InterSystems XEP による Java オブジェクトの永続化

Version 2023.1
2024-01-02

InterSystems Corporation　1 Memorial Drive　Cambridge MA 02142　www.intersystems.com

InterSystems XEP による Java オブジェクトの永続化
InterSystems IRIS Data Platform　Version 2023.1　2024-01-02

インターシステムズ社の製品に関するサポートやご質問は、以下にお問い合わせください：

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

目次

Java での XEP の使用の概要

InterSystems IRIS® は、リレーショナル・テーブル (JDBC および SQL)、オブジェクト、および多次元ストレージを使用した簡単なデータベース・アクセスを実現する軽量の Java SDK を提供します。JDBC によるリレーショナル・テーブル・アクセスについては、"InterSystems ソフトウェアでの Java の使用法" を、多次元ストレージ・アクセスについては、"Native SDK for Java の使用法" を参照してください。このドキュメントでは、オブジェクト・アクセスに XEP を使用する方法について説明します。

XEP は、Java オブジェクト階層に高性能な永続性テクノロジを提供します。XEP は、Java オブジェクトのデータを永続イベント (データ・フィールドの永続コピーを格納するデータベース・オブジェクト) として投影します。これは、InterSystems IRIS データベースにアクセスします。XEP は、高速なデータの永続性と取得が必要となるトランザクション処理アプリケーション用に最適化されています。

このドキュメントで説明する項目は以下のとおりです。

- はじめに ー プラットフォーム・アーキテクチャの概要および一般的なインストール手順について説明します。
- XEP Event Persistence の使用法 ー XEP について詳しく説明し、コード例を示します。
- XEP クラスのクイック・リファレンス ー XEP クラスのメソッドのクイック・リファレンスを提供します。

詳細な目次もあります。

関連ドキュメント

以下のドキュメントにも関連資料が含まれています。

- XEP クラスの最新情報は、Java XEP API オンライン・クラス・ドキュメント (javadoc) を参照してください。
- "InterSystems ソフトウェアでの Java の使用法" では、InterSystems JDBC ドライバを使用して外部アプリケーションから InterSystems IRIS に接続する方法、および SQL を介して InterSystems IRIS から外部 JDBC データ・ソースにアクセスする方法について説明します。
- "Native SDK for Java の使用法" では、Native SDK を使用する方法について説明しています。
- "Java サード・パーティ API 用 InterSystems IRIS 実装リファレンス" の "Hibernate のサポート" では、Hibernate の InterSystems IRIS 言語について説明しています。この言語は、Java プロジェクト内の大規模で複雑なオブジェクト階層で推奨される永続性テクノロジ JPA (Java Persistence Architecture) のサポートを実装します。

1

XEP の概要

XEP は、単純なオブジェクト階層からやや複雑なオブジェクト階層までを永続化するための高性能な Java テクノロジを提供する軽量の Java SDK です。XEP は、Java オブジェクトのデータを永続イベント (データ・フィールドの永続コピーを格納するデータベース・オブジェクト) として InterSystems IRIS™ データベースに投影します。XEP は、可能な限り高速にデータを取得および永続化する必要があるアプリケーション向けに最適化されています。

InterSystems IRIS は、簡単なリレーショナルおよびオブジェクト・データベース・アクセスを実現する Java SDK を提供します。このドキュメントでは、XEP オブジェクト・アクセスについて説明します。JDBC リレーショナル・アクセスについては、″InterSystems ソフトウェアでの Java の使用法″ を参照してください。

1.1 要件と構成

このセクションでは、要件を示し、XEP を使用できるようにご使用の環境を構成する手順を示します。

1.1.1 要件

- このリリースの InterSystems IRIS でサポートされている 64 ビット Java JDK (このリリース用のオンライン・ドキュメント ″インターシステムズのサポート対象プラットフォーム″ の “サポート対象 Java テクノロジ” を参照してください)。
- InterSystems IRIS

Java クライアント・アプリケーションを実行するコンピュータに InterSystems IRIS は必要ありませんが、InterSystems IRIS サーバへの接続が必要です。また、サポートされているバージョンの Java JDK がコンピュータで実行されている必要があります。

1.1.2 インストール

- InterSystems IRIS をインストールするときに、以下のように開発環境を選択します。
  - Windows では、インストール中に `Setup Type: Development` オプションを選択します。
  - UNIX® および関連オペレーティング・システムでは、インストール中に `1) Development - Install InterSystems IRIS server and all language bindings` オプションを選択します (″インストール・ガイド″ の ″UNIX® および Linux″ のセクションにある “InterSystems IRIS の標準インストール手順” を参照してください)。

- InterSystems IRIS をセキュリティ・レベル 2 でインストールした場合、管理ポータルを開いて、System Administration > Security > Services に移動し、%Service_CallIn を選択して、[ ] ボックスにチェックが付いていることを確認します。

  InterSystems IRIS をセキュリティ・レベル 1 (最小) でインストールした場合、これには既にチェックが付いているはずです。

1.1.3 構成

すべての XEP アプリケーションで JAR ファイル intersystems-jdbc-3.0.0.jar および intersystems-xep-3.0.0.jar が必須です (ファイルの場所とその他の詳細は、"InterSystems ソフトウェアでの Java の使用法" の "InterSystems Java クラス・パッケージ" を参照してください)。CLASSPATH 環境変数には、これらのファイルへのフル・パスが含まれている必要があります。または、これらは Java コマンド行の classpath 引数で指定できます。

注釈 **Windows の追加構成**

Windows 上の Java 仮想マシンの既定のスタック・サイズは、XEP アプリケーションを実行するには小さすぎます (XEP アプリケーションを既定のスタック・サイズで実行すると、Java が EXCEPTION_STACK_OVERFLOW をレポートします)。パフォーマンスを最適化するために、ヒープ・サイズも増やす必要があります。XEP アプリケーションを実行するときにスタック・サイズとヒープ・サイズを一時的に変更するには、以下のコマンド行引数を追加します。

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

2

XEP Event Persistence の使用法

XEP は、Java 構造化データの非常に高速な格納および取得を可能にします。XEP は、複雑なデータ構造の最適なマッピングのためのスキーマ生成を制御する方法を提供しますが、単純なデータ構造のスキーマは、多くの場合、変更することなく生成および使用できます。

この章で説明する項目は以下のとおりです。

- Event Persistence の概要 – 永続イベントの概念および用語について紹介し、XEP API を使用するコードの簡単な例を示します。
- EventPersister の作成と接続 – EventPersister クラスのインスタンスを作成する方法、およびそれを使用してデータベース接続を開き、テストし、閉じる方法について説明します。
- スキーマのインポート – Java クラスを分析するため、および対応する永続イベントのスキーマを生成するためのメソッドとアノテーションについて説明します。
- イベントの格納と変更 – 永続イベントの格納、変更、および削除に使用する Event クラスのメソッドについて説明します。
- クエリの使用法 – クエリ結果セットを作成および処理する XEP クラスのメソッドについて説明します。
- スキーマのカスタマイズとマッピング – Java クラスがデータベース・スキーマにどのようにマップされるかについて詳しく説明するほか、最適なパフォーマンスを得るためのカスタマイズ・スキーマの生成方法についても詳細に説明します。

注釈 **“永続イベント”を使用する利点**

永続イベントという用語は元来、リアル・タイム・イベントから取得されたデータを指していました。XEP の現在の実装は、単純なイベント処理をはるかに超える機能を提供するように設計されているので、XEP の“永続イベント”は実際には、XEP が任意のソースから永続化したデータベース・オブジェクトです。

2.1 Event Persistence の概要

永続イベントは、Java オブジェクトにデータ・フィールドの永続コピーを格納する InterSystems IRIS データベース・オブジェクトです。既定では、XEP は、各イベントを標準の %Persistent オブジェクトとして格納します。ストレージは、その他の手段 (オブジェクト、SQL、直接グローバル・アクセスなど) でデータが InterSystems IRIS にアクセスできるように自動的に構成されます。

永続イベントを作成して格納するには、事前に XEP が対応する Java クラスを分析して、スキーマをインポートする必要があります。このスキーマは、Java オブジェクトのデータ構造をデータベースの永続オブジェクトにどのように投影するのかを定義します。スキーマは、以下の 2 つのオブジェクト・プロジェクション・モデルのうちのいずれかを使用できます。

- 既定のモデルは、フラット・スキーマです。そのスキーマでは、参照オブジェクトがすべてシリアル化されて、インポートされたクラスの一部として格納され、スーパークラスから継承したフィールドはすべて、それらが、インポートされたクラスのネイティブ・フィールドであるかのように格納されます。これは、最も高速かつ効率的なモデルですが、元の Java クラス構造に関する情報は何も保持されません。
- 構造情報を保持する必要がある場合は、フル・スキーマ・モデルを使用できます。このモデルでは、Java ソース・クラスと InterSystems IRIS の投影されたクラスとの間に 1 対 1 のリレーションシップを作成することで、完全な Java 継承構造が保持されますが、パフォーマンスがわずかに劣化することがあります。

両方のモデルの詳細は“スキーマ・インポート・モデル”を参照してください。各種のデータ型の投影方法の詳細は“スキーマ・マッピング・ルール”を参照してください。

XEP は、スキーマのインポート時に Java クラスを分析して基本情報を取得します。XEP がインデックスを生成 (“IdKey の使用法”を参照してください) して、フィールドのインポートに関する既定のルールをオーバーライド (“アノテーションの使用法”と“InterfaceResolver の実装”を参照してください) できるようにする追加情報を提供することができます。

スキーマがインポートされると、XEP を使用して非常に高いレートでデータの格納、照会、更新、および削除を行えます。格納されたイベントは、照会に対して、あるいは完全オブジェクトまたはグローバル・アクセスに対してすぐに使用できます。EventPersister、Event、および EventQuery<T> クラスは、XEP API のメイン機能を提供します。これらのクラスは、以下の順序で使用されます。

- EventPersister クラスは、データベース接続を確立して制御するメソッドを提供します (“EventPersister の作成と接続”を参照してください)。
- 接続が確立されたら、他の EventPersister メソッドを使用して、スキーマをインポートできます (“スキーマのインポート”を参照してください)。
- Event クラスは、イベントの格納、更新、または削除、クエリの作成、インデックス更新の制御を行うメソッドを提供します (“イベントの格納と変更”を参照してください)。
- EventQuery<T> クラスは、データベースから一連のイベントを取得する単純な SQL クエリを実行するために使用されます。このクラスは、結果セットを繰り返し処理し、個別のイベントの更新および削除を行うためのメソッドを提供します (“クエリの使用法”を参照してください)。

次のセクションでは、これらの機能をすべて示す非常に簡潔なアプリケーションについて説明します。

2.1.1 XepSimple サンプル・アプリケーション

以下のプログラムは、XEP と InterSystems IRIS の操作方法を示しています。この XepSimple は、小さなプログラムですが、XEP の主要機能のほとんどについて例を提供しているので、XEP API を使用する方法の概要を学ぶことができます。

このプログラムは 4 つのセクションに分かれており、それぞれが 4 つの主要 XEP クラス EventPersister、Event、EventQuery、および EventQueryIterator のいずれかを紹介しています。インポートされる xep.samples.SingleStringSample クラスは、このセクションの最後にリストされます。

クラス 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 は、以下のタスクを実行します。

- まず、サンプルのデータ・クラス xep.samples.SingleStringSample のメソッドを呼び出すことで、サンプルのオブジェクトがいくつか生成されます。
- EventPersister は、XEP の主なエントリ・ポイントで、データベースへの接続を提供します。

  このクラスは、xepPersister という名前のインスタンスを作成します。このインスタンスは、データベース内の User ネームスペースへの JDBC 接続を確立して、既存のサンプル・データをすべて削除します。次に、このクラスは、importSchema() を呼び出して、サンプル・クラスを分析し、スキーマをデータベースに送信します。その結果、SingleStringSample 永続オブジェクトを保持するエクステントが作成されます。
- Event は Java オブジェクトと対応するデータベース・オブジェクトとの間のインタフェースをカプセル化します。

  スキーマが生成されると、xepPersister は、サンプル・クラスに xepEvent という名前の Event オブジェクトを作成できます。ループで、SingleStringSample の各インスタンスに変更が加えられた後、これらのインスタンスがデータベースに永続化されます。xepEvent store() メソッドは、xepPersister で定義されている接続およびスキーマを利用します。
- EventQuery は、SQL クエリの準備と実行に使用されます。

  xepEvent オブジェクトの createQuery() メソッドにクエリ文字列を渡すことによって、xepQuery という名前の EventQuery<SingleStringSample> オブジェクトが作成されます。この文字列によって、2 つのパラメータ (? 文字) を受け取る SQL クエリが定義されます。setParameter() の呼び出しによってパラメータ値が定義され、execute() の呼び出しによってクエリの結果セットがフェッチされます。
- EventQueryIterator は、結果セットからの行の読み取り、および対応する永続オブジェクトの更新または削除に使用されます。

  xepQuery にはクエリの結果セットが含まれているので、getIterator() を呼び出すことによって、このオブジェクト用に xepIter という名前の反復子を作成できます。ループでこの反復子の next() メソッドが、データの各行を取得し、それを SingleStringSample オブジェクトに割り当てる処理に使用されます。次にこのオブジェクトに変更が加えられ、反復子の set() メソッドによって、データベース内の対応する永続オブジェクトが更新されます。
- 処理が完了したら、XEP オブジェクトに対して close() メソッドを呼び出すことによって、クリーン・アップが行われます。

以下に、XepSimple アプリケーションで使用されるサンプル・クラスのリストを示します。

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 EventPersister の作成と接続

EventPersister クラスは、XEP API の主なエントリ・ポイントです。その API は、データベースへの接続、スキーマのインポート、トランザクションの処理、および Event のインスタンスの作成を行い、データベース内のイベントにアクセスするためのメソッドを提供します。

EventPersister のインスタンスは、次のメソッドによって作成され、破棄されます。

- PersisterFactory.createPersister() ― EventPersister の新しいインスタンスを返します。
- EventPersister.close() ― この EventPersister インスタンスを閉じ、関連するリソースを解放します。

以下のメソッドを使用して、接続を作成します。

- EventPersister.connect() ― host、port、namespace、username、password の各引数を受け取り、指定された InterSystems IRIS ネームスペースへの接続を確立します。

以下の例では、接続を確立します。

EventPersister の作成と接続 : 接続の作成

```
// 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();
```

EventPersister.connect() メソッドは、ホスト・マシンの指定されたポートとネームスペースへの接続を確立します。現在のプロセス内に接続が存在しない場合は、新しい接続が作成されます。接続が既に存在している場合は、そのメソッドによって既存の接続オブジェクトへの参照が返されます。

サーバ・アドレスが `127.0.0.1` または `localhost` の場合、接続は既定でローカル共有メモリ接続になります。この接続は、標準の TCP/IP 接続よりも高速です ("InterSystems JDBC ドライバでの Java の使用法" の "共有メモリ接続" を参照してください)。

注釈 **リソースを解放するために必ず close() を呼び出す**

接続に関連するすべてのロック、ライセンス、およびその他のリソースを確実に解放するために、範囲外になる前に必ず EventPersister のインスタンスで close() を呼び出します。

2.3 スキーマのインポート

Java クラスのインスタンスを永続イベントとして格納するには、その前にそのクラスのスキーマをインポートする必要があります。スキーマによって、イベントが格納されるデータベース構造が定義されます。XEP は、フラット・スキーマとフル・スキーマの 2 つの異なるスキーマ・インポート・モデルを提供しています。これらのモデル間の主な相違は、Java オブジェクトが InterSystems IRIS イベントに投影される方法です。パフォーマンスが重要であり、イベント・スキーマが比較的単純である場合、フラット・スキーマが最適な選択になります。フル・スキーマは、より複雑なスキーマ用のより豊富な機能を提供しますが、パフォーマンスに影響を与える可能性があります。スキーマ・モデルと関連テーマの詳細は、“スキーマのカスタマイズとマッピング”を参照してください。

以下のメソッドを使用して、Java クラスを分析し、目的のタイプのスキーマをインポートします。

- EventPersister.importSchema() ― フラット・スキーマをインポートします。jar ファイル名、完全修飾クラス名、またはクラス名の配列を指定する引数を取り、すべてのクラスおよび指定された場所で検出されたすべての依存性をインポートします。正常にインポートされたすべてのクラスの名前を含む String 配列を返します。

- EventPersister.importSchemaFull() ― フル・スキーマをインポートします。importSchema() と同じ引数を取り、同じクラス・リストを返します。このメソッドによってインポートされるクラスは、ユーザ生成 IdKey を宣言する必要があります(“IdKey の使用法”を参照してください)。

- Event.isEvent() ― 静的 Event メソッド。引数としてすべての型の Java オブジェクト名またはクラス名を受け取り、指定されたオブジェクトを有効な XEP イベント(“インポートされるクラスの要件”を参照)として投影可能かどうかを調べるためのテストを行い、それが有効でない場合は適切なエラーをスローします。

インポート・メソッドは、使用されるスキーマ・モデルを除いて同一です。以下の例は、単純なテスト・クラスおよびその依存クラスをインポートします。

スキーマのインポート : クラスとその依存関係のインポート

パッケージ test の以下のクラスがインポートされます。

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

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

以下のコードは、isEvent() を呼び出してメイン・クラス test.MainClass が投影可能であることを確認してから、importSchema() を使用してそのメイン・クラスをインポートします。test.MainClass がインポートされるときに、依存クラス test.Address も自動的にインポートされます。

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

この例では、依存クラス test.Address のインスタンスは、シリアル化されて test.MainClass の他のフィールドとして同じ InterSystems IRIS オブジェクトに埋め込まれます。代わりに importSchemaFull() が使用された場合、格納された test.MainClass のインスタンスには、別の InterSystems IRIS クラス・エクステントに格納されている test.Address のインスタンスへの参照が含まれます。

2.4 イベントの格納と変更

クラスのスキーマがインポートされると（“スキーマのインポート” を参照）、Event のインスタンスを作成して、そのクラスのイベントを格納し、それらにアクセスできます。Event クラスは、永続イベントの格納、更新、または削除、そのクラス・エクステントに対するクエリの作成、インデックス更新の制御を行うメソッドを提供します。このセクションでは、以下の項目について説明します。

- イベントの作成と格納 – Event のインスタンスの作成方法およびそれを使用して指定したクラスの永続イベントを格納する方法について説明します。
- 格納されたイベントへのアクセス – 指定したクラスの永続イベントをフェッチ、変更、および削除する Event メソッドについて説明します。
- インデックス更新の制御 – インデックス・エントリを更新するタイミングを制御することで処理の効率性を高めることができる Event メソッドについて説明します。

2.4.1 イベントの作成と格納

Event クラスのインスタンスは、以下のメソッドによって作成され、破棄されます。

- EventPersister.getEvent() – className String 引数を取り、指定したクラスのイベントを格納し、それにアクセスできる Event のインスタンスを返します。オプションで、インデックス・エントリを更新する既定の方法を指定する indexMode 引数を取ります（詳細は、“インデックス更新の制御” を参照してください）。

  注釈　**ターゲット・クラス**

  Event のインスタンスは、getEvent() の呼び出しで className 引数によって指定されたクラスのイベントのみを格納、アクセス、または照会できます。この章では、クラス className をターゲット・クラスと呼びます。各例では、ターゲット・クラスは常に SingleStringSample です。

- Event.close() – この Event インスタンスを閉じ、それに関連するネイティブ・コード・リソースを解放します。

以下の Event メソッドは、ターゲット・クラスの Java オブジェクトを永続イベントとして格納します。

- store() – ターゲット・クラスの 1 つ以上のインスタンスをデータベースに追加します。引数として 1 つのイベント、または複数のイベントからなる 1 つの配列を取り、格納されているイベントごとに long データベース ID（データベース ID を返すことができない場合は 0）を返します。

  重要　イベントが格納されるときは、どのようなテストも行われず、既存のデータが変更されたり、上書きされることはありません。各イベントは、可能な限り速い速度でエクステントに追加されるか、指定されたキーを持つイベントが既にデータベースに存在する場合は暗黙的に無視されます。

以下の例では、ターゲット・クラスとして SingleStringSample を指定して Event のインスタンスを作成し、それを使用して Java SingleStringSample オブジェクトの配列を永続イベントとして投影します。この例では、myPersister が既に作成されて接続されており、SingleStringSample クラス（前述の “XepSimple サンプル・アプリケーション” にリストされています）のスキーマがインポートされていることを前提としています。

イベントの格納と変更 : オブジェクトの配列の格納

```
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);}
```

- SingleStringSample の generateSampleData() メソッドは、12 個の SingleStringSample オブジェクトを生成し、それらを eventItems という名前の配列に格納します。
- EventPersister.getEvent() メソッドは、ターゲット・クラスとして SingleStringSample を指定して newEvent という名前の Event インスタンスを作成します。
- Event.store() メソッドが呼び出され、eventItems 配列内の各オブジェクトをデータベース内の永続イベントとして投影します。

  このメソッドは、itemIdList という名前の配列を返します。この配列には、正常に格納された各イベントの場合は long オブジェクト ID が、格納できなかったオブジェクトの場合は 0 が含まれます。変数 itemCount は、itemIdList の 0 より大きい各 ID に対して 1 回インクリメントされ、その合計が出力されます。
- ループが終了すると、Event.close() メソッドが呼び出されて、関連するリソースが解放されます。

注釈 **リソースを解放するために必ず close() を呼び出す**

接続に関連するすべてのロック、ライセンス、およびその他のリソースを確実に解放するために、範囲外になる前に必ず EventPersister のインスタンスで close() を呼び出します。

2.4.2 格納されたイベントへのアクセス

永続イベントが格納されると、そのターゲット・クラスの Event インスタンスが、イベントを読み取り、更新、および削除するための以下のメソッドを提供します。

- deleteObject() ー 引数としてデータベース・オブジェクト ID または IdKey を取り、データベースから指定したイベントを削除します。
- getObject() ー 引数としてデータベース・オブジェクト ID または IdKey を取り、指定したイベントを返します。
- updateObject() ー 引数としてデータベース・オブジェクト ID または IdKey およびターゲット・クラスの Object を取り、指定したイベントを更新します。

ターゲット・クラスが標準オブジェクト ID を使用する場合、それは long 値として指定されます (前のセクションで説明した store() メソッドで返されるとおり)。ターゲット・クラスが IdKey を使用する場合、それは Object の配列として指定され、その配列の各項目はその IdKey を構成するフィールドの 1 つの値です (“IdKey の使用法” を参照してください)。

以下の例では、配列 itemIdList には、前に格納された SingleStringSample イベントのオブジェクト ID 値のリストが含まれています。例では、リストの最初の 6 個の永続イベントを適宜更新し、残りを削除します。

注釈 itemIdList 配列を作成した例は、“イベントの作成と格納” を参照してください。この例では、myPersister という名前の EventPersister インスタンスが既に作成され、データベースに接続されていることも前提としています。

SingleStringSample クラスのリストについては、“XepSimple サンプル・アプリケーション” を参照してください。

イベントの格納と変更 : イベントのフェッチ、更新、および削除

```
// 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);}
```

- EventPersister.getEvent() メソッドは、ターゲット・クラスとして SingleStringSample を指定して newEvent という名前の Event インスタンスを作成します。
- 配列 itemIdList には、前に格納された SingleStringSample イベントのオブジェクト ID 値のリストが含まれています (itemIdList を作成した例は、“イベントの作成と格納” を参照してください)。

  ループでは、itemIdList の各項目が処理されます。最初の 6 つの項目は変更および更新され、残りの項目は削除されます。以下の処理が実行されます。

  - Event.getObject() メソッドが、`itemIdList[i]` に指定されたオブジェクト ID を持つ SingleStringSample オブジェクトをフェッチして、これを変数 eventObject に割り当てます。
  - eventObject name フィールドの値が変更されます。
  - eventObject がリストの最初の 6 項目の 1 つである場合は、Event.updateObject() が呼び出されてデータベース内でこれが更新されます。そうでない場合は、Event.deleteObject() が呼び出されて、データベースからオブジェクトが削除されます。
- itemIdList のすべての ID が処理されると、ループが終了し、更新されたイベントの数を示すメッセージが表示されます。
- Event.close() メソッドが呼び出されて、関連するリソースが解放されます。

簡単な SQL クエリによってフェッチされた永続イベントへのアクセス方法、およびその変更方法の詳細は、“クエリの使用法” を参照してください。

テスト・データの削除

テスト・データベースを初期化する際は、多くの場合、クラス全体を削除するか、指定されたエクステントのすべてのイベントを削除すると便利です。以下の EventPersister メソッドは、クラスおよびエクステントを InterSystems IRIS データベースから削除します。

- deleteClass() ー 引数として className 文字列を取り、指定した InterSystems IRIS クラスを削除します。
- deleteExtent() ー 引数として className 文字列を取り、指定したクラスのエクステントのすべてのイベントを削除します。

これらのメソッドは主としてテスト用に提供されているので、実際に使用するコードでは使用しないようにしてください。これらの用語の詳しい定義については、″サーバ側プログラミングの入門ガイド″ の “クラスとエクステント” を参照してください。

2.4.3 インデックス更新の制御

既定では、データベース内のイベントに対して動作する Event メソッドの 1 つの呼び出しを実行するときに、インデックスは更新されません（“格納されたイベントへのアクセス” を参照してください）。インデックスは非同期に更新され、更新は、すべてのトランザクションが完了し、Event インスタンスが閉じられた後にのみ実行されます。一意のインデックスに対して一意性のチェックは実行されません。

注釈　このセクションは、標準オブジェクト ID または生成された IdKey を使用するクラスにのみ適用されます（“IdKey の使用法” を参照してください）。ユーザ割り当て IdKey を持つクラスは、同期でのみ更新できます。

この既定のインデックス作成動作を変更する方法はいくつかあります。Event インスタンスが EventPersister.getEvent() によって作成される場合（“イベントの作成と格納” を参照）、オプションの indexMode パラメータを設定して既定のインデックス作成動作を指定できます。以下のオプションを使用できます。

- Event.INDEX_MODE_ASYNC_ON ー 非同期のインデックス作成を可能にします。これは、indexMode パラメータが指定されていない場合の既定の設定です。
- Event.INDEX_MODE_ASYNC_OFF ー startIndexing() メソッドが呼び出されない限り、インデックス作成は実行されません。
- Event.INDEX_MODE_SYNC ー インデックス作成は、エクステントが変更されるたびに実行されます。これは多数のトランザクションの場合は非効率的になる可能性があります。クラスにユーザ割り当て IdKey がある場合は、このインデックス・モードを指定する必要があります。

以下の Event メソッドを使用すると、ターゲット・クラスのエクステントに対する非同期インデックス更新を制御できます。

- startIndexing() ー ターゲット・クラスのエクステントに対して非同期インデックス作成を開始します。インデックス・モードが Event.INDEX_MODE_SYNC である場合は、例外をスローします。
- stopIndexing() ー エクステントに対する非同期インデックス作成を停止します。Event インスタンスを閉じるときにインデックスを更新しないようにする場合は、Event.close() を呼び出す前にこのメソッドを呼び出します。
- waitForIndexing() ー int timeout 値を引数として取り、非同期インデックス作成が完了するまで待機します。timeout 値は、待機する秒数を指定します（`-1` の場合は永久に待機し、`0` の場合は直ちに返します）。インデックス作成が完了した場合は `true` を返し、インデックス作成が完了する前に待機がタイムアウトした場合は `false` を返します。インデックス・モードが Event.INDEX_MODE_SYNC である場合は、例外をスローします。

2.5 クエリの使用法

Event クラスは、ターゲット・クラスのエクステントに対して制限付き SQL クエリを実行できる EventQuery<T> のインスタンスを作成する方法を提供します。EventQuery<T> メソッドは、SQL クエリの実行、クエリ結果セット内の個別の項目の取得、更新、または削除を行うために使用します。

以下の項目について説明します。

- クエリの作成と実行 ー EventQuery<T> クラスのメソッドを使用してクエリを実行する方法について説明します。
- クエリ・データの処理 ー EventQuery<T> 結果セットの項目にアクセスし、それを変更する方法について説明します。
- フェッチ・レベルの定義 ー クエリによって返されるデータの量の制御方法について説明します。

注釈　このセクションの例では、EventPersister オブジェクトの myPersister が既に作成されて接続されており、SingleStringSample クラスのスキーマがインポートされていることを前提としています。この方法の例と SingleStringSample のリストについては、“XepSimple サンプル・アプリケーション”を参照してください。

2.5.1 クエリの作成と実行

以下のメソッドは、EventQuery<T> のインスタンスを作成および破棄します。

- Event.createQuery() ー SQL クエリのテキストを含む String 引数を取り、EventQuery<T> のインスタンスを返します。パラメータ T は、親 Event のターゲット・クラスです。
- EventQuery<T>.close() ー この EventQuery<T> インスタンスを閉じ、それに関連するネイティブ・コード・リソースを解放します。

EventQuery<T> のインスタンスによって実行されるクエリは、指定した汎用タイプ T (クエリ・オブジェクトを作成した Event インスタンスのターゲット・クラス) の Java オブジェクトを返します。EventQuery<T> クラスによってサポートされるクエリは、以下に示すとおり、SQL の SELECT 文のサブセットです。

- クエリは、SELECT 節、FROM 節、および (オプションで) WHERE、ORDER BY などの標準 SQL 節から構成される必要があります。
- SELECT 節および FROM 節は、構文的に正当ですが、現実的にはクエリ実行中には無視されます。マップされているすべてのフィールドは、常に、ターゲット・クラス T のエクステントからフェッチされます。
- SQL 式は、任意の型の配列、組み込みオブジェクト、または組み込みオブジェクトのフィールドのいずれも参照できません。
- InterSystems IRIS システム生成オブジェクト ID は、%ID と呼ばれることがあります。先頭の % のおかげで、これはフィールドが呼び出した id と Java クラス内で競合しません。

以下の EventQuery<T> メソッドは、クエリを定義および実行します。

- setParameter() ー この EventQuery<T> と関連付けられた SQL クエリのパラメータを結合します。int index および Object value を引数として取ります。ここで index は設定するパラメータを指定し、value は、指定されているパラメータに結合する値です。
- execute() ー この EventQuery<T> と関連付けられた SQL クエリを実行します。クエリが正常に実行された場合、この EventQuery<T> には、後で説明するメソッドでアクセスできる結果セットが含まれます (“クエリ・データの処理”を参照してください)。

以下の例では、xep.samples.SingleStringSample エクステント内のイベントに対して単純なクエリを実行します。

クエリの使用法 : クエリの作成と実行

```
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
```

EventPersister.getEvent() メソッドは、ターゲット・クラスとして SingleStringSample を指定して newEvent という名前の Event インスタンスを作成します。

Event.createQuery() メソッドは、SQL クエリを実行してその結果セットを保持する myQuery という名前の EventQuery<T> のインスタンスを作成します。sql 変数には、2 つのパラメータ値の間の ID を持つターゲット・クラス内のすべてのイベントを選択する SQL 文が含まれています。

EventQuery<T>.setParameter() メソッドが 2 回呼び出されて、2 つのパラメータに値を割り当てます。

EventQuery<T>.execute() メソッドが呼び出されると、ターゲット・クラスのエクステントに対して指定されたクエリが実行され、結果セットが myQuery に格納されます。

既定では、結果セットの各オブジェクトのすべてのデータがフェッチされ、各オブジェクトが完全に初期化されます。各オブジェクトでフェッチされるデータの量および型を制限するオプションについては、“フェッチ・レベルの定義”を参照してください。

2.5.2 クエリ・データの処理

クエリが実行された後、ここに記載されているメソッドを使用して、クエリ結果セット内の項目にアクセスし、データベース内の対応する永続イベントを更新または削除できます。EventQueryIterator<T> クラスは、java.util.Iterator<T> を実装します (ここで T は親 EventQuery<T> インスタンスのターゲット・クラスです)。次の EventQuery<T> メソッドは、EventQueryIterator<T> のインスタンスを作成します。

- getIterator() ー 現在の結果セットの EventQueryIterator<T> 反復子を返します。

EventQueryIterator<T> は、java.util.Iterator<T> メソッドである hasNext()、next()、および remove() と以下のメソッドを実装します。

- set() ー ターゲット・クラスのオブジェクトを取り、それを使用して next() によって最後にフェッチされた永続イベントを更新します。

以下の例は、EventQueryIterator<T> のインスタンスを作成し、それを使用して結果セットの各項目を更新します。

クエリの使用法 : EventQueryIterator<T> での反復

```
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);
}
```

EventQuery<T>.execute() の呼び出しによって、前の例で説明したクエリが実行されます (“クエリの作成と実行”を参照してください)。また、結果セットは myQuery に格納されます。結果セットの各項目は SingleStringSample オブジェクトです。

getIterator() を呼び出すと、現在 myQuery に格納されている結果セットの反復子 myIter が作成されます。

`while` ループにおいて、hasNext() は、結果セットのすべての項目が処理されるまで `true` を返します。

- next() の呼び出しによって、結果セットから次の SingleStringSample オブジェクトが返され、それが currentEvent に割り当てられます。
- currentEvent.name プロパティは変更されます。
- set() メソッドが呼び出され、更新された currentEvent オブジェクトがデータベースに格納されます。

SingleStringSample クラスのリストについては、“XepSimple サンプル・アプリケーション”を参照してください。

2.5.2.1 代替クエリ反復メソッド

EventQuery<T> クラスも、EventQueryIterator<T> を使用せずに結果セットを処理するのに使用できるメソッドを提供します(これは、ObjectScript で提供されるものと類似する反復メソッドを好む開発者向けの代替メソッドです)。クエリが実行された後、以下の EventQuery<T> メソッドを使用して、クエリ結果セット内の項目にアクセスし、データベース内の対応する永続イベントを更新または削除できます。

- getNext() ー ターゲット・クラスの次のオブジェクトを結果セットから返します。結果セットにそれ以上項目がない場合は、null を返します。引数として null またはターゲット・クラスのオブジェクトが必要です(このリリースでは、引数はクエリに影響しないプレースホルダです)。
- updateCurrent() ー 引数としてターゲット・クラスのオブジェクトを取り、それを使用して getNext() によって最後に返された永続イベントを更新します。
- deleteCurrent() ー getNext() によって最後に返された永続イベントをデータベースから削除します。
- getAll() ー getNext() を使用して結果セットからすべての項目を取得し、それらの項目を List で返します。更新または削除には使用できません。getAll() および getNext() は同じ結果セットにアクセスできません。一方のメソッドが呼び出されると、他方のメソッドは execute() が再び呼び出されるまで使用できません。

Id または IdKey によって特定された永続イベントへのアクセス方法、およびその変更方法の詳細は、“格納されたイベントへのアクセス”を参照してください。

重要 **EventQuery<T> および EventQueryIterator<T> 反復メソッドは同時に使用しない**

クエリ結果には、EventQuery<T> メソッドを直接呼び出すか、または EventQueryIterator<T> のインスタンスを取得してそのメソッドを使用することによってアクセスできますが、これらのアクセス・メソッド両方を同時に使用しないでください。反復子を取得してそのメソッドを呼び出し、同時に EventQuery<T> メソッドも直接呼び出すと、予期できない結果を招く可能性があります。

クエリの使用法 : クエリ・データの更新と削除

```
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();
```

この例では、EventQuery<T>.execute() の呼び出しによって前の例で説明したクエリが実行されることが前提になっています(“クエリの作成と実行”を参照)。また、結果セットは myQuery に格納されます。結果セットの各項目は SingleStringSample オブジェクトです。

getNext() の最初の呼び出しによって、結果セットから最初の項目が取得され、それが currentEvent に割り当てられます。

while ループで、以下の処理が結果セットの各項目に適用されます。

- currentEvent.name が文字列 "finished" で始まる場合、deleteCurrent() は対応する永続イベントをデータベースから削除します。
- それ以外の場合、currentEvent.name の値が変更され、updateCurrent() が呼び出されます。これは currentEvent を引数として取り、これを使用してデータベースの永続イベントを更新します。
- getNext() の呼び出しによって、結果セットから次の SingleStringSample オブジェクトが返され、それが currentEvent に割り当てられます。

ループが終了した後、close() が呼び出され、myQuery に関連付けられたネイティブ・コード・リソースを解放します。

注釈 **リソースを解放するために必ず close() を呼び出す**

接続に関連するすべてのロック、ライセンス、およびその他のリソースを確実に解放するために、範囲外になる前に必ず EventPersister のインスタンスで close() を呼び出します。

2.5.3 フェッチ・レベルの定義

フェッチ・レベルは、クエリを実行するときに返されるデータの量を制御するために使用できる Event プロパティです。これは、基礎となるイベントが複雑で、かつイベント・データの小さいサブセットのみが必要な場合に特に便利です。

以下の EventQuery<T> メソッドは、現在のフェッチ・レベルを設定し、返します。

- getFetchLevel() ー Event の現在のフェッチ・レベルを示す int を返します。
- setFetchLevel() ー 引数として Event フェッチ・レベル列挙の値の 1 つを取り、そのフェッチ・レベルを Event に対して設定します。

以下のフェッチ・レベル値がサポートされます。

- Event.OPTION_FETCH_LEVEL_ALL ー これが既定です。すべてのデータがフェッチされ、返されるイベントは完全に初期化されます。
- Event.OPTION_FETCH_LEVEL_DATATYPES_ONLY ー データ型フィールドのみがフェッチされます。これには、すべてのプリミティブ型、すべてのプリミティブ・ラッパ、java.lang.String、java.math.BigDecimal、java.util.Date、java.sql.Date、java.sql.Time、java.sql.Timestamp、および enum 型が含まれます。その他のフィールドはすべて null に設定されます。
- Event.OPTION_FETCH_LEVEL_NO_ARRAY_TYPES ー 配列を除くすべてのタイプがフェッチされます。ディメンジョンに関係なく、配列タイプのすべてのフィールドは null に設定されます。すべてのデータ型、オブジェクト・タイプ (シリアル化タイプを含む) およびコレクションがフェッチされます。
- Event.OPTION_FETCH_LEVEL_NO_COLLECTIONS ー java.util.List、java.util.Map、および java.util.Set の実装を除くすべてのタイプがフェッチされます。
- Event.OPTION_FETCH_LEVEL_NO_OBJECT_TYPES ー オブジェクト・タイプを除くすべてのタイプがフェッチされます。シリアル化タイプもオブジェクト・タイプと見なされ、フェッチされません。すべてのデータ型、配列タイプ、およびコレクションがフェッチされます。

2.6 スキーマのカスタマイズとマッピング

このセクションでは、Java クラスが InterSystems IRIS イベント・スキーマにどのようにマップされるか、また最適なパフォーマンスを得るためにスキーマをどのようにカスタマイズできるかについて詳細に説明します。多くの場合、メタ情報を提供することなく、単純なクラスのスキーマをインポートできます。しかし、状況によって、スキーマのインポート方法のカスタマイズが必要となったり要望されたりすることがあります。以下のセクションには、カスタマイズ・スキーマおよびその生成方法に関する情報が掲載されています。

- スキーマ・インポート・モデル ー XEP によってサポートされる 2 つのスキーマ・インポート・モデルについて説明します。
- アノテーションの使用法 ー XEP アノテーションを Java クラスに追加して、作成する必要があるインデックスを指定できます。また、アノテーションを追加して、インポートしないフィールドまたはシリアル化する必要があるフィールドを指定してパフォーマンスを最適化することもできます。
- IdKey の使用法 ー アノテーションを使用して、IdKey (既定のオブジェクト ID の代わりに使用するインデックス値) を指定できます。IdKey はフル・スキーマをインポートする場合に必要です。
- InterfaceResolver の実装 ー 既定では、フラット・スキーマは、インタフェースとして宣言されたフィールドをインポートしません。スキーマのインポート時に、InterfaceResolver インタフェースの実装を使用して、インタフェースとして宣言されたフィールドの実際のクラスを指定することができます。

- スキーマ・マッピング・ルール – Java クラスが InterSystems IRIS イベント・スキーマにどのようにマップされるかについて詳しく説明します。

2.6.1 スキーマ・インポート・モデル

XEP は、フラット・スキーマとフル・スキーマの 2 つの異なるスキーマ・インポート・モデルを提供しています。これらのモデル間の主な相違は、Java オブジェクトが InterSystems IRIS イベントに投影される方法です。

- 埋め込みオブジェクト・プロジェクション・モデル (フラット・スキーマ) – フラット・スキーマ をインポートします。そのスキーマでは、インポートされたクラスによって参照されるオブジェクトがすべてシリアル化されて埋め込まれ、すべての上位クラスで宣言されたフィールドはすべて、それらが、インポートされたクラス自体で宣言されているかのように収集され、投影されます。そのクラスのインスタンスのすべてのデータは、単一の InterSystems IRIS **%Library.Persistent** オブジェクトとして格納され、元の Java クラス構造に関する情報は保持されません。
- フル・オブジェクト・プロジェクション・モデル (フル・スキーマ) – フル・スキーマ をインポートします。そのスキーマでは、インポートされたクラスによって参照されるオブジェクトがすべて別の InterSystems IRIS **%Persistent** オブジェクトとして投影されます。継承されたフィールドは、同様に InterSystems IRIS **%Persistent** クラスにインポートされている上位クラスのフィールドへの参照として投影されます。Java ソース・クラスと InterSystems IRIS の投影されたクラスの間には 1 対 1 の対応があるため、Java クラス継承構造が保持されます。

フル・オブジェクト・プロジェクションでは、元の Java クラスの継承構造が保持されますが、パフォーマンスに影響する場合があります。パフォーマンスが重要であり、イベント・スキーマが比較的単純である場合、フラット・オブジェクト・プロジェクションが最適な選択になります。パフォーマンスの低下を許容できる場合、より豊富な機能および複雑なスキーマのために、フル・オブジェクト・プロジェクションを使用できます。

2.6.1.1 埋め込みオブジェクト・プロジェクション・モデル (フラット・スキーマ)

既定では、XEP は、平坦化することで参照オブジェクトを投影するスキーマをインポートします。つまり、インポートされたクラスによって参照されるオブジェクトがすべてシリアル化されて埋め込まれ、すべての上位クラスで宣言されたフィールドはすべて、それらが、インポートされたクラス自体で宣言されているかのように収集され、投影されます。対応する InterSystems IRIS イベントは、**%Library.Persistent** を拡張し、シリアル化されて埋め込まれたオブジェクト (元の Java オブジェクトに外部オブジェクトへの参照が含まれていた) を含んでいます。

つまり、フラット・スキーマでは、厳密な意味での継承は InterSystems IRIS 側に保持されません。例えば、以下の 3 つの Java クラスを考えてみます。

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

クラス **C** をインポートすると、以下の InterSystems IRIS クラスができます。

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

対応する InterSystems IRIS イベントは、特別にインポートしない限り、A と B のいずれのクラスに対しても生成されません。InterSystems IRIS 側のイベント C は、A も B も拡張しません。インポートされると A および B は、以下の構造になります。

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

すべての処理は、対応する InterSystems IRIS イベントに対してのみ実行されます。例えば、タイプ C のオブジェクトに対して store() を呼び出すと、対応する C InterSystems IRIS イベントのみが格納されます。

Java の子クラスが、そのスーパークラスでも宣言される同じ名前のフィールドを非表示にすると、XEP エンジンは常にその子フィールドの値を使用します。

2.6.1.2 フル・オブジェクト・プロジェクション・モデル (フル・スキーマ)

フル・オブジェクト・モデルでは、一致する継承構造を InterSystems IRIS に作成することで、Java 継承モデルを保持するスキーマをインポートします。すべてのオブジェクト・フィールドをシリアル化してすべてのデータを単一の InterSystems IRIS オブジェクトに格納するのではなく、スキーマによって Java ソース・クラスと InterSystems IRIS の投影されたクラスとの間に 1 対 1 のリレーションシップが確立されます。フル・オブジェクト・プロジェクション・モデルでは、参照されるクラスがそれぞれ別々に格納され、指定されているクラスのフィールドが、対応する InterSystems IRIS クラスのオブジェクトへの参照として投影されます。

参照されるクラスには、ユーザ定義 IdKey (@Id または @Index ー “IdKey の使用法” を参照してください) を作成するアノテーションが含まれている必要があります。オブジェクトが格納されるときは、参照されるすべてのオブジェクトが最初に格納され、結果の IdKey がその親オブジェクトに格納されます。残りの XEP と同様に、一意性チェックは実行されず、既存のデータの変更または上書きは試みられません。データは単に、可能な限り速い速度で追加されます。IdKey 値が、既に存在しているイベントを参照している場合、それは単にスキップされ、その既存のイベントの上書きが試みられることはありません。

@Embedded クラス・レベル・アノテーションを使用すると、アノテーションで指定されたクラスのインスタンスを別々に格納するのではなく、シリアル化されたオブジェクトとして埋め込むことで、フル・スキーマを最適化できます。

2.6.2 アノテーションの使用法

XEP エンジンは、Java クラスを調べることで、XEP イベント・メタデータを推測します。追加情報は、アノテーションを使用して Java クラスで指定できます。アノテーションは com.intersystems.xep.annotations パッケージにあります。Java オブジェクトは、XEP 永続イベントの定義に準拠している限り (“インポートされるクラスの要件” を参照してください)、InterSystems IRIS イベントとして投影され、カスタマイズは必要ありません。

アノテーションには、投影されるクラス内の個別のフィールドに適用されるものと、クラス全体に適用されるものがあります。

- フィールド・アノテーション ー インポートされるクラスのフィールドに適用されます。
  - @Id ー そのフィールドが IdKey として機能するようになることを指定します。
  - @Serialized ー フィールドをシリアル化形式で格納および取得する必要があることを示します。
  - @Transient ー フィールドをインポートから除外する必要があることを示します。
- クラス・アノテーション ー インポートされるクラス全体に適用されます。
  - @Embedded ー フル・スキーマでこのクラスのフィールドを、参照ではなく (フラット・スキーマと同様に) 埋め込みとする必要があることを示します。
  - @Index ー クラスのインデックスを宣言します。

- @Indices — 同じクラスの複数のインデックスを宣言します。

@Id (フィールド・レベル・アノテーション)

@Id でマークされたフィールドの値は、標準のオブジェクト ID を置換する IdKey として使用されます (“IdKey の使用法” を参照してください)。このアノテーションは、クラスごとに 1 つのフィールドでのみ使用でき、そのフィールドは String、int、または long である必要があります (double は許容されますが推奨されません)。複合 IdKey を作成するには、代わりに @Index アノテーションを使用します。@Id でマークされたクラスは、@Index を使用して複合プライマリ・キーを宣言することもできません。両方のアノテーションが同じクラスに対して使用されている場合、例外がスローされます。

以下のパラメータを指定する必要があります。

- generated — XEP がキー値を生成する必要があるかどうかを指定する boolean。
  - generated = true — (既定の設定) キー値は InterSystems IRIS によって生成され、@Id でマークされたフィールドは Long である必要があります。このフィールドは、挿入または保存の前は null であることが前提となっており、そのような操作の完了時に XEP によって自動的に入力されます。
  - generated=false — マークされたフィールドのユーザ割り当て値が IdKey 値として使用されます。フィールドは、String、int、Integer、long、または Long にできます。

次の例では、ssn フィールドのユーザ割り当て値がオブジェクト ID として使用されます。

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

@Serialized (フィールド・レベル・アノテーション)

@Serialized アノテーションは、該当のフィールドをシリアル化形式で格納および取得する必要があることを示します。

このアノテーションは、java.io.Serializable インタフェース (配列を含みます。これは暗黙的にシリアル化可能です) を実装するフィールドの格納を最適化します。XEP エンジンは、データの格納または取得のための既定のメカニズムを使用するのではなく、シリアル・オブジェクト用の関連する読み取りまたは書き込みメソッドを呼び出します。マークされたフィールドがシリアル化可能でない場合は、例外がスローされます。シリアル化フィールドの投影の詳細は、“データ型のマッピング” を参照してください。

例 :

```
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 (フィールド・レベル・アノテーション)

@Transient アノテーションは、該当のフィールドをインポートから除外する必要があることを示します。アノテーションを付けられたフィールドは、InterSystems IRIS に投影されず、イベントが格納される、またはロードされるときに無視されます。

例 :

```
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 (クラス・レベル・アノテーション)

@Embedded アノテーションは、フル・スキーマが生成される場合に使用できます (“スキーマ・インポート・モデル” を参照してください)。InterSystems IRIS に投影するときに、このクラスのフィールドを、参照ではなく (フラット・スキーマと同様に) シリアル化して埋め込む必要があることを示します。

例 :

```
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 (フィールド・レベル・アノテーション)

@Index アノテーションを使用して、インデックスを宣言できます。

以下のパラメータに引数を指定する必要があります。

- name – 複合インデックスの名前を含む String。
- fields – 複合インデックスを構成するフィールドの名前を含む String の配列。
- type – インデックス・タイプ。xep.annotations.IndexType 列挙には、以下の使用可能なタイプがあります。
  - IndexType.none – 既定値。インデックスがないことを示します。
  - IndexType.bitmap – ビットマップ・インデックス (″InterSystems SQL の使用法″ の “ビットマップ・インデックス” を参照してください)。
  - IndexType.bitslice – ビットスライス・インデックス (″InterSystems SQL の使用法″ の “概要” を参照してください)。
  - IndexType.simple – 1 つ以上のフィールドに対する標準インデックス。
  - IndexType.idkey – 標準 ID の代わりに使用されるインデックス (“IdKey の使用法” を参照してください)。

例：

```
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（クラス・レベル・アノテーション）

`@Indices` アノテーションを使用すると、1 つのクラスに対してさまざまなインデックスからなる 1 つの配列を指定できます。その配列の各要素は、@Index タグです。

例：

```
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 IdKey の使用法

IdKey は、既定のオブジェクト ID の代わりに使用されるインデックス値です。単純 IdKey と複合 IdKey のどちらも XEP によってサポートされており、フル・スキーマでインポートされる Java クラスにはユーザ生成 IdKey が必要です（“スキーマのインポート”を参照してください）。単一フィールドに対する IdKey は @Id アノテーションで作成できます。複合 IdKey を作成するには、IndexType `idkey` と共に @Index アノテーションを追加します。例えば、以下のクラスを指定したとします。

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

既定のストレージ構造では、添え字として標準オブジェクト ID を使用します。

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

以下のアノテーションは、name および id フィールドを使用して、標準オブジェクト ID を置き換える newIdKey という名前の複合 IdKey を作成します。

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

これは、以下のグローバル構造になります。

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

XEP では、SQL コマンドなど他の方法で追加された IdKey も処理されます（″InterSystems SQL の使用法″ の “インデックスでの Unique、PrimaryKey、IDKey キーワードの使用” を参照してください）。XEP エンジンは、基本クラスに IdKey が含まれているかどうかを自動的に判別し、適切なグローバル構造を生成します。

IdKey の使用法にはいくつかの制限があります。

- IdKey 値は一意である必要があります。IdKey がユーザ生成である場合、一意性は呼び出し元のアプリケーションで確保する必要があり、XEP によって強制されることはありません。アプリケーションで、データベースに既に存在しているキー値を持つイベントの追加が試みられると、その試みは暗黙的に無視され、既存のイベントは変更されません。
- IdKey を宣言するクラスには、それが他のインデックスも宣言している場合、非同期にインデックスを付けることはできません。
- 複合 IdKey ではフィールドの数に制限はありませんが、フィールドが String、int、Integer、long、または Long であることが必要です。double も使用できますが、非推奨です。
- 非常に高い継続的な挿入レートを必要とするまれな特定の状況では、パフォーマンスが低下することがあります。

IdKey に基づくイベントの取得、更新、および削除を可能にする Event メソッドの説明は、“格納されたイベントへのアクセス”を参照してください。

IdKey および標準 InterSystems IRIS ストレージ・モデルの詳細は、"グローバルの使用法" の “多次元ストレージの SQL およびオブジェクトの使用法”を参照してください。SQL での IdKey の詳細は、"InterSystems SQL の使用法" の “インデックスの定義と構築”を参照してください。

2.6.4 InterfaceResolver の実装

フラット・スキーマがインポートされるとき、継承階層に関する情報は保持されません(“スキーマ・インポート・モデル”を参照してください)。これにより、インタフェースとして宣言されているタイプを持つフィールドを処理するときに問題が発生します。これは、XEP エンジンは、フィールドの実際のクラスを把握する必要があるためです。既定では、そのようなフィールドはフラット・スキーマにインポートされません。この動作は、com.intersystems.xep.InterfaceResolver の実装を作成し、処理中に特定のインタフェース・タイプを解決することで、変更できます。

注釈　InterfaceResolver は、フラット・スキーマ・インポート・モデルにのみ関連しており、Java クラス継承構造は保持されません。フル・スキーマ・インポート・モデルでは、Java と InterSystems IRIS のクラスの間に 1 対 1 のリレーションシップが確立されるため、インタフェースの解決に必要な情報が保持されます。

InterfaceResolver の実装は、フラット・スキーマ・インポート・メソッド importSchema() を呼び出す前に EventPersister に渡されます(“スキーマのインポート”を参照してください)。これにより、XEP エンジンに、処理中にインタフェース・タイプを解決する方法が提供されます。以下の EventPersister メソッドは、使用される実装を指定します。

- EventPersister.setInterfaceResolver() ー 引数として InterfaceResolver のインスタンスを取ります。importSchema() が呼び出されると、指定されたインスタンスを使用して、インタフェースとして宣言されたフィールドを解決します。

以下の例は、クラスごとに InterfaceResolver の異なるカスタマイズされた実装を呼び出して、2 つの異なるクラスをインポートします。

スキーマのカスタマイズ : InterfaceResolver の適用

```
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);}
```

setInterfaceResolver() の最初の呼び出しによって、インポート・メソッドの呼び出し中に使用する実装として MyFirstInterfaceResolver (次の例で説明) の新しいインスタンスが設定されます。この実装は、setInterfaceResolver() が再び呼び出されて別の実装が指定されるまで、importSchema() のすべての呼び出しで使用されます。

importSchema() の最初の呼び出しで、クラス test.MyMainClass がインポートされます。これには、インタフェース test.MyFirstInterface として宣言されたフィールドが含まれています。MyFirstInterfaceResolver のインスタンスが、そのインポート・メソッドで使用され、このフィールドの実際のクラスが解決されます。

setInterfaceResolver() の 2 回目の呼び出しによって、importSchema() が再び呼び出されたときに使用する新しい実装として別の InterfaceResolver クラスのインスタンスが設定されます。

InterfaceResolver のすべての実装で、以下のメソッドを定義する必要があります。

- InterfaceResolver.getImplementationClass() は、インタフェースとして宣言されたフィールドの実際のデータ型を返します。このメソッドには以下のパラメータがあります。
  - interfaceClass ー 解決されるインタフェース。
  - declaringClass ー interfaceClass として宣言されたフィールドが含まれているクラス。
  - fieldName ー インタフェースとして宣言された declaringClass のフィールドの名前が含まれている文字列。

次の例は、インタフェース、そのインタフェースの実装、およびそのインタフェースのインスタンスを解決する InterfaceResolver の実装を定義します。

スキーマのカスタマイズ：InterfaceResolver の実装

この例では、解決されるインタフェースは、test.MyFirstInterface です。

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

test.MyFirstImpl クラスは、InterfaceResolver によって返される必要がある test.MyFirstInterface の実装です。

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

InterfaceResolver の以下の実装は、インタフェースが test.MyFirstInterface である場合はクラス test.MyFirstImpl を返し、それ以外の場合は null を返します。

```
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;
  }
}
```

MyFirstInterfaceResolver のインスタンスが setInterfaceResolver() によって指定されている場合、importSchema() の後続の呼び出しでは自動的にそのインスタンスが使用されて、test.MyFirstInterface として宣言されているすべてのフィールドが解決されます。そのようなフィールドごとに、そのフィールドを含むクラスにパラメータ declaringClass を設定し、fieldName をそのフィールドの名前に設定し、interfaceClass を test.MyFirstInterface に設定して、getImplementationClass() メソッドが呼び出されます。このメソッドは、インタフェースを解決し、test.MyFirstImpl か null のいずれかを返します。

2.6.5 スキーマ・マッピング・ルール

このセクションでは、XEP スキーマがどのように構築されるかについて説明します。以下の項目について説明します。

- インポートされるクラスの要件 ー 永続イベントとして投影できるオブジェクトを作成するために Java クラスが満たす必要がある構造ルールについて説明します。
- 名前付け規約 ー Java クラスおよびフィールドの名前を InterSystems IRIS の名前付け規則に従うように変換する方法について説明します。
- データ型のマッピング ー 使用できる Java データ型をリストし、これらが対応する InterSystems IRIS のデータ型にどのようにマップされるかについて説明します。

2.6.5.1 インポートされるクラスの要件

XEP スキーマ・インポート・メソッドは、以下の要件を満たさない限り、Java クラスに対して有効なスキーマを作成できません。

- インポートされる InterSystems IRIS クラスまたは派生クラスが、格納されたイベントにクエリを実行してそれにアクセスするために使用される場合、Java ソース・クラスは引数のないパブリック・コンストラクタを明示的に宣言する必要があります。
- Java ソース・クラスは、java.lang.Object として宣言されたフィールド、または java.lang.Object を宣言の一部として使用する配列、リスト、セットまたはマップを含むことはできません。XEP エンジンがこのようなフィールドを検出すると、例外がスローされます。`@Transient` アノテーション (“アノテーションの使用法” を参照してください) を使用して、これらがインポートされないようにします。

Event.isEvent() メソッドを使用して、Java クラスまたはオブジェクトを分析し、それが XEP の意味で有効なイベントを作成できるかどうかを判定できます。前述の条件に加えて、以下の条件が検出された場合、このメソッドは XEPException をスローします。

- 循環した依存関係
- 決まった形式のない List または Map
- String、プリミティブ、プリミティブ・ラッパのいずれでもない Map キー値

永続イベントのフィールドは、プリミティブで、そのラッパ、一時データ型、オブジェクト (組み込み/連続オブジェクトとして投影)、列挙、および java.util.List、java.util.Set、および java.util.Map から派生したデータ型にできます。これらのデータ型を配列、入れ子になったコレクション、および配列のコレクションに含めることもできます。

既定では、投影されるフィールドは、Java クラスの一部の機能を保持できません。特定のフィールドが以下の方法で変更されます。

- Java クラスに静的フィールドが含まれていても、これらは既定でプロジェクションから除外されます。対応する InterSystems IRIS プロパティはありません。追加のフィールドは、`@Transient` アノテーションを使用して除外できます (“アノテーションの使用法” を参照してください)。
- フラット・スキーマ (“スキーマ・インポート・モデル” を参照してください) では、内部 (入れ子にされた) Java クラスを含むすべてのオブジェクト・タイプは、%SerialObject クラスとして InterSystems IRIS に投影されます。オブジェクト内のフィールドは、別の InterSystems IRIS プロパティとして投影されません。このオブジェクトは、ObjectScript の観点からは不明瞭です。
- フラット・スキーマは、すべての継承されたフィールドを、それらが子クラスで宣言されたかのように投影します。

さまざまなデータ型を投影する方法の詳細は、“データ型のマッピング” を参照してください。

2.6.5.2 名前付け規約

対応する InterSystems IRIS クラスおよびプロパティの名前は Java での名前と同じです。ただし、Java では許可されているが InterSystems IRIS では許可されていない以下の 2 つの特別な文字は例外です。

- $ (ドル記号) は、InterSystems IRIS 側では "d" の 1 文字に投影されます。
- _ (アンダースコア) は、InterSystems IRIS 側では "u" の 1 文字に投影されます。

クラス名は 255 文字に制限されます。この文字数はほとんどのアプリケーションで十分のはずです。ただし、対応するグローバル名には 31 文字の制限があります。これは通常 1 対 1 マッピングには不十分なため、XEP エンジンは、31 文字よりも長いクラス名用に透過的に一意のグローバル名を生成します。生成されるグローバル名は元の名前と同じではありませんが、それでも簡単に認識できるはずです。例えば、xep.samples.SingleStringSample クラス (“XepSimple サンプル・アプリケーション” にリストされています) は、グローバル名 xep.samples.SingleStrinA5BFD を受け取ります。

2.6.5.3 データ型のマッピング

永続イベントのフィールドは、以下のデータ型のいずれかにできます。

- プリミティブ・タイプ、プリミティブ・ラッパおよび java.lang.String
- 一時データ型 (java.sql.Time、java.sql.Date、java.sql.Timestamp、および java.util.Date)
- オブジェクト・タイプ (フラット・スキーマで埋め込み/シリアル・オブジェクトとして投影される)
- Java enum データ型
- java.util.List、java.util.Set および java.util.Map から派生したデータ型
- 入れ子になったコレクション (例えばマップのリスト) および配列のコレクション
- 上記のいずれかの配列

以下のセクションでは、現在サポートされている Java タイプおよび対応する InterSystems IRIS タイプをリストします。

プリミティブおよびプリミティブ・ラッパ

以下の Java プリミティブおよびラッパは InterSystems IRIS %String としてマップされます。

- char、java.lang.Character、java.lang.String

以下の Java プリミティブおよびラッパは InterSystems IRIS %Integer としてマップされます。

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

以下の Java プリミティブおよびラッパは InterSystems IRIS %Double としてマップされます。

- double、java.lang.Double
- float、java.lang.Float

一時データ型

以下の Java 一時データ型は InterSystems IRIS %String としてマップされます

- java.sql.Date
- java.sql.Time

- java.sql.Timestamp
- java.util.Date

オブジェクト・タイプ

インポートされる Java クラス (importSchema() または importFullSchema() の呼び出しで指定されるターゲット・クラス) は、InterSystems IRIS %Persistent クラスとして投影されます。必須情報も、スーパークラスおよび依存クラスからインポートされますが、スキーマ・インポート・モデル (“スキーマ・インポート・モデル” を参照してください) によってこの情報が InterSystems IRIS にどのように格納されるのかが決定されます。

- フラット・スキーマでは、インポートされたクラスでフィールド・タイプとして現れたクラスは、%SerialObject InterSystems IRIS クラスとして投影され、親 %Persistent クラスに埋め込まれます。インポートされたクラスのスーパークラスは投影されません。代わりに、スーパークラスから継承されたすべてのフィールドが、インポートされたクラスのネイティブ・フィールドであるかのように投影されます。
- フル・スキーマでは、スーパークラスおよび依存クラスは、独立した %Persistent InterSystems IRIS クラスとして投影され、インポートされるクラスには、それらのクラスへの参照が含まれるようになります。

java.lang.Object クラスはサポートされていないクラスです。XEP エンジンが java.lang.Object として宣言されたフィールド、またはこれを使用する配列、リスト、セット、マップを検出すると例外がスローされます。

シリアル化

@Serialized アノテーション (“アノテーションの使用法” を参照してください) のマークが付けられたフィールドはすべて %Binary としてシリアル化形式で投影されます。

配列

以下の規則が配列に適用されます。

- バイト配列および文字配列を除いて、プリミティブ型、プリミティブ・ラッパ型、および一時データ型のすべての 1 次元配列は、基本ベース型のリストとしてマップされます。
- 1 次元バイト配列 (byte[] および java.lang.Byte[]) は %Binary としてマップされます。
- 1 次元文字配列 (char[] および java.lang.Character[]) は %String としてマップされます。
- オブジェクトの 1 次元配列はオブジェクトのリストとしてマップされます。
- すべての多次元配列は、%Binary としてマップされ、ObjectScript の観点からは不明瞭です。
- 配列は暗黙的にシリアル化可能であり、@Serialized のアノテーションを付けることができます。

列挙

Java enum 型は、InterSystems IRIS %String として投影され、名前のみが格納されます。InterSystems IRIS から取得されると、Java enum オブジェクト全体が再構成されます。列挙の配列、リスト、およびその他のコレクションもサポートされます。

コレクション

java.util.List および java.util.Set から派生したクラスは、InterSystems IRIS リストとして投影されます。java.util.Map から派生したクラスは、InterSystems IRIS 配列として投影されます。決まった形式のない Java リスト、セットおよびマップはサポートされません (タイプ・パラメータを使用する必要があります)。入れ子になったリスト、セットおよびマップ、配列のリスト、セットおよびマップ、ならびにリスト、セットまたはマップの配列はすべて、%Binary として投影され、InterSystems IRIS に関しては不明瞭と見なされます。

3

XEP クイック・リファレンス

この章は、このドキュメントで説明しているパブリック API を含む **com.intersystems.xep** パッケージのクイック・リファレンスです。

注釈　この章は、このドキュメントの読者の利便性を目的としたものであり、XEP の最終的なリファレンスではありません。これらのクラスに関する最も包括的な最新情報については、Java XEP API オンライン・ドキュメントを参照してください。

3.1 XEP クイック・リファレンス

このセクションは、XEP API (ネームスペース **com.intersystems.xep**) のリファレンスです。この API の使用法の詳細は、"XEP Event Persistence の使用法" を参照してください。これには、以下のクラスおよびインタフェースが含まれています。

- クラス PersisterFactory ー **EventPersister** オブジェクトを作成するために必要なファクトリ・メソッドを提供します。
- クラス EventPersister ー XEP データベース接続をカプセル化します。また、XEP オプションの設定、接続の確立または既存の接続オブジェクトの取得、スキーマのインポート、XEP イベント・オブジェクトの作成、トランザクションの制御を行うメソッドを提供します。
- クラス Event ー XEP 永続イベントへの参照をカプセル化します。また、イベントの格納または削除、クエリの作成、インデックス作成の開始または停止を行うメソッドを提供します。
- クラス EventQuery<T> ー 更新または削除するためにデータベースから特定のタイプの個別のイベントを取得するクエリをカプセル化します。
- クラス EventQueryIterator<T> ー Java **Iterator** のメソッドに類似したメソッドを使用して XEP イベントの取得、更新および削除を行う **EventQuery<T>** の代替手段を提供します。
- インタフェース InterfaceResolver ー フィールドがインタフェースとして宣言された場合、フラット・スキーマのインポート中にフィールドの実際のデータ型を解決します。
- クラス XEPException ー ほとんどの XEP メソッドによってスローされる例外です。

3.1.1 XEP メソッドのリスト

このリファレンスでは、以下の XEP API のクラスおよびメソッドを説明します。

PersisterFactory

- createPersister() ー 新しい EventPersister オブジェクトを作成します。

EventPersister

- close() ー このインスタンスで保持されているすべてのリソースを解放します。
- commit() ー 1 レベルのトランザクションをコミットします。
- connect() ー 指定された引数を使用し、TCP/IP を介して InterSystems IRIS® に接続します。
- deleteClass() ー InterSystems IRIS クラスを削除します。
- deleteExtent() ー 指定されたエクステントのすべてのオブジェクトを削除します。
- getEvent() ー 指定されたクラス名に対応するイベント・オブジェクトを返します。
- getInterfaceResolver() ー 現在指定されている InterfaceResolver のインスタンスを返します。
- getTransactionLevel() ー 現在のトランザクション・レベル (トランザクション内でない場合は 0) を返します。
- importSchema() ー フラット・スキーマをインポートします。
- importSchemaFull() ー フル・スキーマをインポートします。
- rollback() ー 指定された数のトランザクション・レベルをロールバックするか、レベルが指定されていない場合、すべてのレベルのトランザクションをロールバックします。
- setInterfaceResolver() ー 使用する InterfaceResolver オブジェクトを指定します。

Event

- close() ー このインスタンスで保持されているすべてのリソースを解放します。
- createQuery() ー EventQuery<T> インスタンスを作成します。
- deleteObject() ー データベース ID または IdKey を指定されたイベントを削除します。
- getObject() ー データベース ID または IdKey を指定されたイベントを返します。
- isEvent() ー オブジェクト (またはクラス) が XEP の意味でイベントかどうかを確認します。
- startIndexing() ー 基本クラスのインデックス作成を開始します。
- stopIndexing() ー 基本クラスのインデックス作成を停止します。
- store() ー 指定されたオブジェクトまたはオブジェクトの配列を格納します。
- updateObject() ー データベース ID または IdKey を指定されたイベントを更新します。
- waitForIndexing() ー このクラスの非同期インデックス作成が完了するまで待機します。

EventQuery<T>

- close() ー このインスタンスで保持されているすべてのリソースを解放します。
- deleteCurrent() ー getNext() によって最後にフェッチされたイベントを削除します。
- execute() ー この XEP クエリを実行します。
- getAll() ー 結果セット内のすべてのイベントを配列としてフェッチします。
- getFetchLevel() ー 現在のフェッチ・レベルを返します。

- getIterator() ー クエリ結果に対して繰り返し処理を実行するために使用できる EventQueryIterator<T> を返します。
- getNext() ー 結果セット内の次のイベントをフェッチします。
- setFetchLevel() ー 返されるデータの量を制御します。
- setParameter() ー このクエリのパラメータを結合します。
- updateCurrent() ー getNext() によって最後にフェッチされたイベントを更新します。

EventQueryIterator<T>

- hasNext() ー クエリ結果セットに他の項目がある場合、`true` を返します。
- next() ー 結果セット内の次のイベントをフェッチします。
- remove() ー next() によって最後にフェッチされたイベントを削除します。
- set() ー next() によって最後にフェッチされたイベントに新しい値を割り当てます。

InterfaceResolver

- getImplementationClass() ー フィールドがインタフェースとして宣言された場合、このメソッドの実装を使用して、スキーマのインポート中に実際のフィールド・タイプを解決できます。

3.1.2 クラス PersisterFactory

クラス com.intersystems.xep.PersisterFactory は新しい EventPersister オブジェクトを作成します。

PersisterFactory() コンストラクタ

PersisterFactory の新しいインスタンスを作成します。

```
PersisterFactory()
```

createPersister()

PersisterFactory.createPersister() は EventPersister のインスタンスを返します。

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

関連項目 :

EventPersister の作成と接続

3.1.3 クラス EventPersister

クラス com.intersystems.xep.EventPersister は、XEP モジュールの主なエントリ・ポイントです。このクラスは、XEP オプションの制御、接続の確立、スキーマのインポート、および XEP イベント・オブジェクトの作成を行うために使用できるメソッドを提供します。また、トランザクションを制御するメソッドおよびその他のタスクを実行するメソッドも提供します。

大部分のアプリケーションでは、EventPersister のインスタンスは、PersisterFactory.createPersister() によって作成する必要があります。コンストラクタは、クラスを拡張するためにのみ使用してください。

EventPersister() コンストラクタ

EventPersister の新しいインスタンスを作成します。

```
EventPersister()
```

close()

EventPersister.close() は、このインスタンスで保持されているすべてのリソースを解放します。接続に関連するすべてのロック、ライセンス、およびその他のリソースを確実に解放するために、範囲外になる前に必ず EventPersister オブジェクトで close() を呼び出します。

```
void close()
```

commit()

EventPersister.commit() は、1 レベルのトランザクションをコミットします。

```
void commit()
```

connect()

EventPersister.connect() は、指定された InterSystems IRIS ネームスペースへの接続を確立します。

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

パラメータ：

- `host` ― TCP/IP 接続のホスト・アドレス。
- `port` ― TCP/IP 接続のポート番号。
- `namespace` ― アクセス先のネームスペース。
- `username` ― この接続のユーザ名。
- `password` ― この接続のパスワード。

ホスト・アドレスが `127.0.0.1` または `localhost` の場合、接続は既定で共有メモリ接続になります。この接続は、標準の TCP/IP 接続よりも高速です（"InterSystems JDBC ドライバでの Java の使用法" の "共有メモリ接続" を参照してください）。

関連項目：

EventPersister の作成と接続

deleteClass()

EventPersister.deleteClass() は、InterSystems IRIS クラス定義を削除します。エクステントに関連付けられたオブジェクトは削除しません（オブジェクトは複数のエクステントに属することができるため）。また、依存関係（内部クラス、埋め込みクラスなど）も削除しません。

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

パラメータ：

- `className` ― 削除されるクラスの名前。

指定されたクラスが存在しない場合、呼び出しは、通知なしで失敗します（エラーはスローされません）。

関連項目：

"格納されたイベントへのアクセス" の "テスト・データの削除"

deleteExtent()

EventPersister.deleteExtent() は、Java イベントに関連付けられたエクステント定義を削除しますが、関連付けられているデータは破棄しません (オブジェクトは、複数のエクステントに属することができるため)。エクステントの管理の詳細は、"クラスの定義と使用" の "エクステント" を参照してください。

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

- `className` — エクステントの名前。

このメソッドを、すべてのエクステント・データをエクステント定義と共に破棄する非推奨の Event.deleteExtent() と混同しないでください。

関連項目 :

"格納されたイベントへのアクセス" の "テスト・データの削除"

getEvent()

EventPersister.getEvent() は、指定されたクラス名に対応する Event オブジェクトを返し、オプションで、使用するインデックス作成モードを指定します。

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

パラメータ :

- `className` — 返されるオブジェクトのクラス名。
- `indexMode` — 使用するインデックス作成モード。

以下の indexMode オプションを使用できます。

- Event.`INDEX_MODE_ASYNC_ON` — 非同期のインデックス作成を可能にします。これは、indexMode パラメータが指定されていない場合の既定の設定です。
- Event.`INDEX_MODE_ASYNC_OFF` — startIndexing() メソッドが呼び出されない限り、インデックス作成は実行されません。
- Event.`INDEX_MODE_SYNC` — インデックス作成は、エクステントが変更されるたびに実行されます。これは多数のトランザクションの場合は非効率的になる可能性があります。クラスにユーザ割り当て IdKey がある場合は、このインデックス・モードを指定する必要があります。

Event の同じインスタンスを使用して、1 つのクラスのすべてのインスタンスを格納または取得できるため、1 つのプロセスで getEvent() メソッドを呼び出す必要があるのはクラスごとに 1 回のみです。単一クラスに対する複数の Event オブジェクトのインスタンス化は避けてください。パフォーマンスに影響を及ぼしかねず、メモリ・リークの原因となることがあります。

関連項目 :

イベント・インスタンスの作成と永続イベントの格納、インデックス更新の制御

getInterfaceResolver()

EventPersister.getInterfaceResolver()― importSchema() によって使用されるようになる InterfaceResolver の現在設定されているインスタンスを返します (“InterfaceResolver の実装” を参照してください)。設定されているインスタンスがない場合、null を返します。

```
InterfaceResolver getInterfaceResolver()
```

関連項目 :

setInterfaceResolver()、 importSchema()

getTransactionLevel()

EventPersister.getTransactionLevel() は、現在のトランザクション・レベル (トランザクション内でない場合は 0) を返します。

```
int getTransactionLevel()
```

importSchema()

EventPersister.importSchema() は、シリアル化オブジェクトとしてすべての参照オブジェクトを埋め込むフラット・スキーマ (“スキーマ・インポート・モデル” を参照) を作成します。このメソッドは、クラスで宣言された各イベントのスキーマまたは指定されている .jar ファイル (依存関係を含む) をインポートし、インポートされたイベントのクラス名の配列を返します。

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

パラメータ :

- classes ― インポートされるクラスの名前を含む配列。
- classOrJarFileName ― クラス名またはインポートされるクラスを含む .jar ファイルの名前。.jar ファイルが指定される場合、ファイル内のすべてのクラスがインポートされます。

引数がクラス名である場合、対応するクラスおよびすべての依存関係がインポートされます。引数が .jar ファイルである場合、ファイル内のすべてのクラスおよびすべての依存関係がインポートされます。そのようなスキーマが既に存在していて、それが Java スキーマと同期していると思われる場合、インポートはスキップされます。スキーマが既に存在しているが、別なものと思われる場合は、データがあるか調べるためのチェックが実行されます。データがない場合、新しいスキーマが生成されます。既存のデータがある場合、例外がスローされます。

関連項目 :

スキーマのインポート

importSchemaFull()

EventPersister.importSchemaFull()― ソース・クラスのオブジェクト階層を保持するフル・スキーマ (“スキーマ・インポート・モデル” を参照) を作成します。このメソッドは、クラスで宣言された各イベントのスキーマまたは指定されている .jar ファイル (依存関係を含む) をインポートし、インポートされたイベントのクラス名の配列を返します。

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

パラメータ :

- classes ― インポートされるクラスの名前を含む配列。

- `classOrJarFileName` — クラス名またはインポートされるクラスを含む jar ファイルの名前。jar ファイルが指定される場合、ファイル内のすべてのクラスがインポートされます。

引数がクラス名である場合、対応するクラスおよびすべての依存関係がインポートされます。引数が jar ファイルである場合、ファイル内のすべてのクラスおよびすべての依存関係がインポートされます。そのようなスキーマが既に存在していて、それが Java スキーマと同期していると思われる場合、インポートはスキップされます。スキーマが既に存在しているが、別なものと思われる場合は、データがあるか調べるためのチェックが実行されます。データがない場合、新しいスキーマが生成されます。既存のデータがある場合、例外がスローされます。

関連項目 :

スキーマのインポート

rollback()

EventPersister.rollback() は、指定された数のレベルのトランザクションをロールバックします。ここで、level は正の整数です。レベルが指定されていない場合、すべてのレベルのトランザクションをロールバックします。

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

パラメータ :

- `level` — ロールバックするレベルの数 (オプション)。

このメソッドは、level が `0` 未満の場合は何もせず、level が最初のトランザクション・レベルより大きい場合、トランザクション・レベルが `0` に到達するとロールバックを停止します。

setInterfaceResolver()

EventPersister.setInterfaceResolver()— importSchema() によって使用される InterfaceResolver のインスタンスを設定します (“InterfaceResolver の実装” を参照してください)。この EventPersiser によって作成される Event のすべてのインスタンスは、指定された InterfaceResolver (このメソッドが呼び出されない場合は既定で `null` になる) を共有します。

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

パラメータ :

- `interfaceResolver` — インタフェースとして宣言されるフィールドの実際のタイプを決定するために importSchema() によって使用される InterfaceResolver の実装。この引数は、`null` にすることができます。

関連項目 :

getInterfaceResolver()、importSchema()

3.1.4 クラス Event

クラス com.intersystems.xep.Event は、XEP イベント (イベントの格納、クエリの作成、インデックスの作成他) を処理するメソッドを提供します。これは EventPersister.getEvent() メソッドにより作成されます。

close()

Event.close() は、このインスタンスで保持されているすべてのリソースを解放します。接続に関連するすべてのロック、ライセンス、およびその他のリソースを確実に解放するために、範囲外になる前に必ず Event オブジェクトで close() を呼び出します。

```
void close()
```

createQuery()

Event.createQuery() は、SQL クエリのテキストを含む String 引数を取り、EventQuery<T> のインスタンスを返します。パラメータ T は、親 Event のターゲット・クラスです。

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

パラメータ：

- `sqlText` ― この SQL クエリのテキスト。

関連項目：

クエリの作成と実行

deleteObject()

Event.deleteObject() は、データベース・オブジェクト ID または IdKey によって特定されたイベントを削除します。

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

パラメータ：

- `id` ― データベース・オブジェクト ID
- `idkeys` ― IdKey を構成するオブジェクトの配列 (“IdKey の使用法” を参照してください)。基礎となるクラスに IdKey がないか、提供されたキーのいずれかが null であるか無効なタイプである場合は XEPException がスローされます。

関連項目：

格納されたイベントへのアクセス

getObject()

Event.getObject() は、データベース・オブジェクト ID または IdKey によって特定されたイベントをフェッチします。指定したオブジェクトが存在しない場合、`null` を返します。

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

パラメータ：

- `id` ― データベース・オブジェクト ID
- `idkeys` ― IdKey を構成するオブジェクトの配列 (“IdKey の使用法” を参照してください)。基礎となるクラスに IdKey がないか、提供されたキーのいずれかが null であるか無効なタイプである場合は XEPException がスローされます。

関連項目：

格納されたイベントへのアクセス

isEvent()

Event.isEvent() は、オブジェクト (またはクラス) が XEP の意味でイベントでない場合、XEPException をスローします (“インポートされるクラスの要件” を参照してください)。例外メッセージにより、このオブジェクトが XEP イベントでない理由が説明されます。

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

パラメータ :

- `objectOrClass` – テストされるオブジェクト。

startIndexing()

Event.startIndexing() は、ターゲット・クラスのエクステントに対して非同期インデックス作成を開始します。インデックス・モードが Event.INDEX_MODE_SYNC である場合は、例外をスローします (“インデックス更新の制御” を参照してください)。

```
void startIndexing()
```

stopIndexing()

Event.stopIndexing() は、エクステントに対する非同期インデックス作成を停止します。Event インスタンスを閉じるときにインデックスを更新しないようにする場合は、Event.close() を呼び出す前にこのメソッドを呼び出します。

```
void stopIndexing()
```

関連項目 :

インデックス更新の制御

store()

Event.store() は、Java オブジェクトまたはオブジェクトの配列を永続イベントとして格納します。新しく挿入されたオブジェクトごとに 1 つの long データベース ID を返し、ID を返せない場合またはイベントが IdKey を使用している場合は `0` を返します。

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

パラメータ :

- `object` – データベースに追加される Java オブジェクト。
- `objects` – データベースに追加される Java オブジェクトの配列。すべてのオブジェクトは同じ型である必要があります。

updateObject()

Event.updateObject() は、データベース ID または IdKey によって特定されたイベントを更新します。

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

パラメータ :

- `id` – データベース・オブジェクト ID

- `idkeys` – IdKey を構成するオブジェクトの配列 (“IdKey の使用法” を参照してください)。基礎となるクラスに IdKey がないか、提供されたキーのいずれかが null であるか無効なタイプである場合は XEPException がスローされます。
- `object` – 指定したイベントを置き換える新しいオブジェクト。

関連項目 :

格納されたイベントへのアクセス

waitForIndexing()

Event.waitForIndexing() は、非同期インデックス作成の完了を待機します。インデックス作成が完了した場合は `true` を返し、インデックス作成が完了する前に待機がタイムアウトした場合は `false` を返します。インデックス・モードが Event.`INDEX_MODE_SYNC` である場合は、例外をスローします。

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

パラメータ :

- `timeout` – タイムアウトになるまで待機する秒数 (`-1` の場合は永久に待機し、`0` の場合は直ちに返します)。

関連項目 :

インデックス更新の制御

3.1.5 クラス EventQuery<T>

クラス com.intersystems.xep.EventQuery<T> は、データベースから個別のイベントを取得、更新および削除するために使用できます。

close()

EventQuery<T>.close() は、このインスタンスで保持されているすべてのリソースを解放します。接続に関連するすべてのロック、ライセンス、およびその他のリソースを確実に解放するために、EventQuery<T> オブジェクトが範囲外になる前に必ず close() を呼び出します。

```
void close()
```

deleteCurrent()

EventQuery<T>.deleteCurrent() は、getNext() によって最後にフェッチされたイベントを削除します。

```
void deleteCurrent()
```

関連項目 :

クエリ・データの処理

execute()

EventQuery<T>.execute() は、この EventQuery<T> と関連付けられた SQL クエリを実行します。クエリが正常に実行された場合、この EventQuery<T> には、他の EventQuery<T> または EventQueryIterator<T> メソッドでアクセスできる結果セットが含まれます。

```
void execute()
```

関連項目 :

クエリの作成と実行

getAll()

EventQuery<T>.getAll() は、単一のリストとして結果セット内のすべての行からターゲット・クラス T のオブジェクトを返します。

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

getNext() を使用して、結果セット内のターゲット・クラス T のオブジェクトをすべて取得し、それらのオブジェクトを List で返します。このリストは、更新または削除には使用できません (ただし、各オブジェクトの Id または IdKey を取得する方法がある場合、Event メソッドの updateObject() および deleteObject() は使用できます)。getAll() および getNext() は同じ結果セットにアクセスできません。一方のメソッドが呼び出されると、他方のメソッドは execute() が再び呼び出されるまで使用できません。

関連項目 :

"クエリ・データの処理"、Event.updateObject()、Event.deleteObject()

getFetchLevel()

EventQuery<T>.getFetchLevel() は、現在のフェッチ・レベルを返します ("フェッチ・レベルの定義" を参照してください)。

```
int getFetchLevel()
```

getIterator()

EventQuery<T>.getIterator() は、クエリ結果に対して繰り返し処理を実行するために使用できる EventQueryIterator<T> を返します ("クエリ・データの処理" を参照してください)。

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

getNext()

EventQuery<T>.getNext() は、ターゲット・クラス T のオブジェクトを結果セットから返します。引数が null の場合、結果セットの最初の項目を返します。または、前の getNext() の呼び出しによって返されたオブジェクトを引数として取り、結果セットの次の項目を返します。結果セットにそれ以上項目がない場合は、null を返します。

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

パラメータ :

- obj ― 前の getNext() の呼び出しによって返されたオブジェクト (または、結果セットの最初の項目を返す場合は null)。

関連項目 :

クエリ・データの処理

setFetchLevel()

EventQuery<T>.setFetchLevel() は、フェッチ・レベルを設定することで返されるデータの量を制御します ("フェッチ・レベルの定義" を参照してください)。

例えば、フェッチ・レベルを Event.FETCH_LEVEL_DATATYPES_ONLY に設定すると、このクエリによって返されるオブジェクトは、それらのデータ型フィールドのみ設定されるようになり、オブジェクト・タイプ、配列、またはコレクションのフィールドには何も生成されません。このオプションを使用すると、クエリ・パフォーマンスが大幅に向上する場合があります。

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

パラメータ：

- level ― フェッチ・レベル定数 (Event クラスで定義される)。

サポートされるフェッチ・レベルは以下のとおりです。

- Event.FETCH_LEVEL_ALL ― 既定。すべてのフィールドが生成されます。
- Event.FETCH_LEVEL_DATATYPES_ONLY ― データ型フィールドのみが生成されます。
- Event.FETCH_LEVEL_NO_ARRAY_TYPES ― すべての配列がスキップされます。
- Event.FETCH_LEVEL_NO_OBJECT_TYPES ― すべてのオブジェクト・タイプがスキップされます。
- Event.FETCH_LEVEL_NO_COLLECTIONS ― すべてのコレクションがスキップされます。

setParameter()

EventQuery<T>.setParameter() は、この EventQuery<T> と関連付けられた SQL クエリのパラメータを結合します。

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

パラメータ：

- index ― クエリ文内のこのパラメータのインデックス。
- value ― このクエリに使用される値。

関連項目：

クエリの作成と実行

updateCurrent()

EventQuery<T>.updateCurrent() は、getNext() によって最後にフェッチされたイベントを更新します。

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

パラメータ：

- obj ― 現在のイベントを置き換える Java オブジェクト。

関連項目：

クエリ・データの処理

3.1.6 クラス EventQueryIterator<T>

クラス com.intersystems.xep.EventQueryIterator<T> は、XEP イベントの取得、更新、および削除を行う主要手段です (同じタスクは、EventQuery<T> メソッドを直接使用しても実行できます)。

hasNext()

EventQueryIterator<T>.hasNext() は、クエリ結果セットに他の項目がある場合、true を返します。

```
boolean hasNext()
```

next()

EventQueryIterator<T>.next() は、クエリ結果セット内の次のイベントをフェッチします。

```
E next()
```

remove()

EventQueryIterator<T>.remove() は、next() によって最後にフェッチされたイベントを削除します。

```
void remove()
```

set()

EventQueryIterator<T>.set() は、next() によって最後にフェッチされたイベントを置換します。

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

パラメータ：

- obj – next() によって最後にフェッチされたイベントを置き換えるターゲット・クラスのオブジェクト。.

3.1.7 インタフェース InterfaceResolver

既定では、インタフェースとして宣言されたフィールドはスキーマ生成の際に無視されます。この動作を変更するには、InterfaceResolver の実装を importSchema() メソッドへ渡して、インタフェース・タイプを正しい具体的なデータ型で置換できる情報を提供すると、できます。

getImplementationClass()

InterfaceResolver.getImplementationClass() は、インタフェースとして宣言されたフィールドの実際のデータ型を返します。詳細は、“InterfaceResolver の実装”を参照してください。

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

パラメータ：

- declaringClass – fieldName が interfaceClass として宣言されるクラス。
- fieldName – インタフェースとして宣言された declaringClass のフィールド名。
- interfaceClass – 解決されるインタフェース。

3.1.8 クラス XEPException

クラス com.intersystems.xep.XEPException は、Event、EventPersister、および EventQuery<T> のほとんどのメソッドによりスローされる例外を実装します。このクラスは、java.lang.RuntimeException から継承されます。

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