



# Data Serialization with the InterSystems Persister for Java

Version 2025.1  
2025-06-03

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble®, InterSystems HealthShare®, InterSystems IRIS®, and TrakCare are registered trademarks of InterSystems Corporation. HealthShare® CMS Solution Pack™, HealthShare® Health Connect Cloud™, InterSystems® Data Fabric Studio™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, and InterSystems TotalView™ For Asset Management are trademarks of InterSystems Corporation. TrakCare is a registered trademark in Australia and the European Union.

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

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Congress Street, Boston, MA 02114, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

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

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

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

For Support questions about any InterSystems products, contact:

**InterSystems Worldwide Response Center (WRC)**

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

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

# Table of Contents

<b>1 The InterSystems Persister for Java .....</b>	<b>1</b>
<b>2 Overview of the Java Persister .....</b>	<b>3</b>
<b>3 Serializing Data with Persister .....</b>	<b>5</b>
3.1 Persister Threading and Buffering .....	6
3.1.1 Using Local Buffers with PersisterBuffer .....	6
3.1.2 Threading with BufferedWriter .....	7
3.2 Persister Buffer Statistics .....	8
<b>4 Implementing Schemas with SchemaManager .....</b>	<b>11</b>
4.1 Synchronizing Schema Definitions .....	11
4.2 Acquiring Schemas from the Server .....	12
4.3 Schema and Extent Utilities .....	13
<b>5 Designing Schemas with SchemaBuilder .....</b>	<b>15</b>
<b>6 Java Persister Examples .....</b>	<b>17</b>
6.1 Hello Persister .....	17
6.2 Continents - Load Data from a Local Array .....	18
6.3 DivvyTrip - CSV bicycle sharing dataset (small - 1.5 million records) .....	19
6.4 ThreadLoader (large - 20 million records) .....	20
<b>7 Quick Reference for Java Persister Classes .....</b>	<b>23</b>
7.1 Class Persister .....	23
7.1.1 Persister Constructor .....	23
7.1.2 Persister Methods .....	24
7.2 class SchemaManager .....	28
7.2.1 SchemaManager Constructor .....	28
7.2.2 SchemaManager Methods .....	28
7.3 class SchemaBuilder .....	31
7.3.1 SchemaBuilder Methods .....	31
7.3.2 SchemaBuilder LogicalSchema Methods .....	36

# List of Figures

Figure 3–1: Persister Architecture ..... 5

# 1

## The InterSystems Persister for Java

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

The InterSystems IRIS Persister for Java is a high speed data persistence engine, featuring a multi-threaded loader capable of ingesting and serializing nearly a million real time records per second. It uses a format based on Apache Avro [<https://avro.apache.org/>], a robust open source data serialization framework.

- Schema-based serialization format allows extremely fast, flexible, reliable data persistence. Since the schema is always stored with the records, data can be exchanged with systems that have no prior knowledge of the data structure.
- Flexible, language agnostic design eases data exchange between systems, programming languages, and processing frameworks.
- Data is stored as serialized instances of ObjectScript classes. Sharded extents can be used, enabling shard factor buffering for parallel writes.
- Schemas can be generated from an existing ObjectScript class, created by inference from source data, or designed using the SchemaBuilder class.

The Persister is the successor to InterSystems XEP. Compared to XEP, the Persister is faster, much easier to implement, and far more powerful.

See the following topics for detailed information:

- [Overview of the Java Persister](#) — provides a quick introduction to important Persister SDK features.
- [Serializing Data with Persister](#) — describes how the [Persister](#) is used to buffer and store serialized data.
- [Implementing Schemas with SchemaManager](#) — describes how a [SchemaManager](#) is used to manage schemas and interface with the server.
- [Designing Schemas with SchemaBuilder](#) — describes how a [SchemaBuilder](#) is used to design custom schemas.
- [Java Persister Examples](#) — lists and describes several small applications that demonstrate various Persister SDK features.
- [Quick Reference for Java Persister Classes](#) — Provides a quick reference for methods discussed in this document.

### Related Documents

The following documents contain related material about InterSystems solutions for Java:

- [Using Java with InterSystems Software](#) provides an overview of all InterSystems Java technologies enabled by the InterSystems JDBC driver, and describes how to use the driver.
- [Using the Native SDK for Java](#) describes how to use Java with the InterSystems Native SDK to access and manipulate ObjectScript classes, objects, and multidimensional global arrays.



# 2

## Overview of the Java Persister

The InterSystems IRIS® Persister for Java is designed to ingest data streams and persist them to a database at extremely high speed. Each thread-safe Persister instance consumes a data stream, serializes each record, and writes each serialized record to an output buffer or pool of buffers. Each buffer in a pool maintains a separate connection to an InterSystems IRIS server.

The Persister SDK uses a format based on the [Apache Avro](#) schema-based data serialization format, which enables extremely fast, flexible, reliable data persistence. The format consists of two parts:

- a *schema*, which describes the structure the data
- data *records*, serialized in a compact format without repetitive structural information

Since the data and the schema are stored together, records can always be serialized or deserialized without any previous knowledge of their structure.

The Persister SDK provides several ways to create schemas. They can be generated from an existing ObjectScript class, created by inference from source data, or designed using the `SchemaBuilder` class.

When data is serialized to an InterSystems IRIS database, schemas are stored on the server in a Schema Registry, where they are available to any Persister application. Each schema in the registry defines a corresponding ObjectScript class, and records are stored as serialized instances of those classes.

The serialized classes are immediately usable through standard access methods, including ObjectScript and SQL. Indexes can be generated during serialization or deferred until later.

The Persister SDK has three main classes:

- [Persister](#) handles all aspects of reading, serializing, buffering, and writing data. Each instance of Persister is bound to one specific schema and its associated database extent. Persisters are thread-safe, and allow precise control and monitoring of buffers and buffer queues. See [Serializing Data with Persister](#) for details.
- [SchemaManager](#) implements schemas and makes them available to Persisters. It maintains a local cache of schemas for the current application, and synchronizes the cache with a persistent Schema Registry on the server. It provides Persisters with a connection to the database, and includes tools for creating and changing both schemas and their corresponding database extents. See [Implementing Schemas with SchemaManager](#) for details.
- [SchemaBuilder](#) is a utility that provides methods to construct schema definitions, which are returned as JSON strings. All construction methods are static and calls can be nested. Field types can be specified directly, or can be inferred from Java types, classes, or objects. See [Designing Schemas with SchemaBuilder](#) for details.

The following code fragments demonstrate the basic steps from schema creation to data serialization. The code uses the three main Persister SDK classes to create a schema, synchronize it to the server, and persist the serialized data (see [Hello Persister](#) in the [Java Persister Examples](#) section for a complete listing of the source application).

## Persister Workflow

The test data for this example consists of three `String[]` objects that are used to create a stream for the Persister to ingest.

```
String[][] data = new String[][]{{"Hello"}, {"Bonjour"}, {"Guten Tag"}};
Stream<Object[]> stream = Arrays.stream(data);
```

`SchemaBuilder` examines the first *data* record to infer the data structure, and returns the resulting schema as JSON string *schemaJson*. The schema name is `Demo.Hello`.

```
String[] fieldnames = new String[]{"greeting"}
String schemaJson = SchemaBuilder.infer(data[0], "Demo.Hello", fieldnames);
```

Next, a `SchemaManager` is created and connected to the InterSystems IRIS server (this example assumes that [JDBC connection object](#) *irisConn* already exists). The schema manager makes the JSON schema definition available to the application by synchronizing it to the Schema Registry on the server.

```
SchemaManager manager = new SchemaManager(irisConn);
RecordSchema schemaRec = manager.synchronizeSchema(schemaJson);
```

The synchronized schema is returned as *schemaRec*, a canonical `RecordSchema` object that identifies the extent of the associated `ObjectScript` class on the server. Synchronizing a schema creates a new extent if one does not already exist. The `ObjectScript` class has the same name as the schema, `Demo.Hello`.

Finally, a `Persister` object is created. It is bound to the `Demo.Hello` extent identified by *schemaRec*, and accesses the server through the connection provided by *manager*.

```
Persister persister = Persister.createPersister(manager, schemaRec,
    Persister.INDEX_MODE_DEFERRED);
```

Each item in the data stream is passed to *persister*, which serializes the data and inserts each serialized record into the database extent.

```
persister.deleteExtent(); // delete old test data
stream.map(d -> new ArrayRecord(d, schemaRec)).forEach(persister::insert);
```

Once the data has been persisted, it can be retrieved by standard database access methods such as an SQL query.

```
Statement statement = irisConn.createStatement();
ResultSet rs = statement.executeQuery( "SELECT %ID, * FROM Demo.Hello");
while (rs.next()) {
    System.out.printf( "\n Greeting: %s", rs.getString("greeting"));
}
```

The following sections discuss how the main Persister SDK classes are typically used:

- [Serializing Data with Persister](#) — describes how a `Persister` reads, serializes, buffers, and writes data.
- [Implementing Schemas with SchemaManager](#) — describes how a `SchemaManager` implements a schema on the server and makes it available to `Persisters`.
- [Designing Schemas with SchemaBuilder](#) — describes the structure of a schema and demonstrates how to create one with the `SchemaBuilder` utility.

See [Java Persister Examples](#) for complete program listings that are the source for many of the examples shown in other parts of this document.



# 3

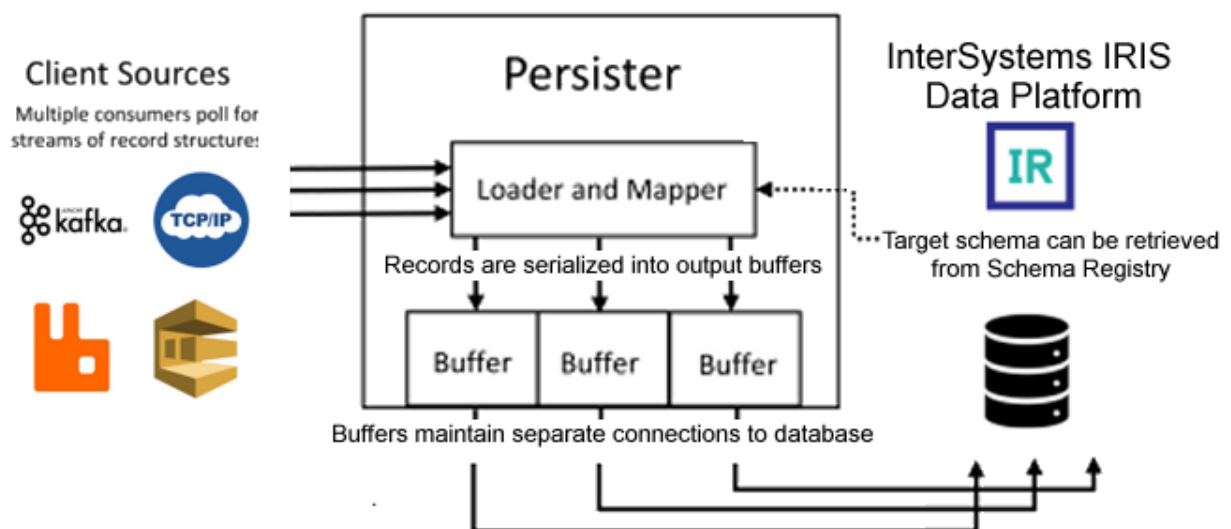
## Serializing Data with Persister

**Persister** handles all aspects of reading, serializing, buffering, and writing data. Each instance of Persister is bound to one specific schema and its associated database extent. Persisters are thread-safe, and allow precise control and monitoring of buffers and buffer queues.

- A multi-threaded loader can be used to ingest large data sets. The loader consumes a data stream, serializing each record and writing each serialized record to a pool of output buffers.
- Each buffer maintains a separate connection to an InterSystems IRIS® server.
- If the targeted extent is a sharded extent then at least one buffer per shard factor is allocated, resulting in parallel writes to the factors.
- The size of the serialized object queues, the number of buffers allocated and the size of the buffers can be configured.
- The loader maintains its own connection pool which, in the case of sharded extents, may include connections to multiple servers.

Schemas define the content and structure of data messages and are used by serializers (data message producers) and deserializers (data message consumers).

**Figure 3–1: Persister Architecture**



The Persister can be used by Java applications to rapidly ingest large data sets into an InterSystems IRIS® server.

Data is organized as a set of records and is described by a Schema. Schemas are managed by the [SchemaManager](#). The SchemaManager interacts with the InterSystems IRIS Schema Manager to synchronize schemas between the local schema cache and the Schema Repository on the server. A Persister instance is created by passing a SchemaManager instance and a RecordSchema (an implementation of the Record interface).

## 3.1 Persister Threading and Buffering

A Persister instance accepts data from the caller, serializes that data and writes it to a buffer. Buffers are automatically written to the InterSystems IRIS Server when full, on demand, and on close. Buffers to be written to the server can be either written immediately or placed in a queue that is monitored by a separate thread. The Persister constructor automatically creates an output buffer, but it is also possible to create local buffers for use in threads.

Thread-local instances of PersisterBuffer are created by calling Persister.createLocalBuffer().

- **createLocalBuffer()** creates an instance of PersisterBuffer for use exclusively within a thread. If a *bufferQueue* is specified, filled buffers will be placed in the buffer queue instead of being written directly to the server

```
PersisterBuffer createLocalBuffer ()
PersisterBuffer createLocalBuffer (LinkedBlockingQueue< BufferWrite > bufferQueue)
```

- **flush()** flushes a buffer to the server or *bufferQueue* if there are objects in the buffer. Optional *localBuffer* specifies a local buffer to be flushed. This method can be used to finish a sequence of calls to add().

```
synchronized void flush ()
void flush (PersisterBuffer localBuffer)
```

### 3.1.1 Using Local Buffers with PersisterBuffer

PersisterBuffer holds data that is intended to be written to a server using the provided connection. The buffer data is flushed whenever it is full, by direct call and when it is closed.

This class does not use any concurrency protections as it is primarily used by Persister, where protections are in place. The exception is for direct use by applications wishing to use a local buffer in a single thread. The local buffer can be flushed a buffer queue to be written to the database. The buffer queue is expected to be consumed by a separate thread running the BufferWriter runnable (see [Threading with BufferWriter](#) for details).

PersisterBuffer accumulates statistics, including number of objects written, total number of bytes written, and the number of buffer flushes (see [Persister Buffer Statistics](#) for details).

#### class PersisterBuffer

The following PersisterBuffer methods are available:

```
PersisterBuffer (long persisterFunction, int bufferSize, IRISConnection connection,
    ListWriter headerSuffix)
PersisterBuffer (long persisterFunction, int bufferSize, IRISConnection connection,
    ListWriter headerSuffix, LinkedBlockingQueue bufferQueue)

    final void add (ListWriter vList)
    final void addAndWrite (ListWriter vList)
    final void close () throws IOException
final PersisterBuffer combine (PersisterBuffer otherBuffer)
    final long [] finish ()
    void flush ()
static ListWriter getListWriter ()
static ListWriter getListWriter (byte[] byteArray)
    final long [] getStatistics ()
static void recycleListWriter (ListWriter idList)
    final void write ()
```

### 3.1.2 Threading with BufferWriter

PersisterBuffer does not use any concurrency protections as it is primarily used by Persister where protections are in place. The exception is for direct use by applications wishing to use a local buffer in a single thread.

The buffer can be placed in a queue of buffers to be written. That queue is expected to be consumed by a separate thread running the BufferWriter runnable (BufferWriter.BufferWriterRun.run()).

#### BufferWriter

BufferWriter consumes a BlockingQueue of BufferWrite objects and writes each one to the server. The following methods are available:

```
BufferWriter (BlockingQueue< BufferWrite > bufferQueue, IRISConnection connection)
void close () throws IOException
void stopBufferWriter ()
static BufferWriter startBufferWriter (BlockingQueue< BufferWrite > bufferQueue, IRISDataSource
dataSource)
static BufferWriter startBufferWriter (BlockingQueue< BufferWrite > bufferQueue, IRISConnection
connection)
```

#### Using a buffer queue with BufferWriter

Persister instances are thread safe. To improve performance, it is possible to create buffers that are used in a single thread. The local buffers are not thread safe, so they are passed to a buffer queue rather than being written directly to the database. When the queue is full, BufferWriter writes the buffered data to the database.

Create bufferQueue to hold local buffers, then start bufferWriter.

```
LinkedBlockingQueue<BufferWrite> bufferQueue =
    new LinkedBlockingQueue<BufferWrite>(200_000);
BufferWriter bufferWriter = BufferWriter.startBufferWriter(bufferQueue, dataSource);
```

Generate Runnable *producer* which provides a custom **run()** function to be passed to the threads. The runnable overrides function BufferWriter.BufferWriterRun.run().

```
Runnable producer = new Runnable() {
    @Override
    public void run() { // create separate local buffer for each thread
        PersisterBuffer localBuffer = persister.createLocalBuffer(bufferQueue);
        for (int i = 0; i < chunks; i++) { // get data from somewhere and insert it
            Object[] data = new Object[3]{"Smith-Jones", true, 922285477L};
            persister.insert(data, localBuffer);
        }
        persister.flush(localBuffer);
        statistics.addThreadStat(persister.getStatistics(localBuffer));
    }
};
```

Now call *producer.run()* from each thread and buffer the results:

```
Thread[] producerThreads = new Thread[producerThreadCount];
for (int tp = 0; tp < producerThreadCount; tp++) {
    producerThreads[tp] = new Thread(producer); // pass producer.run() to thread
    producerThreads[tp].start();
}
for (int tp = 0; tp < producerThreadCount; tp++) {
    producerThreads[tp].join();
}
bufferWriter.stopBufferWriter();
```

See [ThreadLoader in Java Persister Examples](#) for a complete listing of the source program that provides this example.

## 3.2 Persister Buffer Statistics

Persister statistics are collected by the Persister and can be reported. Statistics are stored in a [PersisterStatistics](#) object (described below). The following Persister statistics methods are available:

- **getStatistics()** — returns a [PersisterStatistics](#) object containing the current statistics from the buffer or a specified local buffer.
- **reportStatistics()** — displays a console message listing server write statistics since this Persister instance was created or reset (see details at end of this section).
- **resetStatistics()** — resets buffer *startTime* and refreshes *baseStatistics*. This will cause a future call to **reportStatistics()** to report on the activity starting from the time when this method is called. If *localBuffer* is specified, resets statistics for the local buffer.

These methods are typically called just before and after a set of Persister inserts. For example::

```
persister.resetStatistics();
PersisterStatistics statistics = new PersisterStatistics();
statistics.setStartTime((new Date(System.currentTimeMillis())).getTime());
;
//start BufferWriter / run Persister / stop BufferWriter

statistics.setStopTime((new Date(System.currentTimeMillis())).getTime());
statistics.reportStatistics();
```

### class PersisterStatistics

This is a small class that tracks statistics for a specified buffer: start time, stop time, number of objects, total bytes written, and number of buffers written to the server.

```
PersisterStatistics ()
PersisterStatistics (long startTime, long stopTime, long[] rawStat)
PersisterStatistics (long startTime, long stopTime, ConcurrentLinkedQueue< long[]> rawStats)

synchronized void addThreadStat (PersisterStatistics persisterStatistics)
    long [] getCumulativeRaw ()
    long getDuration ()
    long getStartTime ()
    long getStopTime ()
    void reportStatistics ()
    void setStartTime (long startTime)
    void setStopTime (long stopTime)
```

- **addThreadStat()** — add a [PersisterStatistics](#) instance to this Persister instance, accumulating the statistics into the *cumulativeStats* and adding it to the *threadStats* List.
- **getCumulativeRaw()** — return a `long[]` containing the accumulated statistics. The first element is the total object count, the second is the total byte count and the third is the total number of buffers used.
- **getDuration()** — return the duration (stopTime - startTime) in milliseconds.
- **getStartTime()** — start time in milliseconds.
- **getStopTime()** — stop time in milliseconds.
- **reportStatistics()** — display console message listing server write statistics since this Persister instance was created or reset.
- **setStartTime()** — start time in milliseconds.
- **setStopTime()** — stop time in milliseconds.

**reportStatistics()** writes the following statistics to the console:

```
load() executed on [threadStats.size()] threads...
Elapsed time (seconds) = [getDuration()/1000.0f]
Number of objects stored = [objectCount]
Store rate (obj/sec)    = [objectCount * 1000.0/getDuration()]
Total bytes written     = [byteCount]
Total buffers written   = [bufferWrites]
MB per second           = [(byteCount / getDuration() * 1000f) / 1048576f]

Avg object size         = [byteCount / objectCount]
```

Where:

```
objectCount = getCumulativeRaw()[0]
byteCount   = getCumulativeRaw()[1]
bufferWrites = getCumulativeRaw()[2]
```



# 4

## Implementing Schemas with SchemaManager

In order to persist data, the Persister requires a schema structure that is understood by both the Persister application and the server. The application must have a way to retrieve valid schemas from the server's persistent Schema Registry, and to update existing schemas or add new ones. On the server side, the Schema Registry synchronization process must ensure that each schema has a corresponding ObjectScript class extent, creating a new extent if one does not already exist.

[SchemaManager](#) is the primary interface between a Persister application and the server. The SchemaManager provides each Persister instance with a validated schema and a connection to the server. The SchemaManager also communicates with the Schema Registry process, ensuring that application and server schema definitions are synchronized. To accomplish this, the SchemaManager provides the following services to the Persister application:

- [Synchronizes schema definitions](#) with the Schema Registry, making a local cache of validated schemas available to the application.
- Provides the [IRISConnection](#) object that connects the Persister to the server.
- Adds new schema definitions to the Schema Registry (see [Designing Schemas with SchemaBuilder](#))
- Retrieves existing schemas by name, and creates new ones from existing ObjectScript classes (see [Acquiring Schemas from the Server](#))
- Includes [schema and extent utilities](#) that provide current information and allow schemas or extents to be deleted.

The following sections provide detailed information on [synchronizing](#) schemas, [acquiring](#) them from the server, and managing them with [various utilities](#).

### 4.1 Synchronizing Schema Definitions

Each SchemaManager object maintains a local cache of schemas for the current application. In each InterSystems IRIS namespace, schema records are persisted in a Schema Registry, and are implemented in associated ObjectScript class extents. A schema is said to be *synchronized* between the application and the server when:

- The named schema is present in the Schema Registry on the server.
- The Schema Registry record is implemented by an existing ObjectScript class extent.
- The SchemaManager object in the application has a schema in its local cache that matches the one in the Schema Registry.

A schema is implemented by calling the [synchronizeSchema\(\)](#) method, which passes it to the Schema Registry process on the server. If the server process is able to synchronize the schema, it is returned as canonical RecordSchema object, which is also stored in the SchemaManager's local cache.

The synchronization process on the server can act in several different ways, depending on what information is already on the server:

- If the requested schema does not exist in the Schema Registry, the process adds a new record to the Registry and creates a corresponding ObjectScript class extent.
- If a matching schema is already defined in the Schema Registry and is implemented by a matching ObjectScript extent, the process simply returns the currently defined RecordSchema.

If a schema of the same name is in the Registry, but does not match the parameter passed by [synchronizeSchema\(\)](#), the differences are resolved. The resolution may require generating a new version of the local implementation class. In this case, the new class will be kept compatible with existing data.

- If the schema does not exist in the Registry but an ObjectScript class with the same name exists, then a schema is generated from that ObjectScript class and synchronized.

Once the schema is synchronized, a Persister can be used to store data in the extent of the implementing ObjectScript class (see [Serializing Data with Persister](#)).

## 4.2 Acquiring Schemas from the Server

SchemaManager has two methods to acquire schemas from existing server information:

- [getSchema\(\)](#) gets and synchronizes an existing schema from the Schema Registry.
- [getSchemaForClass\(\)](#) creates and synchronizes a schema from an existing ObjectScript class.

### getSchema()

Given a schema name, [getSchema\(\)](#) will first check to see if the requested schema is already in the local cache. If not, it checks the Schema Registry on the server. If the schema is present there then it is retrieved, placed in the local cache and returned to the caller.

```
Schema demoSchema = mgr.getSchema("Demo.Hello");
System.out.println(demoSchema.toJson());
```

The resulting schema may contain some metadata added by the Schema Registry.

```
{ "name": "Hello", "namespace": "Demo", "final": false, "importFlags": 0, "category": "persistent",
  "type": "record", "fields": [ { "name": "greeting", "type": "string" } ] }
```

### getSchemaForClass()

[getSchemaForClass\(\)](#) — A schema can also be generated from an existing ObjectScript class. If the schema for that class already exists and is up to date then the previously defined schema is retrieved. Otherwise, a new schema is generated from the class and returned to the caller.

```
RecordSchema someSchema = schemaManager.getSchemaForClass("Demo.someClass");
```



## 4.3 Schema and Extent Utilities

The following examples assume that there is a schema with namespace `Test.Demo` and name `Hello`. By default the corresponding ObjectScript class will have a package name and short name identical to the schema namespace and name. The corresponding SQL table name would be `Test_Demo.Hello` (see [Designing Schemas with SchemaBuilder](#) for more information on schema names).

### Schema Utilities

All of these `SchemaManager` methods act only on schemas stored in the Schema Registry on the server. They do not affect or require a schema with the same name on the application side.

- **`deleteIrisSchema()`** — deletes the schema definition from the Schema Registry on the server. Also deletes the corresponding ObjectScript class (if it exists), and all data in the class extent.

```
manager.deleteIrisSchema("Test.Demo.Hello");
```

- **`isIrisSchemaDefined()`** — returns true if a schema with the specified name is defined in the Schema Registry.

```
boolean isDefined = manager.isIrisSchemaDefined("Test.Demo.Hello");
```

- **`isSchemaUpToDate()`** — returns true if the structure of the specified ObjectScript class matches the corresponding schema in the Schema Registry.

```
boolean isCurrent = manager.isSchemaUpToDate("Test.Demo.Hello");
```

### Extent Utilities

- **`deleteIrisExtent()`** — given a schema name, deletes the extent of the associated ObjectScript class.

```
manager.deleteIrisExtent("Test.Demo.Hello");
```

- **`isIrisClassDefined()`** — returns true if the ObjectScript class is defined on the server. The class does not have to match an entry in the Schema Registry.

```
boolean isDefined = manager.isIrisClassDefined("Test.Demo.Hello");
```

- **`getIrisTableClass()`** — gets the name of the ObjectScript class that projects the specified SQL table.

```
String tableClass = manager.getIrisTableClass("Test_Demo.Hello");
```

In this example, a query on table name `Test_Demo.Hello` would return class name `Test.Demo.Hello`, where `Test.Demo` is the class package name and `Hello` is the unqualified class name.



# 5

## Designing Schemas with SchemaBuilder

**SchemaBuilder** is a utility that provides simple calls to construct schema definitions, which are returned as JSON strings. All construction methods are static and calls can be nested. Field types can be specified directly or inferred from Java types, classes, or objects.

There are several ways to create schemas. One of the easiest is to use **SchemaBuilder.infer()** to construct a schema from sample data and corresponding field names. For example:

```
Object[] values = new Object[]{"Apple", 2};
String[] labels = new String[]{"item", "count"};
String schemaFruit = SchemaBuilder.infer(values, "Demo.Fruit", labels);
```

Schemas can also be designed using **SchemaBuilder.record()**, which provides builder methods such as **addField()** to create schema components. Builder methods can be chained until **complete()** is called to return the JSON schema string.

The following call to **record()** produces a JSON schema string identical to the one produced by **infer()**:

```
String schemaFruit = SchemaBuilder.record()
    .withName("Test.Demo.Fruit")
    .addField("item", "string")
    .addField("count", "int")
    .complete();
```

Both of the examples above will return the same JSON schema string (line breaks added for clarity):

```
{ "type": "record",
  "namespace": "Test.Demo",
  "name": "Fruit",
  "category": "persistent",
  "fields": [
    { "name": "item", "type": "string" },
    { "name": "count", "type": "int" }
  ]
}
```

This schema contains the following components:

- **type** — Schemas can have various types (in the example above, notice that each field is a schema with its own type), but Persister schemas will always be stored in the server Schema Registry as record types (class `RecordSchema`).
- **namespace** — A schema name qualifier. It is important to note that schema namespaces have nothing to do with InterSystems IRIS database namespaces. Schema namespaces are part of the qualified schema name, which also determines the fully qualified name of the corresponding ObjectScript class. For example, the `Test.Demo.Fruit` schema has namespace `Test.Demo` and name `Fruit`. The corresponding class would be `Test.Demo.Fruit` (package name `Test.Demo` and short class name `Fruit`). This class could be stored in any InterSystems IRIS namespace (for example, the `USER` namespace).
- **name** — The unqualified schema name. This is the final part of the identifier you specify for the schema. For example, if you specify `.withname(Test.Demo.Fruit)`, the namespace will be `Test.Demo`, and the name will be `Fruit`.

The corresponding SQL table will be named `Test_Demo.Fruit` (see Table Names and Schema Names in *Using InterSystems SQL* for related information on naming conventions).

- **category** — Since InterSystems IRIS stores records as serialized instances of ObjectScript classes, Persister schemas need to differentiate between classes that extend `%Library.Persistent` and those that extend `%Library.SerialObject` (embedded objects). The default is `Persistent` if no value is specified.
- **fields** — Each field entry is specified as a schema with its own name and type. The Persister supports primitive types `string`, `bytes`, `short`, `int`, `long`, `float`, `double`, `boolean`, and `null`. Complex type declarations can include their own fields, nested as deep as necessary.

`SchemaBuilder.record()` provides an extra level of control when specifying field types. For example, a schema for the following object could be created by either **`infer()`** or **`record()`**:

```
Object[] data = new Object[];
data[0] = "Wilber";
data[1] = true;
data[2] = new Object[][] {{0,1},{2,3}};
data[3] = java.util.UUID.randomUUID();
data[4] = new java.util.Date();
```

It is very simple to create the schema by inference from data:

```
String[] names = {"Name", "isActive", "Scores", "MemberID", "DateJoined"};
String schemaJson = SchemaBuilder.infer(data, "Demo.ClubMember", names);
```

But the **`record()`** builder methods allow individual field types to be specified in several different ways. Types can still be inferred from data, but they can also be inferred from Java class names or specified directly as Java types. Some complex types such as dates and times can be also be specified using [logical type methods](#). The following example demonstrates all of these options:

```
String schemaJson = SchemaBuilder.record()
    .withName("Demo.ClubMember")
    .addField("Name", SchemaBuilder.infer("java.lang.String"))
    .addField("isActive", SchemaBuilder.infer(true))
    .addField("Scores", java.lang.Integer[].class)
    .addField("MemberID", SchemaBuilder.uuid())
    .addField("DateJoined", SchemaBuilder.date())
    .complete();
```

In the first two fields, *Name* and *isActive*, types are determined by calling **`infer()`** on class name `"java.lang.String"` and value `true`. The *Scores* field directly specifies class type `java.lang.Integer[].class`. The last two fields use logical type methods **`uuid()`** and **`date()`**. This produces the following JSON schema string:

```
{ "type": "record",
  "name": "ClubMember",
  "namespace": "Demo",
  "category": "persistent",
  "fields": [
    { "name": "Name", "type": "string" },
    { "name": "isActive", "type": "boolean" },
    { "name": "Scores", "type": { "type": "array", "items": "int" } },
    { "name": "MemberID", "type": { "logicalType": "uuid", "type": "string" } },
    { "name": "DateJoined", "type": { "logicalType": "date", "type": "int" } },
  ] }
```

This schema is almost identical to one produced by simple inference, except for the last field. The **`date()`** method produces a logical type for `java.util.Date`, while inference would produce a logical type for `java.sql.Timestamp`.

# 6

## Java Persister Examples

This section provides full listing for several working Persister programs:

- [Hello Persister](#) — a quick demonstration of the basic Persister workflow.
- [Continents](#) - load data from a local array
- [DivvyTrip](#) - Chicago Bicycle Sharing CSV file (small - 1.5 million records)
- [ThreadLoader](#) — load a large data set- 20 million records

**Note:** The Persister, like all InterSystems Java drivers, connects to the database with a standard InterSystems JDBC IRISConnection object (see [Using IRISDataSource to Connect](#) in *Using Java with InterSystems Software*). IRISDataSource is the recommended way to create a the connection because a DataSource is required to use the Persister's multi-threaded loader.

### 6.1 Hello Persister

This very short Persister application demonstrates all of the basic Persister mechanisms from schema creation to data serialization, using the three main Persister classes: [SchemaBuilder](#) creates a schema, [SchemaManager](#) synchronizes the schema to the Schema Registry on the server, and [Persister](#) serializes and stores records in the extent specified by the Registry.

#### Hello.java

```
package com.intersystems.demo;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.intersystems.jdbc.*;
import com.intersystems.persister.Persister;
import com.intersystems.persister.SchemaBuilder;
import com.intersystems.persister.SchemaManager;
import com.intersystems.persister.ArrayRecord;
import com.intersystems.persister.schemas.RecordSchema;
import java.util.Arrays;
import java.util.stream.*;
import java.sql.*;

public class Hello {
    public static void main(String[] args) throws SQLException, JsonProcessingException {
        IRISDataSource dataSource = new IRISDataSource();
        dataSource.setURL("jdbc:IRIS://127.0.0.1:1972/USER");
        IRISConnection irisConn = (IRISConnection) dataSource.getConnection("_SYSTEM", "SYS");

        // Create a data stream and use SchemaBuilder to infer a schema
        String[][] data = new String[][]{{"Hello"}, {"Bonjour"}, {"Guten Tag"}};
        Stream<Object[]> stream = Arrays.stream(data);
        String schemaJson = SchemaBuilder.infer(data[0], "Demo.Hello", new String[]{"greeting"});
```

```
// Create a SchemaManager and synchronize the schema to the database
SchemaManager mgr = new SchemaManager(irisConn);
RecordSchema schemaRec = mgr.synchronizeSchema(schemaJson);

// Create a persister, passing it the manager and the schema record
Persister persister = Persister.createPersister(mgr, schemaRec,
Persister.INDEX_MODE_DEFERRED);
persister.deleteExtent(); // delete old test data

// Pass the stream to the persister and insert each item into the database
stream.map(d -> new ArrayRecord(d, schemaRec)).forEach(persister::insert);

// Use standard SQL calls to display the persisted data
ResultSet rs = irisConn.createStatement().executeQuery("SELECT %ID, * FROM Demo.Hello");
while (rs.next()) { System.out.printf("\n Greeting: %s", rs.getString("greeting")); }
}
```

The call to `SchemaBuilder.infer()` produces the following JSON schema string:

```
{ "type": "record", "name": "Hello", "namespace": "Demo", "category": "persistent",
  "fields": [ { "name": "greeting", "type": "string" } ] }
```

The final SQL printf statement produces the following output:

```
Greeting: Hello
Greeting: Bonjour
Greeting: Guten Tag
```

## 6.2 Continents - Load Data from a Local Array

Example of loading a local array of strings. Each string is delimited. This example uses `CsvRecord` to model the source data.

### Continents.java

```
package com.intersystems.demo;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.intersystems.jdbc.*;
import com.intersystems.persister.*;
import com.intersystems.persister.schemas.*;
import java.util.Arrays;
import java.util.stream.*;
import java.sql.*;

public class Continent {
    public static void main(String[] args) throws SQLException, JsonProcessingException {
        IRISDataSource dataSource = new IRISDataSource();
        dataSource.setURL("jdbc:IRIS://127.0.0.1:1972/USER");
        IRISConnection irisConn = (IRISConnection) dataSource.getConnection("_SYSTEM", "SYS");

        // create a set of delimited strings (instead of reading a CVS file)
        String[] continents = new String[]{
            "NA,North America", "SA,South America", "AF,Africa", "AS,Asia",
            "EU,Europe", "OC,Oceania", "AT,Atlantis", "AN,Antarctica"
        };
        Stream<String> dataStream = Arrays.stream(continents);

        // Create a schema
        String schemaSource = SchemaBuilder.record()
            .WithName("Demo.Continent")
            .addField("code", "string")
            .addField("name", "string")
            .complete();
        System.out.println("\nschemaSource:\n" + schemaSource + "\n");

        // Parse and synchronise the schema
        SchemaManager schemaManager = new SchemaManager(irisConn);
        RecordSchema schemaRecord = (RecordSchema) schemaManager.parseSchema(schemaSource);
        RecordSchema continentsSchema = schemaManager.synchronizeSchema(schemaRecord);
        System.out.println("\ncontinentsSchema:\n" + continentsSchema.toJson() + "\n");
    }
}
```

```

        // Prepare the User.Continent extent and parse the data stream to a buffer
        Persister persister = Persister.createPersister(schemaManager, continentsSchema,
        Persister.INDEX_MODE_DEFERRED);
        persister.deleteExtent();
        dataStream.map(CsvRecord.getParser(",", schemaRecord)).forEach(record ->
        persister.add(record));

        // Flush buffer to the database and print buffer statistics
        persister.flush();
        System.out.println("REPORT STATISTICS");
        persister.reportStatistics();

        // Query the new data
        Statement query = irisConn.createStatement();
        java.sql.ResultSet rs = query.executeQuery("select code, name from Demo.Continent order by
        name");
        int colnum = rs.getMetaData().getColumnCount();
        while (rs.next()) {
            for (int i=1; i<=colnum; i++) {
                System.out.print(rs.getString(i) + " ");
            }
            System.out.println();
        }
    }
}

```

## 6.3 DivvyTrip - CSV bicycle sharing dataset (small - 1.5 million records)

The City of Chicago's Divvy bicycle sharing data sets are publicly available. This example shows how a local file containing data in CSV format can be loaded using streams and Persister. The schema used here is generated from an existing InterSystems IRIS class (RowDB.DivvyTrip) created by a series of DDL statements (see listing in [TABLE DivvyTrip](#)). The data set used in this example concatenates several months of Divvy data. To simplify the example, the data was preprocessed to eliminate rows that contain null entries.

### DivvyTrip.java

```

package com.intersystems.demo;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.intersystems.jdbc.*;
import com.intersystems.persister.*;
import com.intersystems.persister.schemas.*;
import java.util.Arrays;
import java.util.stream.*;
import java.util.function.Function;
import java.sql.*;
import java.io.*;

public class DivvyTrip {
    public static void main(String[] args) throws SQLException, JsonProcessingException {
        IRISDataSource dataSource = new IRISDataSource();
        dataSource.setURL("jdbc:IRIS://127.0.0.1:1972/USER");
        IRISConnection irisConn = (IRISConnection) dataSource.getConnection("_SYSTEM", "SYS");

        SchemaManager divvyManager = new SchemaManager(irisConn);

        // Generate a schema from an existing ObjectScript class
        RecordSchema divvySchema = divvyManager.getSchemaForClass("RowDB.DivvyTrip");
        System.out.println("\ndivvySchema:\n" + divvySchema + "\n");
        System.out.println();

        // Create a Persister and delete any old test data
        Persister persister = Persister.createPersister(divvyManager, divvySchema,
        Persister.INDEX_MODE_DEFERRED);
        persister.deleteExtent();

        // Get a data stream from a CSV file
        Stream<String> dataStream = Stream.empty();
        try {

```

```

        dataStream = new BufferedReader(new FileReader("../divvy-tripdata.csv")).lines();
    } catch (Exception ignore) {}
    System.out.println("\n STARTING TO PERSIST STREAM\n");

    // Create a CSV parser, then parse and add each record to the database.
    Function<String, CsvRecord> parser = CsvRecord.getParser(",", divvySchema);
    persister.resetStatistics();
    try {
        dataStream.skip(1)
            .map(parser)
            .forEach(record -> persister.add(record));
    } catch (Exception e) {
        e.printStackTrace();
    }

    // Flush the buffer to the database, report buffer statistics after indexing has finished
    persister.flush();
    persister.waitForIndexing();
    persister.reportStatistics();

    // Query the new data and print first 10 records
    Statement query = irisConn.createStatement();
    java.sql.ResultSet rs = query.executeQuery(
        "select top 10 * from RowDB.DivvyTrip order by started_at");
    int colnum = rs.getMetaData().getColumnCount();
    while (rs.next()) {
        for (int i=1; i<=colnum; i++) {
            System.out.print(rs.getString(i) + " ");
        }
        System.out.println();
    }
}

```

## TABLE DivvyTrip

The RowDB.DivvyTrip class is created by a series of DDL statements. To create a fresh table, open the Terminal and paste the following lines exactly as shown. (The first command may be changed if you want to use a namespace other than USER. The single empty line after DROP TABLE is required to enter SQL.Shell() multi-line mode):

```

zn "USER"
DO $SYSTEM.SQL.Shell()
DROP TABLE RowDB.DivvyTrip

CREATE TABLE RowDB.DivvyTrip (
    ride_id VARCHAR(16),
    rideable_type VARCHAR(11),
    started_at TIMESTAMP,
    ended_at TIMESTAMP,
    start_station_name VARCHAR(50),
    start_station_id VARCHAR(4),
    end_station_name VARCHAR(50),
    end_station_id VARCHAR(4),
    start_lat DOUBLE,
    start_lng DOUBLE,
    end_lat DOUBLE,
    end_lng DOUBLE,
    member_casual VARCHAR(10) )
GO
CREATE BITMAP INDEX StartTimeIndex ON RowDB.DivvyTrip (started_at)
CREATE BITMAP INDEX EndTimeIndex ON RowDB.DivvyTrip (ended_at)
CREATE BITMAP INDEX StartStationIndex ON RowDB.DivvyTrip (start_station_id)
CREATE BITMAP INDEX EndStationIndex ON RowDB.DivvyTrip (end_station_id)
CREATE BITMAP INDEX RideIDIndex ON RowDB.DivvyTrip (ride_id)
TUNE TABLE RowDB.DivvyTrip
q

```

## 6.4 ThreadLoader (large - 20 million records)

This example uses generated data stored in an Object array, multiple threads, local buffers and a BufferWriter. See [Serializing Data with Persister](#) for more information on threading, buffering, and buffer statistics.



**ThreadLoader.java**

```

package com.intersystems.demo;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.intersystems.jdbc.*;
import com.intersystems.persister.*;
import com.intersystems.persister.schemas.*;
import java.sql.*;
import java.util.concurrent.LinkedBlockingQueue;

public class ThreadLoader {
public static void main(String[] args) throws SQLException, JsonProcessingException {
    try {
        //=====
        // Initialize Persister, SchemaManager, statistics, and buffer queue
        //=====
        System.out.println("ThreadLoader - write arrays of generated data " +
            "to localBuffer, BufferWriter writes to server");
        IRISDataSource dataSource = new IRISDataSource();
        dataSource.setURL("jdbc:IRIS://127.0.0.1:1972/USER");
        IRISConnection irisConn = (IRISConnection) dataSource.getConnection("_SYSTEM","SYS");

        SchemaManager schemaManager = new SchemaManager(irisConn);
        try {
            schemaManager.deleteIrisSchema("Demo.ThreadLoader");
        } catch (Exception ignore) {}

        RecordSchema sourceType = schemaManager.synchronizeSchema(SchemaBuilder.record()
            .WithName("Demo.ThreadLoader")
            .addField("ID", "int")
            .addField("FirstName", "string")
            .addField("LastName", "string")
            .addField("aBool", "boolean")
            .addField("along", "long")
            .addField("afloat", "float")
            .addField("adouble", "double")
            .addField("abytes", "bytes")
            .complete());
        Persister persister = Persister.createPersister(schemaManager, sourceType,
            Persister.INDEX_MODE_DEFERRED, 32_000);
        persister.deleteExtent();
        persister.flush();

        System.out.println("\nStarting load, resetting statistics");
        persister.resetStatistics();
        PersisterStatistics statistics = new PersisterStatistics();
        statistics.setStartTime(new Date(System.currentTimeMillis()).getTime());

        // Create queue to hold local buffers
        LinkedBlockingQueue<BufferWrite> bufferQueue =
            new LinkedBlockingQueue<BufferWrite>(200_000);
        BufferWriter bufferWriter = BufferWriter.startBufferWriter(bufferQueue, dataSource);

        // Set loader constants
        int producerThreadCount = 2;
        int objectCount = 24_000_000;
        int chunkSize = 10_000;
        int chunks = objectCount / chunkSize / producerThreadCount;
        System.out.format("Loading %d objects using %d producer threads, %d chunks of " +
            "%d objects.%n", objectCount, producerThreadCount, chunks, chunkSize);

        //=====
        // Generate Runnable producer (provides function to be passed to threads)
        //=====
        Runnable producer = new Runnable() {
            @Override
            public void run() { // create separate local buffer for each thread
                PersisterBuffer localBuffer = persister.createLocalBuffer(bufferQueue);
                for (int i = 0; i < chunks; i++) { // generate some test data
                    Object[][] data = new Object[chunkSize][8];
                    for (int j = 0; j < chunkSize; j++) {
                        data[j][0] = i * chunkSize + j;
                        data[j][1] =
                            "123456789012345678901234567890123456789012345678901234567890";
                        data[j][2] = "Smith-Jones";
                        data[j][3] = true;
                        data[j][4] = 922285477L;
                        data[j][5] = 767876231.123F;
                        data[j][6] = 230.134;
                        data[j][7] = 233;
                    }
                    persister.insert(data, localBuffer);
                }
            }
        }
    }
}

```

```
        persister.flush(localBuffer);
        statistics.addThreadStat(persister.getStatistics(localBuffer));
    }
};

//=====
// Call producer.run() from each thread and buffer results
//=====
Thread[] producerThreads = new Thread[producerThreadCount];

for (int tp = 0; tp < producerThreadCount; tp++) {
    producerThreads[tp] = new Thread(producer); // pass producer.run() to thread
    producerThreads[tp].start();
}
for (int tp = 0; tp < producerThreadCount; tp++) {
    producerThreads[tp].join();
}

System.out.println("Stopping the BufferWriter");
bufferWriter.stopBufferWriter();
statistics.setStopTime((new Date(System.currentTimeMillis())).getTime());
statistics.reportStatistics();
} catch (Exception e) {
    e.printStackTrace();
}
}
```

This code produces output similar to the following:

```
Demo.ThreadLoader - write arrays of generated data to localBuffer, BufferWriter writes to server

Starting load, resetting statistics
Loading 24000000 objects using 2 producer threads, 1200 chunks of 10000 objects.
Stopping the BufferWriter
loaded 24,000,000 objects, using 2 threads in 37.51 seconds, 639,863 obj/sec, 76.956749 MB/sec
load() executed on 2 threads...

Elapsed time (seconds)      =          37.51
Number of objects stored =    24,000,000
Store rate (obj/sec)       =       639,863
Total bytes written        =   3,026,714,094
Total buffers written      =         94,856
MB per second              =         76.96

Avg object size            =           126
```

# 7

## Quick Reference for Java Persister Classes

This section is a reference for the InterSystems Persister for Java SDK (namespace `com.intersystems.persister`). This quick reference covers the following SDK classes:

- class [Persister](#) — main interface for serializing records to the database.
- class [SchemaManager](#) — main interface for coordinating schemas between Persister applications and the server.
- class [SchemaBuilder](#) — utility for creating schema definitions.

**Note:** This quick reference is intended as a convenient guide to the Java Persister classes and methods discussed in this document. It does not cover all public classes, and is not a complete or definitive reference for the Java Persister. For the most complete and up-to-date information, see the Java Persister online documentation. In many cases, documentation and examples from Avro implementations may also be helpful.

### 7.1 Class Persister

The `com.intersystems.persister.Persister` class defines and implements the main interface for the InterSystems IRIS Persister for Java SDK. Persister is used to write data to the extent of a persistent class. See [Serializing Data with Persister](#) for more information and examples.

#### 7.1.1 Persister Constructor

##### Constructor

`Persister.Persister()` is used to write data to the extent of an ObjectScript persistent class. A [SchemaManager](#) provides the connection to the target extent and interfaces with the Schema Registry on the server.

```
Persister (SchemaManager schemaManager, RecordSchema schema, int indexMode) throws  
PersisterException  
Persister (SchemaManager schemaManager, RecordSchema schema, int indexMode, int bufferSize)  
throws PersisterException
```

*parameter:*

- `schemaManager` — a local instance of `SchemaManager` connected to the Schema Registry on the server.

- `schema` — a schema (either a JSON string or a `RecordSchema`) describing the target extent on the server.
- `indexMode` — valid values are attributes `INDEX_MODE_DEFAULT` (= -1), `INDEX_MODE_DEFERRED` (= 0), or `INDEX_MODE_IMMEDIATE` (= 2).
- `bufferSize` — optional buffer size. Defaults to attribute `DEFAULT_BUFFER_SIZE` (= 32\_000)

## 7.1.2 Persister Methods

### **add()**

`Persister.add()` adds one or more records to a buffer. If adding a record causes the buffer to overflow then the buffer automatically flushes to the server. `Persister.flush()` may be used to write a partially filled buffer.

Local buffers are presumed to be thread-private as there are no concurrency guarantees with local buffers.

#### **add ( record )**

serializes and adds one or more `Record` objects to the buffer.

```
synchronized void add (Record record)
synchronized< R extends Record > void add (R[] records)
```

- `record` — the `Record` object to be added to the buffer
- `records` — array of `Record` objects to be added to the buffer

#### **add ( list )**

adds one or more serialized `ListWriter` records to the buffer.

```
synchronized void add (ListWriter vList)
synchronized void add (ListWriter[] lists)
```

- `vList` — serialized `ListWriter` object to be added to the buffer
- `lists` — array of `ListWriter` objects to be added to the buffer

#### **add ( list, localBuffer )**

adds one or more serialized `ListWriter` records to the specified local buffer.

```
void add (ListWriter serial, PersisterBuffer localBuffer)
synchronized void add (ListWriter[] lists, PersisterBuffer localBuffer)
```

- `serial` — serialized `ListWriter` object
- `lists` — array of serialized `ListWriter` objects
- `localBuffer` — local buffer where serialized value is to be added

### **close()**

`Persister.close()` close this instance of `Persister`.

```
void close ()
```

## createLocalBuffer()

`Persister.createLocalBuffer()` creates an instance of `PersisterBuffer` that is expected to be used within a thread and not shared with any other threads as there are no built in concurrency controls. If *bufferQueue* is specified, filled buffers will be placed in the buffer queue instead of being written directly to the server.

```
PersisterBuffer createLocalBuffer ()
PersisterBuffer createLocalBuffer (LinkedBlockingQueue< BufferWrite > bufferQueue)
```

*parameter:*

- `bufferQueue` — optional queue where filled buffers will be placed.

## createPersister()

`Persister.createPersister()` constructs and returns an instance of `Persister` that is bound to the specified Schema. Works exactly like the [Constructor](#).

```
static Persister createPersister (SchemaManager schemaManager, RecordSchema schema, int
indexMode)
static Persister createPersister (SchemaManager schemaManager, RecordSchema schema, int
indexMode, int bufferSize)
```

*parameter:*

- `schemaManager` — a local instance of `SchemaManager` connected to the Schema Registry on the server.
- `schema` — a schema (either a JSON string or a `RecordSchema`) describing the target extent on the server.
- `indexMode` — valid values are attributes `INDEX_MODE_DEFAULT` (= -1), `INDEX_MODE_DEFERRED` (= 0), or `INDEX_MODE_IMMEDIATE` (= 2).
- `bufferSize` — optional buffer size. Defaults to attribute `DEFAULT_BUFFER_SIZE` (= 32\_000)

## delete()

`Persister.delete()` deletes a single object from the extent of the server class bound to this persister.

```
void delete (Long id)
void delete (String id)
```

*parameter:*

- `id` — id of the object to delete. In the case where the `IDKEY` is a composite key, the id is a delimited string serialization of the `IDKEY` key properties using `|` as a delimiter.

## deleteExtent()

`Persister.deleteExtent()` deletes all of the data from the extent of the server class that is bound to this persister.

```
void deleteExtent ()
```

## flush()

`Persister.flush()` flushes a buffer to the server or `bufferQueue` (see [createLocalBuffer\(\)](#)) if there are objects in the buffer. *localBuffer* optionally specifies a local buffer to be flushed. This method can be used to finish a sequence of calls to [add\(\)](#).

```
synchronized void flush ()
void flush (PersisterBuffer localBuffer)
```

*parameter:*

- `localBuffer` — local `PersisterBuffer` to be flushed

### **getStatistics()**

`Persister.getStatistics()` returns a `PersisterStatistics` object containing the current statistics from the buffer or a specified *localBuffer* (see [Persister Buffer Statistics](#) for details).

```
PersisterStatistics getStatistics ()  
PersisterStatistics getStatistics (PersisterBuffer localBuffer)
```

*parameter:*

- `localBuffer` — local buffer where statistics are recorded.

### **insert()**

`Persister.insert()` immediately writes records into the extent of the class that implements this persister's schema (unlike [add\(\)](#), which writes to a buffer that may not be flushed immediately).

#### **insert( object )**

inserts one or more objects into the extent of the class that implements this persister's schema. Each object contains an array of schema field values (where element 0 of the array is the first field in the schema). If *localBuffer* is specified, the *data* array is written to the local buffer, which is then immediately flushed to the server.

```
synchronized void insert (Object[] object)  
synchronized void insert (Object[][] data)  
void insert (Object[][] data, PersisterBuffer localBuffer)
```

*parameter:*

- `object` — an array of field values.
- `data` — an array of field value arrays.
- `localBuffer` — optional local buffer

#### **insert( record )**

inserts one or more new `Record` objects into the extent of the class that implements this persister's schema.

```
synchronized void insert (Record record)  
synchronized void insert (Record[] data)
```

*parameter:*

- `record` — an object implementing the `Record` interface.
- `data` — an array of `Record` objects.

#### **insert( map )**

inserts one or more new `Map` objects into the extent of the class that implements this persister's schema. Each object contains a map of schema field values (for example `JsonRecord` or `MapRecord` values).

```
synchronized void insert (Map< String, Object > map)  
synchronized void insert (Map< String, Object >[] data)
```

*parameter:*

- `map` — Map where key is the field name and value is the field value.
- `data` — an array of field value maps.

### **insert( stream )**

inserts all data in a stream into the extent of the class that implements this persister's schema. The *factory* function maps the data from the stream to a Record. The buffer is automatically flushed either when it becomes full or when the stream is completely consumed.

```
synchronized< R extends Record, T extends Object > void insert (BiFunction< T, RecordSchema,
R > factory, Stream< T > data)
```

*parameter:*

- `factory` — the Record factory function used to instantiate the records to which data is mapped.
  - `<R>` — optional Record implementation class. This can often be inferred from the *factory* function.
  - `<T>` — optional data type. This can often be inferred from the Stream.
- `data` — the data Stream.

### **reportStatistics()**

`Persister.reportStatistics()` writes report to the console. Reports the server write statistics for activity since this persister was constructed. If *localBuffer* is specified, reports server write statistics for the local buffer (see [Persister Buffer Statistics](#) for more information).

```
void reportStatistics ()
void reportStatistics (PersisterBuffer localBuffer)
```

*parameter:*

- `localBuffer` — local buffer containing statistics to be reported

### **resetStatistics()**

`Persister.resetStatistics()` resets buffer *startTime* and refreshes *baseStatistics*. This will cause a future call to **reportStatistics()** to report on the activity starting from the time when this method is called. If *localBuffer* is specified, resets statistics for the local buffer.

```
synchronized void resetStatistics ()
void resetStatistics (PersisterBuffer localBuffer)
```

*parameter:*

- `localBuffer` — local buffer containing statistics to be reset.

### **serialize()**

`Persister.serialize()` serializes an `Object[]`, `Map`, or `Record` and returns the serialization.

```
ListWriter serialize (Object[] value)
ListWriter serialize (Map< String, Object > value)
ListWriter serialize ( Record value )
```

*parameter:*

- `value` — value to be serialized.

**waitForIndexing()**

Persister.**waitForIndexing()** builds deferred indexes for the specified class on the server and waits for completion. Returns true if indexing has completed, false if timed out before index build is completed.

```
final boolean  waitForIndexing ()
```

## 7.2 class SchemaManager

Class `com.intersystems.persister.SchemaManager` is connected to an InterSystems server. Each server maintains a Schema Registry in each namespace. The SchemaManager maintains a cache of schemas that are synchronized with the Schema Registry and available to the application. Once a specified schema is synchronized with the server, a Persister can be used to store data in the extent of the ObjectScript class that implements that schema. See [Implementing Schemas with SchemaManager](#) for more information and examples.

### 7.2.1 SchemaManager Constructor

**Constructor**

Persister.**SchemaManager()** accepts an `IRISConnection` object and returns a `SchemaManager`.

```
SchemaManager (IRISConnection connection) throws SQLException
```

*parameter:*

- `connection` — a connected `IRISConnection` object (standard [InterSystems JDBC connection](#) object).

### 7.2.2 SchemaManager Methods

**deleteIrisExtent()**

`SchemaManager.deleteIrisExtent()` deletes the persistent extent in the current namespace for the server's local ObjectScript implementation class. Throws an exception if the schema does not exist or if there is an error encountered during delete. Also see Persister.[delete\(\)](#), which can delete a single object from the extent.

```
void  deleteIrisExtent (String schema_name)
```

*parameter:*

- `schema_name` — string specifying a schema name.

**deleteIrisSchema()**

`SchemaManager.deleteIrisSchema()` deletes the schema definition from the server. If the associated ObjectScript implementation class exists, it will also be deleted, along with any data in its extent. Throws an exception if any errors are encountered.

```
void  deleteIrisSchema (String schemaName)
```

*parameter:*

- `schemaName` — string specifying a schema name.



**getIrisTableClass()**

SchemaManager.**getIrisTableClass()** gets the name of the class that projects the *tableName* table.

```
String getIrisTableClass (String tableName)
```

*parameter:*

- `tableName` — string specifying a table name.

**getSchema()**

SchemaManager.**getSchema()** gets a schema from the local Schema Registry. If it is a NamedSchema then get it from the local cache or from the server if not present in the cache. Returns an instance of a class that implements Schema.

```
Schema getSchema (String name) throws JsonProcessingException
```

*parameter:*

- `name` — string specifying the schema name.

**getSchemaForClass()**

SchemaManager.**getSchemaForClass()** retrieves the schema from the connected server whose local ObjectScript implementation class is *class\_name*. If the schema for the class already exists and is up to date then that schema is retrieved. Otherwise, a new schema is generated from the class and returned.

```
RecordSchema getSchemaForClass (String className) throws JsonProcessingException
```

*parameter:*

- `className` — name of the ObjectScript class that implements the current schema.

**isIrisClassDefined()**

SchemaManager.**isIrisClassDefined()** return true if the ObjectScript class is defined on the server.

```
boolean isIrisClassDefined (String name)
```

*parameter:*

- `name` — name of the ObjectScript class.

**isIrisSchemaDefined()**

SchemaManager.**isIrisSchemaDefined()** return true if the schema is defined in the Schema Registry

```
boolean isIrisSchemaDefined (String name)
```

*parameter:*

- `name` — string specifying the schema name.

## isSchemaUpToDate()

SchemaManager.**isSchemaUpToDate()** returns true if the schema matches ObjectScript implementation class *class\_name* on the server.

```
boolean isSchemaUpToDate (String class_name)
```

*parameter:*

- *class\_name* — name of the ObjectScript class.

## parseSchema()

SchemaManager.**parseSchema()** parses the schema source, returning an instance of a class that implements Schema.

```
Schema parseSchema (String source) throws JsonProcessingException  
Schema parseSchema (JsonNode source) throws JsonProcessingException
```

*parameter:*

- *source* — source of the schema; either a String or a JsonNode (an instance of com.fasterxml.jackson.databind.JsonNode)

## synchronizeSchema()

SchemaManager.**synchronizeSchema()** synchronizes a schema (either a JSON string or a RecordSchema) with the connected Schema Registry and returns the synchronized schema as a RecordSchema. If the schema is already defined in the Schema Registry then it is returned. If an ObjectScript class whose name matches the name defined by *schemaSource* exists then a schema is generated from that ObjectScript class and returned. Otherwise, *schemaSource* is added to the Schema Registry and an ObjectScript implementation class is generated. The resulting schema from the server is then returned.

When there is a matching schema already defined in the Schema Registry, it is compared with the *schemaSource*. If there is a difference then those differences are resolved, possibly generating a new version of the local implementation class. Existing data is kept compatible with the new class.

```
RecordSchema synchronizeSchema (String schemaSource) throws PersisterException,  
JsonProcessingException  
RecordSchema synchronizeSchema (RecordSchema schema) throws PersisterException  
RecordSchema synchronizeSchema (String schemaSource, String annotations)  
RecordSchema synchronizeSchema (RecordSchema schema, String annotations) throws  
PersisterException
```

*parameter:*

- *schemaSource* — JSON formatted schema definition
- *schema* — a RecordSchema formatted schema definition
- *annotations* — JSON formatted object with indices field (see SchemaBuilder.[index\(\)](#) and [indexes\(\)](#)).

Throws PersisterException if the server reports a problem. JsonProcessingException is thrown by the Jackson JSON parser.

## synchronizeSchemaWithIris()

SchemaManager.**synchronizeSchemaWithIris()** allows the manager to establish a new connection and synchronize using that connection. Once connected, it synchronizes a schema definition with the connected server and returns the corresponding RecordSchema retrieved from the Schema Registry at that location. If *annotations* is specified, the index definitions contained in the annotation are also applied. If the named schema already exists on the server

then it is simply returned. If it does not exist but a class of the same name does exist then a schema is generated on the server and returned. Otherwise, the schema is saved to the server, compiled, and the resulting schema is then returned. The new connection is closed when this method returns.

```
static RecordSchema synchronizeSchemaWithIris (RecordSchema schema, IRISConnection connection)
    throws SQLException
static RecordSchema synchronizeSchemaWithIris (RecordSchema schema, IRISConnection connection,
    String annotations) throws SQLException
```

*parameter:*

- `schema` — RecordSchema sent to the server
- `connection` — IRISConnection object connected to the server
- `annotations` — JSON formatted object containing index definitions

Throws SQLException if there is an issue obtaining a connection to the server.

## 7.3 class SchemaBuilder

Class `com.intersystems.persisters.SchemaBuilder` is a utility that provides simple calls to construct Schema sources. All methods are static and calls can be nested.

### 7.3.1 SchemaBuilder Methods

#### **array()**

`SchemaBuilder.array()` builds an array schema and returns a JSON string containing the schema.

```
static String array (String itemsType)
static String array (Schema itemsType)
```

*parameter:*

- `itemsType` — Schema object or JSON schema string defining the type of the array items

#### **date()**

`SchemaBuilder.date()` returns a LogicalSchema of type date as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String date ()
```

#### **decimal()**

`SchemaBuilder.decimal()` returns a LogicalSchema of type Decimal as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String decimal (int precision, int scale)
```

*parameter:*

- `precision` — integer specifying the maximum number of significant digits to be stored.
- `scale` — (optional) integer specifying the number of digits to the right of the decimal point. Defaults to 0.

## embedded()

SchemaBuilder.**embedded()** accepts an embedded schema description (in either JSON or RecordSchema format) and returns a string containing a JSON formatted embeddedSchema.

Note: Embedded types are currently not fully supported by the builder.

```
static String  embedded (RecordSchema embeddedSchema)
static String  embedded (String embeddedSchema)
```

*parameter:*

- embeddedSchema — (RecordSchema or String) schema describing the embedded record.

## index()

SchemaBuilder.**index()** returns an SchemaBuilder.IndexBuilder that can be used to build an index definition.

```
static IndexBuilder  index (String indexName)
```

*parameter:*

- indexName — name of the index to build.

### SchemaBuilder.IndexBuilder

The IndexBuilder class has the following methods, all of which can be chained. The final call in the chain must be the **complete()** method, which returns the finished index definition as a JSON string.

```
IndexBuilder  IndexBuilder ()
IndexBuilder  isExtent ()
IndexBuilder  isIdkey ()
IndexBuilder  isPrimaryKey ()
IndexBuilder  isUnique ()
IndexBuilder  on (String field)
IndexBuilder  withName (String name)
IndexBuilder  withType (String type)
IndexBuilder  withType (IndexType type)
String        complete ()
```

## indexes()

SchemaBuilder.**indexes()** returns a SchemaBuilder.AnnotationsBuilder object implementing addIndex() methods used to create a schemas.LogicalSchema annotations object, typically containing one or more index definitions.

Indexes are not part of a schema but can be defined and passed to an IRIS server when synchronizing a schema. Indexes are processed by the server when generating the ObjectScript class.

```
static AnnotationsBuilder  indexes ()
```

### SchemaBuilder.AnnotationsBuilder

AnnotationsBuilder has the following methods, all of which can be chained. The final call in the chain must be the **complete()** method, which returns the finished annotations object as a JSON string.

```
AnnotationsBuilder  addExtent (String name, IndexType type)
AnnotationsBuilder  addId (String name, String field)
AnnotationsBuilder  addIndex (IndexBuilder index)
AnnotationsBuilder  addIndex (String index)
AnnotationsBuilder  addIndex (String name, String field)
AnnotationsBuilder  addIndex (String name, String field, IndexType type)
String              complete ()
```

**infer()**

SchemaBuilder.**infer()** creates a schema by inference from a data value, a class instance, a class type, or a Record.

**infer( value )**

Infers a schema from a value and returns a string containing a JSON formatted schema describing the value.

```
static String infer (Object value)
```

*parameter:*

- value — a data value, can be a simple or complex value.

**infer( type )**

Infers a schema from a Java Type or Java Class object, and returns a string containing a JSON formatted schema describing the type.

```
static String infer (Class type)
static String infer (Type type)
```

*parameter:*

- type — a Java Type or Java Class object.

**infer( record )**

Infers a schema from the types of each element in the specified Object[] and returns a string containing a JSON formatted schema. If *fieldName* is specified, each value in the generated schema is assigned a name from the *fieldName* array.

```
static String infer (Object[] record)
static String infer (Object[] record, String schemaName)
static String infer (Object[] record, String schemaName, String[] fieldName)
```

*parameter:*

- record — Object[] containing representative field values
- schemaName — optional name of the schema to generate.
- fieldName — optional String array of field names to assign to Record fields.

**logical()**

SchemaBuilder.**logical()** returns an instance of SchemaBuilder.LogicalSchemaBuilder, which can be used to create a schemas.LogicalSchema object.

```
static LogicalSchemaBuilder logical ()
```

**SchemaBuilder.LogicalSchemaBuilder**

LogicalSchemaBuilder has the following methods, all of which can be chained. The final call in the chain must be the **complete()** method, which returns the finished logical schema definition as a JSON string.

```
LogicalSchemaBuilder withLogicalType (String logicalType)
LogicalSchemaBuilder withProp (String propName, String value)
LogicalSchemaBuilder withType (Schema type)
LogicalSchemaBuilder withType (String type)
String complete ()
```

## map()

SchemaBuilder.**map()** builds a map schema and returns a string containing a JSON array formatted schema.

```
static String map (Schema keysSchema, Schema valuesSchema)
static String map (String keysType, String valuesType)
```

*parameter:*

- `keysSchema` — Schema instance that is type of the map keys
- `valuesSchema` — Schema instance that is type of the map values
- `keysType` — JSON formatted Schema that is type of the map keys
- `valuesType` — JSON formatted Schema that is type of the map values

## normalizeAsJson()

SchemaBuilder.**normalizeAsJson()** calls `String.trim()` on any *value* starting with `{`, `[`, or `"` and returns the trimmed string. If it begins with any other character, *value* is returned in double quotes.

```
static String normalizeAsJson (String value)
```

*parameter:*

- `value` — String value to be normalized.

## primitive()

SchemaBuilder.**primitive()** builds a primitive schema and returns string containing a JSON formatted schema.

```
static String primitive (String type)
```

*parameter:*

- `type` — String specifying primitive schema type. Valid values are `string`, `bytes`, `short`, `int`, `long`, `float`, `double`, `boolean`, or `null`.

## record()

SchemaBuilder.**record()** returns an instance of subclass `RecordSchemaBuilder`, which can be used to create a JSON schema string.

```
static RecordSchemaBuilder record ()
```

**SchemaBuilder.RecordSchemaBuilder**

RecordSchemaBuilder has the following methods, all of which can be chained. The final call in the chain must be the **complete()** method, which returns the finished record schema definition as a JSON string:

```
RecordSchemaBuilder addField (String name)
RecordSchemaBuilder addField (String name, String type)
RecordSchemaBuilder addField (String name, Schema type)
RecordSchemaBuilder addField (String name, Class type)
RecordSchemaBuilder addField (String name, Type type)
RecordSchemaBuilder addField (RecordField field)
RecordSchemaBuilder asEmbedded ()
RecordSchemaBuilder asFinal ()
RecordSchemaBuilder category (String category)
RecordSchemaBuilder String complete ()
RecordSchemaBuilder extendsClasses (String superClasses)
RecordSchemaBuilder extendsSchema (RecordSchema superSchema)
RecordSchemaBuilder withFinal (boolean isFinal)
RecordSchemaBuilder withName (String name)
RecordSchemaBuilder withName (SchemaName name)
```

**reference()**

SchemaBuilder.**reference()** returns a LogicalSchema of type intersystems-reference as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String reference (String referencedSchema)
```

*parameter:*

- referencedSchema — the schema that is referenced

**timeMicros()**

SchemaBuilder.**timeMicros()** returns a LogicalSchema of type time-micros as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String timeMicros ()
```

**timeMillis()**

SchemaBuilder.**timeMillis()** returns a LogicalSchema of type time-millis as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String timeMillis ()
```

**timestampIrisPosix()**

SchemaBuilder.**timestampIrisPosix()** returns a LogicalSchema of type IRIS POSIX timestamp as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String timestampIrisPosix ()
```

**timeStampMicros()**

SchemaBuilder.**timeStampMicros()** returns a LogicalSchema of type timestamp-micros as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String timeStampMicros ()
```

**timeStampMillis()**

SchemaBuilder.**timeStampMillis()** returns a LogicalSchema of type timestamp-millis as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String timeStampMillis ()
```

**uuid()**

SchemaBuilder.**uuid()** returns a LogicalSchema of type UUID as a JSON formatted string (see [LogicalSchema Types](#)).

```
static String uuid ()
```

## 7.3.2 SchemaBuilder LogicalSchema Methods

SchemaBuilder supports the following logical type methods, which currently return the JSON schema strings shown here:

- **date()** — (returns logical java.util.Date) {"logicalType": "date", "type": "int"}
- **decimal()** — {"logicalType": "decimal", "type": "bytes", "precision": <int>, "scale": <int>}
- **embedded()** — {"logicalType": "intersystems-embedded", "type": "string", "value": {<embedded schema string>}}
- **reference()** — {"logicalType": "intersystems-reference", "type": "string", "value", <referencedSchema>}
- **timeMicros()** — {"logicalType": "time-micros", "type": "int"}
- **timeMillis()** — {"logicalType": "time-millis", "type": "int"}
- **timestampIrisPosix()** — {"logicalType": LogicalSchema.LOGICAL\_TYPE\_TIMESTAMP\_IRIS\_POSIX, "type": "long"}
- **timeStampMicros()** — {"logicalType": "timestamp-micros", "type": "long"}
- **timeStampMillis()** — {"logicalType": "timestamp-millis", "type": "long"}
- **uuid()** — {"logicalType": LogicalSchema.LOGICAL\_TYPE\_UUID, "type": "string"}