



# Persisting .NET Objects with InterSystems XEP

Version 2025.1  
2025-06-03

*Persisting .NET Objects with InterSystems XEP*

PDF generated on 2025-06-03

InterSystems IRIS® Version 2025.1

Copyright © 2025 InterSystems Corporation

All rights reserved.

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® Data Fabric Studio™, 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 Congress Street, Boston, MA 02114, 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>1 Introduction to XEP for .NET</b>	<b>1</b>
1.1 Setup and Configuration	1
1.1.1 XEP Client Assemblies	2
<b>2 Using XEP Event Persistence with .NET</b>	<b>3</b>
2.1 Introduction to Event Persistence	3
2.1.1 Simple Applications to Store and Query Persistent Events	4
2.2 Creating and Connecting an EventPersister	7
2.3 Importing a Schema	7
2.4 Storing and Modifying Events	8
2.4.1 Creating and Storing Events	9
2.4.2 Accessing Stored Events	10
2.4.3 Controlling Index Updating	11
2.5 Using Queries	12
2.5.1 Creating and Executing a Query	12
2.5.2 Processing Query Data	13
2.5.3 Defining the Fetch Level	14
2.6 Schema Customization and Mapping	14
2.6.1 Schema Import Models	15
2.6.2 Using Attributes	16
2.6.3 Using IdKeys	18
2.6.4 Implementing an InterfaceResolver	19
2.6.5 Schema Mapping Rules	21
<b>3 Quick Reference for XEP .NET Classes</b>	<b>23</b>
3.1 XEP Quick Reference	23
3.1.1 List of XEP Methods	23
3.1.2 Class PersisterFactory	25
3.1.3 Class EventPersister	25
3.1.4 Class Event	29
3.1.5 Class EventQuery<T>	31
3.1.6 Interface InterfaceResolver	33
3.1.7 Class XEPException	34



# 1

## Introduction to XEP for .NET

See the [Table of Contents](#) for a detailed listing of the subjects covered in this document.

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

XEP is optimized for transaction processing applications that require extremely high speed data persistence and retrieval. It provides a high-performance Object/Relational Mapping (ORM) persistence framework for .NET object hierarchies. XEP projects the data in .NET objects as *persistent events* (database objects mirroring the data structures in .NET objects) in the InterSystems IRIS database.

The following topics are discussed in this document:

- [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.

Also see [Setup and Configuration](#) later in this section.

### Related Documents

The following documents also contain related material:

- [Using .NET with InterSystems Software](#) — explains how to access InterSystems IRIS from .NET ADO client applications.
- [Using the Native SDK for .NET](#) — describes how to use the .NET Native SDK to access resources formerly available only through ObjectScript.
- [Using XEP with .NET](#) — is a quick tutorial with video and code.
- [Stock Trading with .NET](#) — demonstrates how to combine XEP with other InterSystems .NET client SDKs.

## 1.1 Setup and Configuration

The XEP client assemblies can be copied to your local instance of InterSystems IRIS as part of the installation process:

- When installing InterSystems IRIS, select the `Setup Type: Development` option.
- If InterSystems IRIS has been installed with security option 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 option 1 (minimal) it should already be checked.

You can run your XEP client applications either on the same machine as the instance of InterSystems IRIS, or on a remote machine connected to an InterSystems IRIS server over TCP/IP. The remote machine does not require a local instance of InterSystems IRIS, but must have a copy of the [XEP client assemblies](#) on a path accessible to the application, and must be running a [supported version](#) of .NET or the .NET Framework.

In order to run XEP applications:

- The User namespace must exist on the machine running InterSystems IRIS, and must be writable.
- The *PATH* environment variable on the machine running InterSystems IRIS must include <install-dir>/bin (which contains the main InterSystems IRIS executables). If the *PATH* includes multiple instances of <install-dir>/bin (for example, if you have installed more than one instance of InterSystems IRIS) only the first one will be used, and any others will be ignored.

### 1.1.1 XEP Client Assemblies

Your XEP application must include references to the XEP client assemblies. The assemblies come in multiple versions, compiled under multiple supported versions of .NET. Use the file or files corresponding to the .NET version used to compile your project:

- InterSystems.Data.IRISClient.dll — is always required, and is available for each supported version of .NET and the .NET Framework.
- InterSystems.Data.XEP.dll — is required only for applications using a version of the .NET Framework. .NET 5.0 and later do not use this file.

In your instance of InterSystems IRIS, versions of these files are located in appropriately named subdirectories of <install-dir>/dev/dotnet/bin. For example, the version for .NET 6.0 is located in <install-dir>/dev/dotnet/bin/net6.0.

For a list of currently supported versions, see [Supported .NET Versions](#) in *Using .NET with InterSystems Software*.

# 2

## Using XEP Event Persistence with .NET

XEP for .NET provides extremely rapid storage and retrieval of structured data. It provides ways to control schema generation for optimal mapping of complex data structures. Schemas for simpler data structures can often be generated automatically 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 XEP.
- [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 attributes used to analyze a .NET 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 .NET 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 an advanced Object/Relational Mapping (ORM) framework that can do much more than high speed event processing. XEP “persistent events” can be data objects of significant complexity, and can be persisted from any available data 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 .NET 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 .NET class and *import a schema*, which defines how the data structure of a .NET 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 .NET class structure.
- If structural information must be preserved, the *full schema* model may be used. This preserves the full .NET inheritance structure by creating a one-to-one relationship between .NET 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 .NET 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 Attributes](#)” and “[Implementing an InterfaceResolver](#)”).

Fields of a persistent event can be simple numeric types and their associated System types, strings, objects (projected as embedded/serial objects), enumerations, and types derived from collection classes. These types can also be contained in arrays, nested collections, and collections of arrays.

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 XEP. 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 two very short applications that demonstrate all of these features.

## 2.1.1 Simple Applications to Store and Query Persistent Events

This section describes two very simple applications that use XEP to create and access persistent events:

- [The StoreEvents program](#) — opens a connection to a namespace on the server, creates a schema for the events to be stored, uses an instance of Event to store the array of objects as persistent events, then closes the connection and terminates.
- [The QueryEvents program](#) — opens a new connection accessing the same namespace as StoreEvents, creates an instance of EventQuery<T> to read and delete the previously stored events, then closes the connection and terminates.

**Note:** It is assumed that these applications have exclusive use of the system, and run in two consecutive processes.

### 2.1.1.1 The StoreEvents Program

In StoreEvents, a new instance of EventPersister is created and connected to a specific server namespace. A schema is imported for the SingleStringSample class, and the test database is initialized by deleting all existing events from the extent of the class. An instance of Event is created and used to store an array of SingleStringSample objects as persistent events. The connection is then terminated. The new events will persist in the database, and will be accessed by the QueryEvents program (described in the next section).



## The StoreEvents Program: Creating a schema and storing events

```

using System;
using InterSystems.XEP;
using xep.samples; // compiled XEPTest.csproj

public class StoreEvents {
    private static String className = "xep.samples.SingleStringSample";
    private static SingleStringSample[] eventItems = SingleStringSample.generateSampleData(12);

    public static void Main(String[] args) {
        for (int i=0; i < eventItems.Length; i++) {
            eventItems[i].name = "String event " + i;
        }
        try {
            Console.WriteLine("Connecting and importing schema for " + className);
            EventPersister myPersister = PersisterFactory.CreatePersister();
            myPersister.Connect("127.0.0.1",1972,"User", "_SYSTEM", "SYS");
            try { // delete any existing SingleStringSample events, then import new ones
                myPersister.DeleteExtent(className);
                myPersister.ImportSchema(className);
            }
            catch (XEPException e) { Console.WriteLine("import failed:\n" + e); }
            Event newEvent = myPersister.GetEvent(className);
            newEvent.Store(eventItems); // store array of events

            //Print each item in the resultset
            SingleStringSample mySample = xepQuery.GetNext();
            while (mySample != null){
                Console.WriteLine(mySample.name);
                mySample = xepQuery.GetNext();
            }
            newEvent.Close();
            myPersister.Close();
        }
        catch (XEPException e) { Console.WriteLine("Event storage failed:\n" + e); }
    } // end Main()
} // end class StoreEvents

```

Before `StoreEvents.Main()` is called, the `xep.samples.SingleStringSample.generateSampleData()` method is called to generate sample data array *eventItems*.

In this example, XEP methods perform the following actions:

- `PersisterFactory.CreatePersister()` creates *myPersister*, a new instance of `EventPersister`.
- `EventPersister.Connect()` establishes a connection to the User namespace.
- `EventPersister.ImportSchema()` analyzes the `SingleStringSample` class and imports a schema for it.
- `EventPersister.DeleteExtent()` is called to clean up the database by deleting any previously existing test data from the `SingleStringSample` extent.
- `EventPersister.GetEvent()` creates *newEvent*, a new instance of `Event` that will be used to process `SingleStringSample` events.
- `Event.Store()` accepts the *eventItems* array as input, and creates a new persistent event for each object in the array. (Alternately, the code could have looped through the *eventItems* array and called `Store()` for each individual object, but there is no need to do so in this example.)
- `Event.Close()` and `EventPersister.Close()` are called for *newEvent* and *myPersister* after the events have been stored.

All of these methods are discussed in detail later in this chapter. See “[Creating and Connecting an EventPersister](#)” for information on opening, testing, and closing a connection. See “[Importing a Schema](#)” for details about schema creation. See “[Storing and Modifying Events](#)” for details about using the `Event` class and deleting an extent.

### 2.1.1.2 The QueryEvents Program

This example assumes that QueryEvents runs immediately after the StoreEvents process terminates (see “[The StoreEvents Program](#)”). QueryEvents establishes a new database connection that accesses the same namespace as StoreEvents. An instance of EventQuery<T> is created to iterate through the previously stored events, print their data, and delete them.

#### The QueryEvents Program: Fetching and processing persistent events

```
using System;
using InterSystems.XEP;
using SingleStringSample = xep.samples.SingleStringSample; // compiled XEPTest.csproj

public class QueryEvents {
    public static void Main(String[] args) {
        EventPersister myPersister = null;
        EventQuery<SingleStringSample> myQuery = null;
        try {
            // Open a connection, then set up and execute an SQL query
            Console.WriteLine("Connecting to query SingleStringSample events");
            myPersister = PersisterFactory.CreatePersister();
            myPersister.Connect("127.0.0.1",1972,"User","_SYSTEM","SYS");
            try {
                Event newEvent = myPersister.GetEvent("xep.samples.SingleStringSample");
                String sql = "SELECT * FROM xep_samples.SingleStringSample WHERE %ID BETWEEN 3 AND ?";

                myQuery = newEvent.CreateQuery<SingleStringSample>(sql);
                newEvent.Close();
                myQuery.AddParameter(12); // assign value 12 to SQL parameter
                myQuery.Execute();
            }
            catch (XEPException e) {Console.WriteLine("createQuery failed:\n" + e);}

            // Iterate through the returned data set, printing and deleting each event
            SingleStringSample currentEvent;
            currentEvent = myQuery.GetNext(); // get first item
            while (currentEvent != null) {
                Console.WriteLine("Retrieved " + currentEvent.name);
                myQuery.DeleteCurrent();
                currentEvent = myQuery.GetNext(); // get next item
            }
            myQuery.Close();
            myPersister.Close();
        }
        catch (XEPException e) {Console.WriteLine("QueryEvents failed:\n" + e);}
    } // end Main()
} // end class QueryEvents
```

In this example, XEP methods perform the following actions:

- EventPersister.[CreatePersister\(\)](#) and EventPersister.[Connect\(\)](#) are called again (just as they were in StoreEvents) and a new connection to the User namespace is established.
- EventPersister.[GetEvent\(\)](#) creates *newEvent*, an instance of Event that will be used to create a query on the SingleStringSample extent. After the query is created, *newEvent* will be discarded by calling its [Close\(\)](#) method.
- Event.[CreateQuery\(\)](#) creates *myQuery*, an instance of EventQuery<T> where T is target class SingleStringSample. The SQL statement defines a query that will retrieve all persistent SingleStringSample events with object IDs between 3 and a variable parameter value.
- EventQuery<T>.[AddParameter\(\)](#) assigns value 12 to the SQL parameter.
- EventQuery<T>.[Execute\(\)](#) executes the query. If the query is successful, *myQuery* will now contain a resultset that lists the object IDs of all SingleStringSample events that match the query.
- EventQuery<T>.[GetNext\(\)](#) is called to fetch the first item in the resultset and assign it to variable *currentEvent*.
- In the while loop:
  - The name field of *currentEvent* is printed
  - EventQuery<T>.[DeleteCurrent\(\)](#) deletes the most recently fetched event from the database.

- `EventQuery<T>.GetNext()` is called again to fetch the next event and assign it to variable *currentEvent*.

If there are no more items, `GetNext()` will return `null` and the loop will terminate.

- `EventQuery<T>.Close()` and `EventPersister.Close()` are called for *myQuery* and *myPersister* after all events have been printed and deleted.

All of these methods are discussed in detail later in this chapter. See “[Creating and Connecting an EventPersister](#)” for information on opening, testing, and closing a connection. See “[Using Queries](#)” for details about creating and using an instance of `EventQuery<T>`.

## 2.2 Creating and Connecting an EventPersister

The `EventPersister` class is the main entry point for XEP. 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 `String` arguments for *namespace*, *username*, *password*, and establishes a connection to the specified 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 = 1972;
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.

**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 .NET 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 .NET objects are projected as ObjectScript objects. 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 .NET class and import a schema of the desired type:

- EventPersister.**ImportSchema()** — imports a *flat schema*. Takes an argument specifying a .dll 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 .NET 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 namespace test are to be imported:

```
namespace test {
    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 (XEPException e) {Console.WriteLine("Import failed:\n" + e);}
```

In this example, instances of dependent class test.Address will be serialized and embedded in the same ObjectScript 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 database 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 resources associated with it.

The following `Event` method stores .NET 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 .NET `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. See “[Simple Applications to Store and Query Persistent Events](#)” for an example of how this is done.

### 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++;
    }
    Console.WriteLine("Stored " + itemCount + " of " + eventItems.Length + " events");
    newEvent.Close();
}
catch (XEPException e) { Console.WriteLine("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 Event 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.

### 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) {Console.WriteLine("Failed to process event:\n" + e);}
    }
    Console.WriteLine("Updated " + itemCount + " of " + itemIdList.Length + " events");
    newEvent.Close();
}
catch (XEPException e) {Console.WriteLine("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 database:

- [DeleteClass\(\)](#) — takes a *className* string as an argument and deletes the specified *ObjectScript* 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 “[Simple Applications to Store and Query Persistent Events](#)” for an example of how this is done.

### 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 resources associated with it.

Queries submitted by an instance of `EventQuery<T>` will return .NET 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 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 .NET class.

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

- **AddParameter()** — binds a parameter for the SQL query associated with this `EventQuery<T>`. Takes `Object` *value* as the argument specifying the value to bind to the 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<SingleStringSample>(sql);
myQuery.AddParameter(3); // assign value 3 to first SQL parameter
myQuery.AddParameter(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>.AddParameter()` 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 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.
- **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`.

### Using Queries: Updating and Deleting Query Data

```
myQuery.Execute(); // get resultset
SingleStringSample currentEvent = myQuery.GetNext();
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();
}
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 resources associated with `myQuery`.

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

Always call `Close()` on an instance of `EventQuery<T>` 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 simple numeric types and their associated System types, strings, and enumerations. 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 collection classes.
- `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 .NET 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 Attributes](#) — XEP attributes can be added to a .NET 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 to during schema import to specify the actual class of a field declared as an interface.
- [Schema Mapping Rules](#) — provides a detailed description of how .NET 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 .NET objects are projected to ObjectScript objects.

- [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 %Library.Persistent object, and information about the original .NET 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 %Persistent objects. Inherited fields are projected as references to fields in the ancestor classes, which are also imported as %Persistent classes. There is a one-to-one correspondence between .NET source classes and ObjectScript projected classes, so the .NET class inheritance structure is preserved.

Full object projection preserves the inheritance structure of the original .NET 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 ObjectScript object extends %Library.Persistent, and contains embedded serialized objects where the original .NET 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 .NET classes:

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

Importing class C results in the following ObjectScript class:

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

No corresponding ObjectScript objects will be generated for the A or B classes unless they are specifically imported. ObjectScript object C does not extend either A or B. If imported, A and B would have the following structures:

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

All operations will be performed only on the corresponding ObjectScript object. For example, calling **Store()** on objects of type C will only store the corresponding C ObjectScript objects.

If a .NET 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 .NET inheritance model by creating a matching inheritance structure in the database class. Rather than serializing all object fields and storing all data in a single ObjectScript object, the schema establishes a one-to-one relationship between the .NET source classes and projected ObjectScript 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 ObjectScript class.

Referenced classes must include an attribute 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 attribute can be used to optimize a full schema by embedding instances of the marked class as serialized objects rather than storing them separately.

## 2.6.2 Using Attributes

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

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

- *Field Attributes* — 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 Attributes* — 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.

### [Id] (field level attribute)

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 attribute, and the field must be a `String`, `int`, or `long` (double is permitted but not recommended). To create a compound `IdKey`, use the [\[Index\]](#) attribute instead. A class marked with `[Id]` cannot also declare a compound primary key with `[Index]`. An exception will be thrown if both attributes are used on the same class.

The following parameter must be specified:

- `generated` — a `bool` specifying whether or not XEP should generate key values.
  - `generated = true` — (the default setting) key value will be generated by the server and the field marked as `[Id]` must be `Int64`. 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`, `Int32`, `long` or `Int64`.

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

```
using Id = InterSystems.XEP.Attributes.Id;
public class Person {
    [Id(generated=false)]
    public String ssn;
    public String name;
    public String dob;
}
```

### [Serialized] (field level attribute)

The `[Serialized]` attribute indicates that the field should be stored and retrieved in its serialized form.

This attribute optimizes storage of serializable fields (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.

*Example:*

```
using Serialized = InterSystems.XEP.Attributes.Serialized;
public class MyClass {
    [Serialized]
    public xep.samples.Serialized serialized;
    [Serialized]
    public int[,,,] quadIntArray;
    [Serialized]
    public String[,] doubleStringArray;
}

// xep.samples.Serialized:
[Serializable]
public class Serialized {
    public String name;
    public int value;
}
```

### [Transient] (field level attribute)

The `[Transient]` attribute indicates that the field should be excluded from import. The marked field will not be projected to the `ObjectScript` class, and will be ignored when objects are stored or loaded.

*Example:*

```
using Transient = InterSystems.XEP.Attributes.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 attribute)

The [Embedded] attribute 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 an ObjectScript class.

*Examples:*

```
using Embedded = InterSystems.XEP.Attributes.Embedded;
[Embedded]
public class Address {
    String street;
    String city;
    String zip;
    String state;
}
```

### [Index] (class level attribute)

The [Index] attribute can be used to declare one or more composite indexes.

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.attributes.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:*

```
using Index = InterSystems.XEP.Attributes.Index;
using IndexType = InterSystems.XEP.Attributes.IndexType;

[Index(name="indexOne", fields=new string[] { "ssn", "dob" }, type=IndexType.idkey)]
public class Person {
    public String name;
    public String dob;
    public String ssn;
}
```

## 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 .NET class that is imported with a full schema (see “[Importing a](#)

[Schema](#)). IdKeys on a single field can be created with the [\[Id\]](#) attribute. To create a composite IdKey, add an [\[Index\]](#) attribute with `IndexType idkey`. For example, given the following class:

```
class Person {
    String name;
    int id;
    String dob;
}
```

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

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

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

```
[Index(name="newIdKey", fields=new String[]{"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, or long, or their corresponding System types. 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 storage model. See “Define and Build 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 `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 .NET class inheritance structure. The full schema import model establishes a one-to-one relationship between .NET and ObjectScript 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) {Console.WriteLine("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`:

```
namespace test {
    public interface MyFirstInterface{ }
}
```

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

```
namespace test {
    public class MyFirstImpl : 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:

```
using InterSystems.XEP;
namespace test {
    public class MyFirstInterfaceResolver : InterfaceResolver {
        public MyFirstInterfaceResolver() {}
        public Type GetImplementationType(Type declaringClass,
            String fieldName, Type interfaceClass) {
            if (interfaceClass == typeof(test.MyFirstInterface)) {
                return typeof(test.MyFirstImpl);
            }
            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 .NET class must satisfy to produce objects that can be projected as persistent events.
- [Naming Conventions](#) — describes how .NET class and field names are translated to conform to InterSystems naming rules.

### 2.6.5.1 Requirements for Imported Classes

The XEP schema import methods cannot produce a valid schema for a .NET 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 .NET source class must explicitly declare an argumentless public constructor.
- The .NET source class cannot contain fields declared as `Object`, or arrays or collections that use `Object` as part of their declaration. An exception will be thrown if the XEP engine encounters such fields. Use the `[Transient]` attribute (see “[Using Attributes](#)”) to prevent them from being imported.

The Event.[IsEvent\(\)](#) method can be used to analyze a .NET 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 collection
- a Dictionary key value that is not a `String`, a simple numeric type or its associated `System` type, or an enumeration.

Fields of a persistent event can be simple numeric types or their associated `System` types, objects (projected as embedded/serial objects), enumerations, and types derived from collection classes. 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 .NET class. Certain fields are changed in the following ways:

- Although the .NET class may contain static fields, they are excluded from the projection by default. There will be no corresponding ObjectScript class properties. Additional fields can be excluded by using the `[Transient]` attribute (see “[Using Attributes](#)”).
- In a flat schema (see “[Schema Import Models](#)”), all object types, including inner (nested) .NET classes, are projected as %SerialObject classes in the database. The fields within the objects are not projected as separate ObjectScript 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.

### 2.6.5.2 Naming Conventions

Corresponding ObjectScript class and property names are identical to those in .NET, with the exception of two special characters allowed by .NET but not InterSystems:

- \$ (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 will receive global name `xep.samples.SingleStrinA5BFD`.

# 3

## Quick Reference for XEP .NET Classes

This chapter is a quick reference for members of the InterSystems.XEP namespace, which contains the public classes described in [Using XEP Event Persistence](#).

**Note:** This is not the definitive reference for XEP. For the most complete and up-to-date information, see the help file, located in <install-dir>/dev/dotnet/help/IrisXEP.chm.

### 3.1 XEP Quick Reference

This section is a reference for XEP classes (namespace InterSystems.XEP). See [Using XEP Event Persistence](#) for more details and examples. XEP provides 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 target class T from the database for update or deletion.
- Interface [InterfaceResolver](#) — resolves the actual type of a property during flat schema importation if the property was declared as an interface.
- Class [XEPException](#) — is the exception thrown by most XEP methods.

#### 3.1.1 List of XEP Methods

The following XEP classes and methods 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 using the arguments specified.
- **DeleteClass()** — deletes an InterSystems IRIS class.
- **DeleteExtent()** — deletes all objects in the given extent.
- **GetAdoNetConnection()** — returns the underlying ADO.NET connection.
- **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 an 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>

- **AddParameter()** — binds a parameter for this query.
- **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.
- **GetNext()** — fetches the next event in the resultset.
- **SetFetchLevel()** — controls the amount of data returned.
- **UpdateCurrent()** — updates the event most recently fetched by **GetNext()**

## InterfaceResolver

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

## 3.1.2 Class PersisterFactory

Class 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()
```

*see also:*

[Creating and Connecting an EventPersister](#)

## 3.1.3 Class EventPersister

Class 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.

### 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:*

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

- `port` — port number for TCP/IP 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 the InterSystems Managed Provider for .NET](#)).

*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 .NET 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](#)

## GetAdoConnection()

`EventPersister.GetAdoNetConnection()` returns the ADO.NET connection underlying an `EventPersister` connection.

```
System.Data.Common.DbConnection GetAdoNetConnection()
```

*see also:*

[Creating and Connecting an EventPersister](#)

**Important:** The ADO.NET connection is also used by the XEP engine, so the users should be careful not to close or corrupt the connection obtained by this method as that might cause the XEP engine to fail.

## 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 .dll file specified (including dependencies), and returns an array of class names for the imported events.

```
string[] ImportSchema(string classOrDLLName)
string[] ImportSchema(string[] classes)
```

*parameters:*

- `classes` — an array containing the names of the classes to be imported.
- `classOrDLLName` — a class name or the name of a .dll file containing the classes to be imported. If a .dll 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 .dll 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 .NET 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 .dll file specified (including dependencies), and returns an array of class names for the imported events.

```
string[] ImportSchemaFull(string classOrDLLName)
string[] ImportSchemaFull(string[] classes)
```

*parameters:*

- `classes` — an array containing the names of the classes to be imported.
- `classOrDLLName` — a class name or the name of a .dll file containing the classes to be imported. If a .dll 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 .dll 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 .NET 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 properties declared as interfaces. This argument can be null.

*see also:*

[GetInterfaceResolver\(\)](#), [ImportSchema\(\)](#)

## 3.1.4 Class Event

Class 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.

```
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 .NET 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 obj)
long[] Store(object[] objects)
```

*parameters:*

- `obj` — .NET object to be added to the database.
- `objects` — array of .NET 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 obj)
void UpdateObject(object[] idkeys, object obj)
```

*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.
- `obj` — 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`.

```
bool 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 `InterSystems.XEP.EventQuery<T>` can be used to retrieve, update and delete individual events from the database.

### AddParameter()

`EventQuery<T>.AddParameter()` binds the next parameter in sequence for the SQL query associated with this `EventQuery<T>`.

```
void AddParameter(object val)
```

*parameter:*

- `val` — the value to be used for the next parameter in this query string.

*see also:*

[Creating and Executing a Query](#)

**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>` 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.

```
System.Collections.Generic.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.

This method is more resource-intensive than a loop that explicitly calls **GetNext()** for each item, since there is a high cost associated with maintaining a list of objects.

*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()
```

**GetNext()**

`EventQuery<T>.GetNext()` returns an object of target class `T` from the next row of the resultset. Returns `null` if there are no more items in the resultset.

```
E GetNext()
```

*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 properties set, and any object type, array, or collection properties 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 properties populated
- `Event.FETCH_LEVEL_DATATYPES_ONLY` —only datatype properties 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

### UpdateCurrent()

`EventQuery<T>.UpdateCurrent()` updates the object most recently fetched by `GetNext()`.

```
void UpdateCurrent(T obj)
```

*parameter:*

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

*see also:*

[Processing Query Data](#)

## 3.1.6 Interface InterfaceResolver

By default, properties 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 property declared as an interface. See “[Implementing an InterfaceResolver](#)” for details.

```
Type GetImplementationType(Type declaringClass, string fieldName, Type interfaceClass)
```

*parameters:*

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

- `interfaceClass` — the interface to be resolved.

### 3.1.7 Class XEPException

Class `InterSystems.XEP.XEPException` implements the exception thrown by most methods of `Event`, `EventPersister`, and `EventQuery<T>`. This class inherits methods and properties from `System.SystemException`.