Using Java with Caché

Version 2018.1
2024-04-03

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

Using Java with Caché
Caché Version 2018.1 2024-04-03

For Support questions about any InterSystems products, contact:

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

Table of Contents

List of Figures

List of Tables

1
About This Book

See the Table of Contents for a detailed listing of the subjects covered in this document.

Important: **The Caché Java Binding is Deprecated**

XEP is the recommended persistence technology for high-performance simple to medium complexity object hierarchies in Java projects. See “Using XEP Event Persistence” in *Using Java with Caché XEP*. For existing applications, InterSystems provides a migration path to InterSystems IRIS®. (see “Migrating to InterSystems IRIS”).

Java Persistence Architecture (JPA) is the recommended persistence technology for complex object hierarchies in Java projects. Caché and Ensemble currently support JPA 1.0 and 2.0 via the Hibernate implementations of the JPA specifications. See “Using the Caché Hibernate Dialect” in *Using Caché with JDBC*.

This book is a guide to the Caché Java Binding, which provides a simple, direct way to use Caché objects within a Java application.

The following topics are covered:

- The Caché Java Binding — introduces the Java binding, and describes its architecture and system requirements.
- Using the Java Binding — provides a quick description of how to use the binding.
- Java Proxy Class Mapping — describes how Caché classes and data types are mapped to Java.
- Migrating to InterSystems IRIS — describes how existing Java Binding applications are supported on InterSystems IRIS.

Related Documents

The following documents also contain material related to the Java binding:

- *Using Caché with JDBC* describes how to use the Caché JDBC driver to connect to external JDBC data sources.

2

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. This mechanism requires no object-relational mapping or additional middleware.

- *The Caché JDBC Driver* — Caché includes a level 4 (pure Java) JDBC driver that supports the JDBC version 4.1 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. See *Using Caché with JDBC* for details.

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

2.1 Java Binding Architecture

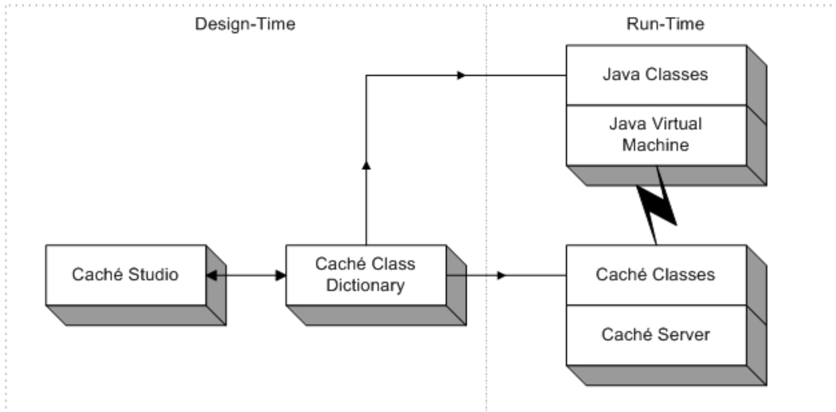
The Caché Java binding gives Java applications a way to access and manipulate objects contained within a Caché server. These objects can be persistent objects stored within the Caché database or they can be transient objects that perform operations within a Caché server.

The Caché Java binding consists of the following components:

- *The Caché Java Class Generator* — an extension to the class compiler that generates pure Java classes from classes defined in the Caché Class Dictionary.
- *The Caché Java Class Packages* — pure Java classes that work in conjunction with the Java classes generated by the Caché Java Class Generator, providing them with transparent connectivity to the objects stored in the Caché database. (See "The Caché Java Class Packages").
- *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 class compiler can automatically create Java client classes for any classes contained within the Caché Class Dictionary. These generated Java classes communicate at runtime 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:

Figure 2–1: Java Client/Server 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.
- The system 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 runtime, 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 runtime 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 Java Downloads for All Operating Systems 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 runtime, 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.

2.2 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. Only the Caché server machine requires a copy of Caché.

2.2.1 Java Client Requirements

The online *InterSystems Supported Platforms* document for this release specifies the current requirements for all Java-based binding applications:

- See "Supported Java Technologies" for supported Java releases.
- See "Supported Client Platforms" for supported Java client platforms.
- If your client application and the Caché server are not running on the same version of Caché, see "Supported Version Interoperability" for information on compatibility between versions.

The core components of the Java binding are files named cache-jdbc-2.0.0.jar and cache-db-2.0.0.jar, which contain the Java classes that provide the connection and caching mechanisms for communication with the Caché server, JDBC connectivity, and reflection support. Client applications do not require a local copy of Caché, but the cache-jdbc-2.0.0.jar and cache-db-2.0.0.jar files must be on the class path of the application when compiling or using Java proxy classes. See "The Caché Java Class Packages" for more information on these files.

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 section 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 username 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 run a Java or JDBC client application, make sure that your installation meets the following requirements:

- The client must be able to access a machine that is currently running a compatible version of the Caché server (see "Supported Version Interoperability" in the online *InterSystems Supported Platforms* document for this release). The client and the server can be running on the same machine.
- Your class path must include the versions of cache-jdbc-2.0.0.jar and cache-db-2.0.0.jar that correspond to your version of the Java JDK (see "The Caché Java Class Packages").
- To connect to the Caché server, the client application must have the following information:
  - 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 to change its connection string and recompile it.
  - 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 to change its connection string and recompile it.
  - A valid SQL username and password. You can manage SQL usernames and passwords on the System Administration > Security > Users page of the Management Portal. The Java sample programs use the administrator username, `"_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 username and password you will need to change it and recompile it.
  - The server namespace containing the classes and data that your client application will use. The Java samples connect to the SAMPLES namespace, which is pre-installed with Caché.

2.3 The Caché Java Class Packages

The files containing the Caché Java class packages are located in <install-dir>\Dev\java\lib\<java-release>, where <install-dir> is the root directory of your Caché installation and <java-release> corresponds to the Java JDK you are using. See "Caché Installation Directory" in the *Caché Installation Guide* for the location of <install-dir> on your system. For supported Java releases, see "Supported Java Technologies" in the online *InterSystems Supported Platforms* document for this release.

The Java class packages are contained in the following files:

- cache-jdbc-2.0.0.jar — all other files in this list are dependent on this file. It can be used by itself for JDBC binding applications.
- cache-db-2.0.0.jar — required for most Java binding applications.
- cache-extreme-2.0.0.jar — required only for Caché XEP binding applications (see *Using Java with Caché XEP*).
- cache-gateway-2.0.0.jar — required for JDBC SQL Gateway applications (see "Using the Caché SQL Gateway with JDBC" in *Using Caché with JDBC*).

See the JavaDoc in <install-dir>\dev\java\doc\ for the latest and most complete description of these packages.

2.4 Other Documentation

The standard Caché installation provides JavaDoc and other documents describing various ways to use Java with Caché.

- Complete JavaDoc is available for the Caché Java class cackages. The main index.html file is located in <install-dir>\dev\java\doc\ (see "Caché Installation Directory" for the location of <install-dir> on your system).
- *Using Caché with JDBC* describes how to use the Caché JDBC driver to connect to external JDBC data sources.

3

Using the Java Binding

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

3.1 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 Studio, establish a connection to the namespace of the class and open the class.
- Add a Java projection definition to the class using the 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 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.

Important: You should never modify the generated Java Projection code directly, or attempt to add custom Java connection code, since this can have serious unintended consequences. The generated code depends on InterSystems internal code that may be changed without notice.

3.1.1 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 Studio*.

To generate a Java class, do the following:

- In the 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 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 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.

3.2 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.

3.2.1 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 DriverManager to Connect in *Using Caché with JDBC*.

3.2.2 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.

3.2.3 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.

3.2.4 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();
```

Note: **Always Close Objects and Connections**

Before closing a connection, it is important to call **closeObject()** on all objects that use the connection. Failure to do so may compromise the integrity of your data on the server. Objects are opened with a default concurrency value of `1`, meaning that a read lock is acquired if the class uses more than one data node (see "Object Concurrency" in *Using Caché Objects*).

You must also call **close()** on all connection instances before they go out of scope. Failure to do so can cause memory leaks and other problems.

3.2.5 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());
        }
    }
}
```

3.3 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

3.4 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.

3.4.1 Class Queries

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

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(parm1);
  ```

  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.

4

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.

4.1 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.

4.1.1 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 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 Rules and Guidelines for Identifiers in the Caché Programming Orientation Guide.

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.

4.1.2 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, the system 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.

4.1.3 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: **idset<Prop>()** and **idget<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 {
   /// 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.

4.2 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:

Table 4–1: 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
ID	Id (a Caché-provided class that represents an Object ID within an extent)	com.intersys.objects	IdHolder
INT	Integer	java.lang	IntegerHolder

ClientDataType	Java Class	Java Package	HolderClass
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.

5

Migrating to InterSystems IRIS

In InterSystems IRIS®, the Caché Java Binding has been replaced by the much more powerful Java Native SDK, which should be used for all new development. However, InterSystems IRIS also offers an easy upgrade path for existing Java Binding applications that would be difficult to replace.

The InterSystems IRIS version of the Java binding is intended to work as much like the old Caché Java binding as possible, but some changes were necessary. Most of the changes were an unavoidable consequence of the move from Caché to InterSystems IRIS. Other changes were introduced to fix bugs and to eliminate the most complex and problematic features of the old binding. For example, all Caché Holder classes were replaced by a single IRISReference class that handles all supported datatypes. The following sections describe these changes in detail.

5.1 New Classes and System Requirements

- The top level Caché com.intersys.* domain has been replaced by the InterSystems IRIS com.intersystems.* domain.
- Not all com.intersys.* classes were ported. The new binding keeps only those that are used directly by projected Java classes.
- All Java binding library classes are consolidated to com.intersystems.binding. These classes depend on com.intersystems.jdbc.
- Since com.intersystems.jdbc is needed, JDK 1.8 or higher must be used.
- In Mac OS and UNIX environments, users should ensure that InterSystems IRIS has write privilege to the folders where proxy .java classes are projected. One way to do this is to set the group-owner of the folders to irisusr.

5.2 Connection Code Changes

Connection code now uses connection class IRISConnection rather than Database, and the connection string starts with `jdbc:IRIS:` instead of `jdbc:Cache:`. The following examples demonstrate these differences:

Legacy Connection Code

```
import com.intersys.objects.Database;

String url = "jdbc:Cache://localhost:1972/User/jdbc.log";
String user = "_SYSTEM";
String pwd = "SYS";
Database conn = CacheDatabase.getDatabase (url, user, pwd);

Person p = new Person(conn);
```

New Connection Code

```
import com.intersystems.jdbc.IRISConnection;

String url = "jdbc:IRIS://localhost:1972/User/Binding.log";
String user = "_SYSTEM";
String pwd = "SYS";
IRISConnection conn = (IRISConnection) java.sql.DriverManager.getConnection(url, user, pwd);

Person p = new Person(conn);
```

5.3 Interface Changes

This section describes interface features that have been removed or renamed.

- Changes to RegisteredObject interface
- Changes to SerialObject interface
- Changes to Pass-By-Reference interface
- Other methods no longer supported
- Projection flags no longer supported

5.3.1 Changes to RegisteredObject interface

The following changes were made in the conversion from com.intersys.classes.RegisteredObject to com.intersystems.binding.RegisteredObject:

Removed methods:

- **createObjects()**: Light binding feature removed.
- **saveObjects()**: Light binding feature removed.
- **getCacheClass()**: expose Caché Reflection API and dynamic binding, for which there is no equivalent in InterSystems IRIS.
- **CheckAllFieldsValid()** / **CheckAllMethods()**: these belong to RegisteredObject class, but were replaced by a no-op in any subclasses. They also use Caché Reflection API.
- **check<propertyname>Valid()**: this method was projected as no-op.
- **addToBatchInsert()** / **executeBatchInsert()**: these methods mass insert data, which should now be accomplished through JDBC.

Changed methods:

- **getCacheClassName()** --> **getIRISClassName()**: changed method name.

5.3.2 Changes to SerialObject interface

The following changes were made in the conversion from com.intersys.classes.SerialObject to com.intersystems.binding.SerialObject:

Removed methods:

- **open(Database, byte[])**: this method depends on Light binding.
- **createClientObject()**: this method again depends on Light binding, and serves the same purpose as constructor.

5.3.3 Changes to Pass-By-Reference interface

The following changes were made in the conversion from com.intersys.objects.<type>Holder to com.intersystems.binding.IRISReference:

- All holder classes are replaced by a single class IRISReference.
- *NOTE:* for binding classes, com.intersystems.binding.IRISReference should be used, rather than superclass com.intersystems.jdbc.IRISReference.

Changed method interface:

- Holder.**setValue(com.intersys.cache.Dataholder)** --> IRISReference.**setValue(Object)**. It is no longer necessary to wrap objects in Dataholder when passing to **setValue()**. Dataholder is not supported.
- Access modifier of local variable "value" was changed from public to *protected*. Accessor methods should be used instead.

5.3.4 Other methods no longer supported

There is no replacement for two com.intersys.classes.ObjectHandle methods:

- **getOref()** / **getZref()**: there are no longer good ways to retrieve this information.

In com.intersys.classes.CacheRootObject (former superclass of RegisteredObject):

- **castTo(CacheClass)**: This method works with the Caché Reflection API, which has no equivalent in InterSystems IRIS.

5.3.5 Projection flags no longer supported

The following projection flags were removed from server-side %Projection.Java:

- `POJO`: this mode depends on code that cannot be directly ported from Caché, and would have to be completely re-implemented.
- `PROJECTBYREFMETHODSTOPOJO`: also cannot be directly ported.
- `PROJECTABSTRACTSTREAM`: this flag was supposed to restrict the Stream type that could be projected, but the flag was never enabled. All streams project appropriately in both old and new binding.

5.4 Replacements for Caché Library Classes

Classes CacheObject, Database, and CacheException have been replaced as follows:

Caché	*InterSystems IRIS*
com.intersys.cache.CacheObject	com.intersystems.jdbc.IRISObject
com.intersys.objects.Database	com.intersystems.jdbc.IRISConnection
com.intersys.objects.CacheException	java.sql.SQLException • Most projected methods now call *com.intersystems.jdbc.IRISConnection*, which throws a *java.sql.SQLException* upon errors.

Classes in com.intersys.objects are replaced with equivalent new classes in com.intersystems.binding.

Caché	*InterSystems IRIS*
com.intersys.objects.Id	com.intersystems.binding.Id
com.intersys.objects.Oid	com.intersystems.binding.Oid
com.intersys.objects.CacheQuery	com.intersystems.binding.IRISQuery
com.intersys.objects.<type>Holder	com.intersystems.binding.IRISReference • Pass-by-reference undergoes some changes in JDBC, refer to the section below.
com.intersys.objects.CacheInputStream	com.intersystems.binding.InputStream
com.intersys.objects.CacheOutputStream	com.intersystems.binding.OutputStream
com.intersys.objects.CacheReader	com.intersystems.binding.Reader
com.intersys.objects.CacheWriter	com.intersystems.binding.Writer
com.intersys.objects.StatusCode	com.intersystems.binding.StatusCode

Classes in com.intersys.classes are replaced with equivalent new classes in com.intersystems.binding.

Caché	*InterSystems IRIS*
com.intersys.classes.AbstractCacheArray	com.intersystems.binding.AbstractIRISArray
com.intersys.classes.AbstractCacheList	com.intersystems.binding.AbstractIRISList
com.intersys.classes.AbstractStream	com.intersystems.binding.AbstractStream

com.intersys.classes.ArrayOfDataTypes	com.intersystems.binding.ArrayOfDataTypes
com.intersys.classes.ArrayOfObjects	com.intersystems.binding.ArrayOfObjects
com.intersys.classes.BinaryStream	com.intersystems.binding.BinaryStream
com.intersys.classes.CharacterStream	com.intersystems.binding.CharacterStream
com.intersys.classes.ListOfDataTypes	com.intersystems.binding.ListOfDataTypes
com.intersys.classes.ListOfObjects	com.intersystems.binding.ListOfObjects
com.intersys.classes.ObjectHandle	com.intersystems.binding.ObjectHandle
com.intersys.classes.Persistent	com.intersystems.binding.Persistent
com.intersys.classes.RegisteredObject	com.intersystems.binding.RegisteredObject
com.intersys.classes.SerialObject	com.intersystems.binding.SerialObject
com.intersys.classes.RelationshipObject	com.intersystems.binding.RelationshipObject
com.intersys.classes.SerialStream	com.intersystems.binding.SerialStream
com.intersys.classes.GlobalCharacter-Stream	com.intersystems.binding.CharacterStream