# Table of Contents

**About This Book** ................................................................................................................................................. 1

1 **Introduction** .................................................................................................................................................... 3
   1.1 Overview ......................................................................................................................................................... 3
   1.2 Installation and Configuration ....................................................................................................................... 3
      1.2.1 Requirements ........................................................................................................................................... 4
      1.2.2 Installation ............................................................................................................................................ 4
      1.2.3 Required Environment Variables for All Platforms .............................................................................. 4
      1.2.4 Windows Configuration .......................................................................................................................... 5
      1.2.5 Mac OS X Configuration ......................................................................................................................... 5
      1.2.6 Unix Configuration ................................................................................................................................ 5
   1.3 eXtreme Sample Applications ...................................................................................................................... 5
      1.3.1 XEP Samples .......................................................................................................................................... 5
      1.3.2 Dynamic Objects Samples ........................................................................................................................ 6
      1.3.3 Globals API Samples ............................................................................................................................. 6
      1.3.4 Sample Data Classes ............................................................................................................................. 6

2 **Using eXtreme Event Persistence** ................................................................................................................ 9
   2.1 Introduction to Event Persistence .................................................................................................................. 9
   2.2 Creating and Connecting an EventPersister ................................................................................................. 13
   2.3 Importing a Schema ....................................................................................................................................... 14
   2.4 Storing and Modifying Events ...................................................................................................................... 15
      2.4.1 Creating and Storing Events .................................................................................................................. 15
      2.4.2 Accessing Stored Events ....................................................................................................................... 17
      2.4.3 Controlling Index Updating .................................................................................................................. 18
      2.4.4 Accessing Stored Events ....................................................................................................................... 18
   2.5 Using Queries ................................................................................................................................................ 18
      2.5.1 Creating and Executing a Query ............................................................................................................. 18
      2.5.2 Processing Query Data ........................................................................................................................ 19
      2.5.3 Using EventQueryIterator<> .............................................................................................................. 20
      2.5.4 Defining the Fetch Level ....................................................................................................................... 21
   2.6 Calling Caché Methods from XEP .................................................................................................................. 21
   2.7 Schema Mapping and Customization ............................................................................................................ 22
      2.7.1 Schema Import Models ........................................................................................................................ 22
      2.7.2 Schema Customization ........................................................................................................................ 23
      2.7.3 Schema Mapping Rules ........................................................................................................................ 29

3 **Using eXtreme Dynamic Objects** ................................................................................................................. 33
   3.1 Opening and Closing an XDO Connection ................................................................................................. 34
   3.2 Using Dynamic Objects .............................................................................................................................. 35
      3.2.1 Creating Dynamic Objects and Database Objects .......................................................................... 35
      3.2.2 Inserting a New Database Object ..................................................................................................... 36
      3.2.3 Modifying an Existing Database Object ............................................................................................ 37
      3.2.4 Using Property Numbers ................................................................................................................... 39
   3.3 Transaction Processing ............................................................................................................................... 39
      3.3.1 Using Transactions ............................................................................................................................... 39

4 **Using the Globals API** ............................................................................................................................... 43
   4.1 Introduction to the Globals API .................................................................................................................. 44
5.3.1 List of Globals API Methods ................................................................. 103
5.3.2 Class ConnectionContext ................................................................. 103
5.3.3 Interface Connection ....................................................................... 104
5.3.4 InterfaceGlobalsDirectory ............................................................. 108
5.3.5 Interface NodeReference ................................................................. 109
5.3.6 Interface ValueList ........................................................................ 117
5.3.7 Class ByteArrayRegion .................................................................. 120
5.3.8 Globals Exception Classes ............................................................. 121
5.3.9 Globals Implementation Classes .................................................... 123
About This Book

Caché eXTreme is a set of Java technologies that enable Caché to be leveraged as a high performance persistence storage engine in XTP (Extreme Transaction Processing) applications. Unlike the standard Caché Java binding, all of the eXTreme APIs provide access to Caché data through an extremely high speed, in-process connection.

The following topics are discussed in this book:

- **Introduction** — provides an overview of the eXTreme platform architecture, and describes common installation procedures.
- **Using eXTreme Event Persistence** — describes the XEP API, which allows simple Java objects to be projected as XTP persistent events.
- **Using eXTreme Dynamic Objects** — describes the XDO API, which provides object oriented access to Caché data.
- **Using the Globals API** — describes the Globals API, which provides direct access to Caché multidimensional data storage.
- **Quick Reference for eXTreme Classes** — Provides a quick reference for methods of the eXTreme API classes.

There is also a detailed Table of Contents.

Related Documents

The following documents also contain related material:

- **JavaDoc for the InterSystems Java Connectivity API** is located in `<cache-root>/dev/java/doc/index.html`.
- **Using Java with Caché** — a guide to the Caché Java Binding, which provides a simple, direct way to use Caché objects within a Java application.
- **Using Caché with JDBC** — describes how to connect to Caché from an external application using the Caché JDBC driver, and how to access external JDBC data sources from Caché.
- **Using Java with Caché Jalapeño** — describes how to use Jalapeño, a lightweight persistence tool that provides an easy way to store and access POJOs (Plain Old Java Objects) without mapping.
- **Caché Java Binding and JDBC QuickStart Tutorial** — provides a quick introduction to working with the Java binding. It includes a complete sample Java binding application.

For general information, see *Using InterSystems Documentation*. 
1

Introduction

Caché eXTreme is a set of technologies that enable Caché to be leveraged as a high performance persistence storage engine optimized for XTP (Extreme Transaction Processing) applications.

Unlike the standard Java binding, the eXTreme APIs do not use TCP/IP to communicate with Caché. Instead, they use a fast in-memory connection (implemented via standard Java JNI and the Caché Callin API), and run in the same process as the Caché instance. Although the Caché server and the Java application must be on the same machine, the application can still use Caché ECP to access data on remote machines.

1.1 Overview

Caché eXTreme components include:

- **IN-Process JDBC** — JDBC running over JNI instead of TCP/IP.
- **eXTreme Event Persistence (XEP)** — allows simple Java objects to be projected as XTP persistent events for rapid storage and processing. This is a lightweight API for low latency object and event stream data access (see “Using eXTreme Event Persistence”).
- **Dynamic Object API** — provides object oriented access to Caché data. Access to data in Caché classes is dynamic, meaning that the classes do not have to be known at Java application compile time, and no code generation is required (see “Using eXTreme Dynamic Objects”).
- **The Globals API** — provides direct access to Caché global arrays, allowing maximum speed and flexibility (see “Using the Globals API”).

The eXTreme APIs are designed for integration with Java platforms such as the following:

- **OSGi** for Event SOA and Dynamic Modules.
- **CEP** (Complex Event Processing) engines such as Esper.
- **Messaging** integration (JMS, AMQP, etc.).

1.2 Installation and Configuration

This section provides specifies requirements and provides instructions for installing Caché and configuring your environment to use the eXTreme APIs.
1.2.1 Requirements

- Java JDK version 1.6.
- Caché 2010.2 or higher.
- The Caché User namespace must exist and must be writable if your application uses XEP (see “Using eXTreme Event Persistence”).

1.2.2 Installation

- When installing Caché, select the Development environment:
  - In Windows, select the Setup Type: Development option during installation.
  - In UNIX®, select the 1) Development - Install Cache server and all language bindings option during installation (see “Run the Installation Script” in the UNIX® and Linux section of the Caché Installation Guide).
- If Caché has been installed with security level 2, open the Management Portal and go to [Home] > [Security Management] > [Services], select %Service_CallIn, and make sure the Service Enabled box is checked.
  If you installed Caché with security level 1 (minimal) it should already be checked.

1.2.3 Required Environment Variables for All Platforms

In order to run eXTreme applications, the following environment variables must be properly set on all platforms:

- Your $GLOBALS_HOME environment variable must be set to the full path of your Caché installation’s `<cache-root>` directory (see “Default Caché Installation Directory” for the location of `<cache-root>` on your system).

  **Note:** $CACHEMGRDIR Deprecated
  In previous releases, the required environment variable was $CACHEMGRDIR, set to the `<cache-root>/mgr` directory rather than `<cache-root>`. This variable is deprecated. If both variables are set, $GLOBALS_HOME will be used.

- Your Path must include `<cache-root>/bin`:
  - In Windows, add it to your `PATH` environment variable.
  - In UNIX®, add it to your `LD_LIBRARY_PATH` environment variable.
  - In Mac OS X, add it to your `DYLD_LIBRARY_PATH` environment variable.
  If your Path variable includes more than one `<cache-root>/bin` path (for example, if you have installed more than one instance of Caché) only the first one will be used, and any others will be ignored.

- your CLASSPATH environment variable must include the full paths to the required JAR files: CacheJDBC.jar, CacheDB.jar, and CacheeXTreme.jar (see “The Caché Java Class Packages” in Using Java with Caché). Alternately, these files can be specified in the Java command line `classpath` argument.
1.2.4 Windows Configuration

- The default stack size of the Java Virtual Machine on Windows is too small for running eXTreme applications (running them with the default stack size causes Java to report EXCEPTION_STACK_OVERFLOW). To optimize performance, heap size should also be increased. To temporarily modify the stack size and heap size when running an eXTreme application, add the following command line arguments:
  
  -Xss1024k
  -Xms2500m -Xmx2500m

1.2.5 Mac OS X Configuration

- Make sure that you have permissions on the Cache binaries (add the user to the cacheusr group).

1.2.6 Unix Configuration

- Make sure that you have permissions on the Cache binaries (add the user to the cacheusr group).
- Set the environment variable LD_PRELOAD to the path of libjsig.so (a library which enables Java to resolve signal handling anomalies) within your Java installation. For example (depending on which shell you are using):

  setenv LD_PRELOAD /my_jdk_path/jdk1.6.0_11/jre/lib/amd64/libjsig.so

  or

  set LD_PRELOAD=/my_jdk_path/jdk1.6.0_11/jre/lib/amd64/libjsig.so

  The path of libjsig.so under the root of a Java installation may vary from platform to platform, or from one Java release to another. You can locate it on your system with the following command:

  find $JAVA_HOME -name libjsig.so -print

  where JAVA_HOME is set to the root directory of your Java installation.

  Note: The LD_PRELOAD variable setting is important if your eXTreme application also uses other third party components that set up signal handlers. It enables Java to chain signal handlers set by Caché with its own signal handlers, so that they do not interfere with each other. Failure to set this variable may result in a system crash.

1.3 eXTreme Sample Applications

Sample applications are available for all three eXTreme APIs. Run the samples with command line argument -h for a list of available command line options.

1.3.1 XEP Samples

XEP sample files are in `<cache-root>/dev/java/samples/extreme/xep/test/`. For convenience, these files are also compiled into extremesamples.jar, located in `<cache-root>/dev/java/samples`. The following sample programs are available:

- `RunAll.java` — is a program that runs all of the other sample programs in sequence.
• Coverage.java — tests basic functionality such as connecting, importing a schema, storing, querying, updating and deleting XEP events. It also exercises most of the supported data types.

• SingleString.java — is the most basic XEP test program. It connects to the database, imports a simple class containing only one string field, then stores and loads a number of events corresponding to that class.

• FlightLog.java — is an example that demonstrates the XEP full inheritance model. It tracks airline flight information such as times, locations, personnel, and passengers.

• Benchmark.java — is a performance test for the XEP API.

• IdKeys.java — extends the Benchmark test by adding the composite IdKey feature.

• Threads.java — is a multithreaded XEP test program. It extends the Java Thread class, and uses the Basic.java test suite to test XEP using multiple threads.

See the Caché JavaDoc (<cache-root>/dev/java/doc/index.html) for detailed documentation of these programs. Supporting files located in <cache-root>/dev/java/samples/extreme/xep/samples/ provide test data for the sample programs.

1.3.2 Dynamic Objects Samples

The following XDO sample program is located in <cache-root>/dev/java/samples/extreme/xdo:

• XDODemo.java — is a demonstration and performance test for the eXTreme Dynamic Object API.

This program is also compiled into extremesamples.jar, located in <cache-root>/dev/java/samples.

1.3.3 Globals API Samples

The Globals API sample application (com.intersys.globals.samples.Sample) is compiled into cacheextreme.jar, located in <cache-root>/dev/java/lib/JDK16. To run the program, enter the following command:

```java
java -Xss1024k -cp %jarpath% com.intersys.globals.samples.Sample -pause
```

where %jarpath% is set to %GLOBALS_HOME%/dev/java/lib/JDK16/cacheextreme.jar

The jar file also contains a copy of the source file, Sample.java, which can be extracted to the current directory by the following command:

```bash
jar -xf %jarpath% com/intersys/globals/samples/Sample.java
```

1.3.4 Sample Data Classes

The examples in this book frequently use the CosTutorial.Person and Sample.Person extents in the Caché Samples namespace to demonstrate data storage and retrieval.

**Note:** The contents of these extents can be viewed in the Management Portal by going to [System] > [Globals], selecting Namespace and Samples from the choices in the left column, and clicking the view link for the desired extent (the CosTutorial.Person extent is listed under PersonD, not CosTutorial.PersonD).

**CosTutorial.Person**

This very simple Caché class contains the following declarations:

- property DOB as %Date;
- property Name as %String;
property Phone as %String(COLLATION="EXACT",MAXLEN=12);

index Phone on Phone [Unique];

The following commands can be copied and pasted into the Caché Terminal to populate the extent or delete all objects in the extent:

```cache
//Switch to the Samples namespace:
zn "Samples"
// Delete all instances of CosTutorial.Person from the extent:
Do ##class(CosTutorial.Person).%KillExtent()
// Create seven new instances of CosTutorial.Person:
Do ##class(CosTutorial.Person).Populate(7)
```

For complete structural details, see the `CosTutorial.Person` entry in the Class Reference. For a Caché ObjectScript example, see the *Caché ObjectScript Tutorial* (starting with “Caché Objects and SQL”).

**Sample.Person**

Some examples in this book use the `Sample.Person` extent as a data source. The following commands can be copied and pasted into the Caché Terminal to populate the extent or delete all objects in the extent:

```cache
//Switch to the Samples namespace:
zn "Samples"
// Delete all instances of Sample.Person from the extent:
Do ##class(Sample.Person).%KillExtent()
// Create seven new instances of Sample.Person:
Do ##class(Sample.Person).Populate(7)
```

For structural details, see the `Sample.Person` entry in the Class Reference. This class cannot be used by the XDO API (see “Using eXTreme Dynamic Objects”) because it contains serial properties.

**Returning a ResultSet from Sample.Person**

The following example demonstrates a simple query that returns a ResultSet containing sample data.

```java
java.sql.ResultSet myResults = null;
try {
    String sql="Select Name, DOB from Sample.Person Order By Name";
    myResults = jdbcConnection.prepareStatement(sql).executeQuery();
} catch (java.sql.SQLException e) { System.out.println("Query failed: " + e.getMessage()); }
```

A ResultSet is generated by using a JDBC connection to query the `Sample.Person` extent (see “Accessing JDBC Databases” in *Using Caché with JDBC* for more examples). Each row in the `myResults` object contains "Name" and "DOB" properties, which could be used as input to populate another extent, such as `CosTutorial.Person`.
Caché eXTreme Event Persistence (XEP) provides extremely rapid storage and retrieval of Java structured data. It implements a fast, efficient object-to-global mapping algorithm that harnesses the power of the Globals API (see “Using the Globals API”), but no knowledge of the Globals API is required. XEP provides ways to control schema generation for optimal mapping of complex data structures, but schemas for simpler data structures can often be generated and used without modification. Unlike the Globals API, XEP can use either in-process communications or a TCP/IP connection.

The following topics are discussed in this chapter:

- **Introduction to Event Persistence** — introduces persistent event concepts and terminology, and provides a simple example of code that uses the XEP API.
- **Creating and Connecting an EventPersister** — describes how to create an instance of the EventPersister class and use it to open, test, and close an eXTreme or TCP/IP database connection.
- **Importing a Schema** — describes the methods used to analyze a Java class and generate a schema for the corresponding Caché 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 EventQuery<> class that create and process query resultsets.
- **Calling Caché Methods from XEP** — describes EventPersister methods that can call Caché ObjectScript methods, functions, and procedures from an XEP application.
- **Schema Mapping and Customization** — provides a detailed description of how Java classes are mapped to Caché event schemas, and how to generate customized schemas for optimal performance.

### 2.1 Introduction to Event Persistence

A *persistent event* is a Caché database object that stores a persistent copy of the data fields in a Java object. By default, the eXTreme Event Persistence API stores each event as regular %Persistent object. Storage is automatically configured so that the data will be accessible to Caché by other means, such as objects, SQL, or direct global access.

Before a persistent event can be created and stored, XEP must analyze the corresponding Java class and *import a schema*, which defines how the data structure of a Java object is projected to a Caché persistent event. A schema can use either of the following two object projection models:
• The default model is the flat schema, where all referenced objects are serialized and stored as part of the imported class, and all fields inherited from superclasses are stored as if they were native fields of the imported class. This is the fastest and most efficient model, but does not preserve any information about the original Java class structure.

• If structural information must be preserved, the full schema model may be used. This preserves the full Java inheritance structure by creating a one-to-one relationship between Java source classes and Caché projected classes, but may impose a slight performance penalty.

See “Schema Import Models” for a detailed discussion of both models.

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<> classes provide the main features of the XEP API. They are used in the following sequence:

• The EventPersister class provides methods to establish and control a database connection (see “Creating and Connecting an EventPersister”).

• Once the connection has been established, other EventPersister methods can be used to import a schema (see “Importing a Schema”).

• The Event class provides methods to store, update, or delete events, create a query, and control index updating (see “Storing and Modifying Events”).

• The EventQuery<> 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 next section (see “Simple Applications to Store and Query Persistent Events”) describes two very short applications that demonstrate all of these features.

When importing a schema, XEP acquires basic information by analyzing the Java class. You can supply additional information that allows XEP to generate indexes and override the default rules for importing fields (see “Schema Customization”).

Fields of a persistent event can be primitives and their wrappers, temporal types, objects (projected as embedded/serial objects), enumerations, and types derived from java.util.List, java.util.Set and java.util.Map. These types can also be contained in arrays, nested collections, and collections of arrays. See “Schema Mapping Rules” for detailed information.

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 an eXTreme connection to a Caché database, 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 eXTreme connection accessing the same namespace as StoreEvents, creates an instance of EventQuery<> 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.

Both programs use instances of xep.samples.SingleStringSample, which is one of the classes defined in the XEP sample programs (see “XEP Samples” for details about the sample programs).

2.1.1.1 The StoreEvents Program

In StoreEvents, a new instance of EventPersister is created and connected to a specific namespace on the Caché server. 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 Caché database, and will be accessed by the QueryEvents program (described in the next section).

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

```java
import com.intersys.xep.*;
import xep.samples.SingleStringSample;

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

    public static void main(String[] args) {
        for (int i=0; i < eventData.length; i++) {
            eventData[i].name = "String event " + i;
        }
        try {
            System.out.println("Connecting and importing schema for " + className);
            EventPersister myPersister = PersisterFactory.createPersister();
            myPersister.connect("User","_SYSTEM","SYS");
            try { // delete any existing SingleStringSample events, then import new ones
                myPersister.deleteExtent(className);
                myPersister.importSchema(className);
            }
            catch (XEPException e) { System.out.println("import failed:
" + e); }
            Event newEvent = myPersister.getEvent(className);
            long[] itemIDs = newEvent.store(eventData);  // store array of events
            System.out.println("Stored " + itemIDs.length + " of " + eventData.length + " objects. Closing connection...");
            newEvent.close();
            myPersister.close();
        }
        catch (XEPException e) {System.out.println("Event storage failed:
" + e);}
    }
}
```

Before `StoreEvents.Main()` is called, the `xep.samples.SingleStringSample.generateSampleData()` method is called to generate sample data array `eventData` (see “XEP Samples” for information on sample classes).

In this example, XEP methods perform the following actions:

- `PersistFactory.createPersister()` creates `myPersister`, a new instance of EventPersister.
- `EventPersister.connect()` establishes an eXTreme in-process 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 `eventData` array as input, and creates a new persistent event for each object in the array. (Alternately, the code could have looped through the `eventData` 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. This is always necessary to release native code resources and prevent memory leaks.

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 an eXTreme 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<>` is created to iterate through the previously stored events, print their data, and delete them.
import com.intersys.xep.*;
import xep.samples.SingleStringSample;

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
            System.out.println("Connecting to query SingleStringSample events");
            myPersister = PersisterFactory.createPersister();
            myPersister.connect("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(sql);
                newEvent.close();  // assign value 12 to SQL parameter
                myQuery.setParameter(1,12);  // assign value 12 to SQL parameter
                myQuery.execute();
            } catch (XEPException e) {System.out.println("createQuery failed:
" + e);}
            // Iterate through the returned data set, printing and deleting each event
            SingleStringSample currentEvent;
            currentEvent = myQuery.getNext(null); // get first item
            System.out.println("Retrieved " + currentEvent.name);
            myQuery.deleteCurrent();
            currentEvent = myQuery.getNext(currentEvent); // get next item
            }
            myQuery.close();
            myPersister.close();
        } catch (XEPException e) {System.out.println("QueryEvents failed:
" + 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, a new 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<> for SingleStringSample events. The SQL statement defines a query that will retrieve all persistent SingleStringSample events with object IDs between 3 and a variable parameter value.

- EventQuery<.setParameter() assigns value 12 to the SQL parameter.

- EventQuery<.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<.getNext() is called with null as the argument, which specifies that the first item in the resultset is to be fetched and assigned to variable currentEvent.

- In the while loop:
  - The name field of currentEvent is printed
  - EventQuery<.deleteCurrent() deletes the most recently fetched event from the database.
  - EventQuery<.getNext() is called again with the most recently fetched event as the argument, specifying that the method should fetch the next event after that one.

If there are no more items, getNext() will return null and the loop will terminate.
• `EventQuery<> .close()` and `EventPersister .close()` are called for `myQuery` and `myPersister` after all events have been printed and deleted. This is always necessary to release native code resources and prevent memory leaks.

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 an eXTreme connection. See “Using Queries” for details about creating and using an instance of `EventQuery<>`.

## 2.2 Creating and Connecting an EventPersister

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

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

• `PersisterFactory .createPersister()` — returns a new instance of `EventPersister`.

• `EventPersister .close()` — closes this `EventPersister` instance and releases the native code resources associated with it.

The following methods are used to create a connection:

• `EventPersister .connect()` — takes `String` arguments for `namespace, username, password`, and establishes an eXtreme in-process connection to the specified Caché namespace.
  
  Optionally takes additional `host` and `port` arguments, and establishes a TCP/IP connection instead of an eXtreme connection.

It is important to understand that only one eXtreme Connection instance can exist in a process, and all `Connection` variables are references to that instance. When the XEP API and other eXtreme APIs are used in the same process, they will share the same underlying connection (see “Using the Globals API with Other eXtreme APIs”).

The following example establishes an in-process eXtreme connection:

```
Creating and Connecting an EventPersister: Creating an eXtreme connection

// Open an eXtreme connection
String namespace = "USER";
String username = "_SYSTEM";
String password = "SYS";
EventPersister myPersister = PersisterFactory.createPersister();
myPersister.connect(namespace,username,password);
// perform event processing here . . .
myPersister.close();
```

The `PersisterFactory .createPersister()` method creates a new instance of `EventPersister`. Only one instance is required in a process.

The `EventPersister .connect()` method establishes an in-process eXtreme connection. If no connection exists in the current process, a new eXtreme connection is created. If a connection already exists, the method returns a reference to the existing connection object.

When the application is ready to exit, the `EventPersister .close()` method must always be called to release resources used by the underlying native code.

**Important:** Always call `close()` to avoid memory leaks

It is important to always call `close()` on an instance of `EventPersister` before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.
Establishing a TCP/IP Connection

When an XEP application requires access to a database on another machine, it may be desirable to establish a standard TCP/IP connection rather than an in-process eXTreme connection. Although the TCP/IP connection is somewhat slower, the difference depends on the complexity of the events that are being stored. For extremely simple events with only one or two fields, the eXTreme connection is much faster. For more complex events, the difference is much less pronounced, and may be negligible for very complex events.

A TCP/IP connection is established when the call to `connect()` specifies the optional `host` and `port` arguments, as demonstrated in the following example:

`Creating and Connecting an EventPersister: Creating a TCP/IP connection`

```java
// Open a TCP/IP connection
String host = "127.0.0.1";
int port = 1972;
myPersister.connect(host, port, "User", "_SYSTEM", "SYS");
// perform event processing here . . .
myPersister.close();
```

The `EventPersister.connect()` method is called with `host` and `port` arguments, followed by the same `namespace`, `username`, and `password` arguments as in the previous example (hard-coded in this example). This establishes a TCP/IP connection to the specified port of the specified host machine.

Accessing the Underlying Connections

XEP is a layer over the Globals API, and uses the same underlying connections. The following methods return the underlying eXTreme and JDBC connections:

- `EventPersister.getConnection()` — returns the underlying eXTreme `intersys.globals.Connection` object. Throws an exception if the connection is TCP/IP rather than eXTreme.
- `EventPersister.get JDBCConnection()` — returns the underlying JDBC connection.

The `eXTreme Connection` object can be useful in XEP applications that also use the other APIs described in this book, since the same underlying connection is used by all of them. See “Creating a Connection” in the Globals API chapter for a more detailed description of eXTreme connections.

2.3 Importing a Schema

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

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

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

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

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

The following classes from package test are to be imported:

```java
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:

```java
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:
" + e);}
```

In this example, instances of dependent class test.Address will be serialized and embedded in the same Caché 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 Caché 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:

- **EventPersistor.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).
• Event.close() — closes the Event instance and releases the native code resources associated with it.

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.

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

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

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

The following example creates an instance of Event with SingleStringSample as the target class, and uses it to project an array of Java SingleStringSample objects as persistent events.

Note: The example assumes that an EventPersister instance named myPersister has already been created and connected to the database, and that a schema has been imported for the SingleStringSample class. See “XEP Samples” for information on SingleStringSample and the sample programs that define and use it.

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

```java
SingleStringSample[] eventData = SingleStringSample.generateSampleData(12);
try {
    Event newEvent = myPersister.getEvent("xep.samples.SingleStringSample");
    itemIdList = newEvent.store(eventData); // 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 "+ itemIdList.length + " events");
    newEvent.close();
} catch (XEPException e) {
    System.out.println("Event storage failed:
" + e);
}
```

The generateSampleData() method of SingleStringSample generates twelve SingleStringSample objects and stores them in an array named eventData.

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 eventData 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. A loop counts each ID greater than 0 in itemIdList, and the total is printed.

When the loop terminates, the Event.close() method is called to release resources used by the underlying native code.

**Important:** Always call close() to avoid memory leaks

It is important to always call close() on instances of Event before they go out of scope or are reused. Failing to close them can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.
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 (see “Creating and Storing Events” for the example that created `itemIdList`). The example arbitrarily updates the first six persistent events in the list and deletes the rest.

**Note:** The example 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

```java
// 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:
" + e);}
        System.out.println("Updated " + itemCount + " of " + itemIdList.length + " events");
    }
    newEvent.close();
} catch (XEPException e) {System.out.println("Event processing failed:
" + e);}
```

**Note:** See “XEP Samples” for information on the sample programs that define and use the SingleStringSample class.

### 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 Caché database:

- **deleteClass()** — takes a `className` string as an argument and deletes the specified Caché class.
- **deleteExtent()** — takes a `className` string as an argument and deletes all events in the extent of the specified class.
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

An instance of Event can execute a limited SQL query on the extent of its target class by creating an instance of EventQuery<>.

The EventQuery<> class provides methods to execute a predefined SQL query on the extent of the target class, and related methods that can access the query resultset to retrieve, update, and delete individual events from the database.

The following topics are discussed:

- Creating and Executing a Query — describes how use methods of the EventQuery<> class to execute queries.
- Processing Query Data — describes how to access and modify items in an EventQuery<> resultset.
- Using EventQueryIterator<> — describes alternate methods (similar to Java Iterator) for accessing and modifying items in a resultset.
- Defining the Fetch Level — describes how to control the amount of data returned by a query.

2.5.1 Creating and Executing a Query

The following methods create and destroy an instance of EventQuery<>:
• **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<>` instance and releases the native code resources associated with it.

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

• Queries must consist of a `SELECT` clause, a `FROM` clause, and (optionally) standard JDBC 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 Caché system-generated object ID may be referred to as `%ID`. Due to the leading `%`, this will not conflict with any field called `id` in a Java class.

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

• **setParameter()** — binds a parameter for the SQL query associated with this `EventQuery<>`. Takes int `index` and Object `value` as arguments, where `index` specifies the parameter to be set, and `value` is the value to bind to the specified parameter.

• **execute()** — executes the SQL query associated with this `EventQuery<>`. If the query is successful, this `EventQuery<>` will contain a resultset that can be accessed by the methods described later (see “Processing Query Data” and “Using `EventQueryIterator<>`”).

The following example executes a simple query on events in the `xep.samples.SingleStringSample` extent (see “XEP Samples” for information on the sample programs that define and use the `SingleStringSample` class).:

**Using Queries: Creating and executing a query**

```java
Event newEvent = myPersister.getEvent("xep.samples.SingleStringSample");
String sql = "SELECT * FROM xep_samples.SingleStringSample WHERE %ID BETWEEN ? AND ?";
EventQuery<SingleStringSample> 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 with 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<>` named `myQuery`, which will use the specified SQL statement to query the extent of target class. The SQL statement selects all events in the target class that have IDs between two parameter values.

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

When the `EventQuery<>execute()` method is called, the specified query is executed, and the resultset is stored in `myQuery`.

### 2.5.2 Processing Query Data

The following `EventQuery<>` methods access and modify items in a query resultset:

• **getNext()** — takes an object of the target class as an argument and returns the next item in the resultset (or returns the first item in the resultset if the argument is `null`). Returns `null` if there are no more items in the resultset.
• **getAll()** — returns the entire resultset as an array of objects.

• **deleteCurrent()** — deletes the event most recently fetched by **getNext()** from the database.

• **updateCurrent()** — takes an object of the target class as an argument and uses it to update the event most recently fetched by **getNext()**.

**Using Queries:**

```java
// Iterate through the returned data set, printing and deleting each event
SingleStringSample currentEvent = new SingleStringSample();
currentEvent = myQuery.getNext(currentEvent);
while (currentEvent != null) {
  System.out.println("Retrieved " + currentEvent.name);
  myQuery.deleteCurrent();
  currentEvent = myQuery.getNext(currentEvent);
}
myQuery.close();
```

**Important:**  **Always call close() to avoid memory leaks**

It is important to always call **close()** on instances of **EventQuery<>** before they go out of scope or are reused. Failing to close them can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

**2.5.3 Using EventQueryIterator<>**

The **EventQueryIterator<>** class is an alternative to the **EventQuery<>** class. It provides exactly the same functionality, but works in a manner that more closely resembles a standard Java **Iterator**.

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

The following **EventQuery<>** method creates an instance of **EventQueryIterator<E>** (where E is the same target class as the parent **EventQuery<E>**):

• **getIterator()** — returns an **EventQueryIterator<E>** that can be used to iterate over the results of queries on the extent of the target class.

The following **EventQueryIterator<>** methods access and modify events of the target class listed in the query resultset:

• **hasNext()** — returns **true** if the query resultset has more rows, or **false** otherwise.

• **next()** — returns the next item in the resultset.

• **remove()** — deletes the event most recently fetched by **next()** from the database.

• **set()** — takes an object of the target class as an argument and assigns the value to the event most recently fetched by **next()**.

**Using Queries: Iteration with EventQueryIterator<>**

```java
// Iterate through the returned data set, printing and deleting each event
EventQueryIterator<xep.samples.SingleStringSample> iterator = myQuery.getIterator();
SingleStringSample currentEvent = null;
while (iterator.hasNext()) {
  currentEvent = iterator.next();
  System.out.println("Retrieved " + currentEvent.name);
  iterator.remove();
}
```
2.5.4 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<> methods set and return the current fetch level:

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

The following fetch level values are supported:

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

2.6 Calling Caché Methods from XEP

The following EventPersister methods call Caché class methods:

- `callClassMethod()` — calls the specified Caché ObjectScript class method. Takes String arguments for `className` and `methodName`, plus 0 or more arguments that will be passed to the Caché method. Returns the Caché method return value as an Object that will be an instance of `String`, `int`, `long`, or `double`.
- `callVoidClassMethod()` — calls the specified Caché ObjectScript void class method. Takes String arguments for `className` and `methodName`, plus 0 or more arguments that will be passed to the Caché method.

The following EventPersister methods call Caché functions and procedures (see “User-defined Code” in Using Caché ObjectScript):

- `callFunction()` — calls the specified Caché ObjectScript function. Takes String arguments for `functionName` and `routineName`, plus 0 or more arguments that will be passed to the Caché method. Returns the Caché function return value as an Object that will be an instance of `String`, `int`, `long`, or `double`.
- `callProcedure()` — calls the specified Caché ObjectScript procedure. Takes String arguments for `procedureName` and `routineName`, plus 0 or more arguments that will be passed to the Caché method.
2.7 Schema Mapping and Customization

This section provides details about how a Java class is mapped to a Caché event schema, and how a schema can be customized for optimal performance. The following subjects are discussed:

- **Schema Import Models** — describes the two schema import models supported by XEP.
- **Schema Customization** — describes various options for customizing a schema by adding indexes or overriding the default rules for importing a field.
- **Schema Mapping Rules** — provides a detailed description of how Java classes are mapped to Caché event schemas.

2.7.1 Schema Import Models

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

- **The Embedded Object Projection Model (Flat Schema)** — imports a flat schema where all objects referenced by the imported class are serialized and embedded, and all fields declared in all ancestor classes are collected and projected as if they were declared in the imported class itself. All data for an instance of the class is stored as a single Caché %Library.Persistent object, and information about the original Java class structure is not preserved.

- **The Full Object Projection Model (Full Schema)** — imports a full schema where all objects referenced by the imported class are projected as separate Caché %Persistent objects. Inherited fields are projected as references to fields in the ancestor classes, which are also imported as Caché %Persistent classes. There is a one-to-one correspondence between Java source classes and Caché projected classes, so the Java class inheritance structure is preserved.

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

2.7.1.1 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 Caché event extends %Library.Persistent, and contains embedded serialized objects where the original Java object contained references to external objects.

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

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

Importing class C results in the following Caché class:

```caché
Class C Extends %Persistent ... { 
    Property a As %String; 
    Property b As %String; 
    Property c As %String; 
}
```
No corresponding Caché events will be generated for the A or B classes unless they are specifically imported. Event C on the Caché side does not extend either A or B. If imported, A and B would have the following structures:

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

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

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

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

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

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

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

**Note:** See the FlightLog sample program (listed in “XEP Samples”) for a demonstration of how to use the full object model.

### 2.7.2 Schema Customization

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 describe various options for generating customized schemas by adding indexes and overriding the default rules for importing fields:

- **Using Annotations** — XEP annotations can be added to a Java class to specify the indexes that should be created. They can also be added to optimize performance by specifying fields that should not be imported or fields that should be serialized.

- **Using IdKeys** — Annotations can be used to specify IdKeys (index values used in place of the default object ID), which are required when importing a full schema.

- **Implementing an InterfaceResolver** — By default, fields declared as interfaces are not imported into a flat schema. Implementations of the `InterfaceResolver` interface can be used to during schema import to specify the actual class of a field declared as an interface.

### 2.7.2.1 Using Annotations

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

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

• **Field Annotations** — are applied to a field in the class to be imported:
  
  –  **@Id** — specifies that the field will act as an IdKey.
  
  –  **@Serialized** — indicates that the field should be stored and retrieved in its serialized form.
  
  –  **@Transient** — indicates that the field should be excluded from import.

• **Class Annotations** — are applied to the entire class to be imported:
  
  –  **@Embedded** — indicates that a field of this class in a full schema should be embedded (as in a flat schema) rather than referenced.
  
  –  **@Index** — declares an index for the class.
  
  –  **@Indices** — declares multiple indexes for the same class.

**@Id (field level annotation)**

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

The following parameter must be specified:

• **generated** — a boolean specifying whether or not XEP should generate key values.

  –  **generated = true** — (the default setting) key value will be generated by Caché and the field annotated as **@Id** must be java.lang.Long. This field is expected to be null prior to insert/store and will be filled automatically by XEP upon completion of such an operation.

  –  **generated=false** — the user-assigned value of the annotated field will be used as the IdKey value. Fields can be String, int, Integer, long or Long.

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

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

**@Serialized (field level annotation)**

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

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

*Example:*
import com.intersys.xep.annotations.Serialized;
public class MyClass {
    @Serialized
    public xep.samples.Serialized serialized;
    @Serialized
    public int[][][] quadIntArray;
    @Serialized
    public String[][] doubleStringArray;
}

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

@Transient (field level annotation)

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

Example:

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

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

@Embedded (class level annotation)

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

Examples:

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

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

@index (class level annotation)

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

Arguments must be specified for the following parameters:

- name — a String containing the name of the composite index
- fields — an array of String containing the names of the fields that comprise the composite index
- type — the index type. The xep.annotations.IndexType enumeration includes the following possible types:
  - IndexType.none — default value, indicating that there are no indexes.
  - IndexType.bitmap — a bitmap index (see “Bitmap Indices” in Using Caché SQL).
- **IndexType.bitslice** — a bitslice index (see “Overview” in *Using Caché 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:*

```java
import com.intersys.xep.annotations.Index;
import com.intersys.xep.annotations.IndexType;

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

**@Indices (class level annotation)**

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

*Example:*

```java
import com.intersys.xep.annotations.Index;
import com.intersys.xep.annotations.IndexType;
import com.intersys.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;
}
```

### 2.7.2.2 Using IdKeys

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

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

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

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

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

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

This would result in the following global structure:

`^PersonD("John",12)=$LB("1976-11-11")`
XEP will also honor IdKeys added by other means, such as SQL commands (see “Using the Unique, PrimaryKey, and IdKey Keywords with Indices” in Using Caché 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:

- A class that declares an IdKey cannot be indexed asynchronously if it also declares other indexes.
- There is no limit of the number of fields in a composite IdKey, but the fields must be string, int, Integer, long or Long. Although double can also be used, it is not recommended.
- 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.
- 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 Caché Globals for information on IdKeys and the standard Caché storage model. See “Defining and Building Indices” in Using Caché SQL for information on IdKeys in SQL.

Sample programs IdKeyTest and FlightLog provide demonstrations of IdKey usage (see “XEP Samples” for details about the sample programs).

2.7.2.3 Implementing an InterfaceResolver

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

Note: InterfaceResolver is only relevant for the flat schema import model, which does not preserve the Java class inheritance structure. The full schema import model establishes a one-to-one relationship between Java and Caché 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**

```java
try {
    myPersister.setInterfaceResolver(new test.MyFirstInterfaceResolver());
    myPersister.importSchema("test.MyMainClass");
    myPersister.setInterfaceResolver(new test.MyOtherInterfaceResolver());
    myPersister.importSchema("test.MyOtherClass");
} 
catch (XEPException e) {System.out.println("Import failed:\n" + e);}
```
The first call to `setInterfaceResolver()` sets a new instance of `MyFirstInterfaceResolver` (described in the next example) as the implementation to be used during calls to the import methods. This implementation will be used in all calls to `importSchema()` until `setInterfaceResolver()` is called again to specify a different implementation.

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

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

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

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

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

**Schema Customization: Implementing an InterfaceResolver**

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

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

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

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

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

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

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

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

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

2.7.3.1 Requirements for Imported Classes

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

- If the imported Caché class will be used to execute queries and access stored events, the Java source class must include an argumentless public constructor.
- The Java source class cannot contain fields declared as java.lang.Object, or arrays, lists, sets or maps that use java.lang.Object as part of its declaration. An exception will be thrown if the XEP engine encounters such fields.

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

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

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

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

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

See “Type Mapping” for more details on how various datatypes are projected.

2.7.3.2 Naming Conventions

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

- `$` (dollar sign) is projected as a single "d" character on the Caché side.
- `_` (underscore) is projected as a single "u" character on the Caché 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.SingleStringA5BFD`.

### 2.7.3.3 Type Mapping

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

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

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

#### Primitives and Primitive Wrappers

The following Java primitives and wrappers are mapped as Caché %String:

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

The following Java primitives and wrappers are mapped as Caché %Integer:

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

The following Java primitives and wrappers are mapped as Caché %Float:

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

#### Temporal Types

The following Java temporal types are mapped as Caché %String

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

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

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

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

Serialized

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

Arrays

The following rules apply to arrays:

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

Enumerations

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

Collections

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

In some situations, rapid access to object properties may be more important than the convenience of using the standard Caché Java binding. The Caché eXTreme Dynamic Object (XDO) classes provide an highly efficient mechanism for inserting, updating, and deleting persistent database objects. The most significant differences between the standard binding and XDO objects (package com.intersys.xdo) are as follows:

- The class to be accessed by a dynamic object is defined at runtime. No compiled proxy classes are required.
- Rather than using TCP/IP to communicate with Caché, an eXTreme connection uses intraprocess calls to the Caché kernel and Object Server.
- Dynamic objects contain actual data, not just pointers to data on the server. Currently supported property datatypes are int, Integer, long, Long, double, Double, BigDecimal, String, Date, Time, and Timestamp.
- Dynamic objects contain only property values, and do not provide access to method calls.

There are also some extra installation and configuration requirements (see “Installation and Configuration”).

The com.intersys.xdo API contains two main classes:

- DatabaseConnection — represents a physical and logical connection to a Cache database. It provides methods to create instances of DynamicObject, update indexes, and control transactions.
- DynamicObject — represents a persistent object in a Caché class extent specified at runtime. The dynamic object has fields corresponding to the properties of the specified class, and provides methods to set or change the values of those fields. It accesses the extent of the class through a DatabaseConnection, and provides methods to create, change, or delete objects in the extent.

This chapter describes how these classes are used. The following topics are discussed:

- Opening and Closing an XDO Connection — describes how to create an instance of DatabaseConnection and establish an eXTreme connection.
- Using Dynamic Objects — describes how to create an instance of DynamicObject and use it to create, change, or delete objects in the database.
- Using Transactions — describes how to use the DatabaseConnection transaction and indexing methods.

Note: This chapter includes summaries of the methods discussed, but does not provide all available information on method overloads and signatures. For more information, click on the method name links or see the listings in the “XDO Quick Reference” section.
3.1 Opening and Closing an XDO Connection

Like all eXTreme connection classes, an XDO DatabaseConnection depends on an underlying JDBC connection. Unlike the others, it does not create the JDBC connection automatically. The following two connection objects must be created:

- The JDBC Connection object provides the underlying eXTreme connection. The connection URL must include the special reserved hostname SPCHOST.
- The DatabaseConnection object provides the interface between the JDBC eXTreme connection and instances of DynamicObject (described later in “Using Dynamic Objects”).

**Note:** It is important to understand that only one eXTreme JDBC Connection instance can exist in a process, and all other Connection and DatabaseConnection variables in the process are references to that instance. When the XDO API and other eXTreme APIs are used in the same process, they will share the same underlying connection (see “Using the Globals API with Other eXTreme APIs”).

A process can have only one eXTreme JDBC connection open at a time. It is established using the standard JDBC connection interface (see “Establishing JDBC Connections” in Using Caché with JDBC), but it uses the special reserved hostname SPCHOST in place of localhost or an IP address. This allows XDO transactions to be part of the same context as SQL queries and other transactions. The connection is otherwise functionally identical to a regular JDBC connection using a local IP address (localhost or 127.0.0.1), and can be used for all the same purposes.

The following methods are used to create or terminate an XDO connection:

- DatabaseConnectionFactory.createJNIDatabaseConnection() — returns an instance of DatabaseConnection.
- DatabaseConnection.connect() — takes namespace, username, and password arguments, and connects to the database in the specified Caché namespace. The namespace must be the same as the one specified in the JDBC connection URL.
- DatabaseConnection.disconnect() — disconnects from the Caché database.

The following code creates a JDBC connection to the Samples namespace using the SPCHOST identifier, then creates a DatabaseConnection object that uses the eXTreme JDBC connection:

```java
java.sql.Connection jdbcConnection = null;
DatabaseConnection xdoConnection = null;
String namespc = "Samples";
String user = "_SYSTEM";
String password = "SYS";
try {
    Class.forName ("com.intersys.jdbc.CacheDriver");
    jdbcConnection = java.sql.DriverManager.getConnection(url, user, password);
    xdoConnection = DatabaseConnectionFactory.createJNIDatabaseConnection();
    xdoConnection.connect(namespc, user, password);
    // insert XDO application code here
} catch (XDOException e) { System.out.println(e.getMessage()); }
catch (Exception e) { System.out.println(e.getMessage()); }
finally {
    try {
        if (xdConnection!=null) xdoConnection.disconnect();
    if (jdbcConnection!=null) jdbcConnection.close();
    }
    catch (XDOException e) { System.out.println(e.getMessage()); }
    catch (Exception e) { System.out.println(e.getMessage()); }
}
```
A call to the `java.sql.getConnection()` method creates the initial eXTreme JDBC connection by specifying `SPCHOST` in the URL (SPC stands for “Single Process Communication.” For a general explanation of how a URL string is structured, see “Defining a JDBC Connection URL” in Using Caché with JDBC).

The call to `createJNIDatabaseConnection()` creates an XDO DatabaseConnection object.

The call to the `connect()` method must specify the same namespace as the one in the URL used by JDBC Connection object. The `connect()` method doesn't have to specify the entire URL because it uses the previously established JDBC connection (identified by the namespace and hostname `SPCHOST`).

After the connection has been opened, the database can be accessed (see the examples in “Inserting a New Database Object” and “Modifying an Existing Database Object” for code that could be inserted here).

In the `finally` clause, the DatabaseConnection object is terminated by calling its `disconnect()` method. This does not affect the JDBC Connection object, which could be used independently, just like a standard JDBC `localhost` connection. In this example the JDBC connection is terminated by calling its `close()` method.

---

### 3.2 Using Dynamic Objects

This section discusses the DynamicObject class, which provides the methods that allow a Java application to access persistent objects in the Caché database. The following topics are discussed:

- **Creating Dynamic Objects and Database Objects** — discusses how a DynamicObject is created, and how it interacts with an associated persistent database object.

- **Inserting a New Database Object** — describes how to create an empty DynamicObject for a specified class, set the values of its properties, and create a new database object with those properties.

- **Modifying an Existing Database Object** — describes how to create an instance of DynamicObject containing the property values of an existing database object, read the values of DynamicObject properties, and update the database object.

- **Using Property Numbers** — describes how to specify DynamicObject properties by number rather than name.

#### 3.2.1 Creating Dynamic Objects and Database Objects

An instance of DynamicObject is used to access persistent database objects in a specified class extent (see “Classes and Extents” in the Caché Programming Orientation Guide for basic information on extents). The dynamic object contains fields corresponding to the properties of the specified class, and provides methods to read and write those fields. It can insert, update, and delete objects in the extent.

**Attached and Detached Target Objects**

When an instance of DynamicObject is created, it is always associated with a specific class extent. It can also be associated with a specific target object in the extent. In this case, the dynamic object is said to be attached to the target, and can update or delete it. Otherwise, the dynamic object is detached, and can create and attach a new target object in the extent.

The DatabaseConnection interface provides the `createNew()` method to create a detached instance of DynamicObject, and the `openId()` method to create an attached instance:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createNew()</code></td>
<td>Creates a detached dynamic object, which can create and attach a new target object in the extent of the class. See “Inserting a New Database Object” for a details and examples.</td>
</tr>
<tr>
<td><code>openId()</code></td>
<td>Creates a dynamic object attached to a target object in the specified class extent. The dynamic object contains the same property values as the target object, and can update or delete the target. See “Modifying an Existing Database Object” for a details and examples.</td>
</tr>
</tbody>
</table>
There are four DynamicObject methods that can make changes in the Caché database. Their behavior depends on whether the dynamic object is attached or detached:

<table>
<thead>
<tr>
<th>Method</th>
<th>if attached</th>
<th>if detached</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert()</td>
<td>detaches from current target, then creates and attaches a new target object.</td>
<td>creates and attaches a new target object.</td>
</tr>
<tr>
<td>update()</td>
<td>updates the data in the target object.</td>
<td>throws exception</td>
</tr>
<tr>
<td>save()</td>
<td>updates the target exactly like update()</td>
<td>creates and attaches a new target like insert()</td>
</tr>
<tr>
<td>delete()</td>
<td>deletes the target object from the database.</td>
<td>throws exception</td>
</tr>
</tbody>
</table>

All four of these methods can take optional deferIndices and timeout arguments, which are used to optimize transaction processing.

**Note: Unique Key Constraints**

Both insert() and save() can insert a new database object, but they behave differently if the insert fails because of a unique key constraint. If insert() fails, it throws an exception. If save() fails, it attaches and updates the database object that caused the insert to fail.

### 3.2.2 Inserting a New Database Object

This section describes how to use the DatabaseConnection.createNew() method to create an empty instance of DynamicObject, how to set the property values of the dynamic object, and how to create a new database object containing those values.

- DatabaseConnection.createNew() — takes a className string and returns a DynamicObject with empty property fields corresponding to the ones in class className. The new dynamic object is detached, and can create and attach a new database object in the extent of the class.

The following DynamicObject methods define the property values of an instance, save values to a target object in the database, and perform garbage collection on the instance when necessary:

- cleanup() — destroys the DynamicObject and releases its underlying JNI native resources to prevent memory leaks. This method must always be called before an instance of DynamicObject is reused or goes out of scope.
- set() — takes property and value arguments, and sets the specified property of the dynamic object to the specified value. The datatype of value must be int, Integer, long, Long, double, Double, BigDecimal, String, Date, Time, or Timestamp. In this section, the examples specify the property argument as a name string, but it can also be specified as a number (see “Using Property Numbers”).
- insert() — creates and attaches a new database object that contains a copy of the dynamic object’s property values. If the DynamicObject is already attached, it will be detached before the new target object is created and attached.
- update() — copies the data in DynamicObject to the target object, or throws an exception if not attached.

The following code creates a new CosTutorial.Person dynamic object, sets the values of its properties, and saves it to the database. It then changes the value of one property and updates the target object. Finally, it uses the same dynamic object to create and attach a different database object.

**Note:** This example does no error checking. Assume that the CosTutorial.Person extent is empty, and that this code fragment is running inside a TRY/CATCH block (see the example in “Opening and Closing an XDO Connection”). See “Sample Data Classes” for instructions on how to view the CosTutorial.Person extent in the Management Portal, and how to delete or populate the extent using the Caché Terminal.
Inserting a New Database Object

// xdoConnection is a previously created DatabaseConnection object
DynamicObject objPerson = xdoConnection.createNew("CosTutorial.Person");
objPerson.set("name", "Jones, Oscar"); // required unique index field
objPerson.set("dob", java.sql.Date.valueOf("1929-06-22"));
objPerson.insert(); // create and attach a new database object

// We entered the wrong dob! Update the target object with the correct dob
objPerson.set("dob", java.sql.Date.valueOf("1959-08-31");
objPerson.update(); // change the dob property in the database object

// Now give "Jones, Oscar" a second phone number
objPerson.set("phone", "123-456-9999"); // required unique index field
objPerson.insert(); // create a second new database object
objPerson.cleanup(); // destroy the DynamicObject when done.

The `createNew()` method creates DynamicObject `objPerson`, which represents a database object of class CosTutorial.Person. All of the fields in `objPerson` are null.

The first three calls to `set()` assign values to the name, phone, and dob properties of `objPerson`. The standard `java.sql.Date.valueOf()` method is used to convert a date string to a Date.

The `insert()` method creates a new persistent object in the CosTutorial.Person extent and fills it with the property values defined in `objPerson`. The dynamic object is now attached to the newly created target object.

The next call to `set()` changes the value of the dob property in `objPerson`, and the call to `update()` saves the new value to the target object.

The last call to `set()` changes the value of the phone property in `objPerson`. The call to `insert()` detaches the current target object, then creates and attaches a new one. The new target object contains a new phone value but has the same name and dob values as before. (If only the name or dob values of `objPerson` had been changed, `insert()` would have thrown an exception because the CosTutorial.Person extent contains a unique index on the phone property).

When `objPerson` is no longer needed, the `cleanup()` method is called to destroy it and release its underlying JNI native resources (which must be done to prevent memory leaks).

**Important:** Always call `cleanup()` to avoid memory leaks

It is important to call `cleanup()` on all instances of DynamicObject before they go out of scope or are reused. When a dynamic object is created, the underlying JNI native code allocates resources that are not managed by Java garbage collection. Memory leaks will be created if `cleanup()` is not called to release these resources.

3.2.3 Modifying an Existing Database Object

This section describes how to use the DatabaseConnection.openId() method to create an instance of DynamicObject containing the same property values as a specified target database object, and how to use the dynamic object to update or delete the target.

- DatabaseConnection.openId() — takes `className` and `Id` arguments that identify a target object in extent `className`, and returns a DynamicObject that contains the same property values as the target. It is attached to the target object, and can update or delete it.

The following DynamicObject methods test whether a call to `openId()` was successful, return the Id and property values of the dynamic object, and delete the target object:

- `isNull()` — tests whether the object reference returned by `openId()` is null, which will be the case if no object with the specified Id value was found.
- `getId()` — returns the Id value of the target object if attached, or an empty string ("") if detached.
• **delete()** — deletes the target object from the database. Throws an exception if no target is attached.

• **Property Getter methods** — take a property argument and return the value of the specified property. The individual method names indicate the datatype of the variable that will be returned. Property getter methods exist for the following datatypes:

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>getInt()</td>
</tr>
<tr>
<td>integer</td>
<td>getIntegerWrapper()</td>
</tr>
<tr>
<td>long</td>
<td>getLong()</td>
</tr>
<tr>
<td>long</td>
<td>getLongWrapper()</td>
</tr>
<tr>
<td>double</td>
<td>getDouble()</td>
</tr>
<tr>
<td>double</td>
<td>getDoubleWrapper()</td>
</tr>
<tr>
<td>String</td>
<td>getString()</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>getBigDecimal()</td>
</tr>
<tr>
<td>Date</td>
<td>getDate()</td>
</tr>
<tr>
<td>Time</td>
<td>getTime()</td>
</tr>
<tr>
<td>Timestamp</td>
<td>getTimestamp()</td>
</tr>
<tr>
<td>Long</td>
<td>getLongWrapper()</td>
</tr>
</tbody>
</table>

In this section, the examples specify the property argument as a name string, but it can also be specified as a number (see “**Using Property Numbers**”).

The following code opens target objects with Id values 1 and 2 in the CosTutorial.Person extent, prints their property values, and then deletes them from the database.

**Note:** This example assumes that the CosTutorial.Person extent contains the two objects created by the example in the previous section (see “**Inserting a New Database Object**”), and that those objects have Id values 1 and 2. In this example, the Id values are provided in a hard-coded list. See “**Transaction Processing**” for more realistic examples that read Id values from a resultset. Assume that this code fragment is inside a TRY/CATCH block (see the example in “**Opening and Closing an XDO Connection**”).

### Modifying an Existing Database Object

```java
dynamicObject objPerson = null;
String[] idList = {"1","2"};  // create a list of Ids to open
for (String newId:idList) {
    objPerson = xdoConnection.openId("CosTutorial.Person", newId);
    if (objPerson.isNull()) {
        System.out.println("Invalid Id:" + newId);
    } else {
        System.out.println("Person " + objPerson.getId() + ":" + " name: " + objPerson.getString("name") + " dob: " + objPerson.getDate("dob"));
        objPerson.delete();
    }
}
```

The **openId()** method returns a DynamicObject that contains a copy of the data in the existing CosTutorial.Person database object with Id value **newId**.

The **isNull()** method tests to make sure that an object with the specified Id value was found. If **isNull()** returns false, this instance of DynamicObject cannot be used, and an error message is printed.

The **getId()** method is used to print the Id of the retrieved database object, and property getter methods **getString()** and **getDate()** are used to print the phone, name, and dob properties.

The **delete()** method is called to delete the current target from the CosTutorial.Person extent after its properties have been printed.

After each loop iteration, the **cleanup()** method must be called to destroy the DynamicObject and release its underlying JNI native resources. A memory leak will be created if a new instance is assigned to the **objPerson** variable before the old one is destroyed, or if the variable goes out of scope before the last instance is destroyed.
3.2.4 Using Property Numbers

The set() method and the property getter methods (discussed in ‘‘Modifying an Existing Database Object’’) can accept either a name string or a number as the property argument. Previous examples in this section have used name strings, but when the same properties are referenced repeatedly, it is more efficient to specify property numbers. The following method of DynamicObject returns a property number for a given property name:

- **getPropertyNumber()** — accepts a propertyName string as an argument, and returns a property number (an arbitrary identification number assigned by the system) that corresponds to the specified name.

The following code creates a new DynamicObject, defines a set of property number variables, and uses the variables in both getter and setter methods. (For simplicity, this example only defines one object. See “Transaction Processing” for examples that process multiple objects.)

**Using Property Numbers**

```java
DynamicObject objPerson = null;
int propName = 0;
int propPhone = 0;
int propDOB = 0;
try {
    // Initialize dynamic object and property number variables
    objPerson = xdoConnection.createNew("CosTutorial.Person");
    propName = objPerson.getPropertyNumber("Name");
    propPhone = objPerson.getPropertyNumber("Phone");
    propDOB = objPerson.getPropertyNumber("DOB");

    // Set the object properties and print them
    objPerson.set(propName, "Fudd, Elmer");
    objPerson.set(propPhone, "123-456-7890");
    objPerson.set(propDOB, java.sql.Date.valueOf("1940-01-01"));

    System.out.println("Current name: " + objPerson.getString(propName) + ", phone: " + objPerson.getString(propPhone) + ", DOB: " + objPerson.getDate(propDOB));
} catch (XDOException e) {System.out.println("XDO error: 
" + e.getMessage()); }
finally { objPerson.cleanup(); }
```

A new DynamicObject is created, and calls to `getPropertyNumber()` define property number variables for each of the three properties in the class. The variables only have to be defined once in the process, and can be used with any instance of DynamicObject that accesses `CosTutorial.Person`.

The calls to set() use the property numbers to set the values of each property.

In the print statement, property getter methods `getString()` and `getDate()` use property numbers to print the newly defined phone, name, and dob properties.

3.3 Transaction Processing

This section discusses the methods most useful for performing and optimizing high speed transaction processing. The following topics are discussed:

- **Using Transactions** — describes how to start, commit, and roll back transactions.

3.3.1 Using Transactions

The DatabaseConnection interface provides several methods to control transaction processing. This section describes how to use the following methods:
• `startTransaction()` — starts a transaction (which may be nested within another transaction).
• `commit()` — commits one level of transaction.
• `rollback()` — rolls back all levels of transaction.
• `transactionLevel()` — returns the current transaction level (0 if not in a transaction).

The following example processes a series of transactions that attempt to insert new `CosTutorial.Person` objects into the database. In each transaction, the dynamic object gets values for `Name` and `DOB` from one source, gets the value of `Phone` from a second source, and attempts to insert a new database object containing these values. The example is designed to recover from a unique key exception, which will be thrown when a transaction attempts to insert an object containing a `Phone` property with the value "123-456-2222".

**Note:** Assume that a query has already returned `java.sql.ResultSet myResults` (see the query example in “Sample Data Classes”), and that property number variables `propName`, `propDOB`, and `propPhone` have already been defined (as previously demonstrated in “Using Property Numbers”).

**Using Transactions: startTransaction(), commit(), rollback() and transactionLevel()**

```java
// Create list of phone numbers to be used during transaction processing
int phoneCount = 0;
String[] phoneList = {"123-456-1111","123-456-2222","123-456-3333",
"123-456-4444","123-456-5555");

// Add an object that will force a unique key exception during processing
try {
    objPerson.set(propPhone,"123-456-2222");
    objPerson.insert();
} catch (XDOException e) { System.out.println("Error initializing: 
" + e.getMessage()); }

// Start transaction processing
try {
    while (myResults.next() && phoneCount<phoneList.length) {
        boolean phoneAdded = false;
        try {
            objPerson.set(propName,myResults.getString("Name"));
            objPerson.set(propDOB,myResults.getDate("DOB");
            do {
                // Assign the next phone number and try to insert.
                objPerson.set(propPhone,phoneList[phoneCount++]);
                try {
                    xdoConnection.startTransaction();
                    objPerson.insert();
                    if (xdoConnection.transactionLevel()>0) {
                        xdoConnection.commit();
                        System.out.println("Committed " + objPerson.getString(propName) +",
" phone: " + objPerson.getString(propPhone) +",
" DOB: " + objPerson.getDate(propDOB) );
                        phoneAdded = true;
                    }
                }
            } catch (XDOException e) { System.out.println("Error assigning to "+ objPerson.getString(propName) +",
" phone " + objPerson.getString(propPhone) +",
" already in use.");
                if (!phoneAdded && phoneCount<phoneList.length) {
                    xdoConnection.rollback();
                    System.out.println("Error assigning to " + objPerson.getString(propName) +",
" phone " + objPerson.getString(propPhone) +",
" already in use.");
                }
            } catch (java.sql.SQLException e) {System.out.println("SQL error: " + e.getMessage());}
        } catch (Exception e) {System.out.println("Unexpected error: " + e.getMessage());}
        finally { objPerson.cleanup(); }
    }
}
```

Before transaction processing is started, an object with `Phone` value "123-456-2222" is inserted into the database. The second item in the `phoneList` array contains the same value, so a unique key exception will be thrown when a transaction attempts to insert it.
The while loop processes transactions until it reaches the end of either `phoneList` or `myResults`. For each row in `ResultSet myResults`, Name and DOB values are retrieved, and `set()` is called to assign them to the corresponding fields in dynamic object `objPerson`.

The following process retrieves a phone number from `phoneList` and assigns it to the Phone property of `objPerson`, starts a transaction, and attempts to create a new object in the database. If the transaction fails, the process repeats with new values of Phone until a transaction is successful or there are no more items in `phoneList`.

- A phone number is retrieved from `phoneList` and assigned to the Phone property in `objPerson`.
- `startTransaction()` is called.
- `insert()` is called to save `objPerson` to the database. If an exception is encountered, control passes to the catch block, where `rollback()` is called and an error message is printed.
- The call to `transactionLevel()` will return 0 if the transaction was rolled back. If this happens, a new transaction will be attempted using the same Name and DOB properties but a different Phone property.
- If the insert is successful, `commit()` is called, the current properties are printed, and processing continues with the next row (if any) in `myResults`.

The code in this example would generate output similar to the following:

```
Error assigning to Jafari, Elizabeth M. - phone 123-456-2222 already in use.
Committed Zevon, Maria V., phone: 123-456-5555, DOB: 1975-09-10
```
Caché multidimensional storage provides a very fast, flexible storage model that can be used to implement a wide variety of data structures. Caché uses multidimensional storage to structure data as objects or SQL tables, but many other structures are equally possible. The Globals API allows direct access to global arrays (the basis of the multidimensional storage model), allowing you to implement your own data structures.

The Globals API is optimized for extremely rapid storage and retrieval of a few elementary datatypes — int, long, double, byte[], String, and ValueList (a serialized list). It can also be used in the same application with the eXTreme Dynamic Object (XDO) API, which provides rapid access to data stored in Caché objects.

The Globals API runs in the same process as the Caché instance, rather than communicating via TCP/IP like the standard Java binding. The Caché server and the Java application must be on the same machine, but the application can still access data on remote machines if necessary.

For a quick overview of the multidimensional storage model, see “Introduction to Global Arrays” later in this chapter. See Using Caché Globals for a detailed discussion of their structure and usage in ObjectScript applications. This chapter covers the following topics:

- **Introduction to the Globals API** — introduces global array concepts and terminology, and provides a simple example of code that uses the Globals API.

- **Creating a Connection** — describes methods of the Connection class that open, test, and close the eXTreme connection.

- **Building a Global Array** — describes methods of the NodeReference class that create, change, or delete nodes in a multidimensional global array.

- **Iterating Over a Global Array** — describes the iteration methods that allow rapid access to the nodes of a global array.

- **Retrieving Global Array Data** — describes the methods that retrieve values from global arrays.

- **Using Serialized Value Lists** — describes using methods of the ValueList class to create and access serialized lists, providing a practical way to store multiple values of all supported datatypes in a single node.

- **Accessing Multiple Global Arrays** — describes using the GlobalsDirectory class and related methods to list and access all global arrays in all available namespaces.

- **Transactions and Locking** — describes how to use the Globals API in transactions.

- **Calling Caché Methods** — describes Connection methods that can call Caché ObjectScript methods, functions, and procedures from a Globals API application.

- **Globals API Requirements and Conventions** — describes naming conventions and interactions between eXTreme APIs.
Note: **Globals Terminology**

In other parts of the Caché documentation, global arrays are frequently referred to as simply *globals*. This book uses the longer term *global array* to avoid any confusion with the Globals API name.

## 4.1 Introduction to the Globals API

This section introduces the basic concepts behind the Globals API. The following topics are discussed:

- **Introduction to Global Arrays** — introduces global array concepts and terminology.
- **Simple Applications to Create and Fetch Nodes** — provides a simple demonstration of how the Globals API is used.

### 4.1.1 Introduction to Global Arrays

A Caché global array, like all sparse arrays, is a tree structure rather than a sequentially numbered list. The basic concept behind sparse arrays can be illustrated by analogy to the Java package naming convention. Each class in a package is uniquely identified by a namespace made up of a series of identifiers. For example, consider a package containing only two classes, `main.foo.SubFoo` and `main.bar.UnderBar`. The five identifiers that make up these namespaces could be viewed as elements of a sparse array:

```plaintext
main -->|--> foo --> SubFoo
|--> bar --> UnderBar
```

Classes could be created that use the other possible namespaces (for example, `main` or `main.bar`), but no resources are wasted if those classes do not exist.

Like Java classes, the elements of a Caché global array are uniquely identified by a namespace consisting of an arbitrary number of identifiers. All namespaces in the global array structure are referred to as *nodes*. A node must either contain data, have child nodes, or both. A node that has child nodes but does not contain data is called a *valueless node*.

The root identifier (like `main` in the Java package) is referred to as the *global name*. All other identifiers in the namespace are called *subscripts*. The complete namespace of the node (global name plus subscripts) is the *node address*.

In Caché notation, a node address is symbolized by a circumflex (`^`) followed by the global name and a comma-delimited *subscript list* in parentheses. For example, given a global array using the same identifiers as the Java package example, the nodes in the array would be represented as follows:

- **(valueless root node)** `^main`
- **(valueless node)** `^main("foo")`
- **(node with value)** `^main("foo","SubFoo") = <value>`
- **(valueless node)** `^main("bar")`
- **(node with value)** `^main("bar","UnderBar") = <value>`

The *root node* of the array is simply `^main`, with no subscript list. The two nodes containing data are `^main("foo","SubFoo")` and `^main("bar","UnderBar")`. Nodes `^main`, `^main("foo")` and `^main("bar")` are all valueless.

All descendants of a given node are referred to as *subnodes* of that node. For example, `^main("foo","SubFoo")` is a subnode of `^main("foo")`, and all four subscripted nodes are subnodes of `^main`.

The term *node level* refers to the number of subscripts in the subscript list. For example, `^main("bar","UnderBar")` is a level 2 node. The address has a level 1 subscript (`"bar"`), and a level 2 subscript (`"UnderBar"`).

In this example, all of the subscripts are strings, but numeric values can also be used. The Globals API allows subscripts to be specified using Java datatypes `String`, `int`, `long`, or `double`, all of which can be used in the same subscript list. For example, `^myNode(1,"two",3.14)` is a valid node address.
4.1.2 Simple Applications to Create and Fetch Nodes

This section describes two very simple applications that use the Globals API to create and access nodes in a global array:

- **The CreateNodes program** — opens an eXTreme connection to a Caché database, creates a persistent global array with two nodes, then closes the connection and terminates.

- **The FetchNodes program** — opens a new eXTreme connection, accesses the database to fetch the values of both nodes created by CreateNodes, then deletes the global array, closes the connection, and terminates.

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

4.1.2.1 The CreateNodes Program

In CreateNodes, a new Globals connection object is created and connected to the User namespace on the Caché server. A new global array with two nodes is added to the Caché database, the connection is closed, and the program terminates.

**The CreateNodes Program: Opening a Connection and Creating Nodes**

```java
import com.intersys.globals.*;

public class CreateNodes {
    public static void main(String[] args) {
        Connection myConn = ConnectionContext.getConnection();
        try {
            myConn.connect("User", "_SYSTEM", "SYS");
            NodeReference nodeRef = myConn.createNodeReference("myGlobal");
            // Create a new global array with two nodes
            nodeRef.set("Hello world"); // create root node ^myGlobal
            nodeRef.appendSubscript("sub1"); // point to node address ^myGlobal("sub1")
            nodeRef.set("This is a subscripted node"); // create subnode ^myGlobal("sub1")
            nodeRef.close();
            myConn.close();
        } catch (GlobalsException e) { System.out.println(e.getMessage()); }
    } // end Main()
} // end class CreateNodes
```

In this example, Globals API methods perform the following actions:

- **ConnectionContext.getConnection()** creates a new Connection object named myConn for this process.

- **Connection.connect()** opens a connection that accesses the database associated with the User namespace.

- **Connection.createNodeReference()** creates NodeReference object nodeRef, which contains a reference to the root node of a global array that will be named ^myGlobal.

- **NodeReference.set()** creates a new persistent node in the database. Global array ^myGlobal does not exist in the database until set() assigns a value at the address specified in the nodeRef object.

- **NodeReference.appendSubscript()** adds a subscript to the node reference, which now points to ^myGlobal("sub1"), and set() is called again to create a persistent node at that address.

- **NodeReference.close()** must be called because the memory allocated for nodeRef by native code cannot be released by Java garbage collection.

- **Connection.close()** terminates the connection and releases native code resources. Both nodes of new global array ^myGlobal will persist in the Caché database after the connection is closed.
Important: Always call close() to avoid memory leaks

It is important to always call close() on instances of Connection and NodeReference before they go out of scope or are reused. Failing to close them can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

All of these methods are discussed in detail later in this chapter. See “Creating a Connection” for information on opening, testing, and closing an eXtreme connection. See “Building a Global Array” for details about using a NodeReference object to create a node or change its value. See “Setting the Target Address” for information on specifying the subscripts of a node reference.

4.1.2.2 The FetchNodes Program

This example assumes that FetchNodes runs immediately after the CreateNodes process terminates. FetchNodes creates a new eXtreme connection that accesses the same namespace as CreateNodes. A new node reference object is used to fetch the values of root node ^myGlobal and subnode ^myGlobal("sub1") from the database, demonstrating that the global array has persisted between processes. The global array is then deleted, the connection is closed, and the program terminates.

The FetchNodes Program: Fetching the Values of Existing Nodes

```java
import com.intersys.globals.*;

class FetchNodes {
   public static void main(String[] args) {
      Connection myConn = ConnectionContext.getConnection();
      try {
         myConn.connect("User", "_SYSTEM", "SYS");
         NodeReference nodeRef = myConn.createNodeReference("myGlobal");
         // Read both existing nodes
         System.out.println("Value of ^myGlobal is " + nodeRef.getString());
         nodeRef.appendSubscript("sub1");   // point to subnode
         System.out.println("Value of ^myGlobal("sub1") is " + nodeRef.getString());
         nodeRef.setSubscriptCount(0);   // point to root node
         nodeRef.kill();   // delete entire array
         nodeRef.close();
         myConn.close();
      }
      catch (GlobalsException e) { System.out.println(e.getMessage()); }
   }
} // end class FetchNodes
```

In this example, Globals API methods perform the following actions:

- The getConnection(), connect(), and createNodeReference() methods are called again, exactly as they were called in CreateNodes. A new eXtreme connection object is created for this process, and a new connection to the User namespace is established. New node reference object nodeRef is created, and initially points to the root node of ^myGlobal (the global array previously created by CreateNodes).

- NodeReference.getString() fetches the value of root node ^myGlobal from the database and returns it as a Java String.

- As in CreateNodes, the appendSubscript() method is used to add subscript "sub1" to the referenced node address. The NodeReference object now points to subnode ^myGlobal("sub1"), and getString() is called again to return the value of that subnode.

- NodeReference.setSubscriptCount() is a method that changes the referenced address by reducing the number of subscripts in the subscript list. In this case, it specifies that the number of subscripts should be zero. Since all subscripts have been deleted from the referenced address, the NodeReference object once again points to root node ^myGlobal.

- NodeReference.kill() is a method that deletes the referenced node and all of its subnodes from the database. In this case, root node ^myGlobal and subnode ^myGlobal("sub1") are both deleted, and the empty global array is deleted from the database.
• The close() methods for both myConn and nodeRef are called again to prevent memory leaks, since the memory allocated for them by native code cannot be released by Java garbage collection.

All of these methods are discussed in detail later in this chapter. See “Fetching Node Values” for information on getString() and other methods that fetch node values of various datatypes. See “Setting the Target Address” for information on changing the subscripts of a node reference. See “Creating and Deleting Nodes” for information on deleting persistent nodes.

4.2 Creating a Connection

This section describes the Connection class, which encapsulates a reference to the underlying eXTreme Caché database connection.

Note: It is important to understand that only one eXTreme Connection instance can exist in a process, and all Connection variables are references to that instance. When the Globals API and other eXTreme APIs are used in the same process, they will share the same underlying connection (see “Using the Globals API with Other eXTreme APIs”).

If no connection exists in the current process, the ConnectionContext.getConnection() method is used to return a new Connection object. If a connection already exists, the method returns a reference to the existing Connection object.

The following methods of Connection are used to connect, test the connection, and destroy the connection:

• connect() — uses the underlying connection object to establish a connection to a specified namespace. Can optionally specify a user name and password. Other methods are available to read and change the namespace that will be accessed (see “Accessing Namespaces” later in this chapter).

• isConnected() — returns true if the connection object has a connection to a database.

• close() — closes the underlying eXTreme connection and releases associated resources allocated by the underlying native code.

The following example creates two Connection variables and shows that they both reference the same underlying connection.

Creating a Connection

```java
Connection myConn1 = ConnectionContext.getConnection();
Connection myConn2 = ConnectionContext.getConnection();
myConn1.connect("User", "_SYSTEM", "SYS");
System.out.println("myConn1 is " + myConn1.isConnected() + "; myConn2 is " + myConn2.isConnected());
// Prints: myConn1 is true; myConn2 is true
myConn1.close();
System.out.println("myConn1 is " + myConn1.isConnected() + "; myConn2 is " + myConn2.isConnected());
// Prints: myConn1 is false; myConn2 is false
myConn1.close(); // release native resources
```

This example assumes that no other connections already exist elsewhere in this process.

Connection object myConn1 is created by a call to ConnectionContext.getConnection(). Connection variable myConn2 is defined in the same way, but since a connection object already exists, myConn2 will reference the same object as myConn1. The second call to getConnection() is functionally equivalent to Connection myConn2 = myConn1;

The myConn1 variable is used to call connect(), establishing a connection to database namespace User. Each connection variable makes a call to isConnected(). Both calls return true, proving that both variables can now access the database, even though only one of them called connect().
The myConn2 variable calls close(), which closes the underlying connection and releases associated resources. This must be done to prevent memory leaks, since memory allocated by the underlying native code cannot be released by Java garbage collection.

Both variables call isConnected() again to demonstrate that they are both disconnected now, even though close() has only been called once.

Even though the connection has been closed, close() is also called on myConn1 to release any remaining native code resources it may be holding.

### 4.3 Building a Global Array

This section describes the methods of the NodeReference class that allow it to specify an address in a global array and add, change, or delete a node value at that address. The following topics are discussed:

- **Introduction to Node References** — provides basic information about how NodeReference instances are created and used.
- **Addressing a Subnode of the Target** — describes how to access a subnode of the current target node.
- **Creating and Deleting Nodes** — provides a detailed description of the four methods that actually change the contents of the database.
- **Setting the Target Address** — describes how to modify the subscript list of a node reference.

**Note:** This section deals primarily with storing data in various global array structures. See “Retrieving Global Array Data” for a detailed discussion of data retrieval methods and techniques.

#### 4.3.1 Introduction to Node References

The NodeReference class encapsulates a reference to a node in a global array. It provides methods to specify the address of the node, and to add, change, or delete the corresponding persistent node in the database. The following method of Connection creates an instance of NodeReference:

- **Connection.createNodeReference()** — returns a new instance of NodeReference and optionally takes a name argument that specifies the global name. When the instance is created, the target address is the root node of the global array specified by name.

**Note:** **Target Address and Target Node**

A node address consists of a global name plus a list of zero or more subscripts. A NodeReference object contains global name and subscript list properties that store a reference to one possible address within the named global array. The address stored in these properties is referred to as the target address. If the database contains a node at the target address, it is referred to as the target node. See “Setting the Target Address” for information about modifying the subscript list.

See “Introduction to Global Arrays” for a quick overview of the terms and concepts discussed here.

Nothing is added or changed in the database when an instance of NodeReference is created. If the database does not already contain the specified global array, it will be created when a NodeReference method such as set() creates a persistent node.

The set() method takes a value argument and assigns the value to a node in the database. If there is no node at the specified address, a new node is created. By default, the set() method acts on the target address, but the value argument can be followed by any number of optional subscript arguments that specify a subnode of the target node (see “Addressing a Subnode of...”)
the Target” for details). The value can be an int, long, double, byte[], String, or ValueList. The following example creates an instance of NodeReference and uses set() to create two nodes in global array "myGlobal.

**Creating Nodes with NodeReference**

```java
nodeRef = myConn.createNodeReference("myGlobal"); // target is root node "myGlobal
nodeRef.set("blue", 3);  // create node "myGlobal(3) = "blue"
nodeRef.set(123);       // set value of root node "myGlobal to 123
nodeRef.set("red");    // change value of root node "myGlobal to "red"
```

The call to `createNodeReference()` creates an instance of NodeReference named `nodeRef`, and specifies "myGlobal" as the value of the global name property. The target address is "myGlobal", but nothing has been stored in the database yet.

The first call to `set()` specifies value "blue" and subscript 3, so the value is assigned to subnode "myGlobal(3). The database now contains persistent global array "myGlobal", which has two nodes: "myGlobal(3) and valueless root node "myGlobal.

The second call to `set()` specifies only value 123, with no optional subscripts, so the value is assigned to target node "myGlobal.

The final call also assigns a value to the target node, so the value of root node "myGlobal" is changed to "red."

The `set()` method is one of four NodeReference methods that can change the contents of the database. See “Creating and Deleting Nodes” for a detailed description of these methods.

### 4.3.2 Addressing a Subnode of the Target

By default, NodeReference methods act on the node at the target address (described in the previous section, “Introduction to Node References”). However, many of these methods can take optional arguments that specify the address of a subnode (a descendant of the target node). The following example sets the values of several different nodes without changing the target address:

**Addressing a Subnode of the Target**

```java
nodeRef = myConn.createNodeReference("myGlobal");
nodeRef.appendSubscript("A"); // Target address is "myGlobal("A")", but set() specifies subnodes
nodeRef.set("myvalue1",1,"x"); // sets "myGlobal("A",1,"x") = "myvalue1"
nodeRef.set("myvalue2",2);     // sets "myGlobal("A",2) = "myvalue2"
nodeRef.set("myvalue3",3,"y","z"); // sets "myGlobal("A",3,"y","z") = "myvalue3"
// Target address of nodeRef is still "myGlobal("A")"
nodeRef.set("myvalue0");      // sets "myGlobal("A") = "myvalue0"
```

The first three calls to `set()` each specify subscript arguments. Since the target address is "myGlobal("A")", these calls set the values of subnodes "myGlobal("A","x",1), "myGlobal("A",2), and "myGlobal("A",3,"y","z")|. None of these calls change the subscription list stored in `nodeRef`.

The last call to `set()` does not specify optional subscripts and sets the value of target node "myGlobal("A")", demonstrating that the previous calls have not changed the target address.

In addition to `set()` and `increment()`, subscript arguments can be used by the following NodeReference methods described elsewhere in this chapter:

- `kill()` and `killNode()` in “Creating and Deleting Nodes”.
- `getInt()`, `getLong()`, `getDouble()`, `getString()`, `getBytes()`, `getList()`, and `getObject()` in “Fetching Node Values”.
- `exists()`, `hasSubnodes()`, `nextSubscript()`, and `previousSubscript()` in “Iterating Over a Global Array”.
- `acquireLock()` and `releaseLock()` in “Locking”.

Using Java with Caché eXTreme
4.3.3 Creating and Deleting Nodes

The Globals API contains numerous methods to connect to a Caché database and read data from the global arrays stored there, but the NodeReference class contains the only four methods that can actually make changes in the database: `set()`, `increment()`, `kill()`, and `killNode()`. This section describes how these methods are used.

Creating and Changing Nodes with `set()` and `increment()`

The following `NodeReference` methods can be used to create a persistent node with a specified value, or to change the value of an existing node:

- `set()` — takes a value argument and stores the value at the target address. If no node exists at that address, a new one is created when the value is stored. The value argument can be an int, long, double, byte[], String, or ValueList.

- `increment()` — takes an int number argument, increments the target node value by that amount, and returns the incremented value as a long. Unlike `set()`, it uses a thread-safe atomic operation to change the value of the node, so the node is never locked. The target node value can be an int, long, or double. If there is no node at the target address, the method creates one and assigns the number argument as the value.

Both methods can take optional subscript arguments to access a subnode of the target node (see “Addressing a Subnode of the Target”). The following example uses both `set()` and `increment()` to create and change node values:

**Creating and Deleting Nodes: Setting and incrementing node values**

```java
// Target address of nodeRef is root node ^myGlobal
nodeRef.set("first");  // create root node ^myGlobal = "first"
nodeRef.set(2);         // change value of ^myGlobal to int 2.
// Use increment() to create and change nodes
nodeRef.increment(2);   // increment value of root node ^myGlobal from 2 to 4.
nodeRef.increment(3,1); // create new subnode ^myGlobal(1) = 3.
// Increment subnode ^myGlobal(1) and create subnodes ^myGlobal(2) and ^myGlobal(3)
for (int subnode = 1; subnode < 4; subnode++) {
    nodeRef.increment(2,subnode); // create node or increment value of node
}
```

The first call to `set()` stores a String value of "first" at the target address, root node ^myGlobal. This creates a new global array (also named ^myGlobal) containing one node. The second call changes the value of the target node, replacing the String value with an int value of 2.

The first call to `increment()` changes the value of target node ^myGlobal from 2 to 4.

The second call specifies a subscript argument, creating new subnode ^myGlobal(1) with a value of 3.

In the for loop, `increment()` is called three times with three different subscript arguments. The first iteration changes the value of existing subnode ^myGlobal(1) from 3 to 5. The next two iterations create new subnodes ^myGlobal(2) and ^myGlobal(3), each with a value of 2.

Deleting Nodes with `kill()` and `killNode()`

A node is deleted from the database when it is valueless and has no subnodes with values. When the last value is deleted from a global array, the entire global array is deleted. The following methods of `NodeReference` are used to delete one or more node values from the database:

- `kill()` — deletes the specified node and all of its subnodes. If the root node is specified, the entire global array is deleted. This method is equivalent to the Caché inclusive KILL command.

- `killNode()` — deletes only the value of the specified node. The node becomes valueless, but is not deleted if it has subnodes. The values of subnodes are not affected. This method is equivalent to the Caché ZKILL command.

Both `kill()` and `killNode()` can take optional subscript arguments to access a subnode of the target node (see “Addressing a Subnode of the Target”), as demonstrated in several of the following examples.
All of these examples assume that root node `^myGlobal` is the target address of `nodeRef`, and that the global array initially contains the following nodes:

```
^myGlobal = 0
^myGlobal("A") = 0
^myGlobal("A",1) = 0
^myGlobal("B") = 0
^myGlobal("B",1) = 0
^myGlobal("B",2) = 0
```

### Creating and Deleting Nodes: Using `killNode()` to delete only the specified node value

```
nodeRef.killNode("A");  // only delete value of ^myGlobal("A")
```

The call to `killNode()` deletes only the value of specified subnode `^myGlobal("A")`. It is now valueless, but it is still part of the global array because `^myGlobal("A",1)` still has a value.

### Creating and Deleting Nodes: Using `kill()` to delete all nodes in one branch of a global array

```
nodeRef.kill("B");  // kill ^myGlobal("B") and all of its subnodes
```

The call to `kill()` deletes the values of specified subnode `^myGlobal("B")` and both nodes under it. The nodes are completely eliminated from the global array (unlike `^myGlobal("A")` in the previous example) because they no longer have values or subnodes with values. The three remaining nodes in the global array are unaffected.

### Creating and Deleting Nodes: Using `kill()` to delete an entire global array

```
// These nodes are left after the previous examples:
//  ^myGlobal = 0
//  ^myGlobal("A")  <valueless node>
//  ^myGlobal("A",1) = 0

nodeRef.kill();  // kill all nodes in global array ^myGlobal
```

No subscript argument is specified, so the call to `kill()` deletes the values of target node `^myGlobal` and all of its subnodes. This causes the entire global array to be deleted because it no longer contains any nodes with values.

### 4.3.4 Setting the Target Address

It is often convenient to access a node by specify the subscripts of the node address in a method call (see “Addressing a Subnode of the Target”). However, a hard-coded list of subscripts quickly becomes difficult to manage when working with large numbers of nodes (as demonstrated later in “Iterating Over a Global Array”). In such cases, it is more practical to change the target address stored in the node reference. This section describes methods that control the subscript list property, allowing an instance of NodeReference to target any address in the global array. (Methods are also provided to control the global name property, as described later in “Accessing Multiple Global Arrays”).

**Note:** Subscript Lists and Node Levels

The term node level refers to the number of subscripts in the subscript list. For example, `^myGlobal("a","b","c")` is a “level 3 node,” which is just another way of saying “a node with three subscripts.”

The following methods add, delete, or change items in the subscript list:

- `appendSubscript()` — takes a subscript argument (int, long, double, or String) and appends it to the end of the list.
- `setSubscriptCount()` — takes a subscriptPosition argument and shortens the list by removing all subscripts after the one at the specified position.
- `setSubscript()` — changes a subscript to the specified value. Takes subscriptPosition and value arguments, which specify the subscript to be replaced and the new subscript value.
The following example constructs a node address in the \texttt{NodeReference} object, and then creates a global array in the Caché database by calling \texttt{set()} to store a value at the target address:

**Setting the Target Address: Adding subscripts with appendSubscript()**

```java
nodeRef = myConn.createNodeReference("myGlobal"); // nodeRef points to ^myGlobal
nodeRef.appendSubscript("A"); // nodeRef points to ^myGlobal("A")
nodeRef.appendSubscript(1); // nodeRef points to ^myGlobal("A",1)
nodeRef.set("hay"); // create persistent level 2 node ^myGlobal("A",1)
```

The call to \texttt{createNodeReference()} specifies "myGlobal" as the global name property in \texttt{nodeRef}, so the initial target address is root node ^\texttt{myGlobal}.

The calls to \texttt{appendSubscript()} add two subscripts to the target address, changing it to "^myGlobal("A",1)". No changes have been made to the database yet.

The call to \texttt{set()} creates a persistent node at target address "^myGlobal("A",1)".

The Caché database now contains global array "^myGlobal" with three nodes: target node "^myGlobal("A",1)", plus valueless node "^myGlobal("A")" and valueless root node "^myGlobal".

Once a subscript has been added to the subscript list, it remains until removed by \texttt{setSubscriptCount()} or changed by \texttt{setSubscript()}. The following example starts with the target address still set to "^myGlobal("A",1)".

**Setting the Target Address: Removing and changing subscripts**

```java
// node reference currently points to ^myGlobal("A",1)
nodeRef.setSubscriptCount(1); // go back to ^myGlobal("A")
nodeRef.setSubscript(1,"B"); // change target to ^myGlobal("B")
nodeRef.set("bee"); // create persistent level 1 node ^myGlobal("B")
nodeRef.setSubscriptCount(0); // go back to root node ^myGlobal
nodeRef.set("this is the root"); // set the value of ^myGlobal
```

The first call to \texttt{setSubscriptCount()} specifies that the subscript list should be truncated after the first subscript, changing the target address to "^myGlobal("A")".

The call to \texttt{setSubscript()} changes the target address to "^myGlobal("B")", and the call to \texttt{set()} creates a persistent node at that address.

The second call to \texttt{setSubscriptCount()} specifies 0 as the node level, removing all subscripts from the subscript list. The target address is now valueless root node ^\texttt{myGlobal}, and the call to \texttt{set()} assigns a value to the target node.

This example has added node "^myGlobal("B")" and changed the value of the root node, so global array ^\texttt{myGlobal} now contains the following nodes and values:

```
^myGlobal = "this is the root"
^myGlobal("A") = (valueless node)
^myGlobal("A",1) = "hay"
^myGlobal("B") = "bee"
```

The \texttt{setSubscript()} method can also add a new subscript to the list if \texttt{subscriptPosition} is set to one more than the current number of subscripts. The following method returns the current number of subscripts in the list:

- \texttt{getSubscriptCount()} — returns an int value indicating the target node level (current number of subscripts in the list).

This allows you to create or access a series of nodes under the same parent node by repeatedly calling \texttt{setSubscript()} with the node level set to \texttt{getSubscriptCount()}+1, as demonstrated in the following example:
Setting the Target Address: Creating a series of subnodes

```java
// node reference currently points to ^myGlobal
nodeRef.appendSubscript("C");
int nodelevel = nodeRef.getSubscriptCount()+1;
for (int ii = 1; ii < 4; ii++) {
    nodeRef.setSubscript(nodelevel,ii);
    nodeRef.set(ii*2);   // create three nodes ^myGlobal("C",<ii>)
}
```

The current target address is ^myGlobal, so the call to `appendSubscript()` sets the target address to ^myGlobal("C"). Variable `nodelevel` will be used to specify the level inside the loop. A call to `getSubscriptCount()` is used to set its value to 2 (current node level +1).

In the loop, the first call to `setSubscript()` adds a level 2 subscript to the subscript list. Each subsequent call to `setSubscript()` changes the existing level 2 subscript to a new value.

The `set()` method is called for each new target address, creating new level 2 nodes ^myGlobal("C",1), ^myGlobal("C",2), and ^myGlobal("C",3).

This example adds the following nodes to ^myGlobal:

- ^myGlobal("C") (valueless node)
- ^myGlobal("C",1) = 2
- ^myGlobal("C",2) = 4
- ^myGlobal("C",3) = 6

4.4 Iterating Over a Global Array

The Globals API provides ways to iterate over part or all of a global array. The following topics describe the various Globals iteration methods:

- **Retrieving a Set of Sibling Nodes** — describes how to iterate over a set of subnodes immediately under the parent node.
- **Controlling the Starting Point and Order of Iteration** — describes how to initialize the cursor and iterate in ascending or descending collation order
- **Accessing All Subnodes of a Target Address** — describes how to test for the existence of node values and subnodes, and how to iterate over all subnodes regardless of node level.

4.4.1 Retrieving a Set of Sibling Nodes

Sibling nodes are sets of nodes immediately under the same parent node. Any sibling of the current target node can be addressed by changing only the final subscript of the target address. For example, the following global array has five level 2 sibling nodes under parent node ^myNames("people"):

```plaintext
^myNames                               (valueless root node)
^myNames("people")                  (valueless level 1 node)
  ^myNames("people","Anna") = 2
  ^myNames("people","Julia") = 4
  ^myNames("people","Misha") = 5
  ^myNames("people","Ruri") = 3
  ^myNames("people","Vlad") = 1
```

Assume that the current target node is ^myNames("people","Misha"). This node has four sibling nodes, which can be accessed by changing the final subscript of the target address to "Anna", "Julia", "Ruri", or "Vlad". In other words, it is possible to access all of the sibling nodes by finding the final subscript for each node. The example in this section uses the `nextSubscript()` method to return the final subscripts of all five sibling nodes.
Finding the First Subscript

The nextSubscript() method starts from the current target address and returns the final subscript of the next sibling node in ascending collation order. For example, if the target address is set to `^myNames("people","Misha")`, the method will return "Ruri".

To return the subscript of the first sibling, you can initialize the target subscript to an empty string ("""). For example, to find subscript "Anna", you would set the target address to `^myNames("people",""")`. You could think of this address as another sibling, but it is important to remember that this not a valid node address, and should only be set immediately before calling nextSubscript().

Finding the Other Subscripts

Once the first valid subscript has been recovered, the process can be repeated to access all of the other sibling nodes in ascending order. The following example finds and prints the final subscript for each of the five sibling nodes:

```
Retrieving a Set of Nodes: Finding all sibling nodes under ^myNames("people")

// Find the first level 2 node under ^myNames("people")
NodeReference nodeRef = myConn.createNodeReference("myNames");
nodeRef.appendSubscript("people");
nodeRef.appendSubscript("");
String subscr = nodeRef.getSubscriptCount(); // Value of subscr is now "Anna"

// Iterate in ascending order until NextSubscript() returns ""
while (!subscr.equals("")) {
    System.out.print("\"" + subscr + "," + nodeRef.getInt() + "\")
    subscr = nodeRef.nextSubscript();
}
```

This code prints the following line:

```
Ascend from first node: "Anna"=2 "Julia"=4 "Misha"=5 "Ruri"=3 "Vlad"=1
```

The call to createNodeReference() creates a node reference for `^myNames`, and the first call to appendSubscript() sets the target address to `^myNames("people")`.

The second call to appendSubscript() sets the level 2 subscript of the target address to an empty string, deliberately creating invalid node reference `^myNames("people",""")`. This starting address is before the first possible child of `^myNames("people")` in ascending order.

The getSubscriptCount() method is used to determine the position of the final subscript in the target address. By definition, this value is constant for all five siblings.

The call to nextSubscript() returns the final subscript from the first sibling node in ascending order. The value of subscr is now "Anna".

Each iteration of the loop performs the following actions:

- setSubscript() is called to set the final subscript of the target address to the returned value.
- The new subscript is printed, and getInt() is used to read and print the node value. (See “Fetching Node Values” for detailed information on getInt()).
- nextSubscript() is called to return the next sibling subscript. An empty string (""") will be returned if there are no more sibling nodes, and the loop will terminate.

This example is extremely simple, and would fail in several situations. What if the global array has data on more than one level? What if the code attempts to fetch a value from a valueless node? What if we don’t want to start with the very first node? The following sections describe how to deal with these situations.
4.4.2 Controlling the Starting Point and Order of Iteration

The example in this section will use a slightly more complex set of subnodes:

```
^myNames                               (valueless root node)
  "myNames("dogs")"                  (valueless level 1 node)
    "myNames("dogs","Balto")" = 6
    "myNames("dogs","Hachiko")" = 8
    "myNames("dogs","Lassie")" = 9
    "myNames("dogs","Lassie","Timmy")" = 10
    "myNames("dogs","Whitefang")" = 7
```

In addition to the four sibling nodes under "myNames("dogs")", this example also includes level 3 node "myNames("dogs","Lassie","Timmy")", which is not a sibling of the level 2 nodes.

The starting point for a search is specified according to the following rules:

- Any valid subscript value can be specified for the beginning subscript, and the resulting target address does not have to be an existing node.
  For example, if the target address is "myNames("dogs","M")", nextSubscript() will return "Whitefang" and previousSubscript() will return "Lassie" (ignoring level 3 subscript "Timmy", which is not part of a sibling node address).

- If the empty string is specified as the beginning subscript, the search will start before the first node in the given direction.
  For example, given "myNames("dogs","")" as the target address, nextSubscript() will return "Balto" and previousSubscript() will return "Whitefang".

**Note:** Be careful when using empty strings as subscripts

Empty string subscripts should never be added except to initialize the target address immediately before a search. Adding an empty string produces an invalid target address that will cause most NodeReference method calls to throw GlobalsException.

The direction of iteration (ascending or descending) is controlled by calling the appropriate NodeReference iteration method:

- **nextSubscript()** — searches in ascending collation order (as demonstrated in the previous example).
- **previousSubscript()** — searches in descending collation order.

Both methods start at the current target node, find the next sibling node, and return its final subscript (or "" if no node is found).

Rather than starting with the first level 2 node and iterating in ascending order, this example will start in the middle and iterate in descending order. It sets the starting address to "myNames("dogs","W")", and fetches all level 2 subnodes of "myNames("dogs")" that have a final subscript less than "W" in collation order.

**Controlling the Starting Point and Order of Iteration: descending from "W"**

```java
nodeRef.setSubscriptCount(0);
nodeRef.appendSubscript("dogs");
nodeRef.appendSubscript("W");
int siblingLevel = nodeRef.getSubscriptCount();
String subscr = nodeRef.previousSubscript();
System.out.print("Start at \"W\" and descend: ");
while (!subscr.equals("")) {
    nodeRef.setSubscript(siblingLevel, subscr);
    System.out.print("\"" + subscr + ",\"" + nodeRef.getInt() + " ");
    subscr = nodeRef.previousSubscript();
}
```

This code prints the following line:

```
Start at "W" and descend: "Lassie"=9 "Hachiko"=8 "Balto"=6
```
The call to `appendSubscript()` sets the starting address to level 2 node `^myNames("dogs","W")`. It does not matter if a node actually exists at that address. It just establishes the point from which the subscript search will begin.

The first call to `previousSubscript()` finds the first level 2 subnode of `^myNames("dogs")` that is lower than `^myNames("dogs","W")` in collation order, and returns its final subscript.

In the while loop, each level 2 subscript lower that "W" will be returned in descending collation order, and the loop will terminate when `previousSubscript()` has returned an empty string.

In this example, the search misses many of the nodes in global array `^myNames`:

- Node `^myNames("dogs","Whitefang")` is not found because the search is in descending order and "Whitefang" is higher in collation order than "W".
- Node `^myNames("dogs","Lassie","Timmy")` is not found because it is not a sibling of the level 2 subnodes.
- Nodes under `^myNames("people")` are not found because they are not siblings of the nodes under `^myNames("dogs")`.

The next section demonstrates how to find all subnodes in a global array and determine which ones contain values.

### 4.4.3 Accessing All Subnodes of a Target Address

The `previousSubscript()` and `nextSubscript()` methods only operate on one node level, but it is possible to traverse all subnodes on every level. The following methods of `NodeReference` can be used to traverse all levels of a global array and determine which nodes contain values:

- `hasSubnodes()` — returns `true` if the target node has any descendants.
- `exists()` — returns `true` if the target node has a defined value, or `false` if it has subnodes but no value.

The example `exampleShowGlobal()` method listed in the following example iterates over all nodes in global array `^myNames`, making recursive calls to process all levels. The main code block of `exampleShowGlobal()` is almost identical to the previous examples, except that `hasSubnodes()` is used to trigger recursive calls and `exists()` is used to test for node values.

#### Accessing Subnodes: exampleShowGlobal() — traversing all node levels recursively

The following code creates a temporary node reference to `^myNames`, with the target address set to the root node, and calls `exampleShowGlobal()` for the first time:

```java
NodeReference tempRef = myConn.createNodeReference("myNames");
exampleShowGlobal(tempRef, "^myNames(");
tempRef.close();
```

The `exampleShowArray()` method calls itself recursively whenever subnodes are detected below the current target node. It iterates over each child node and prints the address and value of each node that has a value.

```java
public static void exampleShowGlobal(NodeReference aNodeRef, String aBaseName) {
    try {
        aNodeRef.appendSubscript("\n");
        String subscr = aNodeRef.nextSubscript();
        while (!subscr.equals("\n")) {
            aNodeRef.setSubscript(aNodeRef.getSubscriptCount(), subscr);
            if (aNodeRef.exists())
                System.out.println(aBaseName + subscr + ") = " + aNodeRef.getInt());
            if (aNodeRef.hasSubnodes())
                exampleShowGlobal(aNodeRef, aBaseName + subscr + ",");
            subscr = aNodeRef.nextSubscript();
        }
    } catch (GlobalsException e) { System.out.println("exampleShowGlobal():" + e.getMessage()); }
    finally { aNodeRef.setSubscriptCount(aNodeRef.getSubscriptCount() - 1); }
} // end exampleShowGlobal()
```

The call to `appendSubscript()` adds an empty string to the subscript list so that `nextSubscript()` will begin its search before the first node in ascending collation order.
The while loop will find each sibling node on the current level. If the call to exists() indicates that the node contains a value, the node address and value will be printed.

The hasSubnodes() method is used to trigger a recursive call to exampleShowGlobal() for each node that has descendants. Each recursion descends one more level and processes all sibling nodes on that level, until no more subnodes are found. In the finally clause, setSubscriptCount() is called at the end of each recursion to return the target address to the previous level.

This example produces the following output:

```
"myNames("dogs","Balto")=6
"myNames("dogs","Hachiko")=8
"myNames("dogs","Lassie")=9
"myNames("dogs","Lassie","Timmy")=10
"myNames("dogs","Whitefang")=7
"myNames("people","Anna")=2
"myNames("people","Julia")=4
"myNames("people","Misha")=5
"myNames("people","Ruri")=3
"myNames("people","Vlad")=1
```

This output is similar to what you would see in the Management Portal by going to [Home] > [Globals], selecting the User namespace from the list on the left, and clicking the View link for "myNames.

Although the nodes are found and printed in collation order, they were not necessarily added to the database in that order. (In fact, the code that created this test array got the value of each node from a loop counter, so the values indicate the order of insertion). See “Collation Order in Multilevel Global Arrays” for details about how Caché determines collation order.

### 4.5 Retrieving Global Array Data

The Globals API allows node references to take subscripts of datatype int, long, double, or String. Node values can be any of those four datatypes plus byte[], or Valuelist. For simplicity, examples in previous sections of this chapter have always used int or String subscripts and node values. This section describes how to assign and retrieve the full range of supported datatypes.

- **Fetching Node Values** — provides a detailed description of the methods that fetch node values from the database and return them as supported Java datatypes.
- **Reading Subscripts from the Target Address** — describes methods that can read the subscript list of a Java NodeReference object and return each subscript as a variable of the appropriate datatype.
- **Using Subscripts to Store Data** — describes why and when it may be useful to store numbers or short strings as subscripts rather than node values.

#### 4.5.1 Fetching Node Values

The NodeReference class provides datatype-specific methods for fetching nodes with values of known datatype int, long, double, String, byte[], or Valuelist. In addition, the getobject() method can fetch node values of unknown datatype and return them as Object variables that can be cast to the appropriate type.

The following topics are discussed in this section:

- **Fetching Numeric Values** — describes the type-specific methods for fetching int, long, and double values.
- **Fetching String and byte[] Values** — describes the type-specific methods for fetching String and byte[] values.
- **Fetching Node Values of Unknown Datatype** — describes the getobject() method, which fetches an Object value that can be cast to the appropriate datatype.
Note: **Fetching ValueList**

The ValueList class encapsulates a serialized list that can store any of the supported datatypes, including a nested ValueList. See “Using Serialized Value Lists” for a complete discussion of the `getList()` method, which fetches a node value as a ValueList.

### 4.5.1.1 Fetching Numeric Values

This section discusses the following methods of NodeReference:

- **getInt()** — fetches the value of the target node and returns it as an int.
- **getLong()** — fetches the value of the target node and returns it as a long.
- **getDouble()** — fetches the value of the target node and returns it as a double.

The `getInt()`, `getLong()`, and `getDouble()` methods assume that the node value is numeric, and attempt to convert it to an appropriate Java variable. They will throw **UndefinedException** if the target node is valueless or does not exist. All of these methods can take optional subscript arguments to access a subnode of the target node (see “Addressing a Subnode of the Target”).

**Important:** **Fetch methods do not check for incompatible datatypes**

These methods are optimized for speed, and never perform type checking. Your application should never depend on an exception being thrown if one of these methods attempts to fetch a value of the wrong datatype. Although an exception may be thrown, it is possible that the method will return an inaccurate or meaningless value in such cases.

Given an int value, all numeric methods return meaningful values. The `getInt()` and `getLong()` methods cannot be applied to double values with reliable results, and may throw an exception for some double values.

**Fetching Node Values: Using getInt(), getLong(), and getDouble()**

```java	nodeRef.set(23);
int intVal = nodeRef.getInt();  // returns 23
long longVal = nodeRef.getLong();  // returns 23
double doubleVal = nodeRef.getDouble();  // returns 23.0
	nodeRef.set(10.1234);
doubleVal = nodeRef.getDouble();  // returns 10.1234
// Do not use getInt() or getLong() to fetch double values!
```

### 4.5.1.2 Fetching String and byte[] Values

This section discusses the following methods of NodeReference:

- **getString()** — fetches the value of the target node and returns it as a String.
- **getBytes()** — fetches the value of the target node and returns it as a byte[].

Both of these methods can take optional subscript arguments to access a subnode of the target node (see “Addressing a Subnode of the Target”).

The `getString()` and `getBytes()` methods assume that the node value is non-numeric, and attempt to convert it appropriately. Both of these methods return null if the target node is valueless or does not exist. These methods do not perform any type checking, and will not usually throw an exception if the argument specifies a node with a numeric value.

Binary data can be stored as an array of byte and fetched with `getBytes()`. The following example creates a byte array and prints it out as a list of unsigned integer values. For convenience, this example uses byte values that correspond to characters.
Fetching Node Values: Using getBytes()

```java
byte[] testBytes = {'C','a','c','h',(byte)'é'};
nodeRef.set(testBytes);
System.out.println("Bytes fetched as byte[]:");
byte[] outBytes = nodeRef.getBytes();
for (int ii = 0; ii < outBytes.length; ii++) {
    System.out.print(" " + ((int)outBytes[ii] & 0xff));  // get byte as unsigned int
}
System.out.println();
// prints: Bytes fetched as byte[]: 67 97 99 104 233
```

By default, byte values greater than 127 are interpreted as negative numbers. This example masks each byte with 0xff to produce an unsigned integer for printing.

4.5.1.3 Fetching Node Values of Unknown Datatype

If a node has a value of datatype int, long, double, or String, the node value can be returned as an Object and cast to the appropriate datatype. The following method of NodeReference returns Object values:

- `getObject()` — fetches the value of the target node and returns it as an Object. Can take optional subscript arguments to access a subnode of the target node (see “Addressing a Subnode of the Target”).

**Note:** In the Caché database, String, byte[], and ValueList objects are all stored as strings, and no information about the original datatype is preserved. Since all three datatypes will be interpreted as instances of String by the `instanceOf` operator, applications should avoid using `getObject()` in situations where the original non-numeric type is unknown.

The following example creates nodes with values of four different datatypes, fetches each of them with `getObject()`, and casts each returned Object to the appropriate datatype. In this example, both `set()` and `getObject()` use subscript arguments to specify the subnode that will be accessed.

Fetching Node Values: Using `getObject()`

```java
nodeRef.set(13, 1); // ^MyGlobal(1) = 13
nodeRef.set(-455666777L, 2); // ^MyGlobal(2) = -455666777L
nodeRef.set(1.234, 3); // ^MyGlobal(3) = 1.234
nodeRef.set("hello", 4); // ^MyGlobal(4) = "hello"

for (int subnode = 1; subnode < 5; subnode++) {
    Object obj = nodeRef.getObject(subnode);
    System.out.print("node ^myGlobal(" + subnode + ") is type ");
    if (obj instanceof String) System.out.print("String: " + (String)obj);
    else if (obj instanceof Integer) System.out.print("Integer: "+((Integer)obj).intValue());
    else if (obj instanceof Long) System.out.print("Long: "+((Long)obj).longValue("+"L");
    else if (obj instanceof Double) System.out.print("Double: " +((Double)obj).doubleValue());
    else System.out.println("unexpected: "+obj.getClass().getName());
    System.out.println();
}
```

The calls to `set()` produce four new nodes of various datatypes.

The `getObject()` method returns each node value as an Object, and the `instanceOf` operator is used to determine the original datatype.

This example produces the following output:

```
node "myGlobal(1)" is type Integer: 13
node "myGlobal(2)" is type Long: -455666777L
node "myGlobal(3)" is type Double: 1.234
node "myGlobal(4)" is type String: "hello"
```

4.5.2 Reading Subscripts from the Target Address

This section describes methods that read a subscript from the subscript list of a NodeReference object, and return it as a value of the appropriate datatype (int, long, double, or String). These methods can be used to print the current target address,
to duplicate the current node reference, or to access the values of individual subscripts in any other situation where that may be useful.

- **Reading Subscripts** — describes the `NodeReference` methods used to read subscripts.
- **Creating a duplicate node reference** — describes how to read and duplicate the target address of a node reference.

### 4.5.2.1 Reading Subscripts

Although the subscript methods may seem similar to the methods for fetching node values (see “Fetching Node Values”), there are some important differences:

- Unlike the node value methods, subscript methods do not fetch persistent data from the Caché database. They just return the current values from the subscript list property of the `NodeReference` instance.
- Since the subscripts are already stored as specific Java datatypes in the subscript list, no type conversion is necessary. An exception is thrown when a type-specific subscript method tries to read a subscript of the wrong datatype (although `getLongSubscript()` can read int subscripts).

The following methods of `NodeReference` return a subscript list item of a specific datatype. Each method takes a `level` argument that specifies the position of the subscript to be read:

- `getIntSubscript()` — reads an int subscript at the specified position.
- `getLongSubscript()` — reads a long or int subscript at the specified position.
- `getDoubleSubscript()` — reads a double subscript at the specified position.
- `getStringSubscript()` — reads a String subscript at the specified position.

Each method will throw `GlobalsException` if it attempts to read a subscript of the wrong datatype, or if the `level` argument is greater than the number of subscripts in the list.

The following example reads each subscript of node `^mySubscripts(101, 9876543210L, 3.14, "string value")` with the appropriate method for the datatype. The returned values are used to print the full node address.

**Reading Subscripts from the Target Address: Subscripts of known datatype**

```java
// nodeRef points to ^mySubscripts(101, 9876543210L, 3.14, "string value")
System.out.println("^mySubscripts(" + nodeRef.getIntSubscript(1) + ", " + nodeRef.getLongSubscript(2) + "L, " + String.valueOf(nodeRef.getDoubleSubscript(3)) + ", " + "\"" + nodeRef.getStringSubscript(4) + "\")")
// prints: ^mySubscripts(101, 9876543210L, 3.14, "string value")
```

The standard Java `String.valueOf()` method is used to convert the double value to an accurate display string.

In cases where the datatype is not known, the following method can be used to return an `Object`, which can then be cast as a subscript of the appropriate datatype:

- `getObjectSubscript()` — reads the subscript at the specified level and returns it as an `Object`. For example, `getObjectSubscript(3)` would return an object containing the level 3 subscript. Throws `GlobalsException` if the specified level is greater than the number of subscripts in the list.

The Java `instanceof` operator can be used to determine the actual datatype, allowing the subscript to be cast appropriately as an int, long, double, or String.

The following code produces exactly the same output as the previous example, but does not depend on knowledge of the subscript datatypes:  

```java
// nodeRef points to ^mySubscripts(101, 9876543210L, 3.14, "string value")
System.out.println("^mySubscripts(" + nodeRef.getIntSubscript(1) + ", " + nodeRef.getLongSubscript(2) + "L, " + String.valueOf(nodeRef.getDoubleSubscript(3)) + ", " + "\"" + nodeRef.getStringSubscript(4) + "\")")
// prints: ^mySubscripts(101, 9876543210L, 3.14, "string value")
```
**Reading Subscripts from the Target Address: Subscripts of any datatype**

```java
Object sub = null;
try {
    System.out.print("^" + nodeRef.getName() + "(");
    for (int ii=1; ii<=nodeRef.getSubscriptCount(); ii++) {
        sub = nodeRef.getObjectSubscript(ii);
        if (sub instanceof Integer) System.out.print(sub.toString());
        else if (sub instanceof Long) System.out.print(sub.toString() + "L");
        else if (sub instanceof Double) System.out.print(String.valueOf((Double)sub));
        else System.out.print("" + sub.toString() + "");
        if (ii < nodeRef.getSubscriptCount()) System.out.print(",");
    }
    System.out.println(")");
} catch (GlobalsException e) {
    System.out.println(e.getMessage());
}
```

If the target address in `nodeRef` is `^mySubscripts(101,9876543210L,3.14,"string value")`, the following output is printed:

`^mySubscripts(101, 9876543210L, 3.14, "string value")`

Before the subscripts are read, `getName()` is called to read and print the global name (see “Accessing Global Names” for more information).

The `getObjectSubscript()` method is called for each subscript, and the `instanceof` operator is used to determine the subscript datatype. Using this information, each subscript is printed in an appropriate format: quotes are placed around a `String`, an `L` is added after a `Long`, and the Java `String.valueOf()` method is used to generate an accurate display string for the `Double`.

### 4.5.2.2 Creating a duplicate node reference

In some cases, it may be useful to retain an accurate copy of a target address. It is important to remember that the typed subscript list exists only in memory, as a property of the `NodeReference` object. In the Caché database, all subscripts are normalized as strings and the original datatype distinctions are lost. When retrieving node addresses from the database (as described previously in “Iterating Over a Global Array”), there is no simple way to recover the original datatype of the subscript.

The following example reads the target address of an existing instance of `NodeReference` and copies the address into a new instance. This technique can be useful when you want to preserve a record of the current target address before performing an operation that may change it.

**Reading Subscripts from the Target Address: Creating a duplicate node reference.**

```java
try {
    NodeReference newNode = myConn.createNodeReference(nodeRef.getName());
    if (nodeRef.getSubscriptCount()>0) {
        for (int ii=1; ii<=nodeRef.getSubscriptCount(); ii++) {
            Object subscr = nodeRef.getObjectSubscript(ii);
            if (subscr instanceof Integer) { newNode.appendSubscript((Integer)subscr); }
            else if (subscr instanceof Long) { newNode.appendSubscript((Long)subscr); }
            else if (subscr instanceof Double) { newNode.appendSubscript((Double)subscr); }
            else { newNode.appendSubscript((String)subscr); }
        }
    }
} catch (GlobalsException e) { System.out.println(e.getMessage()); }
```

The `createNodeReference()` method creates a new node reference object, using the `getName()` method (discussed later in “Accessing Global Names”) to read the global name from `nodeRef`.

The `getObjectSubscript()` method is used to read the subscripts. Each returned subscript is examined with the `instanceof` operator, allowing it to be cast appropriately when it is passed to `appendSubscript()`.
4.5.3 Using Subscripts to Store Data

In some cases, it may be useful to store integer numbers or short strings as subscripts rather than node values. For example, the following code counts the frequency of words in a sentence, storing the counted words as subscripts and the word frequencies as node values:

```java
// Count the words in the sentence
String[] sentence = ("a penny saved is a penny earned").split(" ");
nodeRef = myConn.createNodeReference("myWords");
for (int i = 0; i < sentence.length; i++) {
    nodeRef.setSubscript(1, sentence[i]);
    nodeRef.increment(1);
}

// Print the results
nodeRef.setSubscript(1, "");
String subscr = nodeRef.nextSubscript();
while (!subscr.equals("")) {
    nodeRef.setSubscript(1, subscr);
    System.out.println("^myWords(" + subscr + ")" + nodeRef.getInt());
    subscr = nodeRef.nextSubscript();
}
```

This code produces the following output:

```
^myWords("a") = 2
^myWords("earned") = 1
^myWords("is") = 1
^myWords("penny") = 2
^myWords("saved") = 1
```

Each word in the sentence is stored only once as a unique level 1 subscript. When a word is encountered for the first time, the `increment()` method creates a new node with a value of 1. Each subsequent instance of the word causes the existing node value to be incremented by 1. After the sentence has been processed, the results are printed by iterating over each node in the global array (as described in “Iterating Over a Global Array”).

**Note:** When using subscripts to store data, bear in mind that there are limits to the total number of characters in a node address (see “Global Array Naming Conventions” for details). Although it is theoretically possible to store double values as subscripts, this is not recommended because the Caché database normalizes all subscripts as strings. The subscript strings can be converted back to double, but this may result in a small conversion error.

### 4.6 Using Serialized Value Lists

The `ValueList` class encapsulates a serialized list that can store an arbitrary number of elements. Each element can be any one of the supported datatypes: int, long, double, String, byte[], or nested `ValueList`. In the database, a `ValueList` of any size is stored in a single node, encoded as a binary string that uses the Caché $LIST format. This format is transparent to Java applications, but a node containing a `ValueList` can be interpreted as a $LIST by Caché ObjectScript or any Caché language binding that supports $LIST format.

This section discusses the following topics:

- **Creating and Storing a ValueList** — describes how to use the methods that create and destroy a `ValueList`, add or delete list elements, and fetch a list from the database.

- **Reading ValueList Elements** — describes the `ValueList` methods that allow a list element to be selected and returned as a variable of the appropriate datatype.
4.6.1 Creating, Storing, and Fetching a ValueList

Instances of ValueList are created and fetched by the following methods:

- **Connection.createList()** — returns an empty ValueList object. Takes an optional int bufferSize argument that can optimize memory allocation by specifying the initial byte size of the list buffer. ThrowsGlobalsException if there is no connection to the database.

- **NodeReference.getList()** — fetches the value of the target node from the database and returns it as a ValueList object (see “Fetching Node Values” for a general discussion of methods that fetch node values from the database). Can take optional subscript arguments.

The following ValueList methods are used to populate and destroy a instance:

- **append()** — appends one or more values of any supported type: int, long, double, String, byte[], or nested ValueList.

- **clear()** — clears the list, causing it to contain 0 elements.

- **close()** — destroys the object and releases its underlying native code resources to prevent memory leaks.

The append() and clear() methods are the only ways to change the contents of a list. There are no methods to delete a single element, insert a new element in the middle of a list, or change the value of an existing element.

In the following example, a ValueList is created and populated with elements of datatype Integer, Long, Double, String, and byte[]. The list is stored in root node ^myListNode. The example then clears the list and fills it with new data, fetches the old list, and appends the new ValueList as a nested element in the old list.

```
// Create a list, append elements of different types, and store it in ^myListNode
ValueList myList = myConn.createList();
myList.append(13, -455566777L, 1.234, new String("hello"),
            new String("Hi!").getBytes()); // convert a String to byte[] and append
NodeRef = myConn.createNodeReference("myListNode");
nodeRef.set(myList);

// Delete all elements from myList re-use it to create a different list
myList.clear(); // delete all elements from the list
myList.append(new String("world"), new Integer(7), new Double(3.1),new Long(9876543210L));

// Fetch the old list, then store the new list as a nested element in the old list
ValueList myNewList = nodeRef.getList(); // fetch previously created list
myNewList.append(myList); // append the new list object as a single element
nodeRef.set(myNewList);
myList.close(); // delete all elements from the list
myNewList.close();
```

The Connection.createList() method creates an empty instance of ValueList named myNewList.

The call to append() adds five elements of different types, and then set() stores the list in node ^myListNode.

The call to clear() deletes all existing elements from the myList instance so it can be re-used, and the next call to append() adds four new elements to the list.

The NodeReference.getList() method fetches the list stored in ^myListNode and assigns it to a new instance of ValueList named myNewList. The myList object is appended as the sixth element in myNewList, and the updated list is again stored in persistent node ^myListNode.

After this update process is completed, both instances of ValueList are destroyed by calling their close() methods.

**Important:** Always call close() to avoid memory leaks

It is important to always call close() on instances of ValueList before they go out of scope or are reused. Failing to close them can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.
4.6.2 Reading ValueList Elements

The ValueList class provides methods that return an element value of known datatype as a variable of the desired datatype.

Reading Numeric Elements

The following datatype-specific methods are used to read numeric elements from the ValueList object:

- `getNextDouble()` — returns the next element from the list as a double.
- `getNextInt()` — returns the next element from the list as an int.
- `getNextLong()` — returns the next element from the list as a long.

All of the numeric methods return 0 if the list element is null.

Reading Non-numeric Elements

The following datatype-specific methods are used to read non-numeric elements from the ValueList object.

- `getNextString()` — returns the next element from the list as a String.
- `getNextBytes()` — returns the next element from the list as byte array.
- `getNextList()` — returns the next element from the list as a ValueList. If the optional ValueList `reuseList` argument is specified, the returned list is assigned to the specified variable rather than being placed in a new instance of ValueList. ThrowsGlobalsException if the element is not a valid ValueList.

All of these methods return null if the list element is null. All three non-numeric types (String, byte[], and ValueList) are stored as strings in the database, so no exception will be thrown if `getNextString()` or `getNextBytes()` is applied to the wrong non-numeric datatype.

Reading ValueList Elements of Known Datatype

In the following example, the list created in the previous example (see “Creating, Storing, and Fetching a ValueList”) is fetched from node `^myListNode`, and the first five elements in the list are read and printed. Those five elements are known to be int, long, double, String, and byte[], in that order, so data-specific methods can be used to read them.

Reading ValueList Elements: reading int, long, double, String, and byte[]

```java
// Fetch the list from the database
nodeRef = myConn.createNodeReference("myListNode");
ValueList outList = nodeRef.getList();   // fetch list from node ^myListNode

// Read and print out each element in the list
System.out.println("   int value: " + outList.getNextInt());
System.out.println("  long value: " + outList.getNextLong());
System.out.println("double value: " + outList.getNextDouble());
System.out.println("String value: " + outList.getNextString());
byte[] listBytes = outList.getNextBytes();
```

The NodeReference `getList()` method is used to fetch the ValueList from node `^myListNode`.

The type-specific methods `getNextInt()`, `getNextLong()`, `getNextDouble()`, `getNextString()`, and `getNextBytes()` each read the current list element as a specific type, and then advance the cursor to the next element in the list.

This example produces the following output:

```
int value: 13
long value: -4555666777
double value: 1.234
String value: "hello"
byte[] value: {72,105,33}
```
Controlling the List Cursor

When a ValueList is created, it contains a list cursor that points to the first element of the list. Each call to one of the read methods returns the current element and advances the cursor to the next element. After the last element has been read, the cursor will not point to a valid element until a new element is appended or the resetToFirst() method is called.

The following methods can also be used to control the cursor position:

- ValueList.length() — returns the number of elements in the list.
- ValueList.skipNext() — advances the cursor past the specified number of list elements without getting their values.
- ValueList.resetToFirst() — resets the cursor to the beginning of the list.

Reading a Nested ValueList

At the end of the previous example, the cursor points to the final element in the list, which is a nested ValueList. The next example will use getNextList() to return the nested list, the cursor will be moved to the final element of the new list, and the final element will be printed:

Reading ValueList Elements: reading a nested ValueList

```java
// Reuse the ValueList object to get the nested ValueList
outList.getNextList(outList);   // re-use list object to store the nested list
outList.skipNext(outList.length()-1);  // move cursor to last element in the new list
System.out.println("\nNested ValueList: last element is Long value " + outList.getNextLong());
```

The getNextList() method is called to get the nested list. Since the main list is no longer needed, the outList object is reused by specifying it as the place to store the nested list. The existing contents of outList are replaced by the nested list.

The code uses skipNext() and length() to move the cursor to the last element of the new list, and prints the value of the element, producing the following output:

```
Nested ValueList: last element is long value 9876543210
```

Reading ValueList Elements of Unknown Datatype

The following methods return one or more list elements as objects:

- getAll() — returns all elements from the list as an array of Object. After this method is called, the list cursor will be positioned after the last element in the list.
- getNextObject() — returns the next element from the list as an Object and advances the list cursor.

By default, both of these methods return each object as an instance of Integer, Long, Double, or String. They will return strings as instances of byte[] if the optional returnBytes parameter is set to true.

The following example gets each element of the list as an Object, uses instanceof to determine the datatype, and prints the type and value of the element.

Reading ValueList Elements: reading elements of unknown datatype

```java
System.out.println("Value in nested list:");
outList.resetToFirst();
for (int ii = 0; ii < outList.length(); ii++) {   
  Object item = outList.getNextObject();
  if (item instanceof String) 
    System.out.println(" String: \""+(String)item+"\"\n");
  else if (item instanceof Integer) 
    System.out.println("Integer: \""+(Integer)item.intValue())
  else if (item instanceof Long) 
    System.out.println(" Long: \""+(Long)item.longValue()+"L\"");
  else if (item instanceof Double) 
    System.out.println(" Double: \""+(Double)item.doubleValue());
  }
```
The `resetToFirst()` method moves the cursor to the first item in the list.

The `getNextObject()` method returns the current item as an Object, and advances the cursor to the next item.

This example generates the following printout:

```
Value in nested list:
  String: "world"
  Integer: 7
  Double: 3.1
  Long: 9876543210L
```

### 4.7 Accessing Multiple Global Arrays

Most of the examples in this chapter act on only one global array in a single namespace. In those examples, the namespace is specified when the connection is created and the global name is defined when the node reference is created. This section describes methods that allow your application to specify the global array that will be accessed by a node reference, browse a list of all global array names in a given namespace, and change the namespace accessed by the connection. The following topics are discussed:

- Accessing Global Names — describes how to change which global array is accessed by an instance of `NodeReference`.
- Using a Directory of Global Arrays — describes how to use the `GlobalsDirectory` class to browse a list of all global array names in a given namespace.
- Accessing Namespaces — describes how to read and change the namespace property of the connection.

#### 4.7.1 Accessing Global Names

The following `NodeReference` methods provide a way to access the global name property of the node reference:

- `NodeReference.setName()` — sets or changes the global name property in the node reference, allowing a single instance of `NodeReference` to create or access any global array in the current namespace. A global name is a String that must conform to certain rules (see “Global Array Naming Conventions”).
- `NodeReference.getName()` — returns a String containing the current global name.

Changes in the global name property do not affect the subscript list property. The node reference continues to use the current list of subscripts until the list is changed (as described in “Setting the Target Address”). The following example uses a single instance of `NodeReference` to create two global arrays. After the first array is created, the subscript list of node reference object `nodeRef` contains subscript "a". Although the global name and the node value are changed when the second array is created, the subscript list is never modified:

**Accessing Global Names: Creating a new array with the existing subscripts**

```java
nodeRef = myConn.createNodeReference("myGlobal");
nodeRef.appendSubscript("a");
nodeRef.set("new node"); // create ^myGlobal("a")
nodeRef.setName("mySecondGlobal");
nodeRef.set("second node"); // create ^mySecondGlobal("a")
```

Node reference object `NodeRef` points to root node `^myGlobal` when it is created. A subscript is added, and `set()` is called to create a new global array containing persistent node `^myGlobal("a")`.

The call to `setName()` changes the array name to `^mySecondGlobal`, but does not alter the subscript list of `NodeRef`. The next call to `set()` therefore creates a second global array containing persistent node `^mySecondGlobal("a")`. 

It is sometimes useful to create multiple instances of NodeReference that access different parts of the same global array. In the following example, the getName() method is used to specify the same global name for two different instances:

**Accessing Global Names: Using two node reference objects to access the same global array**

```java
NodeReference firstNodeRef = myConn.createNodeReference("myGlobal");
NodeReference secondNodeRef = myConn.createNodeReference(firstNodeRef.getName());
firstNodeRef.set("level zero"); // create ^myGlobal
firstNodeRef.appendSubscript("a");
System.out.println("firstNodeRef=" + firstNodeRef.getString() + ", secondNodeRef=" + secondNodeRef.getString()); // Prints: firstNodeRef="level one", secondNodeRef="level zero"
```

The first call to createNodeReference() assigns global name "myGlobal" to firstNodeRef. When the secondNodeRef object is created, getName() is used to assign the same global name as firstNodeRef.

The firstNodeRef object is used to create a global array containing two persistent nodes. After the second node is created, firstNodeRef points to ^myGlobal("a"). Since the address of secondNodeRef has never been changed, it still points to root node ^myGlobal.

### 4.7.2 Using a Directory of Global Arrays

This section describes how use methods of the GlobalsDirectory class to browse a list of all global array names in a given namespace. Instances of GlobalsDirectory are created by Connection.createGlobalsDirectory(). GlobalsException will be thrown if there is no connection to the database when createGlobalsDirectory() is called.

The following GlobalsDirectory methods are discussed here:

- **nextGlobalName()** — gets the global name that is next in ascending collation order.
- **previousGlobalName()** — gets the global name that is next in descending collation order.
- **refresh()** — updates the directory to reflect any additions or deletions since the object’s creation or most recent refresh.
- **close()** — releases the object's underlying resources to avoid resource leaks when the object is destroyed.

**Important:** Always call close() to avoid memory leaks

It is important to always call close() on instances ofGlobalsDirectory before they go out of scope or are reused. Failing to close them can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

Both nextGlobalName() and previousGlobalName() can take an optional String argument that specifies where the search is to start. If no string is specified, the search starts from the beginning of the directory list or from the most recently found global name. Both methods return an empty string when no more global array names are found. This iteration technique is identical to the one previously described for the methods used to iterate over the nodes of a global array (see “**Controlling the Starting Point and Order of Iteration**”).

### Browsing a Directory List

This section provides two examples using GlobalsDirectory. Both examples scan the directory list and print any global names that start with "myArray". The first example creates some test data and reads the contents of the namespace in descending order. The second example adds and deletes some global arrays in the same namespace, and then reads the contents of the changed namespace in ascending order. The examples produce the following combined output:

```
List in descending order after creating ^myArray1 and ^myArray2
^myArray2
^myArray1

List in ascending order after killing ^myArray1, adding ^myArray3, and refreshing
^myArray2
^myArray3
```
The Management Portal can display the current list of global arrays in a namespace, and the value of each node in an array. Go to [System Explorer] > [Globals] and select a namespace from the list displayed on the left.

The following example reads the entire directory list in descending collation order, printing all global names that start with "myArray".

### Using a Directory of Global Arrays: Creating the list and browsing in descending order

```java
nodeRef.setName("myArray1");
nodelRef.set("");
nodelRef.setName("myArray2");
nodelRef.set("");
System.out.println("List in descending order after creating ^myArray1 and ^myArray2");
GlobalsDirectory globDir = myConn.createGlobalsDirectory();
String globname = "n";
while (!globname.equals("") { 
    if (globname.startsWith("myArray")) System.out.println("  ^" + globname);
    globname = globDir.previousGlobalName();
}
```

Two global arrays, ^myArray1 and ^myArray2, are created in the current namespace. The NodeReference.setName() method (described previously in “Accessing Global Names”) specifies each global name before set() creates the new node.

The Connection.createGlobalsDirectory() method is used to create GlobalsDirectory object globDir, which starts with the cursor positioned just before the first possible item in collation order.

The first call to previousGlobalName() positions the cursor to "n" (greater than "myArray" in collation order). In the loop, each call to previousGlobalName() returns the next global name in descending order. The standard Java startsWith() method is used to ensure that only global names starting with "myArray" will be printed.

After the previous example has finished processing, the following example changes the contents of the User namespace before reading it again. It deletes ^myArray1, creates ^myArray3, refreshes the directory list, and prints names starting with "myArray" in ascending order from the beginning of the list.

### Using a Directory of Global Arrays: Refreshing the list and browsing in ascending order

```java
nodeRef.setName("myArray1");
nodeRef.kill();
nodelRef.setName("myArray3");
nodelRef.set("");
globDir.refresh();
System.out.println("List in ascending order after killing ^myArray1, adding ^myArray3, and refreshing");
globname = globDir.nextGlobalName(""); // Start before first name and ascend.
while (!globname.equals("")) { 
    if (globname.startsWith("myArray")) System.out.println("  ^" + globname);
    globname = globDir.nextGlobalName();
}
globDir.close();
```

Global array ^myArray1 is deleted, ^myArray3 is created, and refresh() is called to update the directory list.

The first call to nextGlobalName() sets the cursor before the first name in collation order by specifying an empty string (""). In the loop, each call to nextGlobalName() returns the next global name in ascending order. The two global names starting with "myArray" will be printed.

The call to close() destroys globDir and releases all associated resources.

### 4.7.3 Accessing Namespaces

A Caché namespace is a logical view of a physical database. Each installed Caché instance contains definitions of several namespaces (see “Configuring Namespaces” in the Caché System Administration Guide). The Globals API can access data in any available namespace of the Caché instance specified by the GLOSALS_HOME environment variable (see “Required Environment Variables for All Platforms”). A database on another machine can be accessed if a namespace
has been mapped to that database. In such cases, access speed will by limited by the physical connection to the database (via ECP rather than direct access to local hardware).

The following Connection methods are used to read and change the namespace reference of a connection object:

- `Connection.setNamespace()` — sets the namespace accessed by the underlying eXTreme connection.
- `Connection.getNamespace()` — returns the current namespace string.

The following example assumes that connection object `myConn` already exists, and may or may not be connected to the database. The code tests the connection and ensures that it connects to the User namespace.

**Accessing Namespaces: Testing a connection to a namespace**

```java
if(!myConn.isConnected()) myConn.connect("User", "_SYSTEM", "SYS");
else if (!myConn.getNamespace().equals("User")){
    System.out.print("Switching connection from " + myConn.getNamespace());
    myConn.setNamespace("User");
    System.out.println(" to " + myConn.getNamespace() + ".");
    // Prints: Switching connection from "[current namespace]" to "User".
}
```

The call to `isConnected()` checks to see if `myConn` is connected to a database. If not, `connect()` connects to the database associated with the User namespace. Otherwise, `getNamespace()` returns the current namespace name. If necessary, `setNamespace()` is called to switch to User.

**Note:** Since this call acts on the underlying eXtreme connection, every `Connection` variable in this process now accesses the User namespace (this even includes eXtreme XDO and XEP connections running in the same process, as described in “Using the Globals API with Other eXtreme APIs”). See “Creating a Connection” for a discussion of how the underlying eXtreme connection works.

### 4.8 Transactions and Locking

The following topics are discussed in this section:

- **Controlling Transactions** — describes using methods of the `Connection` class to process transactions.
- **Acquiring and Releasing Locks** — describes how to use the various lock methods.
- **Using Locks in a Transaction** — provides examples of locking within a transaction.

#### 4.8.1 Controlling Transactions

The `Connection` class provides the following methods to control transactions:

- `commit()` — commits one level of transaction.
- `startTransaction()` — starts a transaction (which may be a nested transaction).
- `transactionLevel()` — returns an int value indicating the current transaction level (0 if not in a transaction).
- `rollback()` — rolls back the specified number of transaction levels, or rolls back all levels of transaction if no number is specified.

The following example starts three levels of nested transaction, setting the value of a different node in each transaction level. All three nodes are printed to prove that they have values. The example then rolls back the second and third levels and commits the first level. All three nodes are printed again to prove that only the first node still has a value.
Controlling Transactions: Using three levels of nested transaction

```java
NodeReference nodeRef = myConn.createNodeReference("myGlobal");
myConn.startTransaction();
nodeRef.set("firstValue", myConn.transactionLevel());
// transactionLevel() is 1 and "myGlobal(1) = "firstValue"

myConn.startTransaction();
nodeRef.set("secondValue", myConn.transactionLevel());
// transactionLevel() is 2 and "myGlobal(2) = "secondValue"

myConn.startTransaction();
nodeRef.set("thirdValue", myConn.transactionLevel());
// transactionLevel() is 3 and "myGlobal(3) = "thirdValue"

System.out.println("Node values before rollback and commit:");
for (int ii=1;ii<4;ii++) {
    System.out.print("^myGlobal(" + ii + ") = ");
    if (nodeRef.exists(ii)) System.out.println(nodeRef.getString(ii));
    else System.out.println("<valueless>");
}   
// prints: Node values before rollback and commit:
//         ^myGlobal(1) = firstValue
//         ^myGlobal(2) = secondValue
//         ^myGlobal(3) = thirdValue

myConn.rollback(2);  // roll back 2 levels to transactionLevel 1
myConn.commit();  // transactionLevel() after commit will be 0

System.out.println("Node values after the transaction is committed:");
for (int ii=1;ii<4;ii++) {
    System.out.print("^myGlobal(" + ii + ") = ");
    if (nodeRef.exists(ii)) System.out.println(nodeRef.getString(ii));
    else System.out.println("<valueless>");
}
// prints: Node values after the transaction is committed:
//         ^myGlobal(1) = firstValue
//         ^myGlobal(2) = <valueless>
//         ^myGlobal(3) = <valueless>
```

4.8.2 Acquiring and Releasing Locks

The following methods of NodeReference are used to acquire and release locks. Both methods take a lockType argument to specify whether the lock is shared or exclusive:

- **acquireLock()** — Takes lockType, lockMode, and optional subscripts arguments, and locks the node. The lockMode argument specifies whether any previously held locks should be released. This method will time out after a predefined interval (see “Controlling the Timeout Interval”) if the lock cannot be acquired.

- **releaseLock()** — Takes lockType, releaseMode, and optional subscripts arguments, and releases the lock on a node. The releaseMode argument specifies whether the lock should be released immediately or at the end of the transaction. Connection methods close() or releaseAllLocks() (see “Using Locks in a Transaction”) will release all currently held locks.

The following argument values can be used:

- **lockType** — both acquireLock() and releaseLock() use these values to specify whether the lock is shared or exclusive:
  - NodeReference.SHARED_LOCK — A shared lock allows other callers to acquire shared locks on the node, but prevents any caller from acquiring an exclusive lock on it. The node value cannot be changed until all shared locks are released.
  - NodeReference.EXCLUSIVE_LOCK — An exclusive lock prevents other callers from acquiring a lock on the node, allowing the node value to be changed safely.

- **lockMode** — acquireLock() uses these values to specify whether previously acquired locks should be released:
  - NodeReference.LOCK_INCREMENTALLY — An incremental lock does not release any previously held locks.
  - NodeReference.LOCK_NON_INCREMENTALLY — A non-incremental lock releases all previously held locks.
• **releaseMode** — releaseLock() uses these values to specify when the lock should be released:
  - NodeReference.RELEASE_IMMEDIATELY — Release the lock immediately.
  - NodeReference.RELEASE_AT_TRANSACTION_END — Wait until the end of the current transaction.

• **subscripts** — acquireLock() and releaseLock() can both use optional subscript arguments to lock or release a subnode of the target node (see “Addressing a Subnode of the Target”).

The following example will lock and release node ^myGlobal("a",1). The target address in nodeRef is ^myGlobal, so both methods use subscript arguments to specify the subnode.

**Acquiring and Releasing Locks: Locking and releasing a node**

```java
nodeRef.acquireLock(NodeReference.SHARED_LOCK, NodeReference.LOCK_INCREMENTALLY,"a",1);
// perform transaction on locked node here...
nodeRef.releaseLock(NodeReference.SHARED_LOCK, NodeReference.RELEASE_IMMEDIATELY,"a",1);
```

**Note:** You can use the Management Portal to examine locks. Go to [System Operation] > [Locks] to see a list of the locked items on your system.

### 4.8.2.1 Controlling the Timeout Interval

A NodeReference object maintains an internal timeout setting that specifies how long the acquireLock() method should wait to acquire a lock before timing out and throwing LockException. The default interval is 10 seconds.

• **getOption()** — takes an option argument and returns the current lock timeout interval in seconds.

• **setOption()** — takes option and value arguments, and sets the number of seconds that acquireLock() should wait for a lock before timing out.

The following argument values can be used:

• **option** — used by both getOption() and setOption(). Currently, the only valid argument value is NodeReference.LOCK_TIMEOUT.

• **value** — an int specifying the timeout value for setOption(). Any int value may be specified, and the following predefined values are available:
  - NodeReference.DEFAULT_LOCK_TIMEOUT — Default value is 10.
  - NodeReference.NO_LOCK_TIMEOUT — Value is -1, which specifies that the call will never time out (acquireLock() will wait forever).

The following example reads the timeout interval, and sets it to 5 seconds if it is less than 5.

**Acquiring and Releasing Locks: Controlling the timeout interval**

```java
if (nodeRef.getOption(NodeReference.LOCK_TIMEOUT)<5) {
    nodeRef.setOption(NodeReference.LOCK_TIMEOUT, 5);
}
```

### 4.8.3 Using Locks in a Transaction

This section demonstrates incremental locking within a transaction, using the methods previously described (see “Controlling Transactions” and “Acquiring and Releasing Locks”). You can see a list of the locked items on your system by opening the Management Portal and going to [Home] > [Locks]. The calls to read(pressEnter) in the following code will pause execution so that you can look at the list whenever it changes.
The following Connection methods will release all currently held locks:

- `releaseAllLocks()` — releases all locks currently held by this connection.
- `close()` — releases all locks and other connection resources before it closes the connection.

### Using Locks in a Transaction: Using incremental locking

```java
byte[] pressEnter = new byte[4];
NodeReference nodeRef1 = myConn.createNodeReference("nodeRef1");
nodeRef1.appendSubscript("my-node");
NodeReference nodeRef2 = myConn.createNodeReference("nodeRef2");
nodeRef2.appendSubscript("shared-node");
try {
    myConn.startTransaction();
    // lock ^nodeRef1("my-node") exclusively
    nodeRef1.acquireLock(NodeReference.EXCLUSIVE_LOCK, NodeReference.LOCK_INCREMENTALLY);
    // lock ^nodeRef2 shared
    nodeRef2.acquireLock(NodeReference.SHARED_LOCK, NodeReference.LOCK_INCREMENTALLY);
    System.out.println("Exclusive lock on ^nodeRef1("my-node") and shared lock on ^nodeRef2");
    System.out.println("Press return to release locks individually");
    System.in.read(pressEnter); // Wait for user to press Return
    // release ^nodeRef1("my-node") after transaction
    nodeRef1.releaseLock(NodeReference.EXCLUSIVE_LOCK, NodeReference.RELEASE_AT_TRANSACTION_END);
    // release ^nodeRef2 immediately
    nodeRef2.releaseLock(NodeReference.SHARED_LOCK, NodeReference.RELEASE_IMMEDIATELY);
    System.out.println("Press return to commit transaction");
    System.in.read(pressEnter);
    myConn.commit();
} catch (LockException e) { System.out.println(e.getMessage()); }
catch (java.io.IOException e) { System.out.println(e.getMessage()); } 
```

The following example demonstrates non-incremental locking.

### Using Locks in a Transaction: Using non-incremental locking

```java
// lock ^nodeRef1("my-node") non-incremental
nodeRef1.acquireLock(NodeReference.EXCLUSIVE_LOCK, NodeReference.LOCK_NON_INCREMENTALLY);
System.out.println("Exclusive lock on ^nodeRef1("my-node")", return to lock ^lockRef2
non-incrementally");
System.in.read(pressEnter);

// lock ^nodeRef2 shared non-incremental
nodeRef2.acquireLock(NodeReference.SHARED_LOCK, NodeReference.LOCK_NON_INCREMENTALLY);
System.out.println("Verify that only ^nodeRef2 is now locked, then press return");
System.in.read(pressEnter);
```

The following example demonstrates releasing all incremental locks.

### Using Locks in a Transaction: Using releaseAllLocks() to release all incremental locks

```java
// lock ^nodeRef1("my-node") shared incremental
nodeRef1.acquireLock(NodeReference.SHARED_LOCK, NodeReference.LOCK_INCREMENTALLY);

// lock ^nodeRef2 exclusive incremental
nodeRef2.acquireLock(NodeReference.EXCLUSIVE_LOCK, NodeReference.LOCK_INCREMENTALLY);
System.out.println("Two locks are held (one with lock count 2), return to release both locks");
System.in.read(pressEnter);
myConn.releaseAllLocks();
System.out.println("Verify both locks have been released");
System.in.read(pressEnter);
```
4.9 Calling Caché Methods

The following Connection methods call Caché ObjectScript class methods:

- **callClassMethod()** — calls the specified Caché ObjectScript class method. Takes String arguments for `className` and `methodName`, plus 0 or more arguments that will be passed to the Caché method. Returns the Caché method return value as an Object that will be an instance of String, int, long, or double.

- **callBytesClassMethod()** — identical to `callClassMethod()` except that string values are returned as byte[].

- **callVoidClassMethod()** — calls the specified Caché ObjectScript void class method. Takes String arguments for `className` and `methodName`, plus 0 or more arguments that will be passed to the Caché method.

The following Connection methods call Caché functions and procedures (see “User-defined Code” in Using Caché ObjectScript):

- **callFunction()** — calls the specified Caché ObjectScript function. Takes String arguments for `functionName` and `routineName`, plus 0 or more arguments that will be passed to the Caché method. Returns the Caché function return value as an Object that will be an instance of String, int, long, or double.

- **callBytesFunction()** — identical to `callFunction()` except that string values are returned as byte[].

- **callProcedure()** — calls the specified Caché ObjectScript procedure. Takes String arguments for `procedureName` and `routineName`, plus 0 or more arguments that will be passed to the Caché method.

4.10 Globals API Requirements and Conventions

The following topics are discussed in this section:

- **Global Array Naming Conventions** — provides detailed rules for naming global arrays and subscripts.

- **Using the Globals API with Other eXTreme APIs** — describes how to use the Globals API, XEP, and XDO in the same application.

- **Understanding Node Order in Multilevel Global Arrays** — discusses how the structure of multidimensional global arrays influences the order in which they are traversed.

- **Considerations for Safe Threading** — discusses best practices for safe use of multithreading.

4.10.1 Global Array Naming Conventions

This section describes how the standard Caché global naming conventions apply to the Globals API. See “Logical Structure of Globals” in Using Caché Globals for details about how the same conventions are enforced in Caché ObjectScript.

Global names must obey the following rules:

- Global names cannot contain Unicode characters.

- The first character in a global name can be an letter or a percent character (%). A percent character can only be used as the first character, and should be avoided in most cases (see “Global Variable Names to Avoid” in the Caché Programming Orientation Guide).

- The other characters of a global name may be letters, numbers, or the period (.) character. The period character cannot be used as the first or last character of a global name.
• Global names are case-sensitive, and consist of up to 31 significant characters. You can specify global names that are significantly longer, but the extra characters are ignored.

Subscript identifiers must obey the following rules:

• A subscript identifier can be an int, long, double, or String.

• Numeric identifiers are converted to their canonical form unless the number is in quotes. For example, leading and trailing zeros would be stripped from double value 00123.10. However, if the same number is specified as a quoted numeric string ("00123.10"), the zeros will be retained and the value will be collated as a string, not a number.

• String identifiers are case-sensitive, and can be any string except the empty string (""). They can include characters of all types, including spaces, non-printing characters, and Unicode characters.

• There is no fixed limit on the number of subscripts, but a node address (global name plus all of its subscripts) is limited to 1023 bytes. Unicode characters in subscripts count as two or more bytes (see “Determining the Maximum Length of a Subscript” in Using Caché Globals).

4.10.2 Using the Globals API with Other eXTreme APIs

The XDO, XEP, and eXTreme JDBC APIs may be used in the same applications as the Globals API, allowing object access and SQL to be part of the same transaction context. There are some common installation and configuration requirements for all eXTreme APIs (see “Installation and Configuration”).

All of these APIs use the same underlying connection, but each one has its own methods for connecting to and disconnecting from Caché. The following rules apply when using more than one of these in the same application:

• Opening connections

If you have already opened an eXTreme JDBC connection, the Globals API can be used without explicitly opening a Globals connection. If you do not want your Globals code to be dependent on the JDBC connection, you can connect to Globals explicitly by calling Connection.connect() either before or after the JDBC connection is opened. Both connections must specify the same namespace, or an exception is thrown.

• Changing namespaces

In Globals applications that also use another eXTreme API, a call to Connection.setNamespace() will change the effective namespace for all APIs. Subsequent SQL statements or eXTreme API calls will operate against the new namespace, and may fail if they expect to use resources that only exist in the old namespace. An eXTreme JDBC connection cannot be opened after setNamespace() is called, even if the same namespace is specified for that connection.

• Closing connections

You must explicitly close all connections that you explicitly open. Globals and JDBC connections are closed by calling Connection.close(). An XDO connection is closed by calling DatabaseConnection.disconnect(). You can call these in either order, but the underlying connection is not closed (and associated resources are not released) until the last one is called.

4.10.3 Collation Order in Multilevel Global Arrays

The nodes in a global array do not have to be added in a sequential or hierarchical order, but since a global array is essentially a tree structure, a hierarchy is automatically generated whenever persistent nodes are added. For example, a global array might contain three persistent nodes: `^myGlobal("a","23")`, `^myGlobal("a"," x")`, and `^myGlobal("b")`. The subscript lists are arbitrary, and the nodes can be added to the array in any order. However, the pointer structure of the array must also
contain two valueless nodes: root node `myGlobal`, and level 1 node `myGlobal("a")`, which is the parent of the first two persistent nodes and the sibling of the third. These nodes create the following hierarchy:

```
^myGlobal                      (valueless root node)
  ^myGlobal("a")              (valueless level 1 node)
    ^myGlobal("a","23") = value
    ^myGlobal("a"," x") = value  (second subscript contains a space character!)
  ^myGlobal("b") = value
```

As demonstrated in this example, the hierarchy is based on node level and on the collation sequence of the subscripts at each level.

In the physical database, valueless nodes `myGlobal` and `myGlobal("a")` are created automatically when a value is assigned to one of their child nodes. The valueless nodes exist only as part of the pointer structure, and no extra space is allocated for data. Although any attempt to read a value from them would throw an exception, they can still be used by the Globals iteration methods to traverse the array.

Child nodes under the same parent are organized by their subscripts in collation order, with numbers first, followed by alphabetic strings. For example, the following list of level 1 nodes is in collation order:

**Collation Order in Caché Databases**

```
^myGlobal("-23.1")  (subscript is a string containing a numeric value)
^myGlobal("99")     (subscript is a string containing only digits)
^myGlobal(" ")      (subscript is a space character)
^myGlobal(" 22")    (subscript starts with a space)
^myGlobal("0123")   (subscript contains a leading zero)
^myGlobal("11 ")    (subscript contains a trailing space)
^myGlobal("23-2")   (subscript is a numeric expression, not a value)
```

The first two subscripts are numeric values, and therefore come before all alphabetic subscripts. A space character (" ") is the first alphabetic subscript in this list. For more information on collation order, see “Sorting Data within Globals” in Using Caché Globals.

4.10.4 Considerations for Safe Threading

Separate instances of Globals API classes can be used concurrently in multiple threads, in a thread-safe manner. Individual instances of Globals API classes are not thread safe, in the sense that some of their methods modify their state in a way that persists between method calls, so that the same sequence of method calls in one thread may have different results, depending on whether another thread calls methods of the same instance during the course of the sequence. For example, a call to NodeReference.appendSubscript() in another thread changes which node is referenced by the NodeReference instance, which could cause a call to NodeReference.set() to set the value of a different node than the one intended.

The single Connection instance may be used safely in multiple threads with these caveats:

- Calls to the transaction methods startTransaction(), commit(), and rollback(), and releaseAllLocks() affect the transaction and locking state of the Connection for all threads.
- Connection.setNamespace() changes the current namespace for all threads. If there is an active XEP EventPersister in the same application, its current namespace is also changed by a call to setNamespace().
- Connection.close() closes the connection for all threads. Subsequent method calls on the Connection instance, other than connect(), will throw exceptions indicating that connect() must be called before they can be invoked.
Quick Reference for eXTreme Classes

This chapter is a quick reference for the classes that are most important to an understanding of the following Caché eXTreme APIs:

- **XEP Quick Reference**
  The `com.intersys.xep` package contains the public API described in Using eXTreme Event Persistence.

- **XDO Quick Reference**
  The `com.intersys.xdo` package contains the public API described in Using eXTreme Dynamic Objects.

- **Globals API Quick Reference**
  The `com.intersys.globals` package contains the public API described in Using the Globals API.

**Note:** This is not the definitive reference for these APIs. For the most complete and up-to-date information, see the JavaDoc for the InterSystems Java Connectivity API, located in `<cache-root>/dev/java/doc/index.html`.

### 5.1 XEP Quick Reference

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

- **Class PersisterFactory** — provides a factory method to create `EventPersister` objects.

- **Class EventPersister** — encapsulates an XEP database connection. It provides methods that set XEP options, establish an XEP connection or get an existing connection object, import schema, produce XEP event objects, call Caché functions and methods on the server, 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<>** — encapsulates a query that retrieves individual events of a specific type from the database for update or deletion.

- **Class EventQueryIterator<>** — provides an alternative to `EventQuery<>` for retrieving, updating and deleting XEP events, using methods similar to those in Java `Iterator`.

- **Interface InterfaceResolver** — resolves the actual type of a field during flat schema importation if the field was declared as an interface.

- **Class XEPException** — is the exception thrown by most XEP methods.
5.1.1 List of XEP Methods

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

**PersisterFactory**
- `createPersister()` — creates a new `EventPersister` object.

**EventPersister**
- `callClassMethod()` — calls a Caché class method.
- `callFunction()` — calls a Caché function.
- `callProcedure()` — calls a Caché procedure.
- `callVoidClassMethod()` — calls a Caché class method with no return value.
- `close()` — releases all resources held by this instance.
- `commit()` — commits one level of transaction.
- `connect()` — connects to Caché using the arguments specified.
- `deleteClass()` — deletes a Caché class.
- `deleteExtent()` — deletes all objects in the given extent.
- `getConnection()` — Returns the underlying eXTreme connection.
- `getEvent()` — returns an event object that corresponds to the class name supplied.
- `getInterfaceResolver()` — returns the currently specified instance of `InterfaceResolver`.
- `getJDBCConnection()` — returns a JDBC connection object.
- `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.
- `startTransaction()` — starts a transaction (which may be a nested transaction).

**Event**
- `close()` — releases all resources held by this instance.
- `createQuery()` — creates a `EventQuery<>` 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<>**

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

**EventQueryIterator<>**

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

**InterfaceResolver**

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

### 5.1.2 Class PersisterFactory

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

**PersisterFactory() Constructor**

Creates a new instance of `PersisterFactory`.

```java
PersisterFactory()
```

**createPersister()**

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

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

**see also:**

Creating and Connecting an `EventPersister`
5.1.3 Class EventPersister

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

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

**EventPersister() Constructor**

Creates a new instance of EventPersister.

EventPersister()

callClassMethod()

EventPersister.callClassMethod() — calls a Caché ObjectScript class method, passing 0 or more arguments and returning the method’s return value as an instance of String, Integer, Long, or Double (use callVoidClassMethod() to call a method that doesn’t return a value).

Object callClassMethod(String className, String methodName, Object... args)

parameters:

- className — fully qualified name of the Caché class to which the called method belongs.
- methodName — name of the Caché class method.
- args — a list of 0 or more arguments to pass to the method.

Arguments may be of any of the types String, Integer, Long, Double, byte[], or com.intersys.globals.ByteArrayRegion. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing null for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

callFunction()

EventPersister.callFunction() calls a Caché ObjectScript function (see “User-defined Code” in Using Caché ObjectScript).

Object callFunction(String functionName, String routineName, Object... args)

parameters:

- functionName — name of the function.
- routineName — name of the routine containing the function.
- args — arguments to be passed.

Arguments may be of any of the types String, Integer, Long, Double, byte[], or com.intersys.globals.ByteArrayRegion. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing null for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

callProcedure()

void callProcedure(String procedureName, String routineName, Object... args)

parameters:
• procedureName — name of the procedure.
• routineName — name of the routine containing the procedure.
• args — arguments to be passed.

Arguments may be of any of the types String, Integer, Long, Double, byte[], or com.intersys.globals.ByteArrayRegion. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing null for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

callVoidClassMethod()

EventPersister.callVoidClassMethod() — calls a Caché ObjectScript class method with no return value, passing 0 or more arguments. This method may be used to call any Caché class method (regardless of whether it normally returns a value) when the caller does not need the return value. Use callClassMethod() to call a method that returns a value.

void callVoidClassMethod(String className, String methodName, Object... args)

parameters:
• className — fully qualified name of the Caché class to which the called method belongs.
• methodName — name of the Caché class method.
• args — list of 0 or more arguments to pass to the method.
• Arguments may be of any of the types String, Integer, Long, Double, byte[], or com.intersys.globals.ByteArrayRegion. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing null for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

close()

EventPersister.close() releases all resources held by this instance.

void close()

It is important to always call close() on an instance of EventPersister before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

commit()

EventPersister.commit() commits one level of transaction

void commit()

connect()

EventPersister.connect() connects to Caché using either an eXTreme connection or a TCP/IP connection, depending on the arguments specified. If only namespace, username, and password arguments are specified, an in-process eXTreme connection is established. If host and port are also specified, a TCP/IP connection is established.

void connect(String namespace, String username, String password)
void connect(String host, int port, String namespace, String username, String password)
parameters:
- namespace — namespace to be accessed.
- username — user name for this connection.
- password — password for this connection.
- host — host address for TCP/IP connection.
- port — port number for TCP/IP connection.

*see also:*
Creating and Connecting an EventPersister

**deleteClass()**

EventPersister.deleteClass() deletes a Caché 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).

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

**parameter:**

- className — name of the class to be deleted.

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

*see also:*
“Deleting Test Data” in Accessing Stored Events

**deleteExtent()**

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

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

**parameter:**

- 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

**getConnection()**

EventPersister.getConnection() — returns the instance of com.intersys.globals.Connection underlying an EventPersister eXTreme connection. Throws an exception if the EventPersister has a TCP/IP connection.

```java
com.intersys.globals.Connection getConnection()
```

*see also:*
Creating and Connecting an EventPersister
**getEvent()**

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

```java
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.

```java
InterfaceResolver getInterfaceResolver()
```

*see also:*
setInterfaceResolver(), importSchema()

**getJDBCConnection()**

EventPersister.getJDBCConnection() returns the JDBC Connection object underlying an EventPersister connection.

```java
java.sql.Connection getJDBCConnection()
```

*see also:*
Creating and Connecting an EventPersister

**getTransactionLevel()**

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

```java
int getTransactionLevel()
```
importSchema()

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

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

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

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

see also:
Importing a Schema

importSchemaFull()

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

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

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

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

see also:
Importing a Schema

rollback()

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

void rollback()
void rollback(int level)

parameter:
• level — optional number of levels to roll back.

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

setInterfaceResolver()

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

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

parameters:

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

see also:

getInterfaceResolver(), importSchema()

startTransaction()

EventPersister.startTransaction() starts a transaction (which may be a nested transaction)

```java
void startTransaction()
```

5.1.4 Class Event

Class com.intersys.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.

```java
void close()
```

It is important to always call close() on an instance of Event before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

createQuery()

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

```java
<E> EventQuery<E> 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.

```java
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.

```java
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.

```java
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”).

```java
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()`.

```java
void stopIndexing()
```
see also:

Controlling Index Updating

store()

Event.store() stores a Java object or array of objects as persistent events. There is no significant performance difference between passing an array and passing individual objects in a loop, but all objects in the array must be of the same type. 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.

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

parameters:

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

updateObject()

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

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

parameter:

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

see also:

Accessing Stored Events

waitForIndexing()

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

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

parameter:

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

see also:

Controlling Index Updating

5.1.5 Class EventQuery<>

Class com.intersys.xep.EventQuery<> can be used to retrieve, update and delete individual events from the database.
**close()**

`EventQuery<> .close()` releases all resources held by this instance.

```java
double close()
```

It is important to always call `close()` on an instance of `EventQuery<>` before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

**deleteCurrent()**

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

```java
double deleteCurrent()
```

*see also:*

Processing Query Data

**execute()**

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

```java
double execute()
```

*see also:*

Creating and Executing a Query

**getAll()**

`EventQuery<> .getAll()` returns all rows in the resultset as a single array.

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

This is a convenience method that can be used to get all data that matches the given query. It cannot be combined with calls to `getNext()`. There is a high cost associated with maintaining a list of objects. For best performance, or if you need to process one element at the time only, choose `getNext()` instead.

*see also:*

Processing Query Data

**getFetchLevel()**

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

```java
double getFetchLevel()
```

**getIterator()**

`EventQuery<> .getIterator()` returns an `EventQueryIterator<>` that can be used to iterate over query results (see “Using `EventQueryIterator<>`”).

```java
EventQueryIterator<E> getIterator()
```
getNext()

EventQuery<>.getNext() takes an object of the target class as an argument and returns the next item in the resultset (or returns the first item in the resultset if the argument is null). Returns null if there are no more items in the resultset.

E getNext(E obj)

parameter:
• obj — the object that the iteration cursor currently points to (or null to return the first item in the resultset).

see also:
Processing Query Data

setFetchLevel()

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

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

void setFetchLevel(int level)

parameter:
• level — fetch level constant (defined in the Event class).

Supported fetch levels are:
• Event.FETCH_LEVEL_ALL — default, all fields populated
• Event.FETCH_LEVEL_DATATYPES_ONLY — only datatype fields filled in
• Event.FETCH_LEVEL_NO_ARRAY_TYPES — all arrays will be skipped
• Event.FETCH_LEVEL_NO_OBJECT_TYPES — all object types will be skipped
• Event.FETCH_LEVEL_NO_COLLECTIONS — all collections will be skipped

setParameter()

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

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

parameters:
• index — the index of this parameter within the query statement.
• value — the value to be used for this query.

see also:
Creating and Executing a Query

updateCurrent()

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

void updateCurrent(E obj)
parameter:
  • obj — the Java object that will replace the current event.

see also:
Processing Query Data

5.1.6 Class EventQueryIterator<>

Class com.intersys.xep.EventQueryIterator<> is an alternative way of retrieving, updating and deleting XEP events (the same task can be also achieved by direct use of EventQuery<> methods).

hasNext()

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

boolean hasNext()

del

del

next()

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

E next()

remove()

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

void remove()

set()

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

void set(E obj)

parameter:
  • obj — an object of the target class that will replace the last event fetched by next().

5.1.7 Interface InterfaceResolver

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

getImplementationClass()

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

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

parameters:
  • declaringClass — class where fieldName is declared as interfaceClass.
  • fieldName — name of the field in declaringClass that has been declared as an interface.
  • interfaceClass — the interface to be resolved.
5.1.8 Class XEPException

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

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

5.2 XDO Quick Reference

This section is a reference for the XDO API (eXTreme Dynamic Objects — namespace com.intersys.xdo). See Using eXTreme Dynamic Objects for a details on how to use the API. It contains the following public classes and interfaces:

- Class DatabaseConnectionFactory — supplies the method that returns a DatabaseConnection object.
- Interface DatabaseConnection — encapsulates a physical and logical connection to a Caché database. It provides methods to create XDO instances, update indexes, and control transactions.
- Interface DynamicObject — encapsulates a reference to a persistent Caché object of a specified class. It provides methods to access and manipulate a persistent object in the database.
- XDO Exception Classes— XDOException is thrown by most DatabaseConnection and DynamicObject methods. XDONullValueException is thrown if the application attempts to get an int, double, or long when the node value is null.

5.2.1 List of XDO Methods

The following classes, interfaces, and methods of the XDO API are described in this reference:

DatabaseConnectionFactory
- createJNI.DatabaseConnection() — returns a DatabaseConnection object.

DatabaseConnection
- buildIndices() — rebuilds the indices for a specified class.
- commit() — commits one level of transaction.
- connect() — connects to the Caché database.
- createNew() — creates an empty DynamicObject of a specified persistent class.
- disconnect() — disconnects from the Caché database.
- getDefaultConcurrencyLevel() — returns the current default locking level.
- openId() — creates a DynamicObject that contains a copy of data from a persistent object of a specified class.
- rollback() — rolls back all levels of transaction.
- setDefaultConcurrencyLevel() — specifies the default locking level during transactions.
- startTransaction() — starts a transaction (which may be a nested transaction).
- transactionLevel() — returns the current transaction level (0 if not in a transaction).
DynamicObject

- **cleanup()** — releases the object's underlying resources to avoid resource leaks when the object is destroyed.
- **delete()** — deletes the target object from the database.
- **getBigDecimal()** — returns the value of a property as a BigDecimal variable.
- **getDate()** — returns the value of a property as a Date variable.
- **getDouble()** — returns the value of a property as a double variable.
- **getDoubleWrapper()** — returns the value of a property as a Double variable.
- **getId()** — returns the Id of the database object corresponding to this dynamic object.
- **getInt()** — returns the value of a property as an int variable.
- **getIntegerWrapper()** — returns the value of a property as an Integer variable.
- **getLong()** — returns the value of a property as a long variable.
- **getLongWrapper()** — returns the value of a property as a Long variable.
- **getPropertyNumber()** — given a property name, returns an integer indicating the property number of the property.
- **getString()** — returns the value of a property as a String variable.
- **getTime()** — returns the value of a property as a Time variable.
- **getTimestamp()** — returns the value of a property as a Timestamp variable.
- **insert()** — creates a new target database object that contains a copy of the dynamic object's data.
- **isNull()** — tests whether a DynamicObject returned by openId() is null.
- **save()** — saves the dynamic object's data to a persistent database object.
- **set()** — sets the specified property of a dynamic object to the specified value.
- **update()** — updates the data in the target database object.

### 5.2.2 Class DatabaseConnectionFactory

Class `com.intersys.xdo.DatabaseConnectionFactory` provides a method that creates instances of DatabaseConnection. See the section on “Connecting and Disconnecting” in “Using eXTreme Dynamic Objects” for information on usage. See the Caché JavaDoc (<cache-root>/dev/java/doc/index.html) for more details on connection options.

**createJNIDatabaseConnection()**

`DatabaseConnectionFactory.createJNIDatabaseConnection()` returns a new instance of DatabaseConnection.

```java
public static com.intersys.internal.lcbjni.LCBJNIDatabaseConnection createJNIDatabaseConnection()
```

### 5.2.3 Interface DatabaseConnection

Interface `com.intersys.xdo.DatabaseConnection` represents a physical and logical connection to a Caché database. It provides methods to create XDO instances, update indexes, and control transactions.
**buildIndices()**

`DatabaseConnection.buildIndices()` rebuilds the indices for a specified class. This method would be used after a series of saves/inserts/deletes/updates with `deferIndices=true`.

```java
void buildIndices(String className) throws XDOException
```

**Parameter:**
- `className` — class name of the extent to be indexed.

**commit()**

`DatabaseConnection.commit()` commits one level of transaction.

```java
void commit() throws XDOException
```

**connect()**

`DatabaseConnection.connect()` opens an eXTreme database connection.

```java
void connect(String namespace, String user, String password) throws XDOException
```

**Parameters:**
- `namespace` — namespace to be accessed.
- `user` — user name for this connection.
- `password` — password for this connection.

**createNew()**

`DatabaseConnection.createNew()` returns an empty DynamicObject of the specified persistent class.

```java
DynamicObject createNew(String className) throws XDOException
```

**Parameter:**
- `className` — class name of the object to be created.

**disconnect()**

`DatabaseConnection.disconnect()` terminates the eXTreme database connection.

```java
void disconnect() throws XDOException
```

**getDefaultConcurrencyLevel()**

`DatabaseConnection.getDefaultConcurrencyLevel()` returns an int that indicates the current default locking level.

```java
int getDefaultConcurrencyLevel() throws XDOException
```

**openId()**

`DatabaseConnection.openId()` returns a DynamicObject that contains a copy of data from an existing object of a specified class.

```java
DynamicObject openId(String className, String id) throws XDOException
DynamicObject openId(String className, String id, int concurrency, int timeout) throws XDOException
```

**Parameters:**
• className — class name of the object to be opened.
  id — Id of the object to be opened.
  concurrency — locking level to be used for this object.
  timeout — number of seconds to wait for access to the object.

rollback()

DatabaseConnection.rollback() rolls back all levels of transaction.

void rollback() throws XDOException

setDefaultConcurrencyLevel()

DatabaseConnection.setDefaultConcurrencyLevel() specifies the default locking level during transactions.

void setDefaultConcurrencyLevel(int level) throws XDOException

  parameter:
  • level — specifies the default locking level.

startTransaction()

DatabaseConnection.startTransaction() starts a transaction (which may be a nested transaction).

void startTransaction() throws XDOException

transactionLevel()

DatabaseConnection.transactionLevel() returns an int that indicates the current transaction level (0 if not in a transaction).

int transactionLevel() throws XDOException

5.2.4 Interface DynamicObject

Interface com.intersys.xdo.DynamicObject encapsulates a reference to a persistent Caché object of a specified class. The in-process DynamicObject has the same properties as the Caché class, and can update the data in the referenced object. It also provides methods to create a new database object or to delete an existing object from the database.

cleanup()

DynamicObject.cleanup() releases the object’s underlying memory and other resources to avoid problems such as memory leaks when the object is destroyed.

void cleanup()

delete()

DynamicObject.delete() deletes the target object from the database.

void delete() throws XDOException
void delete(boolean deferIndices, int timeout) throws XDOException

  parameters:
  • deferIndices — specifies if indexes should be rebuilt when the object is deleted.
• timeout — number of seconds to wait if the object is locked when the delete is attempted. A value of \(-1\) indicates that the default timeout should be used, and \(0\) means no wait.

This method does not affect the data contained in the dynamic object. If the call is successful, the dynamic object becomes detached, but still exists in memory and retains all of its data.

**getBigDecimal()**

`DynamicObject.getBigDecimal()` returns the value of the specified property as a `BigDecimal` variable.

```java
BigDecimal getBigDecimal(int propertyNumber) throws XDOException
BigDecimal getBigDecimal(String propertyName) throws XDOException
```

**parameters:**

• `propertyNumber` — property number of the property.
• `propertyName` — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.

**getDouble()**

`DynamicObject.getDouble()` returns the value of the specified property as a `double` variable.

```java
double getDouble(int propertyNumber) throws XDOException, XDONullValueException
double getDouble(String propertyName) throws XDOException, XDONullValueException
```

**parameters:**

• `propertyNumber` — property number of the property.
• `propertyName` — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.

**getDoubleWrapper()**

`DynamicObject.getDoubleWrapper()` returns the value of the specified property as a `Double` variable.

```java
Double getDoubleWrapper(int propertyNumber) throws XDOException
Double getDoubleWrapper(String propertyName) throws XDOException
```

**parameters:**
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.

getId()
DynamicObject::getId() returns a String containing the Id of the database object referenced by this dynamic object.

String getId()

The dynamic object will have an Id only if it was created by `openId()` or was saved after being created by `createNew()`.

getInt()
DynamicObject::getInt() returns the value of the specified property as an int variable.

int getInt(int propertyNumber) throws XDOException, XDONullValueException
int getInt(String propertyName) throws XDOException, XDONullValueException

parameters:
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.

getIntegerWrapper()
DynamicObject::getIntegerWrapper() returns the value of the specified property as an Integer variable.

Integer getIntegerWrapper(int propertyNumber) throws XDOException
Integer getIntegerWrapper(String propertyName) throws XDOException

parameters:
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.

getLong()
DynamicObject::getLong() returns the value of the specified property as a long variable.

long getLong(int propertyNumber) throws XDOException, XDONullValueException
long getLong(String propertyName) throws XDOException, XDONullValueException

parameters:
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.
getLongWrapper()

DynamicObject.getLongWrapper() returns the value of the specified property as a Long variable.

Long getLongWrapper(int propertyNumber) throws XDOException
Long getLongWrapper(String propertyName) throws XDOException

parameters:
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see getPropertyName()). Throws XDOException if the property is the wrong datatype.

getPropertyNumber()

DynamicObject.getPropertyNumber() returns an integer indicating the property number of the property with the specified name.

int getPropertyNumber(String propertyName) throws XDOException

parameter:
• propertyName — name of the property.

If the same property will be referenced more than once, it is much more efficient to specify a property by number than by name. See “Using Transactions” for examples that use this method.

getString()

DynamicObject.getString() returns the value of the specified property as a String variable.

String getString(int propertyNumber) throws XDOException
String getString(String propertyName) throws XDOException

parameters:
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see getPropertyName()). Throws XDOException if the property is the wrong datatype.

getTime()

DynamicObject.getTime() returns the value of the specified property as a Time variable.

Time getTime(int propertyNumber) throws XDOException
Time getTime(String propertyName) throws XDOException

parameters:
• propertyNumber — property number of the property.
• propertyName — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see getPropertyName()). Throws XDOException if the property is the wrong datatype.
**getTimestamp()**

DynamicObject.getTimestamp() returns the value of the specified property as a Timestamp variable.

```java
timestamp getTimestamp(int propertyNumber) throws XDOException
timestamp getTimestamp(String propertyName) throws XDOException
```

**parameters:**
- `propertyNumber` — property number of the property.
- `propertyName` — name of the property.

The method argument can be specified as either a property name string or a number assigned by the system (see `getPropertyNumber()`). Throws XDOException if the property is the wrong datatype.

**insert()**

DynamicObject.insert() creates a new target database object that contains a copy of the dynamic object’s data.

```java
void insert() throws XDOException
void insert(boolean deferIndices, int timeout) throws XDOException
```

**parameters:**
- `deferIndices` — specifies if indexes should be rebuilt when the object is inserted.
- `timeout` — number of seconds to wait to acquire a lock if the extent is locked when the insert is attempted. A value of -1 indicates that the default timeout should be used, and 0 means no wait.

Creates a new object. The dynamic object will target the new database object even if it previously targeted a different object. An exception is thrown if the new object has the same unique key as an existing database object.

**isNull()**

DynamicObject.isNull() returns a boolean that indicates whether a DynamicObject object returned by `openId()` is null (which will be the case if no object with the specified Id was found).

```java
boolean isNull()
```

**save()**

DynamicObject.save() saves the dynamic object’s data to a persistent database object.

```java
void save() throws XDOException
void save(boolean deferIndices, int timeout) throws XDOException
```

**parameters:**
- `deferIndices` — specifies if indexes should be rebuilt when the object is saved.
- `timeout` — number of seconds to wait to acquire a lock when the save is attempted. A value of -1 indicates that the default timeout should be used, and 0 means no wait.

If the target database object already exists, it is updated. Otherwise, the method will attempt to insert and target a new database object.

If the insert fails because of a unique key constraint, the dynamic object will target and update the database object that caused the insert to fail.

**set()**

DynamicObject.set() sets the specified property of a dynamic object to the specified value.
void set (int propertyNumber, Double value) throws XDOException
void set (int propertyNumber, double value) throws XDOException
void set (int propertyNumber, Integer value) throws XDOException
void set (int propertyNumber, Long value) throws XDOException
void set (int propertyNumber, int value) throws XDOException
void set (int propertyNumber, String value) throws XDOException
void set (int propertyNumber, long value) throws XDOException
void set (int propertyNumber, BigDecimal value) throws XDOException
void set (int propertyNumber, Date value) throws XDOException
void set (int propertyNumber, Time value) throws XDOException
void set (int propertyNumber, Timestamp value) throws XDOException
void set (String propertyName, Double value) throws XDOException
void set (String propertyName, double value) throws XDOException
void set (String propertyName, Integer value) throws XDOException
void set (String propertyName, Long value) throws XDOException
void set (String propertyName, int value) throws XDOException
void set (String propertyName, String value) throws XDOException
void set (String propertyName, Date value) throws XDOException
void set (String propertyName, Time value) throws XDOException
void set (String propertyName, Timestamp value) throws XDOException

parameters:
- propertyNumber — property number of the property.
- propertyName — name of the property.
- value — value to be assigned to the object.

The property can be specified by either name or number. This method affects only the dynamic object. No change is made to the database until the dynamic object is saved.

update()

DynamicObject.update() updates the data in the target database object.

void update () throws XDOException
void update (boolean deferIndices, int timeout) throws XDOException

parameters:
- deferIndices — specifies if indexes should be rebuilt when the object is updated.
- timeout — number of seconds to wait if the object is locked when the update is attempted. A value of -1 indicates that the default timeout should be used, and 0 means no wait.

Throws an exception if the target database object does not exist.

5.2.5 XDO Exceptions

The following classes implement exceptions thrown by most methods of DatabaseConnection and DynamicObject. These classes extend java.lang.Exception.

com.intersys.xdo.XDOException

Implements the exception thrown by most XDO methods.
5.3 Globals Quick Reference

This section is a reference for the Globals API (namespace com.intersys.globals). See Using the Globals API for a details on how to use the API. It contains the following public classes and interfaces:

Namespace com.intersys.globals:

- **Class `ConnectionContext`** — supplies the method that returns a Connection object.
- **Interface `Connection`** — encapsulates aGlobals database connection, and supplies methods for session control and transaction processing.
- **Interface `GlobalsDirectory`** — represents a browsable directory of the global names in the current namespace.
- **Interface `NodeReference`** — encapsulates a reference to a global node, and supplies numerous methods to access and manipulate nodes.
- **Interface `ValueList`** — encapsulates a Java representation of a Caché $LIST object.
- **Class `ByteArrayRegion`** — is used by a few Connection methods to specify a subset of a byte array.
- **Globals Exception Classes** — implement exceptions thrown by most Globals API methods.
- **Globals Implementation Classes** — are classes in the com.intersys.globals.imp namespace that implement the Globals API interfaces, allowing them to be extended for special purposes.

5.3.1 List of Globals API Methods

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

**Class `ConnectionContext`**

- `getConnection()` — creates a `Connection` object.
Interface Connection

- **callBytesClassMethod()** — calls a Caché class method. Identical to **callClassMethod()** except that string values are returned as `byte[]`.
- **callBytesFunction()** — calls a Caché function. Identical to **callFunction()** except that string values are returned as `byte[]`.
- **callClassMethod()** — calls a Caché class method.
- **callFunction()** — calls a Caché function.
- **callProcedure()** — calls a Caché procedure.
- **callVoidClassMethod()** — calls a Caché class method with no return value.
- **close()** — releases all resources held by this instance.
- **commit()** — commits one level of transaction.
- **connect()** — connects to Caché using the arguments specified.
- **createGlobalsDirectory()** — creates a browsable directory of the global names in the current namespace.
- **createList()** — returns an empty `ValueList` object.
- **createNodeReference()** — returns a `NodeReference` object. Optionally specifies a global name.
- **getNamespace()** — returns the current namespace string.
- **getProductVersion()** returns the version number of the connected Caché instance.
- **isConnected()** — returns `true` if this object is connected to a database.
- **releaseAllLocks()** — releases all locks currently held in this session.
- **rollback()** — rolls back the specified number of transaction levels, or all levels if no number is specified.
- **setNamespace()** — sets the current namespace.
- **startTransaction()** — starts a transaction (which may be a nested transaction).
- **transactionLevel()** — returns the current transaction level (0 if not in a transaction).

InterfaceGlobalsDirectory

- **close()** — releases the object’s underlying resources to avoid resource leaks when the object is destroyed.
- **nextGlobalName()** — gets the next global name in collating sequence after the specified string.
- **previousGlobalName()** — gets the previous global name in collating sequence before the specified string.
- **refresh()** — updates the directory to reflect any additions or deletions since the object’s creation or most recent refresh.

Interface NodeReference

- **acquireLock()** — locks the current node using the specified locking arguments.
- **appendSubscript()** — adds the specified subscript at the end of the subscript list.
- **close()** — releases the object's underlying resources to avoid resource leaks when the object is destroyed.
- **exists()** — returns `true` if this node contains data.
- **getBytes()** — returns the value of this node as a `byte[]`. 

Using Java with Caché eXtreme
• `getDouble()` — returns the value of this node as a double.
• `getDoubleSubscript()` — returns the value of the specified subscript as a double.
• `getInt()` — returns the value of this node object as an int.
• `getIntSubscript()` — Returns the value of the specified subscript as an int.
• `getList()` — returns the value of this node as a ValueList.
• `getLong()` — returns the value of this node as a long.
• `getLongSubscript()` — returns the value of the specified subscript as a long.
• `getName()` — returns the current global name.
• `getObject()` — returns the value of this node as an Object.
• `getObjectSubscript()` — returns the value of the specified subscript as an Object.
• `getOption()` — returns the current lock timeout setting.
• `getString()` — returns the value of this node as a String.
• `getStringSubscript()` — returns the value of the specified subscript as a String.
• `getSubscriptCount()` — returns an int that indicates the current number of subscripts
• `hasSubnodes()` — returns true if this node has descendants.
• `increment()` — increments the node value by the specified number and returns the new value as a long.
• `kill()` — kills the node and any descendants.
• `killNode()` — kills the node, but not its descendants.
• `nextSubscript()` — returns a String containing the subscript of the next node.
• `previousSubscript()` — returns a String containing the subscript of the previous node.
• `releaseLock()` — releases the lock on the current node, either immediately or at the end of the session.
• `set()` — sets the node's value to the specified value.
• `setName()` — sets the name of the global to be referenced by this node object.
• `setOption()` — sets the number of seconds that acquireLock() should wait for a lock before timing out.
• `setSubscript()` — sets a specified subscript's value.
• `setSubscriptCount()` — reduces the number of subscripts (discarding excess trailing subscripts)

Also see “Acquiring and Releasing Locks” in the chapter on “Using the Globals API” for a list of valid arguments used by the acquireLock() and releaseLock(). See “Controlling the Timeout Interval” for more on the arguments used by SetOption(), and GetOption().

**Interface ValueList**

• `append()` — appends a value of the specified datatype to the list. Multiple values can be specified in a single call.
• `clear()` — clears the list, causing it to contain 0 items.
• `close()` — releases the object's underlying resources to avoid resource leaks when the object is destroyed.
• `getAll()` — returns all items from the list as an array of Object. Can optionally return strings as byte[].
• `getNextBytes()` — returns the next item from the list as byte array.
- `getNextDouble()` — returns the next item from the list as a double.
- `getNextInt()` — returns the next item from the list as an int.
- `getNextList()` — returns the next item from the list as a `ValueList`.
- `getNextLong()` — returns the next item from the list as a long.
- `getNextObject()` — returns the next item from the list as an Object. Can optionally return a string as byte[].
- `getNextString()` — returns the next item from the list as a String.
- `length()` — returns the number of items in the list.
- `resetToFirst()` — resets the cursor to the beginning of the list.
- `skipNext()` — advances the cursor past the specified number of list items without getting their values.

**Class ByteArrayRegion**

- `getLength()` — returns the number of bytes in the specified region.
- `getOffset()` — returns the starting offset of the region within the source array.
- `getSourceArray()` — returns the source array containing the region.
- `set()` — defines all attributes of the region.
- `setLength()` — sets the number of bytes to be included in the region.
- `setOffset()` — sets the starting offset of the region within the source array.
- `setSourceArray()` — sets the source array containing the region.

**Globals Exception Classes**

- `GlobalsException` — thrown by most Globals API methods.
- `LockException` — thrown by `NodeReference` method `acquireLock()`.
- `UndefinedException` — thrown by methods that get the value of a node.

**Globals Implementation Classes**

- Class `ConnectionImpl` — implements the `Connection` interface.
- Class `NodeReferenceImpl` — implements the `NodeReference` interface.
- Class `ValueListImpl` — implements the `ValueList` interface.

### 5.3.2 Class ConnectionContext

Class `com.intersys.globals.ConnectionContext` is required to create a Globals API Connection object. See the section on “Creating a Connection” in “Using the Globals API” for information on usage. See the Caché JavaDoc (<cache-root>/dev/java/doc/index.html) for more details on connection options.

`getConnection()` creates an instance of `com.intersys.globals.Connection`.

```java
static synchronized Connection getConnection()
```
### 5.3.3 Interface Connection

Interface `com.intersys.globals.Connection` represents a connection to the Caché database. Instances of `Connection` are created by calls to the `getConnection()` method of class `ConnectionContext`. All instances of `Connection` in a process reference the same underlying connection. Use `Connection.isConnected()` to determine if the underlying connection already exists. See “Creating a Connection” for more information on how connections work.

**callBytesClassMethod()**

Connection `callBytesClassMethod()` — calls a Caché ObjectScript class method, passing 0 or more arguments and returning the method's return value as an instance of `byte[]`, `Integer`, `Long`, or `Double`.

This method is identical to `callClassMethod()` except that it returns string values as instances of `byte[]` rather than `String`.

```java
Object callBytesClassMethod(String className, String methodName, Object... args)
```

**parameters:**

- `className` — fully qualified name of the Caché class to which the called method belongs.
- `methodName` — name of the Caché class method.
- `args` — a list of 0 or more arguments to pass to the method.

Arguments may be of any of the types `String`, `Integer`, `Long`, `Double`, `byte[]`, or `ByteArrayRegion`. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

**callBytesFunction()**

Connection `callBytesFunction()` calls a Caché ObjectScript function (see “User-defined Code” in Using Caché ObjectScript).

This method is identical to `callFunction()` except that it returns string values as instances of `byte[]` rather than `String`.

```java
Object callBytesFunction(String functionName, String routineName, Object... args)
```

**parameters:**

- `functionName` — name of the function.
- `routineName` — name of the routine containing the function.
- `args` — arguments to be passed.

Arguments may be of any of the types `String`, `Integer`, `Long`, `Double`, `byte[]`, or `ByteArrayRegion`. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

**callClassMethod()**

Connection `callClassMethod()` — calls a Caché ObjectScript class method, passing 0 or more arguments and returning the method's return value as an instance of `String`, `Integer`, `Long`, or `Double` (use `callVoidClassMethod()` to call a method that doesn’t return a value).
Also see the `callBytesClassMethod()` method, which is identical to this method except that it returns string values as instances of byte[] rather than String.

```
Object callClassMethod(String className, String methodName, Object... args)
```

**parameters:**
- `className` — fully qualified name of the Caché class to which the called method belongs.
- `methodName` — name of the Caché class method.
- `args` — a list of 0 or more arguments to pass to the method.

Arguments may be of any of the types String, Integer, Long, Double, byte[], or `ByteArrayRegion`. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

**callFunction()**

Connection.`callFunction()` calls a Caché ObjectScript function (see “User-defined Code” in Using Caché ObjectScript).

Also see the `callBytesFunction()` method, which is identical to this method except that it returns string values as instances of byte[] rather than String.

```
Object callFunction(String functionName, String routineName, Object... args)
```

**parameters:**
- `functionName` — name of the function.
- `routineName` — name of the routine containing the function.
- `args` — arguments to be passed.

Arguments may be of any of the types String, Integer, Long, Double, byte[], or `ByteArrayRegion`. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

**callProcedure()**


```
void callProcedure(String procedureName, String routineName, Object... args)
```

**parameters:**
- `procedureName` — name of the procedure.
- `routineName` — name of the routine containing the procedure.
- `args` — arguments to be passed.

Arguments may be of any of the types String, Integer, Long, Double, byte[], or `ByteArrayRegion`. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.
callVoidClassMethod()  
Connection.callVoidClassMethod() — calls a Caché ObjectScript class method with no return value, passing 0 or more arguments. This method may be used to call any Caché class method (regardless of whether it normally returns a value) when the caller does not need the return value. Use callClassMethod() to call a method that returns a value.

void callVoidClassMethod(String className, String methodName, Object... args)

parameters:
• className — fully qualified name of the Caché class to which the called method belongs.
• methodName — name of the Caché class method.
• args — list of 0 or more arguments to pass to the method.
• Arguments may be of any of the types String, Integer, Long, Double, byte[], or ByteArrayRegion. Trailing arguments may be omitted, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing null for trailing arguments. Throws an exception if a non-null argument is passed to the right of a null argument.

close()  
Connection.close() releases all resources held by this instance.

void close()

It is important to always call close() on an instance of Connection before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

commit()  
Connection.commit() commits one level of transaction (see transactionLevel()) for the current session.

void commit()

If the level is greater than 1, the current transaction is merged with the enclosing transaction, and can still be rolled back by calling rollback(). When the transaction level is 1, any changes made during the transaction are permanently committed.

connect()  
Connection.connect() connects the Connection object to the specified database. GlobalsException is thrown if the object is already connected.

void connect()
void connect(String namespace, String user, String password)

parameters:
• namespace — namespace to be accessed.
• user — user name for this connection.
• password — password for this connection.

If no arguments are specified, method call defaults to connect ("User", "", "). This method uses theGLOBALS_HOME environment variable to locate the Caché instance (see “Required Environment Variables for All Platforms”).
createGlobalsDirectory()

Connection.createGlobalsDirectory() creates a browsable directory of the global names in the current namespace, with the cursor positioned before the first global name in collating sequence.

GlobalsDirectory createGlobalsDirectory()

cREATELIST()

Connection.createList() returns a ValueList object, and optionally specifies a buffer size.

ValueList createList()
ValueList createList(int bufferSize)

parameter:
• bufferSize — initial size in bytes of underlying buffer.

If bufferSize is not specified, the ValueList is created with a 1 kbyte buffer, which grows if needed but never gets smaller. Specifying a smaller bufferSize may save memory if a list is known to need significantly less than 1 kbyte. Specifying a larger bufferSize may enhance performance by avoiding repeated reallocation to grow the buffer, if a list is known to need significantly more than 1 kbytes.

cREATENODEREFERENCE()

Connection.createNodeReference() returns a NodeReference object, and optionally specifies the name of the global to be referenced.

NodeReference createNodeReference()
NodeReference createNodeReference(String name)

parameter:
• name — optional String specifying the name of the global to be referenced.

If name is not specified, the node reference object's setName() method must be called in order to specify a global array (see “Setting and Changing Global Names”).

cGETNAMESPACE()

Connection.getNamespace() returns the name of the currently connected namespace.

String getNamespace()

ThrowsGlobalsException if called when this instance is not connected.

cGETPRODUCTVERSION()

Connection.getProductVersion() returns the version number of the connected Caché instance.

String getProductVersion()

isConnected()

Connection.isConnected() returns true if this object is connected to a database.

boolean isConnected()

releaseAllLocks()

Connection.releaseAllLocks() releases all locks currently held in this connection.

void releaseAllLocks()
rollback()

Connection.rollback() rolls back levelCount levels of transaction, or roll back all levels of transaction if levelCount is not specified.

void rollback()
void rollback(int levelCount)

parameter:
• levelCount — number of levels to roll back.

Stops rolling back once transaction level reaches 0, if levelCount is greater than initial transaction level. Does nothing if levelCount is less than 1.

see also: Transactions and Locking

setNamespace()

Connection.setNamespace() sets the current namespace

void setNamespace(String namespace)

parameter:
• namespace — namespace for this connection.

startTransaction()

Connection.startTransaction() starts a transaction (which may be a nested transaction).

void startTransaction()

see also: Transactions and Locking

transactionLevel()

Connection.transactionLevel() returns an int that indicates current transaction level (0 if not in a transaction).

int transactionLevel()

see also: Transactions and Locking

5.3.4 Interface GlobalsDirectory

Interface com.intersys.globals.GlobalsDirectory represents a browsable directory of the names of globals in the current namespace. Global names may be browsed in ascending or descending collating sequence. These names may be passed to Connection.createNodeReference(String) to create instances of NodeReference that can be used to perform operations on globals.

Instances of GlobalsDirectory are created by calling the Connection.createGlobalsDirectory() method, or by using the GlobalsDirectoryImpl constructor.

close()

GlobalsDirectory.close() releases all resources held by this instance.

void close()
It is important to always call `close()` on an instance of `GlobalsDirectory` before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

nextGlobalName()

`GlobalsDirectory.nextGlobalName()` returns the next global name in collating sequence after the specified name (or after the current position if no global name is specified). Returns an empty string if there is no next name.

```java
String nextGlobalName()
String nextGlobalName(String globalName)
```

**parameter:**
- `globalName` — string to use as the beginning position (does not have to specify an existing global name). Specify an empty string (""") to get the first global name in the collating sequence.

previousGlobalName()

`GlobalsDirectory.previousGlobalName()` returns the previous global name in collating sequence before the specified name (or before the current position if no global name is specified). Returns an empty string if there is no previous name.

```java
String previousGlobalName()
String previousGlobalName(String globalName)
```

**parameter:**
- `globalName` — string to use as the beginning position (does not have to specify an existing global name). Specify an empty string (""") to get the last global name in the collating sequence.

refresh

`GlobalsDirectory.refresh()` updates the list of global names, adding or deleting names to reflect any changes made since the instance was created or `refresh()` was last called.

```java
void refresh()
```

### 5.3.5 Interface NodeReference

Interface `com.intersys.globals.NodeReference` represents a reference to a global node. It provides methods for specifying the node's name and subscripts, as well as methods to manipulate the node, including setting and getting its value, killing it, determining its state, and incrementing its value.

Instances of `NodeReference` are created by calls to the `Connection` `createNodeReference()` method. See the chapter on “Using the Globals API” for information on usage.

acquireLock()

`NodeReference.acquireLock()` locks an object, using the specified locking arguments.

```java
void acquireLock(int lockType, int lockMode)
void acquireLock(int lockType, int lockMode, Object... subscripts)
```

**parameters:**
- `lockType` — specify one of the following values:
– **NodeReference.SHARED_LOCK** — A shared lock allows other callers to acquire shared locks, but prevents any caller from acquiring an exclusive lock. The node value cannot be changed until all shared locks are released.

– **NodeReference.EXCLUSIVE_LOCK** — An exclusive lock prevents other callers from acquiring any lock, allowing the node value to be changed safely.

- **lockMode** — specify one of the following values:
  - **NodeReference.LOCK_INCREMENTALLY** — An incremental lock is acquired without releasing any previously held locks.
  - **NodeReference.LOCK_NON_INCREMENTALLY** — A non-incremental lock releases all previously held locks.

- **subscripts** — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

**see also:** Transactions and Locking

## appendSubscript()

**NodeReference.appendSubscript()** adds a new subscript at the end of the subscript list of a node reference object.

```java
void appendSubscript(int sub)
void appendSubscript(long sub)
void appendSubscript(double sub)
void appendSubscript(String sub)
```

**parameter:**

- **sub** — the specified subscript can be an int, long, double, or String.

For example, the method could add subscript "a" to `^myGlobal(1)`, changing the reference to `^myGlobal(1,"a")`.

## close()

**NodeReference.close()** releases all resources held by this instance.

```java
void close()
```

It is important to always call **close**() on an instance of **NodeReference** before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

## exists()

**NodeReference.exists()** returns true if the node has data. Returns false if the node is valueless or does not exist.

```java
boolean exists()
boolean exists(Object... subscripts)
```

**parameter:**

- **subscripts** — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).
getBytes()

NodeReference.getNumber() returns the value of this node as a byte[]. A null is returned if the referenced node is valueless or does not exist.

byte[] getBytes()
byte[] getBytes(Object... subscripts)

**parameter:**
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

getDouble()

NodeReference.getNumber() returns the value of this node as a double. This method should never be used if the node value might be a String.

double getDouble()
double getDouble(Object... subscripts)

**parameter:**
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

String values are not converted even if they are numeric. The method will attempt to interpret the string as a valid double, and will return a meaningless and unpredictable value.

Throws UndefinedException if the referenced node is valueless or does not exist.

getDoubleSubscript()

NodeReference.getNumber() reads the subscript specified by subscriptPosition and returns it as a double value.

double getDoubleSubscript(int subscriptPosition)

**parameter:**
- subscriptPosition — the level number of the subscript to be read (counting subscripts up from level 1).

Throws GlobalsException if subscriptPosition is greater than the current number of subscripts, or if the subscript is the wrong datatype.

getInt()

NodeReference.getNumber() returns the value of this node as an int.

int getInt()
int getInt(Object... subscripts)

**parameter:**
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

If the node value was originally set as a double, the returned value will be truncated to a long value. If the node value is a non-numeric string (one that does not start with +, -, or a digit), a 0 will be returned.

Throws UndefinedException if the referenced node is valueless or does not exist.
**getIntSubscript()**

NodeReference.getIntSubscript() reads the subscript specified by `subscriptPosition` and returns it as an int value.

```java
int getIntSubscript(int subscriptPosition)
```

**parameter:**

- `subscriptPosition` — the level number of the subscript to be read (counting subscripts up from level 1).

Throws `GlobalsException` if `subscriptPosition` is greater than the current number of subscripts, or if the subscript is the wrong datatype.

**getList()**

NodeReference.getList() returns the value of this node as a `ValueList`. A `null` is returned if the referenced node is valueless or does not exist.

```java
ValueList getList()
ValueList getList(Object... subscripts)
ValueList getList(ValueList reuseList)
ValueList getList(ValueList reuseList, Object... subscripts)
```

**parameter:**

- `reuseList` — optional existing instance of `ValueList` to be modified and returned.
- `subscripts` — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

Throws `GlobalsException` if the value is not a string in valid SLIST format.

**getLong()**

NodeReference.getLong() returns the value of this node as a `long`.

```java
long getLong()
long getLong(Object... subscripts)
```

**parameter:**

- `subscripts` — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

Throws `UndefinedException` if the referenced node is valueless or does not exist.

**getLongSubscript()**

NodeReference.getLongSubscript() reads the subscript specified by `subscriptPosition` and returns it as a `long` value.

```java
long getLongSubscript(int subscriptPosition)
```

**parameter:**

- `subscriptPosition` — the level number of the subscript to be read (counting subscripts up from level 1).

Throws `GlobalsException` if `subscriptPosition` is greater than the current number of subscripts, or if the subscript is the wrong datatype (int values are permitted).
getName

NodeReference.getName() returns the name of the global array containing this node.

String getName()

ggetObject

NodeReference.getObject() returns the node value as an Object, which will be an instance of Integer, Long, Double, or String (instances of byte[] and ValueList are returned as instances of String). Returns null if the node is valueless.

Object getObject()
Object getObject(Object... subscripts)

parameter:
• subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

ggetObjectSubscript

NodeReference.getObjectSubscript() returns the subscript specified by subscriptPosition as an Object, which will be an instance of Integer, Long, Double, or String.

Object getObjectSubscript(int subscriptPosition)

Throws GlobalsException if subscriptPosition is greater than current number of subscripts.

ggetOption

NodeReference.getOption() returns the current lock timeout setting (see setOption()).

int getOption(int option)

parameter:
• option — must be set to NodeReference.LOCK_TIMEOUT. Other options will be added in future releases.

see also: Transactions and Locking

getString

NodeReference.getString() returns the value of this node as a String. A null is returned if the referenced node is valueless or does not exist.

String getString()
String getString(Object... subscripts)

parameter:
• subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

Any global value can be accessed as a string, although the string may be meaningless or misleading in the context of your application. For example, if a node is set to a double value of 19.95, the string returned by this method will be 19.949999999999999289, which is the internal representation of the double.

getStringSubscript

NodeReference.getStringSubscript() reads the subscript specified by subscriptPosition and returns it as a String value.
String getStringSubscript(int subscriptPosition)

parameter:
- subscriptPosition — the level number of the subscript to be read (counting subscripts up from level 1).

ThrowsGlobalsException if subscriptPosition is greater than the current number of subscripts, or if the subscript is the wrong datatype.

getSubscriptCount()

NodeReference.getSubscriptCount() returns an int that indicates the current number of subscripts

int getSubscriptCount()

For example, if the node reference points to ^myGlobal(1,"a"), the method returns 2.

hasSubnodes()

NodeReference.hasSubnodes() returns true if the referenced node has descendants.

boolean hasSubnodes()
boolean hasSubnodes(Object... subscripts)

parameter:
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

increment()

NodeReference.increment() increments or decrements the value of the current node by the specified int value, and returns the new value as a long. If the incremented node value is a double, the returned value will be truncated. This is an atomic operation (lock-free, thread-safe).

long increment(int number)
long increment(int number, Object... subscripts)

parameter:
- number — specifies the amount by which the node value should be incremented.
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

If no node exists at the current reference, a node with a value of 0 is created and incremented.

kill()

NodeReference.kill() deletes the node at the current node reference and all of its descendants.

void kill()
void kill(Object... subscripts)

parameter:
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

This method is equivalent to the Caché KILL command.
**killNode()**

`NodeReference.killNode()` deletes the value of the node at the current node reference, but does not affect its subnodes. If the node does not have subnodes, it is deleted from the database.

```java
void killNode()
void killNode(Object... subscripts)
```

**parameter:**

- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

This method is equivalent to the Caché ZKILL command.

**nextSubscript()**

`NodeReference.nextSubscript()` returns a `String` containing the subscript of the next node in collation order on the current level. Returns an empty string ("") if there is no next subscript.

```java
String nextSubscript()
String nextSubscript(Object... subscripts)
```

**parameter:**

- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

**previousSubscript()**

`NodeReference.previousSubscript()` returns a `String` containing the subscript of the previous node in collation order on the current level. Returns an empty string ("") if there is no previous subscript.

```java
String previousSubscript()
String previousSubscript(Object... subscripts)
```

**parameter:**

- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

**releaseLock()**

`NodeReference.releaseLock()` releases the lock on the current node.

```java
void releaseLock(int lockType, int releaseMode)
void releaseLock(int lockType, int releaseMode, Object... subscripts)
```

**parameter:**

- lockType — specify one of the following values:
  - `NodeReference.SHARED_LOCK` — A shared lock allows other callers to acquire shared locks, but prevents any caller from acquiring an exclusive lock. The node value cannot be changed until all shared locks are released.
  - `NodeReference.EXCLUSIVE_LOCK` — An exclusive lock prevents other callers from acquiring any lock, allowing the node value to be changed safely.

- releaseMode — specify one of the following values:
  - `NodeReference.RELEASE_IMMEDIATELY` — Release the lock immediately.
NodeReference.RELEASE_AT_TRANSACTION_END — Wait until the end of the current transaction before releasing the lock.

- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

see also: Transactions and Locking, Connection.releaseAllLocks()

set()

NodeReference.set() assigns the specified value to the referenced node.

```java
void set(int value)
void set(int value, Object... subscripts)
void set(long value)
void set(long value, Object... subscripts)
void set(double value)
void set(double value, Object... subscripts)
void set(String value)
void set(String value, Object... subscripts)
void set(byte[] value)
void set(byte[] value, Object... subscripts)
void set(ValueList value)
void set(ValueList value, Object... subscripts)
```

parameter:
- value — value to be assigned to the node. The value can be an int, long, double, String, byte[], or ValueList.
- subscripts — 0 or more optional subscripts to qualify the node reference (see “Addressing a Subnode of the Target”).

If no persistent node exists at the current reference, a new one is created and set to the specified value.

setName()

NodeReference.setName() specifies the name of the global to be referenced by this node object.

```java
void setName(String name)
```

parameter:
- name — name of the global to be referenced.

setOption()

NodeReference.setOption() sets the number of seconds that acquireLock() should wait for a lock before timing out. Other options will be added in future releases.

```java
void setOption(int option, int value)
```

parameter:
- option — must be set to NodeReference.LOCK_TIMEOUT.
- value — number of seconds to wait before timing out. Any int value may be specified, and the following predefined values are also available:
  - NodeReference.DEFAULT_LOCK_TIMEOUT — Default value is 10.
  - NodeReference.NO_LOCK_TIMEOUT — Value is -1, which specifies that the call will never time out (acquireLock() will wait forever).
setSubscript()

`NodeReference.setSubscript()` changes the specified subscript, or adds a new subscript.

```java
void setSubscript(int subscriptPosition, double value)
void setSubscript(int subscriptPosition, int value)
void setSubscript(int subscriptPosition, long value)
void setSubscript(int subscriptPosition, String value)
```

**Parameter:**

- `subscriptPosition` — the level number of the subscript to set (counting subscripts up from level 1).
- `value` — value of the subscript. The subscript can be an int, long, double, or String.

For example, if the node reference pointed to `"myGlobal(1,"a")`, a call to `setSubscript(1,8)` would set the reference to `"myGlobal(8,"a")`.

This method is equivalent to `appendSubscript()` if `subscriptPosition` is equal to `getSubscriptCount()+1`. For example, a call to `setSubscript(3,"x")` would set it to `"myGlobal(1,"a","x")`.

If `subscriptPosition` is less than 1, a value of 1 will be used.

Throws `GlobalsException` if `subscriptPosition` is greater than `GetSubscriptCount()+1`.

setSubscriptCount()

`NodeReference.setSubscriptCount()` shortens the subscript list by removing all subscripts after the one specified by `num`.

```java
void setSubscriptCount(int num)
```

**Parameter:**

- `num` — the level number of the last subscript to retain (counting subscripts up from level 1).

For example, if the reference points to `"myGlobal(1,"a",35)` calling `setSubscriptCount(1)` would remove all subscripts except the first, changing the reference to `"myGlobal(1)`.

Throws `GlobalsException` if `num` is greater than the current number of subscripts.

### 5.3.6 Interface ValueList

The `com.intersys.globals.ValueList` encapsulates a Java representation of a simple Caché $LIST object. The following datatypes are supported: int, Integer, long, Long, double, Double, String, byte[], and nested instances of `ValueList`. See “Storing Multiple Items in a Node with ValueList” for information on usage. `ValueList` objects are created by calling the `Connection.createList()` method, or by using `ValueListImpl` constructors.

append()

`ValueList.append()` appends a value of the specified datatype to the list.

```java
void append(int value)
void append(Integer value)
void append(long value)
void append(Long value)
void append(double value)
void append(Double value)
void append(String value)
void append(byte[] value)
void append(ValueList value)
```

Multiple values can be specified in a single call to `append()`:

```java
void append(Object... objects)
```
parameter:

• value — the value of the item to be appended. An exception is thrown if the value is not one of the permitted datatypes.

• objects — a list of one or more Object instances. Each Object must evaluate to datatype Integer, Long, Double, or String, or an exception is thrown. Instances of byte[] and ValueList are valid because they evaluate to String.

Throws GlobalsException if the argument is an invalid datatype.

clear()

ValueList.clear() clears the list, causing it to contain 0 items.

void clear()

Throws GlobalsException.

close()

ValueList.close() releases all resources held by this instance.

void close()

It is important to always call close() on an instance of ValueList before it goes out of scope. Failing to close it can cause serious memory leaks because Java garbage collection cannot release resources allocated by the underlying native code.

Throws GlobalsException if the instance of ValueList calls any other method after this one.

getAll()

ValueList.getAll() returns all items from the list as an array of Object, or returns null if the list is empty. Each returned Object is an instance of Integer, Long, Double, String, or byte[], depending on the type and value of the item in the list. An Object is null if the corresponding list item is null. This method leaves the list cursor positioned beyond last item in the list.

Object[] getAll()
Object[] getAll(boolean returnBytes)

parameter:

• returnBytes — if true, return strings as byte[], else return them as String.

Throws GlobalsException.

getNextBytes()

ValueList.getNextBytes() returns the next item from the list as byte[]. Returns null if list item is null.

byte[] getNextBytes()

Throws GlobalsException if the cursor is already at the end of the list.

getNextDouble()

ValueList.getNextDouble() returns the next item from the list as a double. Returns 0 if list item is null.

double getNextDouble()

Throws GlobalsException if the cursor is already at the end of the list.
getNextInt()

ValueList.getNextInt() returns the next item from the list as an int. Returns 0 if list item is null.

```java
int getNextInt()
```

Throws GlobalsException if the cursor is already at the end of the list.

getNextList()

ValueList.getNextList() returns the next item from the list as a ValueList. Returns null if list item is null.

```java
ValueList getNextList()
ValueList getNextList(ValueList reuseList)
```

parameter:

- `reuseList` — optional existing instance of ValueList to be modified and returned. The ValueList passed in `reuseList` is modified to contain the item’s value, and is returned, rather than creating a new ValueList instance. It is not modified if the list item is null.

Throws GlobalsException if the cursor is already at the end of the list, or if the list item is not a valid ValueList.

getNextLong()

ValueList.getNextLong() returns the next item from the list as a long. Returns 0 if list item is null.

```java
long getNextLong()
```

Throws GlobalsException if the cursor is already at the end of the list.

getNextObject()

ValueList.getNextObject() returns the next item from the list as an Object. The type of the returned object depends on the type and value of the item in the list, and on the setting of the optional `returnBytes` parameter.

A null is returned if the list item is null. All integer values are returned as Integer if they are within the range of Integer, else they are returned as Long. Double values are always returned as Double. If optional `returnBytes` is specified as true, both byte[] and String are returned as byte[]. If it is false or not specified, both types are returned as String.

```java
Object getNextObject()
Object getNextObject(boolean returnBytes)
```

parameter:

- `returnBytes` — if true, return both byte[] and String as byte[], else return both types as String.

Throws GlobalsException if the cursor is already at the end of the list.

getNextString()

ValueList.getNextString() returns the next item from the list as a String. Returns null if list item is null.

```java
String getNextString()
```

Throws GlobalsException if the cursor is already at the end of the list.

length()

ValueList.length() returns the number of items in the list.

```java
int length()
```
Throws GlobalsException.

resetToFirst()

ValueList.resetToFirst() resets the cursor to the beginning of the list.

void resetToFirst()

Throws GlobalsException.

skipNext()

ValueList.skipNext() advances the cursor past the number of list items specified by count without getting their values.

void skipNext(int count)

parameter:

- count — number of items to skip past.

Throws GlobalsException if fewer than count items remain in the list beyond the cursor position.

5.3.7 Class ByteArrayRegion

Class com.intersys.globals.ByteArrayRegion specifies a region of a byte array, encapsulating the source array, the starting offset, and the length of the region. This class provides a convenient and efficient way to pass a byte array region as an argument to the Connection methods that call ObjectScript methods and functions (see “Calling Caché Methods”) without having to copy the region to a separate byte[] instance.

ByteArrayRegion() Constructors

Create a new instance of ByteArrayRegion. Can optionally specify the source array, offset, and number of bytes in the region.

ByteArrayRegion()
ByteArrayRegion(byte[] sourceArray, int offset, int length)

parameters:

- sourceArray — array containing the region.
- offset — starting offset of the region within sourceArray.
- length — number of bytes in the region.

getLength()

ByteArrayRegion.getLength() returns the number of bytes in the specified region.

int getLength()

getOffset()

ByteArrayRegion.getOffset() returns the starting offset of the region within the source array.

int getOffset()

ggetSourceArray()

ByteArrayRegion.getSourceArray() returns the source array containing the region.
byte[] getSourceArray()

set()

ByteArrayRegion.set() defines all attributes of the region.

void set(byte[] sourceArray, int offset, int length)

parameter:
• sourceArray — array containing the region.
• offset — starting offset of the region within sourceArray.
• length — number of bytes in the region.

setLength()

ByteArrayRegion.setLength() sets the number of bytes to be included in the region.

void setLength(int length)

parameter:
• length — number of bytes in the region.

setOffset()

ByteArrayRegion.setOffset() sets the starting offset of the region within the source array.

void setOffset(int offset)

parameter:
• offset — starting offset of the region within the array.

setSourceArray()

ByteArrayRegion.setSourceArray() sets the source array containing the region.

void setSourceArray(byte[] sourceArray)

parameter:
• sourceArray — array containing the region.

5.3.8 Globals Exception Classes

The Globals API implements the following exceptions:
• GlobalsException — thrown by most Globals API methods.
• LockException — thrown by the NodeReference acquireLock() method.
• UndefinedException — thrown by methods that get the value of a node.

5.3.8.1 Class GlobalsException

Class com.intersys.globals.GlobalsException implements an exception thrown by all classes of the Globals API. This class inherits from java.lang.RuntimeException.
Constructors

In addition to the parameters defined for RuntimeException constructors, the GlobalsException constructors provide an error code parameter for errors specific to the Globals API:

GlobalsException(String msg)
GlobalsException(String msg, int code)
GlobalsException(Throwable cause)
GlobalsException(Throwable cause, String msg)

parameters:
• msg — optional detailed error message.
• code — optional error code, which can be retrieved by GlobalsException method getErrorCode().
• cause — an instance of java.lang.Throwable.

Method getErrorCode()

Returns a numeric error code that can be used to distinguish some specific Globals API errors. Error code UNDEFINED means that no specific error code is defined for this situation.

int getErrorCode()

The following error codes are defined:
• GlobalsException.METHOD_CALLED_AFTER_CLOSE
  A method was called on a Globals API object instance after its close() method was called.
• GlobalsException.NOT_CONNECTED
  A method was called which requires being connected, and no connection was detected.
• GlobalsException.PARAMETER_OUT_OF_RANGE
  A parameter value is out of range.
• GlobalsException.REQUIRED_PARAMETER_NULL
  null was passed for a required method parameter.
• GlobalsException.UNDEFINED
  No numeric error code is defined for the error which caused this GlobalsException to be thrown.

5.3.8.2 Class LockException

Class com.intersys.globals.LockException implements an exception thrown by NodeReference method acquireLock(). It is thrown if a request to acquire a lock fails, due to timing out waiting for the lock to become available. See “Transactions and Locking” for information on usage.

Constructors

LockException()
LockException(String msg)

parameters:
• msg — optional detailed error message. If this argument is not specified, it will default to Cannot acquire lock.

This class inherits the methods and constants defined in GlobalsException.
5.3.8.3 Class UndefinedException

Class `com.intersys.globals.UndefinedException` implements an exception thrown by methods that get the value of a node. The exception is thrown if the referenced node is valueless or does not exist.

**Constructor**

```java
UndefinedException(String msg)
```

*parameters:*

- `msg` — detailed error message.

This class inherits the methods and constants defined in `GlobalsException`.

5.3.9 Globals Implementation Classes

The following classes are implementations of the main Globals API interfaces. They all provide constructors, allowing them to be extended for special purposes:

- Class `ConnectionImpl` — an implementation of interface `ConnectionImpl`.
- Class `GlobalsDirectoryImpl` — an implementation of interface `GlobalsDirectory`.
- Class `NodeReferenceImpl` — an implementation of interface `NodeReferenceImpl`.
- Class `ValueListImpl` — an implementation of interface `ValueListImpl`.

5.3.9.1 Class ConnectionImpl

Class `com.intersys.globals.imp.ConnectionImpl` is an implementation of interface `Connection`. Extend this class to develop specialized connection types for APIs implemented using the Globals API. In addition to the methods listed for the interface, it provides the following constructor:

**ConnectionImpl() Constructor**

Creates a new instance of `ConnectionImpl`, whose underlying connection implementation is a singleton connection instance managed by `ConnectionContext`.

```java
ConnectionImpl()
```

5.3.9.2 Class GlobalsDirectoryImpl

Class `com.intersys.globals.imp.GlobalsDirectoryImpl` is an implementation of interface `GlobalsDirectory`. In addition to the methods listed for the interface, it provides the following constructor:

**GlobalsDirectoryImpl() Constructor**

Creates a directory of the global names in the current namespace, positioned before the first global name in collating sequence.

```java
GlobalsDirectoryImpl()
```

Throws `GlobalsException` if there is no existing connection to the globals database.
5.3.9.3 Class NodeReferenceImpl

Class com.intersys.globals.imp.NodeReferenceImpl is an implementation of interface NodeReference. In addition to the methods listed for the interface, it provides the following constructors:

NodeReferenceImpl() Constructors

Creates a NodeReference instance.

NodeReferenceImpl()
NodeReferenceImpl(String name)

parameter:

• name — optionally specifies the global name to be used. If this argument is not specified, setName() can be called later to set the global name.

Throws GlobalsException if there is no existing connection to the globals database.

5.3.9.4 Class ValueListImpl

Class com.intersys.globals.imp.ValueListImpl is an implementation of interface ValueList. In addition to the methods listed for the interface, it provides the following constructors:

ValueListImpl() Constructors

Creates an empty ValueListImpl instance, optionally specifying the initial buffer size.

ValueListImpl()
ValueListImpl(int bufferSize)

parameter:

• bufferSize — initial size in bytes of underlying buffer.

The bufferSize parameter permits optional tuning of the amount of memory used for a list's underlying implementation. By default, a ValueList is created with a 1 kbyte buffer, which grows if needed but never gets smaller. Specifying a smaller bufferSize may save memory if a list is known to need significantly less than 1 kbytes. Specifying a larger bufferSize may enhance performance by avoiding repeated reallocation to grow the buffer, if a list is known to need significantly more than 1 kbytes.

Throws GlobalsException if there is no existing connection to the globals database.