



# Using Java with Caché

Version 2009.1

30 June 2009

*Using Java with Caché*

InterSystems Version 2009.1 30 June 2009

Copyright © 2009 InterSystems Corporation

All rights reserved.

This book was assembled and formatted in Adobe Page Description Format (PDF) using tools and information from the following sources: Sun Microsystems, RenderX, Inc., Adobe Systems, and the World Wide Web Consortium at [www.w3c.org](http://www.w3c.org). The primary document development tools were special-purpose XML-processing applications built by InterSystems using Caché and Java.



Caché WEBLINK, Distributed Cache Protocol, M/SQL, M/NET, and M/PACT are registered trademarks of InterSystems Corporation.



InterSystems Jalapeño Technology, Enterprise Cache Protocol, ECP, and InterSystems Zen are trademarks of InterSystems Corporation.

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

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

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

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

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

For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Customer Support**

Tel: +1 617 621-0700

Fax: +1 617 374-9391

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

# Table of Contents

<b>About This Book</b> .....	<b>1</b>
<b>1 The Caché Java Binding</b> .....	<b>3</b>
1.1 Java Binding Architecture .....	3
1.2 Installation and Configuration .....	5
1.2.1 Java Client Requirements .....	5
1.2.2 The Caché Java Packages .....	6
1.2.3 Caché Server Configuration .....	6
1.3 Sample Programs .....	7
<b>2 Using the Java Binding</b> .....	<b>9</b>
2.1 Generating Java Proxy Classes .....	9
2.1.1 Generating a Java Class from a Caché Class .....	10
2.2 Using Objects .....	10
2.2.1 Creating a Connection Object .....	11
2.2.2 Creating and Opening Proxy Objects .....	12
2.2.3 Using Methods and Properties .....	12
2.2.4 Saving and Closing .....	13
2.2.5 A Sample Java Binding Application .....	13
2.3 Using Streams .....	15
2.4 Using Queries .....	16
2.4.1 Class Queries .....	16
<b>3 Java Proxy Class Mapping</b> .....	<b>19</b>
3.1 Classes .....	19
3.1.1 Entity Names .....	19
3.1.2 Methods .....	20
3.1.3 Properties .....	22
3.2 Primitive Data Types .....	23

# List of Figures

Java Client/Server Class Architecture .....	4
---	---

# List of Tables

Client Data Type to Java Correspondence ..... 23



# About This Book

This book is a guide to the Caché Java Language Binding.

This book contains the following sections:

- [The Caché Java Binding](#)
- [Using the Java Binding](#)
- [Java Proxy Class Mapping](#)

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

For general information, see *Using InterSystems Documentation*.





# 1

## The Caché Java Binding

The Caché Java binding provides a simple, direct way to use Caché objects within a Java application. You can create Java applications that work with the Caché database in the following ways:

- *The Caché Java Binding* — The Caché Java binding lets Java applications work directly with objects on a Caché server. The binding automatically creates Java proxy classes for the Caché classes you specify. Each proxy class is a pure Java class, containing only standard Java code that provides your Java application with access to the properties and methods of the corresponding Caché class.

The Caché Java binding offers complete support for object database persistence, including concurrency and transaction control. In addition, there is a sophisticated data caching scheme to minimize network traffic when the Caché server and the Java environment are located on separate machines.

- *The Caché JDBC Driver* — Caché includes a level 4 (pure Java) JDBC driver that supports the JDBC version 3.0 API. The Caché JDBC Driver provides high-performance relational access to Caché. For maximum flexibility, applications can use JDBC and the Caché Java binding at the same time.

Each of these methods is discussed in the following chapters.

This document assumes a prior understanding of Java and the Java standard library. Caché does not include a Java compiler or development environment.

### 1.1 Java Binding Architecture

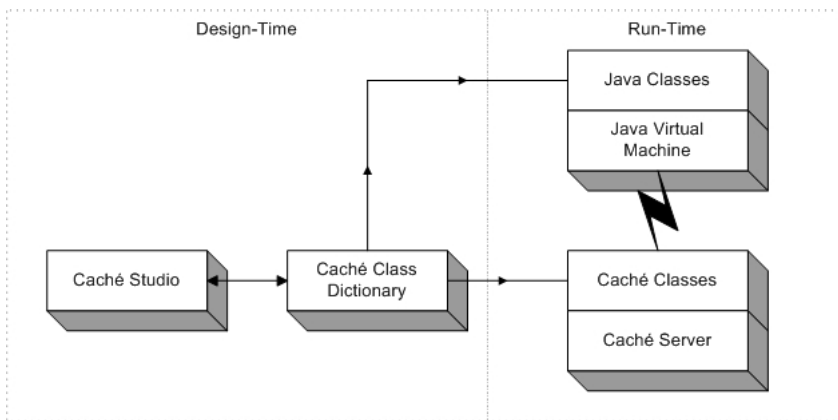
The Caché Java binding gives Java applications a way to access and manipulate objects contained within a Caché server. These objects can be persistent objects stored within the Caché database or they can be transient objects that perform operations within a Caché server.

The Caché Java binding consists of the following components:

- *The Caché Java Class Generator* — an extension to the Caché Class Compiler that generates pure Java classes from classes defined in the Caché Class Dictionary.
- *The Caché Java Class Package* — a package of pure Java classes that works in conjunction with the Java classes generated by the Caché Java Class Generator and provides them with transparent connectivity to the objects stored in the Caché database.
- *The Caché Object Server* — a high performance server process that manages communication between Java clients and a Caché database server. It communicates using standard networking protocols (TCP/IP), and can run on any platform supported by Caché. The Caché Object Server is used by all Caché language bindings, including Java, JDBC, ODBC, C++, Perl, and Python.

The Caché Class Compiler can automatically create Java client classes for any classes contained within the Caché Class Dictionary. These generated Java classes communicate at run time with their corresponding Caché class on a Caché server. The generated Java classes contain only pure Java code and are automatically synchronized with the master class definition. This is illustrated in the following diagram:

**Java Client/Server Class Architecture**



The basic mechanism works as follows:

- You define one or more classes within Caché. These classes can represent persistent objects stored within the Caché database or transient objects that run within a Caché server.
- Caché generates Java classes that correspond to your Caché classes. These Java classes include methods which correspond to Caché methods on the server as well as accessor (get and set) methods for object properties.
- At run time, your Java application connects to a Caché server. It can then create instances of Java objects that correspond to objects within the Caché server. You can use these objects as you would

any other Java objects. Caché automatically manages all communications as well as client-side data caching.

The run time architecture consists of the following:

- A Caché database server (or servers).
- A Java Virtual Machine (JVM) in which your Java application runs. Caché does not provide a JVM but works with a standard JVM. See the Sun Microsystems Java SOFTWARE MANUAL DOWNLOAD page at: <http://Java.com/en/download/manual.jsp> for information on obtaining the Java Runtime Environment for your platform.
- A Java application (including servlet, applets, or Swing-based applications).
- At run time, the Java application connects to Caché using either an object connection interface or a standard JDBC interface. All communications between the Java application and the Caché server use the TCP/IP protocol.

## 1.2 Installation and Configuration

All applications using the Caché Java binding are divided into two parts: a Caché server and a Java client. The Caché server is responsible for database operations as well as the execution of Caché object methods. The Java client is responsible for the execution of all Java code (such as additional business logic or the user interface). When an application runs, the Java client connects to and communicates with a Caché server via a TCP/IP socket. The actual deployment configuration is up to the application developer: the Java client and Caché server may reside on the same physical machine or they may be located on different machines. The Java client can be a simple JVM or an Enterprise Java Bean (EJB) Server.

### 1.2.1 Java Client Requirements

Caché requires that the Java client use the Java SDK, version 1.4 or higher.

The Java components required to connect to Caché are contained within the file `CacheDB.jar`, located in `<cachesys>\Dev\java\lib\`. (For the location of `<cachesys>` on your system, see [Default Caché Installation Directory](#) in the *Caché Installation Guide*).

The Java environment uses the system environment variable `CLASSPATH` to find files that it needs to function. It must include the location of the `CacheDB.jar` file (part of the Caché installation).

## 1.2.2 The Caché Java Packages

Caché provides Java support principally through the `com.intersys.objects` package, which implements the connection and caching mechanisms required to communicate with a Caché server. To take advantage of Java reflection functionality, Caché includes a `com.intersys.objects.reflect` package. Reflection provides information about class members and offers a way to operate on those members. There is also a `com.intersys.jdbc` package, which supplies JDBC connectivity.

Other packages (such as the `com.intersys.cache.*` group) are used internally by Caché.

## 1.2.3 Caché Server Configuration

Very little configuration is required to use a Java client with a Caché server. The Java sample programs provided with Caché should work with no change following a default Caché installation. This sections describes the server settings that are relevant to Java and how to change them.

Every Java client that wishes to connect to a Caché server needs a URL that provides the server IP address, port number, and Caché namespace, plus a user name and password.

The Java sample programs use the following connection information:

```
String url = "jdbc:Cache://127.0.0.1:1972/SAMPLES";  
String user = "_SYSTEM";  
String password = "SYS";
```

To use a Java application using the Java binding or JDBC with a Caché server, do the following:

- Make sure that the Caché server is installed and running.
- Make sure that you know the IP address of the machine on which the Caché server is running. The Java sample programs use "localhost". If you want a sample program to connect to a different system you will need change its connection string and recompile it.
- Make sure that you know the TCP/IP port number on which the Caché server is listening. The Java sample programs use 1972; if you want a sample program to use a different port you will need change its connection string and recompile it.
- Make sure that you have a valid SQL user name and password to use to establish a connection. (You can manage SQL user names and passwords on the **[Home] > [Security Management] > [Users]** page of the System Management Portal). The Java sample programs use the administrator user name, "\_SYSTEM" and the default password of "SYS" or "sys". Typically, you will change the default password after installing the server. If you want a sample program to use a different user name and password you will need change it and recompile it.
- Make sure that your connection URL includes a valid Caché namespace. This should be the namespace containing the classes and data your program uses. The Java samples connect to the SAMPLES namespace, which is pre-installed with Caché.

## 1.3 Sample Programs

Caché comes with a set of sample programs that demonstrate the use of the Caché Java binding. These samples are located in the `<cachsys>/dev/java/Samples/` subdirectory of the Caché installation. (For the location of `<cachsys>` on your system, see [Default Caché Installation Directory](#) in the *Caché Installation Guide*).

The sample programs include:

- `SampleApplication.java` — Demonstrates basic functionality of accessing Caché from Java.
- `CJTest1.java` — Demonstrates how to connect to a Caché database from Java. Once the connection is established, the program displays some server information.
- `CJTest2.java` — Demonstrates how to create a new instance of a Caché object. It also sets some object property values and displays these to the console.
- `CJTest3.java` — Demonstrates how to open an existing instance of a Caché object and access its properties.
- `CJTest4.java` — Demonstrates how to open an existing instance of a Caché object, access its properties, modify their values, and save the object to the Caché database.
- `CJTest5.java` — Demonstrates how to open an existing instance of a Caché object and manipulate a collection property.
- `CJTest6.java` — Demonstrates how to use the `CacheQuery` object to process a pre-defined class query.
- `CJTest7.java` — Demonstrates how to use the `CacheQuery` object to process an SQL statement.
- `CJTest8.java` — Illustrate processing employee subclass and company/employee relationship.
- `CJTest9.java` — Process an SQL query using `openByQuery`.
- `JDBCQuery.java` — Demonstrates the use of JDBC to run SQL query against a Caché database.
- `DumpObject.java` — Use `ObjectDump` to list the contents of an object.
- `DynamicBindingSample.java` — Process an object using dynamic binding.
- `LightDynamicBindingSample.java` — Process an object using light dynamic binding.

All of these applications use the `Sample.Person` class from the `Sample` package in the `SAMPLES` namespace (accessible in Caché Studio). This Caché class has already been projected to Java as `<cachsys>/dev/java/samples/Sample/Person.java`. For your applications, you need to project Java classes that represent the Caché classes (see [Generating Java Proxy Classes](#)). If you haven't used the `Sample` package before, you should open it in Cache Studio and make sure it is properly compiled.



# 2

## Using the Java Binding

This chapter provides detailed examples of how to use the Caché Java binding within a Java application.

### 2.1 Generating Java Proxy Classes

To create a Java projection of a Caché class, specify that the Caché class automatically creates a Java class whenever it is compiled. To do this, add a projection definition to the Caché class. This projection definition specifies that the Class Compiler also generates Java code for this class whenever the class is compiled.

The process is as follows:

- In the Caché Studio, establish a connection to the namespace of the class and open the class.
- Add a Java projection definition to the class using the Caché Studio's New Projection Wizard: Invoke the `New Projection` choice from the `Add` submenu in the `Class` menu.
- After choosing a name for the projection, specify its type as `%Projection.Java`.
- Enter the `ROOTDIR` parameter, which specifies the directory where the generated Java class will be written.
- Compile the Caché class. This generates the Java projection of the class, whether compilation occurs in the Studio or from the Caché Terminal command line.

At this point, you have added a line of code to your class definition similar to the following:

```
Projection PrjName AS %Projection.Java;
```

where `PrjName` is the name of the projection.

You can now compile the Java class using either the `javac` command or some other tool. Whenever you modify and compile the Caché, its projection automatically updates the projected Java class.

### 2.1.1 Generating a Java Class from a Caché Class

To use a Caché class from a Java client, generate a Java class to run on the client side.

You can specify that a Caché class automatically creates a Java class whenever it is compiled; to do this, add a projection definition to the Caché class. This projection definition specifies that the Class Compiler also generates Java code for this class whenever the class is compiled. For more information, see the [Projections](#) chapter in *Using Caché Studio*.

To generate a Java class, do the following:

- In the Caché Studio, establish a connection to the class' namespace and open the class. For instance, to use the `Person` sample class, connect to the `SAMPLES` namespace, open the `Person` class in the `Sample` package.
- Add a Java projection definition to the class using the Caché Studio's New Projection Wizard: Invoke the `New Projection` choice from the `Add` submenu in the `Class` menu.
- After choosing a name for the projection, specify its type as `%Projection.Java`.
- For the `ROOTDIR` parameter, enter the directory where the generated Java class will reside.
- On the last screen of the New Projection Wizard, click `Finish`. At this point, the wizard has added a line of code to your class definition similar to the following:

```
Projection MyProjection As %Projection.Java(ROOTDIR="C:\temp\");
```

- Compile the Caché class. This generates the Java projection of the class, whether compilation occurs in the Studio or from the Caché Terminal command line.
- Compile the Java class. You can do this either using the `javac` command or any other tool that you use for compiling Java classes.

## 2.2 Using Objects

A Caché Java binding application can be quite simple. Here is a complete sample program:

```
import com.intersys.objects.*;
public class TinyBind {
    public static void main( String[] args ) {
        try {
            // Connect to the Cache' database
            String url="jdbc:Cache://localhost:1972/SAMPLES";
            String username="_SYSTEM";
```



```

String password="SYS";
Database dbconnection =
    CacheDatabase.getDatabase (url, username, password);
// Create and use a Cache' object
Sample.Person person = new Sample.Person( dbconnection );
person.setName("Doe, Joe A");
System.out.println( "Name: " + person.getName() );
}
// Handle exceptions
catch (CacheException ex) {
    System.out.println( "Caught exception: " +
        ex.getClass().getName() + ": " + ex.getMessage() );
}
}
}

```

This code performs the following actions:

- Imports the `com.intersys.objects.*` packages.
- Connects to the Caché database:
  - Defines the information needed to connect to the Caché database.
  - Creates a Database object (`dbconnection`).
  - Creates and uses a Caché object:
- Uses the Database object to create an instance of the Caché `Sample.Person` class.
  - Sets the Name property of the `Sample.Person` object.
  - Gets and prints the Name property.
  - Performs standard exception handling.

The following sections discuss these basic actions in more detail.

## 2.2.1 Creating a Connection Object

The `CacheDatabase` class is a Java class that manages a connection to a specific Caché server and namespace. It has a static method, `getDatabase()`, for creating a connection. This method returns a connection to a Caché database that is derived from the `Database` interface.

To establish a connection:

- Import the appropriate packages, which will always include the `com.intersys.objects.*` packages (which include `CacheDatabase`):

```
import com.intersys.objects.*;
```

- Next, declare and initialize variables for use in establishing the connection:

```
Database dbconnection = null;
String url="jdbc:Cache://localhost:1972/NAMESPACE_NAME";
String username="_SYSTEM";
String password="sys";
```

The `getDatabase()` method establishes a TCP/IP connection from the Java client to the Caché server. The method takes three arguments, where the first includes a specified IP address (here, the local host, 127.0.0.1), a specified port number (here, 1972), and to a specified namespace (here, SAMPLES); the second and third arguments are the username and password, respectively, for logging into the server.

It returns a Database object, here called `dbconnection`. You can then use `getDatabase()` to establish the connection:

```
dbconnection = CacheDatabase.getDatabase(url, username, password);
```

**Note:** It is also possible to create a connection and run queries through the Java-standard JDBC connection interface. For details, see [Using a JDBC Connection for Dynamic Queries](#) in *Using Caché with JDBC*.

## 2.2.2 Creating and Opening Proxy Objects

The following code attempts to connect to the local Caché server:

```
String url="jdbc:Cache://localhost:1972/SAMPLES";
String username="_SYSTEM";
String password="sys";
//...
dbconnection = CacheDatabase.getDatabase (url, username, password);
```

Next, the program uses standard Java functionality to prompt the user for an ID to open and to get that value. Once there is an ID, the next step is to open the specified object:

```
person = (Sample.Person)Sample.Person._open(dbconnection, new Id(strID));
```

This code invokes the `_open()` method of the `Person` object in the `Sample` package. This method takes two arguments: the database that contains the object being opened, and the ID of the object being opened. The value being returned is narrowed (cast) as an instance of `Sample.Person` because `_open()` is inherited from the `Persistent` class and, hence, returns an instance of that class.

## 2.2.3 Using Methods and Properties

Once the object is open, the program displays the value of the object's `Name` property.

```
System.out.println("Name: " + person.getName());
```

Note that, unlike on the Caché server, references to object properties are through the `get()` and `set()` methods, rather than through direct references to the properties themselves.

## Embedded Objects

Next, it displays the value of the City property and then gives the City property a new value:

```
System.out.println("City: " + person.getHome().getCity());
person.getHome().setCity("Ulan Bator");
```

The lines of code that manipulate the City property demonstrate the observation and modification of the properties of an embedded object. If a property is an object (such as the Home property), then it has its own properties (such as the City property) with accessor methods. You can invoke these methods using cascading dot syntax.

### 2.2.4 Saving and Closing

Having given the City property a new value, the application then saves the object, displays the value, closes the object, de-assigns it, and then shuts itself down.

```
person._save();

// Report the new residence of this person */
System.out.println( "New City: " + person.getHome().getCity());

// * de-assign the person object */
dbconnection.closeObject(person.getOref());
person = null;

// Close the connection
dbconnection.close();
```

Java performs garbage collection, so it does not require that you close any objects. However, InterSystems recommends that you close all objects, in order to ensure their integrity.

### 2.2.5 A Sample Java Binding Application

SampleApplication.java is a simple Java program that connects to the Caché SAMPLES database, opens and modifies an instance of a Sample.Person object saved within the database, and executes a predefined query against the database. This application is invoked from the operating command line, reads an object id value from the command line, and writes output using the Java system.out object. This example assumes that Caché and Java are running on a Windows machine.

SampleApplication.java is located in the < cachesys >/dev/Java/Samples directory (see [Default Caché Installation Directory](#) in the *Caché Installation Guide* for the location of < cachesys > on your system). In the Samples directory, compile the program:

```
javac SampleApplication.java
```

And then run it:

```
java SampleApplication
```

When executed, this program yields results such as:

```
C:\java> java SampleApplication
Enter ID of Sample.Person object to be opened: 1
Name: Isaacs,Sophia R.
City: Tampa
New City: Ulan Bator

C:\java>
```

Here is the complete Java source for the sample application:

```
/*
 * SampleApplication.java
 */
import java.io.*;
import java.util.*;
import com.intersys.objects.*;

public class SampleApplication {
    public static void main(String[] args){
        Database dbconnection = null;
        String url="jdbc:Cache://localhost:1972/SAMPLES";
        String username="_SYSTEM";
        String password="sys";
        ObjectServerInfo info = null;
        Sample.Person person = null;

        try {
            // Connect to Cache on the local machine, in the SAMPLES namespace
            dbconnection = CacheDatabase.getDatabase (url, username, password);

            // Open an instance of Sample.Person,
            // whose ID is read in from the console
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            System.out.print("Enter ID of Person object to be opened:");
            String strID = br.readLine();

            // Use the entered strID as an Id and use that Id to
            // to open a Person object with the _open() method inherited
            // from the Persistent class. Since the _open() method returns
            // an instance of Persistent, narrow it to a Person by casting.
            person = (Sample.Person)Sample.Person._open(dbconnection, new Id(strID));

            // Fetch some properties of this object
            System.out.println("Name: " + person.getName());
            System.out.println("City: " + person.getHome().getCity());

            // Modify some properties
            person.getHome().setCity("Ulan Bator");

            // Save the object to the database
            person._save();

            // Report the new residence of this person */
            System.out.println("New City: " + person.getHome().getCity());

            /* de-assign the person object */
            dbconnection.closeObject(person.getOref());
            person = null;

            // Close the connection
            dbconnection.close();
        }
    }
}
```

```

    } catch (Exception ex) {
        System.out.println("Caught exception: "
            + ex.getClass().getName()
            + ": " + ex.getMessage());
    }
}
}

```

## 2.3 Using Streams

Caché allows you to create properties known as streams that hold large sequences of characters, either in character or binary format. Character streams are long sequences of text, such as the contents of a free-form text field in a data entry screen. Binary streams are usually image or sound files, and are akin to BLOBs (binary large objects) in other database systems.

The process for using a stream in Java is:

- When creating a Java client, define and initialize a variable of the appropriate type – either `GlobalBinaryStream` or `GlobalCharacterStream`. For instance, if you are using a character stream, define a variable such as:

```
com.intersys.classes.GlobalCharacterStream localnotes = null;
```

You can then read content from an instantiated class' stream:

```
localnotes = myTest.getNotes();
```

- Once you have a local copy of the stream, you can read from it

```

IntegerHolder len = new IntegerHolder( new Integer(8) );
while (len.value.intValue() != 0 ) {
    System.out.println( localnotes._read( len) );
} ;

```

The `_read()` method's argument is an integer hold specifying how many characters to read and its return value is the characters that it reads. It also places the number of characters successfully read (whether the number specified or fewer) in the integer hold variable that was its argument.

When you are writing to or reading from a stream, Caché monitors your position within the stream, so that you can move backward or forward.

The Caché stream classes are `GlobalBinaryStream` and `GlobalCharacterStream` in the `com.intersys.classes` package. The basic methods involving streams are:

### **\_write()**

Adding content

**\_writeLine()**

Adding content (character streams only)

**\_read()**

Reading content

**\_readLine()**

Reading content (character streams only)

**\_moveToEnd()**

Going to the stream's end

**\_rewind()**

Going to the stream's beginning

**atEnd()**

Check if the current position is at the end of the stream

**\_sizeGet()**

Getting the size of the stream

**isNull()**

Checking if the stream has content or not

**\_clear()**

Erasing the stream's content

## 2.4 Using Queries

A Caché query is designed to fit into the framework of JDBC but provides a higher level of abstraction by hiding direct JDBC calls behind a simple and complete interface of a dynamic query. It has methods for preparing an SQL statement, binding parameters, executing the query, and traversing the result set.

### 2.4.1 Class Queries

Caché allows you to define queries as part of a class. These queries are then compiled and can be invoked at run time.

---

To invoke a predefined query, use the `CacheQuery` class:

- Establish a connection to a Caché server (see [Creating a Connection Object](#) for details on this process).
- Create an instance of a `CacheQuery` object using code such as:

```
myQuery = new CacheQuery(factory, classname, queryname);
```

where `factory` specifies the existing connection, `classname` specifies the class on which the query is to be run, and `queryname` specifies the name of the predefined query that is part of the class.

- Once you have instantiated the `CacheQuery` object, you can invoke the predefined query:

```
java.sql.Result ResultSet = myQuery.execute(param1);
```

This method accepts up to three arguments, which it passes directly to the query; if the query accepts four or more arguments, you can pass in an array of argument values. The method returns an instance of a standard JDBC `ResultSet` object.





# 3

## Java Proxy Class Mapping

The Caché Java binding reads the definition of a Caché class and uses the information to generate a corresponding Java class. The generated class provides remote access to an instance of a Caché object from within a Java application. This chapter describes how Caché objects and data types are mapped to Java code, and provides details on the objects and methods provided by the Caché Java binding.

### 3.1 Classes

The projections of Caché classes receive names as Java classes in accordance with the naming conventions listed below. The type of a class (such as persistent or serial) determines its corresponding Java superclass. For example, persistent classes have corresponding Java classes derived from the Java Persistent class.

#### 3.1.1 Entity Names

A Caché identifier, such as class or method name, is usually projected to a corresponding Java identifier with the same name. If a Caché identifier is a Java reserved word, the corresponding Java identifier will be preceded by an underscore ("\_").

For details on Caché Basic and ObjectScript naming conventions, see [Variables](#) in *Using Caché ObjectScript*, [Naming Conventions](#) in *Using Caché Objects*, [Identifiers and Variables](#) in *Using Caché Basic*, and [InterSystems Reserved Names](#) in the *InterSystems Product Miscellany* article.

#### Class Names

All class names are unchanged. All Caché packages become Java packages, and the "%" characters within a package name are translated to "\_". In your code, the Caché and Java package names must match each other exactly.

The `%Library` package is an exception to this rule. There is no one-to-one correspondence between `%Library` and any of Java library packages, but most common `%Library` classes have their stubs in the `com.intersys.classes` package. For example, the `%Library.Persistent` class would be mapped as `"com.intersys.classes.Persistent"`.

### Property Names

You can refer directly to Caché properties. To conform to Java property-handling style, the projection of each Caché property includes two accessor methods: `get<Prop>()` and `set<Prop>()`, where `<Prop>` is the name of the projected property. If the property name starts with "%", it is replaced by "set\_". Hence, the projection of the `Color` property would include `getColor()` and `setColor()` methods. The projection of a `%Concurrency` property would have `get_Concurrency()` and `set_Concurrency()` methods.

### Method Names

Typically, method names are mapped directly, without changes. Exceptions are:

- If the method name starts with "%", this is replaced by "sys\_".
- If the method name is a Java reserved word, "\_" is prepended to the name.
- For methods of classes that are part of the `%Library` package, the leading "%" is replaced with a "\_" and the first letter is converted to lowercase.

### Formal Variable Names

If a variable within a method formal list starts with "%" it is replaced by "\_". If the name is a Java reserved word, "\_" is prepended to the name.

### Packages

In general, the Caché package name for a class is used as its Java package name. If a Caché class defines a class parameter, `JAVAPACKAGE`, then the Java Generator uses the parameter value for a package name.

## 3.1.2 Methods

Methods of Caché classes are projected to Java as stub methods of the corresponding Java classes. Instance methods are projected as Java instance methods; class methods are projected as Java static methods. When called on the client, a method invokes the actual method implementation on the Caché server.

If a method signature includes arguments with default values, Caché generates multiple versions of the method with different numbers of arguments to simulate default argument values within Java.

For example, suppose you define a simple Caché class with one method as follows:

```

Class MyApp.Simple Extends %RegisteredObject {
    Method LookupName(id As %String) As %String {
        // lookup a name using embedded SQL
        Set name = ""
        &sql(SELECT Name INTO :name FROM Person WHERE ID = :id)
        Quit name
    }
}

```

The resulting projected Java class would look something like:

```

public class Simple extends Object {
    //...
    public String LookupName(String id) throws CacheException {
        // generated code to invoke method remotely...
        // ...
        return typedvalue;
    }
}

```

When a projected method is invoked from Java, the Java client first synchronizes the server object cache, then invokes the method on the Caché server, and, finally, returns the resulting value (if any). If any method arguments are specified as call by reference then their value is updated as well.

## System Methods

In addition to any methods defined by a Caché class, the projected Java class includes a number of automatically generated system methods to perform various services:

- **\_close()** — Shuts down an object on the server from the client (by invoking the object's %Close method).
- **\_open()** — For persistent objects, open an instance of object stored within the database using the instance's OID as a handle.
- **\_openId()** — For persistent objects, open an instance of object stored within the database using the instance's class ID value as a handle.

## Passing Null Values and Empty Strings

It is important to remember that Caché represents null values and empty strings differently:

- A Caché NULL value is represented as " " (an empty string).
- A Caché empty string is represented as character \$c(0).

This may seem counter-intuitive to many Java programmers, but is consistent with the way NULL is treated in SQL.

An empty string returned from a method with a non-string return type is converted to a null value, making it necessary to perform a null check on the returned value before using the corresponding object. For example the following line of code could throw a NullPointerException:

```
return myCacheObject.getMyInteger().intValue();
```

To avoid this, you would perform the following check:

```
Integer myInteger = myCacheObject.getMyInteger();
    if (myInteger == null) {
        // handle null value here
    } else
return myInteger.intValue();
```

This is also true for parameters returned by reference.

### 3.1.3 Properties

You can refer to each of a projection's properties using its two accessor methods: **get<Prop>()** to get its value and **set<Prop>()** to set its value.

The values for literal properties (such as strings or integers) are represented using the appropriate Java data type classes (such as `String` or `Integer`).

The values for object-valued properties are represented using the appropriate projected Java class. In addition to the `get` and `set` methods, an object-valued property has additional methods that provide access to the persistent object ID for the object: **idget<Prop>()** and **idset<Prop>()**.

For example, suppose you have defined a persistent class within `Caché` containing two properties, one literal and the other object-valued:

```
Class MyApp.Student Extends %Persistent [ClassType = persistent] {
    // Student's name
    Property Name As %String;
    // Reference to a school object
    Property School As School;
}
```

The Java representation of `MyApp.Student` contains `get` and `set` accessors for both the `Name` and `School` properties. In addition, it provides accessors for the Object Id for the referenced `School` object.

```
public class Student extends Persistent {
    // ...
    public String getName() throws CacheException {
        // implementation...
    }
    public void setName(String value) throws CacheException {
        // implementation...
    }
    public School getSchool() throws CacheException {
        // implementation...
    }
    public void setSchool(School value) throws CacheException {
        // implementation...
    }
    public Id idgetSchool() throws CacheException {
        // implementation...
    }
    public void idsetSchool(Id value) throws CacheException {
        // implementation...
    }
}
```

## Property Caching

When a projected Java object is instantiated within Java, it fetches a copy of its property values from the Caché server and copies them into a local Java-side cache. Subsequent access to the object's property values are made against this cache, reducing the number of messages sent to and from the server. Caché automatically manages this local cache and ensures that it is synchronized with the corresponding object state on the Caché server.

Property values for which you have defined a get or set method within your Caché class definition (such as for a property whose value depends on other properties) are not stored within the local cache. Instead, when you access such properties the corresponding accessor method is invoked on the Caché server. As this can entail higher network traffic, exercise care when using such properties in a Java environment.

## 3.2 Primitive Data Types

Caché uses various literal data types (such as strings or numbers) for properties, method return types, and method arguments. Every Caché data type has an associated client data type. This client data type specifies the Java class to which a variable is mapped. Hence, using its client data type, every Caché data type is represented using an appropriate Java object such as Integer or String.

Regardless of a property's type, if it has unset value, then Java represents it using the `null` keyword. For example, suppose you create a new object with an Age property that is of type Integer. Prior to setting this property's value, invoking the `getAge()` method returns `null`.

By default, the `CLIENTDATATYPE` keyword value of a Caché data type determines which Java class is associated with it. The following table describes this correspondence:

*Client Data Type to Java Correspondence*

ClientDataType	Java Class	Java Package	HolderClass
BINARY	byteArray	null	ByteArrayHolder
BOOLEAN	Boolean	java.lang	BooleanHolder
CURRENCY	BigDecimal	java.math	BigDecimalHolder
DATE	Date	java.sql	DateHolder
DOUBLE	Double	java.lang	DoubleHolder

ClientDataType	Java Class	Java Package	HolderClass
ID	Id (a Caché-provided class that represents an Object ID within an extent)	com.intersys.objects	IdHolder
INT	Integer	java.lang	IntegerHolder
LIST	SysList (A Java implementation of Caché \$List structure)	com.intersys.objects	SysListHolder
LONG VARCHAR	String	java.lang	StringHolder
NUMERIC	BigDecimal	java.math	BigDecimalHolder
OID	Oid (a Caché-provided class that represents a complete Object ID)	com.intersys.objects	OidHolder
STATUS	StatusCode (a Caché-provided class that represents the status)	com.intersys.objects	StatusCodeHolder
TIME	Time	java.sql	TimeHolder
TIMESTAMP	TimeStamp	java.sql	TimeStampHolder
VARCHAR	String	java.lang	StringHolder
Void	void	null	null

Object-valued types (references to other object instances) are represented using the corresponding Java class. Certain Caché objects are treated as special cases. For example, streams are mapped to the Java stream object, and collections are projected as collection objects (a class created by InterSystems for Java client use). If a method argument is passed by reference then a “holder” class is used, such as IntegerHolder or StringHolder.

The `JAVATYPE` parameter allows you to override the default value and associate a property with a specified client Java class.

