is `Event.INDEX_MODE_SYNC`.

```
boolean waitForIndexing(int timeout)
```

*parameter:*

- `timeout` — number of seconds to wait before timing out (wait forever if `-1`, return immediately if `0`).

*see also:*

[Controlling Index Updating](#)

## 3.1.5 Class `EventQuery<T>`

Class `com.intersystems.xep.EventQuery<T>` can be used to retrieve, update and delete individual events from the database.

**close()**

`EventQuery<T>.close()` releases all resources held by this instance. Always call **close()** before the `EventQuery<T>` object goes out of scope to ensure that all locks, licenses, and other resources associated with the connection are released.

```
void close()
```

**deleteCurrent()**

`EventQuery<T>.deleteCurrent()` deletes the event most recently fetched by **getNext()**.

```
void deleteCurrent()
```

*see also:*

[Processing Query Data](#)

**execute()**

`EventQuery<T>.execute()` executes the SQL query associated with this `EventQuery<T>`. If the query is successful, this `EventQuery<T>` will contain a resultset that can be accessed by other `EventQuery<T>` or `EventQueryIterator<T>` methods.

```
void execute()
```

*see also:*

[Creating and Executing a Query](#)

**getAll()**

`EventQuery<T>.getAll()` returns objects of target class `T` from all rows in the resultset as a single list.

```
java.util.List<T> getAll()
```

Uses **getNext()** to get all target class `T` objects in the resultset, and returns them in a `List`. The list cannot be used for updating or deleting (although Event methods **updateObject()** and **deleteObject()** can be used if you have some way of obtaining the `Id` or `IdKey` of each object). **getAll()** and **getNext()** cannot access the same resultset — once either method has been called, the other method cannot be used until **execute()** is called again.

*see also:*

[Processing Query Data](#), `Event.updateObject()`, `Event.deleteObject()`

**getFetchLevel()**

`EventQuery<T>.getFetchLevel()` returns the current fetch level (see “[Defining the Fetch Level](#)”).

```
int getFetchLevel()
```

**getIterator()**

`EventQuery<T>.getIterator()` returns an `EventQueryIterator<T>` that can be used to iterate over query results (see “[Processing Query Data](#)”).

```
EventQueryIterator<T> getIterator()
```



**getNext()**

`EventQuery<T>.getNext()` returns an object of target class `T` from the resultset. It returns the first item in the resultset if the argument is `null`, or takes the object returned by the previous call to `getNext()` as an argument and returns the next item in the resultset. Returns `null` if there are no more items in the resultset.

```
E getNext(E obj)
```

*parameter:*

- `obj` — the object returned by the previous call to `getNext()` (or `null` to return the first item in the resultset).

*see also:*

[Processing Query Data](#)

**setFetchLevel()**

`EventQuery<T>.setFetchLevel()` controls the amount of data returned by setting a fetch level (see “[Defining the Fetch Level](#)”).

For example, by setting the fetch level to `Event.FETCH_LEVEL_DATATYPES_ONLY`, objects returned by this query will only have their datatype fields set, and any object type, array, or collection fields will not get populated. Using this option can dramatically improve query performance.

```
void setFetchLevel(int level)
```

*parameter:*

- `level` — fetch level constant (defined in the `Event` class).

Supported fetch levels are:

- `Event.FETCH_LEVEL_ALL` —default, all fields populated
- `Event.FETCH_LEVEL_DATATYPES_ONLY` —only datatype fields filled in
- `Event.FETCH_LEVEL_NO_ARRAY_TYPES` —all arrays will be skipped
- `Event.FETCH_LEVEL_NO_OBJECT_TYPES` —all object types will be skipped
- `Event.FETCH_LEVEL_NO_COLLECTIONS` —all collections will be skipped

**setParameter()**

`EventQuery<T>.setParameter()` binds a parameter for the SQL query associated with this `EventQuery<T>`.

```
void setParameter(int index, java.lang.Object value)
```

*parameters:*

- `index` — the index of this parameter within the query statement.
- `value` — the value to be used for this query.

*see also:*

[Creating and Executing a Query](#)

**updateCurrent()**

`EventQuery<T>.updateCurrent()` updates the event most recently fetched by `getNext()`.

```
void updateCurrent(E obj)
```

*parameter:*

- `obj` — the Java object that will replace the current event.

*see also:*

[Processing Query Data](#)

## 3.1.6 Class EventQueryIterator<T>

Class `com.intersystems.xep.EventQueryIterator<T>` is the primary way to retrieve, update and delete XEP events (the same task can be also achieved by direct use of `EventQuery<T>` methods).

**hasNext()**

`EventQueryIterator<T>.hasNext()` returns `true` if the query resultset has more items.

```
boolean hasNext()
```

**next()**

`EventQueryIterator<T>.next()` fetches the next event in the query resultset.

```
E next()
```

**remove()**

`EventQueryIterator<T>.remove()` deletes the last event fetched by `next()`.

```
void remove()
```

**set()**

`EventQueryIterator<T>.set()` replaces the last event fetched by `next()`.

```
void set(E obj)
```

*parameter:*

- `obj` — an object of the target class that will replace the last event fetched by `next()`. .

## 3.1.7 Interface InterfaceResolver

By default, fields declared as interfaces are ignored during schema generation. To change this behavior, an implementation of `InterfaceResolver` can be passed to the `importSchema()` method, providing it with information that allows it to replace an interface type with the correct concrete type.

**getImplementationClass()**

`InterfaceResolver.getImplementationClass()` returns the actual type of a field declared as an interface. See [“Implementing an InterfaceResolver”](#) for details.

```
Class<?> getImplementationClass (Class declaringClass, String fieldName, Class<?> interfaceClass)
```

*parameters:*

- `declaringClass` — class where *fieldName* is declared as *interfaceClass*.
- `fieldName` — name of the field in *declaringClass* that has been declared as an interface.
- `interfaceClass` — the interface to be resolved.

### 3.1.8 Class XEPException

Class `com.intersystems.xep.XEPException` implements the exception thrown by most methods of `Event`, `EventPersister`, and `EventQuery<T>`. This class inherits from `java.lang.RuntimeException`.

Constructors

```
XEPException (String message)
XEPException (Throwable x, String message)
XEPException (Throwable x)
```

