ThingSpan

JavaULB

Table of Contents

[Introduction 4](#_Toc4168187)

[Background 4](#_Toc4168188)

[What is ThingSpan 4](#_Toc4168189)

[What is JavaULB 4](#_Toc4168190)

[Prerequisites 4](#_Toc4168191)

[The Objectivity ThingSpan Support Web Site 5](#_Toc4168192)

[Working with ThingSpan Federations 5](#_Toc4168193)

[The **objy** Command 5](#_Toc4168194)

[The Lock Server (ools) 5](#_Toc4168195)

[Starting the Lock Server 5](#_Toc4168196)

[Stopping the Lock Server 6](#_Toc4168197)

[The Page Server (ooams) 6](#_Toc4168198)

[Starting the Page Server 6](#_Toc4168199)

[Starting the Page Server 6](#_Toc4168200)

[Creating a Federated Database 7](#_Toc4168201)

[The License Key 7](#_Toc4168202)

[Connecting to the Federation 9](#_Toc4168203)

[Lab01 9](#_Toc4168204)

[Working with Schema 10](#_Toc4168205)

[Creating a Person Class Using Java ULB 11](#_Toc4168206)

[Lab02a – Creating A Person Class 12](#_Toc4168207)

[Lab02a - Examining the Results Using DO 12](#_Toc4168208)

[Lab02b – Examining the Results Using JavaULB 13](#_Toc4168209)

[Numeric Attributes 14](#_Toc4168210)

[Integer Attributes 14](#_Toc4168211)

[Real Attributes 15](#_Toc4168212)

[Lab02c – Adding Numeric Attributes Using Java ULB 15](#_Toc4168213)

[Lab02c – Examining the Schema Using DO 15](#_Toc4168214)

[Simple Reference Attributes 16](#_Toc4168215)

[Lab02d – Simple Reference Attributes 18](#_Toc4168216)

[Lab02d – Examining the Schema Using DO 18](#_Toc4168217)

[Bidirectional Reference Attributes 18](#_Toc4168218)

[Lab02e – Bidirectional Reference Attributes 19](#_Toc4168219)

[Lab02e - Examining the Schema Using DO 19](#_Toc4168220)

[To-Many Reference Attributes 20](#_Toc4168221)

[Lab02f – To-Many Bidirectional Reference Attributes 20](#_Toc4168222)

[Lab02f - Examining the Schema Using DO 21](#_Toc4168223)

[Many-To-Many Bidirectional Relationships 21](#_Toc4168224)

[Lab02g - Examining the Schema Using DO 22](#_Toc4168225)

1. Introduction
2. Background
   1. What is ThingSpan?
   2. What is JavaULB?
   3. Prerequisites
3. Working with ThingSpan Federations
   1. The **objy** Command
   2. The Lock Server
   3. The Page Server
   4. The License Key
4. Connecting to the Federation
5. Working with Schema
   1. Creating Schema
   2. Reading Schema
   3. Updating Schema
6. Transactions
7. Objects 1.0: Classes, Instances, and Attributes
   1. Creating an Instance
   2. Populating Attributes
8. Objects 2.0: Collections
9. Queries
   1. Simple Lookups
   2. DO Queries
      1. Processing Query Results
10. Advanced Topics
    1. Avoiding Lock Conflicts
    2. Building a Simple REST Server

# Introduction

This document provides the reader with an introduction to the ThingSpan Java ULB API. It covers the major topics that a developer will need to become familiar with in order to develop high performance applications that interface with the ThingSpan Java ULB API.

# Background

## What is ThingSpan

Objectivity’s ThingSpan is a purpose-built, massively scalable graph software platform, powered by Objectivity/DB that leverages the open source stack by natively integrating with Apache Spark and the Hadoop Distributed File System (HDFS). It provides ultra-fast navigation and pathfinding queries against huge distributed graphs. ThingSpan also supports parallel pattern-finding and predictive analytics in combination with Spark components, such as MLlib, GraphX, and Spark SQL.

ThingSpan excels in a mixed workload environment, fusing metadata from real-time streaming and sensor-based data with its distributed graph of stateful information to provide “in-time” context. In essence, ThingSpan bridges the gap of open technologies by realizing the full potential of streaming Fast Data processed by Spark and static Big Data stored in HDFS.

## What is JavaULB

Java ULB is the Java Universal Language Binding for ThingSpan. Java ULB is a second generation Java API for the Objectivity/DB product line and it completely replaces the first generation Java API. Java ULB takes a completely new approach to interfacing with the ThingSpan persistent objects. This new approach relies almost entirely on a Java Native Interface architecture that allows all of the heavy lifting to occur in the highly tuned C++ ThingSpan kernel.

## Prerequisites

In order to use Java ULB you will need:

* A ThingSpan installation,
* A valid license key (oolicense.txt file), and
* Java 1.8 or higher.

You will also need an editor or integrated development environment such as Eclipse or Netbeans.

# The Objectivity ThingSpan Support Web Site

<https://support.objectivity.com/sites/default/files/docs/thingspan/R15_6_2/index.html#page/topics%2FwelcomeThingSpanPlatform.html>

# Working with ThingSpan Federations

## The **objy** Command

The objy command, also known as “Tool Runner” is a wrapper that provides access to a large number of ThingSpan operations. For the purposes of this document we are only concerned with a small subset of those operations. Specifically, we are concerned with the operations that

* Start and stop the lock server,
* Start and stop the page server,
* Create, cleanup and delete federated database,
* Perform DO queries

For a complete list of operations that can be performed see:

<https://support.objectivity.com/sites/default/files/docs/thingspan/R15_6_2/index.html#page/topics%2Fcommon%2Fadministration%2FadmPart-Tools.html%23>

## The Lock Server (ools)

The ThingSpan lock server manages several different kinds of locks on regions of the database to ensure the integrity of the data being written and read.

### Starting the Lock Server

On a Microsoft Windows computer you can start and stop the lock server in the services tool by starting and stopping the **ools** Windows Service or you can do it from the command line as follows.

**Note**: The “startlockserver” command must be run as administrator.

**$ objy startlockserver**

Objectivity/DB (TM) Start Lock Server, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

Lock Server has been started.

Success.

**$ objy startlockserver**

Objectivity/DB (TM) Start Lock Server, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

Lock Server is already running.

Success.

### Stopping the Lock Server

**$ objy stoplockserver**

Objectivity/DB (TM) Stop Lock Server, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

Lock server on host *MyHostName* terminated.

Success.

**$ objy stoplockserver**

Objectivity/DB (TM) Stop Lock Server, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

The Lock Server is not running on host *MyHostName*

## The Page Server (ooams)

The page server is called the Advanced Multithreaded Server or AMS. The process or service name is **ooams**.

Like the lock server, on a Microsoft Windows computer you can start and stop the page server in the services tool by starting and stopping the **ooams** Windows Service or you can do it from the command line as follows.

### Starting the Page Server

**$ objy startams**

Objectivity/DB (TM) Start Ams, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

The AMS has been started (process ID = 0).

Success.

### Starting the Page Server

**$ objy stopams**

Objectivity/DB (TM) Stop AMS, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

AMS on host *MyHostName* terminated.

Success.

PS C:\WINDOWS\system32>

## Creating a Federated Database

ThingSpan data is stored in a ThingSpan federated database. If you don’t have a federated database you can create one using the **objy createFD** command. For a simple federation you simply run:

**$ objy createFD –fdname Sample**

Objectivity/DB (TM) Create FD, Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

Federated Database successfully created:

FD Dir Host : *MyHostName*

FD Dir Path : C:\SampleFDDir

System DB file : Sample.fdb

Boot file : Sample.boot

Lock server host : *MyHostName*

This will create two files:

* Sample.fdb The federated database file
* Sample.boot The boot file used to contain properties of the federation.

The FDB file will contain things like schema definitions and the catalogs of the databases that will be created to contain your objects.

The boot file contains information like where to find the lock server and the FDB file. When you write an application, you will open a connection to the federation by providing the boot file. The ThingSpan connection API will then use the information in the boot file to establish the connection and then you can create interact with the database through this connection.

## The License Key

In order to use a ThingSpan federation you must install a valid license key in the federation. The license key can be obtained from Objectivity, Inc. and comes in a file called oolicense.txt.

On a MS Windows operating system the oolicense.txt file should be placed in “C:\Windows”.

If you run the **objy createFD** command after you place the oolicense.txt file in C:\Windows, the createFD command will find the oolicense.txt file and apply the license the federation when it is created.

If you do not have an oolicense.txt file when you run objy createFD the federation will not be licensed and you will not be able to interact with it. Once you receive your oolicense.txt file and install it in C:\Windows, you can then apply the new license to the existing federation by running the command:

**$ objy license –bootfile Sample.boot**

This assumes that the oolicense.txt file is in the default location, C:\Windows. If you cannot put the oolicense.txt file in C:\Windows, you can place it anywhere on the system that you’d like and then run the command:

**$ objy license –licensefile [path to oolicense.txt] –bootfile Sample.boot**

In most cases the license has an expiration date. When you get an new license you can replace the old license with the new license and run either of the two commands above to apply the new license to your federation and continue running your applications.

Note: Licenses only apply to the federation. There is no licensing for individual applications. Once a federation is licensed, all user defined applications will have access to that federation.

# Connecting to the Federation

In order to perform any operations on a ThingSpan federation you must have an open connection to that federation.

String bootFile = “C:\\Projects\\Sample\\db\\Sample.boot”;

Connection connection;

try {

connection = new Connection(bootFile);

} catch (Exception ex) {

ex.printStackTrace();

}

The Connection constructor will throw an exception if it can’t find the specified boot file, if the FD isn’t properly licensed, or if the license has expired.

## Lab01

The code above is contained in JavaULB\_Lab01. You can run this code by running the following command:

PS D:\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab01 clean build recreatefd run**

This gradlew.bat command cleans and builds the project. Then it recreates the federation and finally it runs the JavaULB\_Lab02 application which includes a call to the function shown above.

# Working with Schema

There are a number of ways to work with schema in ThingSpan. The first and easiest way to create and modify schema is through the DO query language. DO can be used via DO Runner (objy DO …) or you can create and execute DO statements from within Java ULB.

The second way to work with schema is by using the schema APIs in Java ULB. That is what we will demonstrate here.

There are specific steps that need to be performed when building schema programmatically. The steps must be performed after connecting to the federation and also within the scope of an active READ\_UPDATE transaction.

The basic steps are:

1. Using the SchemaProvider, get the defaultPersistent provider, and call refresh(true).

SchemaProvider.getDefaultPersistentProvider().refresh(true);

1. Create an instance of com.objy.data.ClassBuilder, passing the name of the schema class you want to create as a parameter to the constructor.

com.objy.data.ClassBuilder cBuilder

= new com.objy.data.ClassBuilder("Person");

1. On your instance of com.objy.data.ClassBuilder, call addAttribute(…) for each attribute you want to add to your new class definition.

cBuilder.addAttribute(LogicalType.STRING, "FirstName");

The first parameter is the LogicalType of the attribute. The Java ULB class LogicalType is an enumeration of the available logical types.

The second parameter is the case sensitive name of the attribute.

1. Next, you will need to call “build()” on your ClassBuilder object.

com.objy.data.Class cPerson = cBuilder.build();

This give you a transient schema definition of your Person class in a com.objy.data.Class object.

1. Next, you will need to call “represent(…)” on the com.objy.data.Class object that was returned from the call to “build()”.

SchemaProvider.getDefaultPersistentProvider().represent(cPerson);

The call to “represent(…)” actually writes the schema definition into the schema catalog of the federation.

When you call tx.complete() your updates to the federation schema will be complete and the federation schema catalog will contain the new definition for the class Person.

Schema development can become very complex and we’ll cover some of those topics in this training.

### Creating a Person Class Using Java ULB

private void createSchemaPerson() {

int transLCERetryCount = 0;

boolean transactionSuccessful = false;

while (!transactionSuccessful) {

// Create a new TransactionScope that is READ\_UPDATE.

try (TransactionScope tx = new TransactionScope(

TransactionMode.READ\_UPDATE)) {

// Ensure that our view of schema is up to date.

SchemaProvider

.getDefaultPersistentProvider()

.refresh(true);

// Use ClassBuilder to create schema definition.

com.objy.data.ClassBuilder cBuilder

= new com.objy.data.ClassBuilder("Person");

cBuilder

.addAttribute( LogicalType.STRING,

"FirstName");

cBuilder

.addAttribute( LogicalType.STRING,

"LastName");

cBuilder

.addAttribute( LogicalType.STRING,

"MiddleInitial");

cBuilder

.addAttribute( LogicalType.DATE,

"Birthdate");

// Actually build the schema representation.

com.objy.data.Class cPerson = cBuilder.build();

// Represent class into the federated database.

SchemaProvider

.getDefaultPersistentProvider()

.represent(cPerson);

// Complete and close the transaction

tx.complete();

tx.close();

logger.info("Person class created in schema.");

transactionSuccessful = true;

} catch(LockConflictException lce) {

logger.info("LockConflictException. "

+ "Attempting retry... retryCount = "

+ ++transLCERetryCount);

try {

Thread.sleep(10\*transLCERetryCount);

} catch(InterruptedException ie) { }

} catch (Exception ex) {

ex.printStackTrace();

}

}

}

## Lab02a – Creating A Person Class

The code above is contained in JavaULB\_Lab02. You can run this code by running the following command:

PS D:\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 clean build recreatefd run02a**

This gradlew.bat command cleans and builds the project. Then it recreates the federation and finally it runs the JavaULB\_Lab02 application which includes a call to the function shown above.

### Lab02a - Examining the Results Using DO

We can use DO Runner (objy DO) to examine the results of the code above.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs> **objy DO -boot Lab02.boot**

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

DO> show classes;

{

'Person'

}

DO> show class Person;

CLASS Person

{

FirstName: String { Encoding: UTF16, Storage: Variable },

LastName: String { Encoding: UTF16, Storage: Variable },

MiddleInitial: String { Encoding: UTF16, Storage: Variable },

Birthdate: Date

}

DO> from Person return \*;

{}

DO> /exit

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs>

### Lab02b – Examining the Results Using JavaULB

Lab02b shows how to retrieve a schema type from the catalog and list its attributes using the Java ULB APIs.

The key block of code is show below and must be within an active Connection and TransactionScope.

// Lookup the "Person" class from the schema in the database.

com.objy.data.Class cxPerson =

com.objy.data.Class.lookupClass("Person");

// Iterate over the attributes in our Person class.

// getAttrbutes() returns an Interator<Variable> object.

for (Variable v : cxPerson.getAttributes()) {

// We must interpret v as an attribute value.

Attribute at = v.attributeValue();

logger.info(String.format("Attribute: %-20s %s",

at.getName(),

at.getAttributeValueSpecification().getLogicalType()));

}

## Numeric Attributes

ThingSpan provides support for both integer and real numbers as attributes in persistent data types and different sizes of each are supported. The different byte sizes for each type can be found in **com.objy.data.Storage** and are:

* com.objy.data.Storage.B8
* com.objy.data.Storage.B16
* com.objy.data.Storage.B32
* com.objy.data.Storage.B64

### Integer Attributes

Creating an integer attribute where you specify the byte size is slightly different that we saw in Lab02a. In order to specify the size of the integer we now have to specify the encoding as Encoding.Integer.SIGNED or Encoding.Integer.UNSIGNED as well. We do all of this by using call chaining on the IntegerSpecificationBuilder class.

Sample addAttribute calls are shown below:

com.objy.data.ClassBuilder cBuilder

= new com.objy.data.ClassBuilder("NumbersDemo");

// When you don’t use an IntegerSpecificationBuilder you get the

// default settings for the attribute which are B32 and SIGNED.

cBuilder.addAttribute(LogicalType.INTEGER, "SimpleInteger");

cBuilder.addAttribute("MyIntB8\_Signed",

new IntegerSpecificationBuilder(Storage.Integer.B8)

.setEncoding(Encoding.Integer.SIGNED)

.build());

cBuilder.addAttribute("MyIntB16\_Signed",

new IntegerSpecificationBuilder(Storage.Integer.B16)

.setEncoding(Encoding.Integer.SIGNED)

.build());

cBuilder.addAttribute("MyIntB32\_Signed",

new IntegerSpecificationBuilder(Storage.Integer.B32)

.setEncoding(Encoding.Integer.SIGNED)

.build());

cBuilder.addAttribute("MyInt64\_Signed",

new IntegerSpecificationBuilder(Storage.Integer.B64)

.setEncoding(Encoding.Integer.SIGNED)

.build());

cBuilder.addAttribute("MyIntB8\_Unsigned",

new IntegerSpecificationBuilder(Storage.Integer.B8)

.setEncoding(Encoding.Integer.UNSIGNED)

.build());

cBuilder.addAttribute("MyIntB16\_Unsigned",

new IntegerSpecificationBuilder(Storage.Integer.B16)

.setEncoding(Encoding.Integer.UNSIGNED)

.build());

cBuilder.addAttribute("MyIntB32\_Unsigned",

new IntegerSpecificationBuilder(Storage.Integer.B32)

.setEncoding(Encoding.Integer.UNSIGNED)

.build());

cBuilder.addAttribute("MyInt64\_Unsigned",

new IntegerSpecificationBuilder(Storage.Integer.B64)

.setEncoding(Encoding.Integer.UNSIGNED)

.build());

### Real Attributes

Like integer attributes, real attributes also allow the user to use a builder. Real attributes only come in B32 and B64 bit sizes and only have an IEEE encoding.

cBuilder.addAttribute(LogicalType.REAL, "SimpleReal");

cBuilder.addAttribute("MyReal32\_IEEE",

new RealSpecificationBuilder(Storage.Real.B32)

.setEncoding(Encoding.Real.IEEE)

.build());

cBuilder.addAttribute("MyReal64\_IEEE",

new RealSpecificationBuilder(Storage.Real.B64)

.setEncoding(Encoding.Real.IEEE)

.build());

### Lab02c – Adding Numeric Attributes Using Java ULB

Lab02c demonstrates how to create a class that has all of the different numeric data type in a single class using the default, IntegerSpecificationBuilder, and RealSpecificationBuilder.

To run Lab02c from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 clean build run2c**

### Lab02c – Examining the Schema Using DO

Using DO Runner (objy DO) we can examine the NumbersDemo class created in Lab02c.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs> **objy DO -boot Lab02.boot**

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

DO> **show schema;**

{

CLASS NumbersDemo

{

SimpleInteger: Integer { Encoding: Signed, Storage: B32 },

MyIntB8\_Signed: Integer { Encoding: Signed, Storage: B8 },

MyIntB16\_Signed: Integer { Encoding: Signed, Storage: B16 },

MyIntB32\_Signed: Integer { Encoding: Signed, Storage: B32 },

MyInt64\_Signed: Integer { Encoding: Signed, Storage: B64 },

MyIntB8\_Unsigned: Integer { Encoding: Unsigned, Storage: B8 },

MyIntB16\_Unsigned: Integer { Encoding: Unsigned, Storage: B16 },

MyIntB32\_Unsigned: Integer { Encoding: Unsigned, Storage: B32 },

MyInt64\_Unsigned: Integer { Encoding: Unsigned, Storage: B64 },

SimpleReal: Real { Storage: B32 },

MyReal32\_IEEE: Real { Storage: B32 },

MyReal64\_IEEE: Real { Storage: B64 }

}

}

## Simple Reference Attributes

One of the great things about ThingSpan is the ability to link objects together. This capability is made possible because of the fact that every object has an Object Identifier or OID which represents the logical address of the object within the database. A schema class definition can be created with an attribute that is of type “Reference” and then objects of that class can have that attribute populated with the OID of some other object.

For example, in the diagram below we have two classes, Person and Address, where the Person class has a reference attribute called “LivesAt” which contains the OID of Address object. When we query for the person object and read it from disk we can use the OID in the LivesAt attribute to retrieve the Address object without having to search for it. The OID tells us where it is.

Person

FirstName

LastName

MiddleInitial

LivesAt 3-5-7-19

Address

Street1

Street2

City

State

ZIP

OID: 3-5-7-19

OID: 3-8-2-4

Adding a reference attribute to a class is accomplished using a **ReferenceSpecificationBuilder**.

//--------------------------------------------------------------

// Use ClassBuilder to create the schema definition.

com.objy.data.ClassBuilder cBuilderAddress

= new com.objy.data.ClassBuilder("Address");

cBuilderAddress.addAttribute(LogicalType.STRING, "Street1");

cBuilderAddress.addAttribute(LogicalType.STRING, "Street2");

cBuilderAddress.addAttribute(LogicalType.STRING, "City");

cBuilderAddress.addAttribute(LogicalType.STRING, "State");

cBuilderAddress.addAttribute(LogicalType.STRING, "ZIP");

// Actually build the the schema representation.

com.objy.data.Class cAddress = cBuilderAddress.build();

// Represent the new class into the federated database.

SchemaProvider.getDefaultPersistentProvider()

.represent(cAddress);

// Use ClassBuilder to create the schema definition.

com.objy.data.ClassBuilder cBuilderPerson = new com.objy.data.ClassBuilder("Person");

cBuilderPerson.addAttribute(LogicalType.STRING, "FirstName");

cBuilderPerson.addAttribute(LogicalType.STRING, "LastName");

cBuilderPerson.addAttribute(LogicalType.STRING, "MiddleInitial");

cBuilderPerson.addAttribute(LogicalType.DATE, "Birthdate");

cBuilderPerson.addAttribute("LivesAt",

new ReferenceSpecificationBuilder()

.setReferencedClass("Address")

.build());

// Actually build the the schema representation.

com.objy.data.Class cPerson = cBuilderPerson.build();

// Represent the new class into the federated database. SchemaProvider.getDefaultPersistentProvider().represent(cPerson);

### Lab02d – Simple Reference Attributes

Lab02d demonstrates the code shown above.

To run Lab02d from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 clean build run2d**

### Lab02d – Examining the Schema Using DO

Using DO Runner (objy DO) we can examine the Address and Person classes created in Lab02d.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs> **objy DO -boot Lab02.boot**

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

DO> **show schema;**

{

CLASS Address

{

Street1: String { Encoding: UTF16, Storage: Variable },

Street2: String { Encoding: UTF16, Storage: Variable },

City: String { Encoding: UTF16, Storage: Variable },

State: String { Encoding: UTF16, Storage: Variable },

ZIP: String { Encoding: UTF16, Storage: Variable }

},

CLASS Person

{

FirstName: String { Encoding: UTF16, Storage: Variable },

LastName: String { Encoding: UTF16, Storage: Variable },

MiddleInitial: String { Encoding: UTF16, Storage: Variable },

Birthdate: Date,

LivesAt: Reference { Referenced: Address }

}

}

## Bidirectional Reference Attributes

In our example above we create a uni-directional reference attribute in Person that pointed to a single Address object. The Address object did not point back to the Person object. Let’s change that.

Here we are going to create a bidirectional relationship between the two classes by creating reference attributes in each class. Each reference attribute will be created to name the class its referencing with the additional feature of naming the inverse attribute that points back.

Graphically, it will look like this:

Person

FirstName

LastName

MiddleInitial

LivesAt

Address

Street1

Street2

City

State

ZIP

LivesHere

OID: 3-5-7-19

OID: 3-8-2-4

3-8-2-4

3-5-7-19

Adding the attributes to each class will be done with the following code. Note that we are setting the inverse attribute on each ReferenceSpecificationBuilder.

cBuilderAddress.addAttribute("LivesHere",

new ReferenceSpecificationBuilder()

.setReferencedClass("Person")

.setInverseAttribute("LivesAt")

.build());

cBuilderPerson.addAttribute("LivesAt",

new ReferenceSpecificationBuilder()

.setReferencedClass("Address")

.setInverseAttribute("LivesHere")

.build());

### Lab02e – Bidirectional Reference Attributes

Lab02e demonstrates the code shown above.

To run Lab02e from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 clean build run2e**

### Lab02e - Examining the Schema Using DO

Using DO Runner (objy DO) we can examine the Address and Person classes created in Lab02e.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs> **objy DO -boot Lab02.boot**

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

DO> **show schema;**

{

CLASS Address

{

Street1: String { Encoding: UTF16, Storage: Variable },

Street2: String { Encoding: UTF16, Storage: Variable },

City: String { Encoding: UTF16, Storage: Variable },

State: String { Encoding: UTF16, Storage: Variable },

ZIP: String { Encoding: UTF16, Storage: Variable },

LivesHere: Reference { Referenced: Person, Inverse: LivesAt }

},

CLASS Person

{

FirstName: String { Encoding: UTF16, Storage: Variable },

LastName: String { Encoding: UTF16, Storage: Variable },

MiddleInitial: String { Encoding: UTF16, Storage: Variable },

Birthdate: Date,

LivesAt: Reference { Referenced: Address, Inverse: LivesHere }

}

}

### To-Many Reference Attributes

All of the examples thus far have been one-to-one references. Now we will describe the mechanics of a one-to-many attribute.

To-Many attributes are constructed using a list of references. We use the ReferenceSpecificationBuilder we’ve used previously, but now we wrap it in a ListSpecificationBuilder object.

cBuilderAddress.addAttribute("LivesHere",

new ListSpecificationBuilder()

.setElementSpecification(

new ReferenceSpecificationBuilder()

.setReferencedClass("Person")

.setInverseAttribute("LivesAt")

.build())

.build());

Here, we’ve simply extended our definition of Address to include the To-Many list of references to Person. The definition of Person doesn’t change.

### Lab02f – To-Many Bidirectional Reference Attributes

Lab02f demonstrates the code shown above.

To run Lab02e from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab02 clean build run2f**

### Lab02f - Examining the Schema Using DO

Using DO Runner (objy DO) we can examine the Address and Person classes created in Lab02f.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs> **objy DO -boot Lab02.boot**

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

DO> **show schema;**

{

CLASS Address

{

Street1: String { Encoding: UTF16, Storage: Variable },

Street2: String { Encoding: UTF16, Storage: Variable },

City: String { Encoding: UTF16, Storage: Variable },

State: String { Encoding: UTF16, Storage: Variable },

ZIP: String { Encoding: UTF16, Storage: Variable },

LivesHere: **List** { Element: Reference { Referenced: Person, Inverse: LivesAt }, Storage: **Variable** }

},

CLASS Person

{

FirstName: String { Encoding: UTF16, Storage: Variable },

LastName: String { Encoding: UTF16, Storage: Variable },

MiddleInitial: String { Encoding: UTF16, Storage: Variable },

Birthdate: Date,

LivesAt: Reference { Referenced: Address, Inverse: LivesHere }

}

}

This example demonstrate a One-to-Many relationship where one Person LivesAt an Address but an Address can have many LivesHere values.

### Many-To-Many Bidirectional Relationships

It would be easy enough to make the relationship Many-to-Many by simply wrapping the wrapping the LivesAt ReferenceSpecificationBuilder in a ListAttributeBuilder like we did for the LivesHere attribute in Address:

cBuilderPerson.addAttribute("LivesAt",

new ListSpecificationBuilder()

.setElementSpecification(

new ReferenceSpecificationBuilder()

.setReferencedClass("Address")

.setInverseAttribute("LivesHere")

.build())

.build());

### Lab02g - Examining the Schema Using DO

Using DO Runner (objy DO) we can examine the Address and Person classes created in Lab02g.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs> objy DO -boot Lab02.boot

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.7.0

Copyright (c) Objectivity, Inc 2012, 2018. All rights reserved.

DO> show schema;

{

CLASS Address

{

Street1: String { Encoding: UTF16, Storage: Variable },

Street2: String { Encoding: UTF16, Storage: Variable },

City: String { Encoding: UTF16, Storage: Variable },

State: String { Encoding: UTF16, Storage: Variable },

ZIP: String { Encoding: UTF16, Storage: Variable },

LivesHere: List { Element: Reference { Referenced: Person, Inverse: LivesAt }, Storage: Variable }

},

CLASS Person

{

FirstName: String { Encoding: UTF16, Storage: Variable },

LastName: String { Encoding: UTF16, Storage: Variable },

MiddleInitial: String { Encoding: UTF16, Storage: Variable },

Birthdate: Date,

LivesAt: List { Element: Reference { Referenced: Address, Inverse: LivesHere }, Storage: Variable }

}

}

There are a number of advanced topics that will be discussed later including how we can specify a different storage type, which is currently “Variable”.