ThingSpan

JavaULB

Table of Contents

[1 Introduction 6](#_Toc17751044)

[2 Background 6](#_Toc17751045)

[2.1 What is ThingSpan 6](#_Toc17751046)

[2.2 What is JavaULB 6](#_Toc17751047)

[2.3 Prerequisites 6](#_Toc17751048)

[3 The Objectivity ThingSpan Support Web Site 7](#_Toc17751049)

[4 Working with ThingSpan Federations 7](#_Toc17751050)

[4.1 The **objy** Command 7](#_Toc17751051)

[4.2 The Lock Server (ools) 7](#_Toc17751052)

[4.2.1 Starting the Lock Server 7](#_Toc17751053)

[4.2.2 Stopping the Lock Server 8](#_Toc17751054)

[4.3 The Page Server (ooams) 8](#_Toc17751055)

[4.3.1 Starting the Page Server 8](#_Toc17751056)

[4.3.2 Starting the Page Server 8](#_Toc17751057)

[4.4 Creating a Federated Database 9](#_Toc17751058)

[4.5 The License Key 9](#_Toc17751059)

[5 Connecting to the Federation 11](#_Toc17751060)

[5.1 Lab01 11](#_Toc17751061)

[6 Working with Schema 12](#_Toc17751062)

[6.1.1 Creating a Person Class Using Java ULB 13](#_Toc17751063)

[6.2 Lab02a – Creating A Person Class 15](#_Toc17751064)

[6.2.1 Lab02a - Examining the Results Using DO 15](#_Toc17751065)

[6.2.2 Lab02b – Examining the Results Using JavaULB 15](#_Toc17751066)

[6.3 Numeric Attributes 17](#_Toc17751067)

[6.3.1 Integer Attributes 17](#_Toc17751068)

[6.3.2 Real Attributes 18](#_Toc17751069)

[6.3.3 Lab02c – Adding Numeric Attributes Using Java ULB 19](#_Toc17751070)

[6.3.4 Lab02c – Examining the Schema Using DO 19](#_Toc17751071)

[6.4 Simple Reference Attributes 20](#_Toc17751072)

[6.4.1 Lab02d – Simple Reference Attributes 21](#_Toc17751073)

[6.4.2 Lab02d – Examining the Schema Using DO 21](#_Toc17751074)

[6.5 Bidirectional Reference Attributes 22](#_Toc17751075)

[6.5.1 Lab02e – Bidirectional Reference Attributes 23](#_Toc17751076)

[6.5.2 Lab02e - Examining the Schema Using DO 23](#_Toc17751077)

[6.6 LIST Attributes 23](#_Toc17751078)

[6.6.1 Lab02f – List of Strings Attributes 24](#_Toc17751079)

[6.6.2 Lab02f - Examining the Schema Using DO 25](#_Toc17751080)

[6.7 To-Many Reference Attributes 25](#_Toc17751081)

[6.7.1 Lab02g – To-Many Bidirectional Reference Attributes 25](#_Toc17751082)

[6.7.2 Lab02g - Examining the Schema Using DO 26](#_Toc17751083)

[6.7.3 Many-To-Many Bidirectional Relationships 26](#_Toc17751084)

[6.7.4 Lab02h – Many-To-Many Bidirectional Relationships 27](#_Toc17751085)

[6.7.5 Lab02h - Examining the Schema Using DO 27](#_Toc17751086)

[6.8 Embedded Objects as Attributes 27](#_Toc17751087)

[6.8.1 Lab02i – Creating Embeddable Classes 29](#_Toc17751088)

[6.8.2 Lab02i – Examining the Results 29](#_Toc17751089)

[6.9 Define Subclasses 29](#_Toc17751090)

[7 Working with Objects 31](#_Toc17751091)

[7.1 The ThingSpan Technology Stack 31](#_Toc17751092)

[7.2 Working with com.objy.data.Instance Objects 31](#_Toc17751093)

[7.3 Creating an Instance 31](#_Toc17751094)

[7.3.1 Lab03a – Creating an Instance 32](#_Toc17751095)

[7.3.2 Lab03a – Examining the Results 33](#_Toc17751096)

[7.4 Setting Attributes 33](#_Toc17751097)

[7.4.1 Setting String and Date Attributes 33](#_Toc17751098)

[7.4.1.1 Lab03b – Setting Attributes 34](#_Toc17751099)

[7.4.1.2 Lab03b – Examining the Results 35](#_Toc17751100)

[7.4.2 Setting Numeric Attributes 35](#_Toc17751101)

[7.4.2.1 Lab03c – Setting Numeric Attributes 36](#_Toc17751102)

[7.4.2.2 Lab03c – Examining the Results 36](#_Toc17751103)

[7.4.3 Setting DATE and DATE\_TIME Attributes 37](#_Toc17751104)

[7.4.3.1 Setting DATE Attributes 37](#_Toc17751105)

[7.4.3.2 Setting DATE\_TIME Attributes 37](#_Toc17751106)

[7.4.3.3 Lab03d – Setting DATE and DATE\_TIME Attributes 37](#_Toc17751107)

[7.4.3.4 Lab03d – Examining the Results 38](#_Toc17751108)

[7.4.4 Setting LIST Attributes 38](#_Toc17751109)

[7.4.4.1 A List of Strings 38](#_Toc17751110)

[7.4.4.1.1 Lab03e – Setting LIST Attributes 39](#_Toc17751111)

[7.4.4.1.2 Lab03e – Examining the Results 40](#_Toc17751112)

[7.4.4.2 A List of References: To-Many 40](#_Toc17751113)

[7.4.4.2.1 Lab03f – Adding References to a List Attribute 42](#_Toc17751114)

[7.4.4.2.2 Lab03f – Examining the Results 43](#_Toc17751115)

[7.4.4.2.3 Lab03g - Removing References from a List Attribute 44](#_Toc17751116)

[7.4.5 Working with Map Attributes 45](#_Toc17751117)

[7.4.6 Working with Embedded Objects 46](#_Toc17751118)

[7.4.6.1 Lab03h – Creating an Instance Object with an Embedded Object Attribute 46](#_Toc17751119)

[7.4.6.2 Lab03h – Examining the Results 47](#_Toc17751120)

[8 Object Lookup and Queries 47](#_Toc17751121)

[8.1 Object Lookup by OID 47](#_Toc17751122)

[8.1.1 Lab04a – Looking Up an Object Using the OID 48](#_Toc17751123)

[8.1.2 Lab04a – Examining the Results 49](#_Toc17751124)

[8.2 Queries 50](#_Toc17751125)

[8.2.1 Lab04b – Running a DO Query in JavaULB 51](#_Toc17751126)

[8.2.2 An Introduction to Projections 53](#_Toc17751127)

[8.2.2.1 A Simple Projection 53](#_Toc17751128)

[8.2.2.2 OIDs in Projections 55](#_Toc17751129)

[8.2.2.3 Using Dot Notation to Traverse Relationship 55](#_Toc17751130)

[8.2.2.4 Lab04d – Using Dot Notation to Traverse Relationship 56](#_Toc17751131)

[9 Graphs 58](#_Toc17751132)

[9.1 Creating Schema for a Graph 59](#_Toc17751133)

[10 Session Logging 59](#_Toc17751134)

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 federated database using the 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 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 a 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. This will allow you to continue using the federation with 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 run01

> Task :JavaULB\_Lab01:run01

Running 'run1'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab01/build/libs/

log4j:ERROR Could not find value for key log4j.appender.FILE

log4j:ERROR Could not instantiate appender named "FILE".

2019-03-26 15:56:30 INFO main Lab01.java: Running Lab01

2019-03-26 15:56:30 INFO main Lab01.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab01\data\dbs\Lab01.boot>

2019-03-26 15:56:30 INFO main Lab01.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab01\data\dbs\Lab01.boot

2019-03-26 15:56:30 INFO main Lab01.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab01\data\dbs\Lab01.boot

2019-03-26 15:56:30 INFO main Lab01.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab01\data\dbs\Lab01.boot

This gradlew.bat command cleans and builds the project. Then it recreates the federation and finally it runs the JavaULB\_Lab01 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 …**). The second way to create schema is to create and execute DO command from within your Java program.

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

There are specific steps that need to be performed when building schema programmatically. All 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 you wish to create.

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

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

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

1. Next, you will need to call “represent(…)” on the persistent provider returned from the SchemaProvider, passing the com.objy.data.Class object as a parameter.

SchemaProvider.getDefaultPersistentProvider().represent(cPerson);

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

1. Next, using SchemaProvider.getDefautlPersistenceProvider(), you need to activate the edits to the schema by calling activateEdits().

**Note**: This is a crucial step for determining if there are any errors in your schema “proposals”. If you do not call activateEdits() they will be activated during the call to tx.complete() but if there are any problems, the call to tx.complete() will fail without any error messages. By calling activateEdits() you will get descriptive error messages telling you what you’ve done wrong if there are problems.

SchemaProvider.getDefaultPersistentProvider().activateEdits();

1. Finally, you will need to call “complete()” on your transaction to commit you schema updates to the schema catalog in the federated database.

tx.complete()

When you call tx.complete() your updates to the federation schema will be completed 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);

// Activate the proposed edits.

// This will provide error messages if there

// are any problems with your schema.

SchemaProvider

.getDefaultPersistentProvider()

.activateEdits();

// 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 lab 2a 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 shown 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()));

}

The command to run lab 2b is:

PS D:\Root\Workspaces\JavaULBTraining> .\gradlew.bat -p JavaULb\_Lab02 clean build run2b

**Note**: This invocation does not include a **recreateFD** task as this would remove the schema data that was created in Lab02a.

The result is:

> Task :JavaULB\_Lab02:run2b

Running 'run2b'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab02/build/libs/

2019-04-04 14:17:50 INFO main Lab02b.java: Running Lab02b

2019-04-04 14:17:50 INFO main Lab02b.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs\Lab02.boot>

2019-04-04 14:17:50 INFO main Lab02b.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs\Lab02.boot

2019-04-04 14:17:51 INFO main Lab02b.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs\Lab02.boot

2019-04-04 14:17:51 INFO main Lab02b.java: Attribute: FirstName STRING

2019-04-04 14:17:51 INFO main Lab02b.java: Attribute: LastName STRING

2019-04-04 14:17:51 INFO main Lab02b.java: Attribute: MiddleInitial STRING

2019-04-04 14:17:51 INFO main Lab02b.java: Attribute: Birthdate DATE

2019-04-04 14:17:51 INFO main Lab02b.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab02\data\dbs\Lab02.boot

## 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);

// Activate the edits.

SchemaProvider.getDefaultPersistentProvider().activateEdits();

### 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 }

}

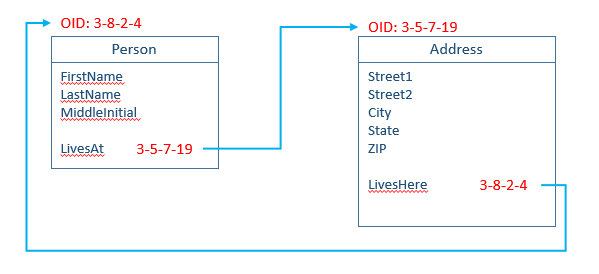
}

## 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:



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 }

}

}

## LIST Attributes

LIST attributes are used to store basic data types like numbers and strings but lists can also be used to store references to other objects. This second mechanism is how we build a to-many relationship attribute. Before we jump into that let’s start with a list of Java Strings.

The following code sample taken from Lab03f.java shows the creation of the scheme for a Person class using the ClassBuilder. Several simple attributes are added and finally a “MyPhoneNumbers” attribute is added using the ListSpecificationBuilder and the StringSpecificationBuilder classes.

The important thing to note here and for future reference is that we have to define the kind of list we’re building and we do that by calling the “setElementSpecification(…)” method on our new ListSpecificationBuilder objects, passing in the results of the “new StringSpecificationBuilder().build()”.

// Use ClassBuilder to create the 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, "DateOfBirth");

cBuilder.addAttribute(LogicalType.DATE\_TIME, "Timestamp");

cBuilder.addAttribute(

"MyPhoneNumbers",

new ListSpecificationBuilder()

.setElementSpecification(

new StringSpecificationBuilder()

.build())

.build());

// Actually build the the schema representation.

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

// Represent the new class into the federated database.

SchemaProvider.getDefaultPersistentProvider().represent(cPerson);

// Activate the edits.

SchemaProvider.getDefaultPersistentProvider().activateEdits();

You can create lists of the different data types by simply using the SpecificationBuilder for the desired type.

### Lab02f – List of Strings Attributes

This lab 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

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 class Person;

CLASS Person

{

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

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

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

DateOfBirth: Date,

Timestamp: DateTime,

MyPhoneNumbers: List { Element: String { Encoding: UTF8, Storage: Variable }, Storage: Variable }

}

## 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.

### Lab02g – To-Many Bidirectional Reference Attributes

This lab 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 run2g**

### Lab02g - 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());

### Lab02h – Many-To-Many Bidirectional Relationships

This lab 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 run2h**

### Lab02h - 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: 12.8.0

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

DO> show schema;

{

CLASS Address

{

Street1: String { Encoding: UTF16 },

Street2: String { Encoding: UTF16 },

City: String { Encoding: UTF16 },

State: String { Encoding: UTF16 },

ZIP: String { Encoding: UTF16 },

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

},

CLASS Person

{

FirstName: String { Encoding: UTF16 },

LastName: String { Encoding: UTF16 },

MiddleInitial: String { Encoding: UTF16 },

Birthdate: Date,

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

}

}

## Map Attributes

ToDo

## Embedded Objects as Attributes

There are many situations where being able to embed one object as an attribute inside another objects can be very useful.

Consider an Address class with your standard attributes, Street1, Street2, City, State, and ZIPCode. It might be useful to create a GeoLocation class and then add an instance of that class as an attribute to the Address class.

In order accomplish this we will need to do two things: First we define our GeoLocation class using ClassBuilder and we will call “cbGeoLoc.setEmbeddable()” to make out GeoLocation class an embeddable object inside our Address object.

com.objy.data.ClassBuilder cBuilderGeoLoc = new com.objy.data.ClassBuilder("GeoLocation");

cBuilderGeoLoc.setEmbeddable();

cBuilderGeoLoc.addAttribute("Latitude",

new RealSpecificationBuilder(Storage.Real.B64)

.setEncoding(Encoding.Real.IEEE)

.build());

cBuilderGeoLoc.addAttribute("Longitude",

new RealSpecificationBuilder(Storage.Real.B64)

.setEncoding(Encoding.Real.IEEE)

.build());

// Actually build the the schema representation.

com.objy.data.Class cGeoLoc = cBuilderGeoLoc.build();

// Represent the new class into the federated database.

SchemaProvider.getDefaultPersistentProvider().represent(cGeoLoc);

// Activate the edits.

SchemaProvider.getDefaultPersistentProvider().activateEdits();

This creates or embeddable GeoLocation object.

Next, we create our Address class with its attributes including the GeoLocation attribute which is an *Instance* of a GeoLocation object. This is accomplished using the InstanceSpecificationBuilder class.

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");

cBuilderAddress.addAttribute("GeoLocation",

new InstanceSpecificationBuilder()

.setClass("GeoLocation")

.build());

// 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);

// Activate the edits.

SchemaProvider.getDefaultPersistentProvider().activateEdits();

### Lab02i – Creating Embeddable Classes

This lab 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 run2i**

### Lab02i – Examining the Results

Using DO Runner (objy DO) we can examine the Address and Person classes created in Lab02i.

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

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.8.0

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

DO> show schema;

{

CLASS GeoLocation EMBEDDABLE

{

Latitude: Real { Storage: B64 },

Longitude: Real { Storage: B64 }

},

CLASS Address

{

Street1: String { Encoding: UTF16 },

Street2: String { Encoding: UTF16 },

City: String { Encoding: UTF16 },

State: String { Encoding: UTF16 },

ZIP: String { Encoding: UTF16 },

GeoLocation: Object { Class: GeoLocation }

}

}

## Define Subclasses

ThingSpan allows you to define subclasses of existing classes as you would expect in any object-oriented environment. Let’s create a subclass of our Person objects call Employee.

In order to define a class as a subclass of an existing schema class definition, the superclass must either already exist or must at least have been “represent’ed” to the schema API, i.e. you can use ClassBuilder to build the superclass and call SchemaProvider.getDefaultPersistentProvider().represent(cBuilderDef) to make the superclass available for use as the base class for subclass definitions even though you haven’t completed the transaction. Base classes and multiple subclasses can be defined within a single TransactionScope.

Once the base class is defined, all you have do is to call “.setSuperclass(superclassname)” on the ClassBuilder being used to create the subclass.

// Use ClassBuilder to create the schema definition.

com.objy.data.ClassBuilder cBuilder

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

// Set the superclass.

cBuilder.setSuperclass("Person");

See Lab02j.java for an example of creating an Employee class as a subclass of a Person class.

There are a number of advanced topics that will be discussed later including how to specify different storage types, which is currently being done using “Variable”.

# Working with Objects

Much of your interaction with ThingSpan will be in working with objects, either creating them and putting them in the database or querying them and reading them from the database. This section will cover both areas.

## The ThingSpan Technology Stack

Your Java application sits on top of a stack of layers that make up ThingSpan. Understanding those layers will help you develop an understanding of the JavaULB API. The image below depicts the layers.

ThingSpan Kernel

ThingSpan JavaULB API

Java Application

Instance

ThingSpan Page Cache

JNI

C++

ThingSpan FD

The JavaULB API is little more than a set of support classes and a JNI wrapper that allow you to work with the ThingSpan kernel which is written in C++. With that in mind, the JavaULB API has been specifically designed to be very light weight and pushes nearly all of the actual work down in to the ThingSpan C++ kernel where the operations will run much faster.

## Working with com.objy.data.Instance Objects

The com.objy.data.Instance class is used to represent a ThingSpan persistent data object within a JavaULB application.

***NOTE****: ThingSpan does not support the storage of POJOs (Plain Old Java Objects) to a ThingSpan federation. Only objects represented as com.objy.data.Instance objects can be persisted to the database.*

As discussed earlier, a ThingSpan application will open a connection to a ThingSpan federated database and then create a Transaction or TransactionScope. These set the stage for interacting with ThingSpan objects.

com.objy.data.Instance is a JNI interface to an actual data object stored in the ThingSpan cache.

## Creating an Instance

Let’s create a new Instance object. It’s a two-step process.

First we need to know what kind of object we are going to create. We do this by instantiating an instance of com.objy.data.Class object using the static “lookupClass(…)” method on com.objy.data.Class. We need to pass in the name of the class we want to look up and this name is the name of the class *as it was defined in the database schema*. (See “Working with Schema”.)

From there we use the static method “createPersistent(…)” on the com.objy.data.Instance class to create an Instance of our “Person” class.

*Note: We have to use “com.objy.data.” before “Class” to differentiate it from java.lang.Class. We don’t have to prefix Instance with the “com.objy.data” because there is not collision on the naming.*

It sounds harder than it is…

// Lookup the Person class from the schema in the ThingSpan federation.

com.objy.data.Class cPerson

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

// Using the cPerson Class object, create a Person Instance.

Instance iPerson = Instance.createPersistent(cPerson);

// The ‘complete’ writes the data out to the database.

tx.complete();

Once we have our Person instance, iPerson, we can set its attributes if we need to and call tx.complete() which “commits” the transaction and ensures that our new Person instance is written to the database.

### Lab03a – Creating an Instance

Lab03a demonstrates the code shown above.

To run Lab03a from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab03 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab03 clean build run3a**

…

> Task :JavaULB\_Lab03:run3a

Running 'run3a'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab03/build/libs/

2019-03-26 11:05:20 INFO main Lab03a.java: Running Lab03a

2019-03-26 11:05:20 INFO main Lab03a.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot>

2019-03-26 11:05:20 INFO main Lab03a.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 11:05:20 INFO main Lab03a.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 11:05:20 INFO main Lab03a.java: Person class created in schema.

2019-03-26 11:05:22 INFO main Lab03a.java: iPerson OID: 3-3-1-4

2019-03-26 11:05:22 INFO main Lab03a.java: Person class created in schema.

2019-03-26 11:05:22 INFO main Lab03a.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

### Lab03a – Examining the Results

Using DO Runner (objy DO) we can examine the results.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot Lab03.boot

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

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

DO> from Person return \*;

{

Person

{

\_\_identifier\_\_:3-3-1-4,

FirstName:null,

LastName:null,

MiddleInitial:null

}

}

## Setting Attributes

After we have an Instance we need to set the values of the attributes on it. We will expand our example above and set the attributes.

### Setting String and Date Attributes

JavaULB allows us to access the value of an attribute of an Instance through a Variable object. We get the Variable object from the instance by calling iPerson.getAttributeValue(attributeName). Then call variable.set(value) to set that attribute to a value.

// We access the value of each attribute in the Instance using

// a variable that we 'associate' with each attribute.

Variable vFirstName = iPerson.getAttributeValue("FirstName");

vFirstName.set("John");

Variable vLastName = iPerson.getAttributeValue("LastName");

vLastName.set("Doe");

Variable vMiddleInitial = iPerson.getAttributeValue("MiddleInitial");

vMiddleInitial.set("Q");

// The complete writes the data out to the database.

tx.complete();

It should be noted here that it is not necessary to use a separate Variable object for each attribute. You can declare a single Variable object and reuse it for each field.

Variable var;

var = iPerson.getAttributeValue("FirstName");

var.set("John");

var = iPerson.getAttributeValue("LastName");

var.set("Doe");

var = iPerson.getAttributeValue("MiddleInitial");

var.set("Q");

The Lab03b demonstrate both techniques.

#### Lab03b – Setting Attributes

Lab03b demonstrates the code shown above.

To run Lab03b from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab03 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab03 clean build run3b**

> Task :JavaULB\_Lab03:run3b

Running 'run3b'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab03/build/libs/

2019-03-26 11:07:51 INFO main Lab03b.java: Running Lab03b

2019-03-26 11:07:51 INFO main Lab03b.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot>

2019-03-26 11:07:51 INFO main Lab03b.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 11:07:51 INFO main Lab03b.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 11:07:51 INFO main Lab03b.java: Person class created in schema.

2019-03-26 11:07:53 INFO main Lab03b.java: iPerson OID: 3-3-1-4

2019-03-26 11:07:53 INFO main Lab03b.java: Person class created in schema.

2019-03-26 11:07:53 INFO main Lab03b.java: iPerson OID: 3-3-1-8

2019-03-26 11:07:53 INFO main Lab03b.java: Person class created in schema.

2019-03-26 11:07:53 INFO main Lab03b.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

#### Lab03b – Examining the Results

Using DO Runner (objy DO) we can examine the results.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot Lab03.boot

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

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

DO> from Person return \*;

{

Person

{

\_\_identifier\_\_:3-3-1-4,

FirstName:'John',

LastName:'Doe',

MiddleInitial:'Q'

},

Person

{

\_\_identifier\_\_:3-3-1-8,

FirstName:'John',

LastName:'Doe',

MiddleInitial:'Q'

}

}

### Setting Numeric Attributes

The Variable .setValue(…) method takes an object parameter, therefore any numeric values will need to be cast to the object version of their data type, i.e. int gets cast to Integer, long to Long, and so on.

*Note: In the following code fragment the function “parseBinary(String, boolean) takes a String representation of a binary number and a binary flag indicating whether the number should be interpreted as signed or unsigned, and generates a Java long from the binary representation.*

This following code fragments were extracted from Lab03c.java.

Variable var;

String sB8 = "11000000";

long intB8 = parseBinary(sB8, SIGNED);

var = iND.getAttributeValue("MyIntB8\_Signed");

var.set((Long)intB8);

String uintSB16 = "1000000000000000";

long uintB16 = parseBinary(uintSB16, UNSIGNED);

var = iND.getAttributeValue("MyIntB16\_Unsigned");

var.set((Long)uintB16);

double avn1 = 602300000000000000000000.00;

var = iND.getAttributeValue("MyReal32\_IEEE");

var.set((Double)avn1);

#### Lab03c – Setting Numeric Attributes

Lab03c demonstrates the code shown above.

To run Lab03c from the command line, use the following command:

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab03 recreateFD**

PS D:\Root\Workspaces\JavaULBTraining> **.\gradlew.bat -p JavaULB\_Lab03 clean build run3c**

#### Lab03c – Examining the Results

Using DO Runner (objy DO) we can examine the results.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot .\Lab03.boot

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

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

DO> from NumbersDemo return \*;

{

NumbersDemo

{

\_\_identifier\_\_:3-3-1-13,

SimpleInteger:33,

MyIntB8\_Signed:-64,

MyIntB16\_Signed:16384,

MyIntB32\_Signed:1073741824,

MyIntB64\_Signed:4611686018427387904,

MyIntB8\_Unsigned:192,

MyIntB16\_Unsigned:32768,

MyIntB32\_Unsigned:2147483648,

MyIntB64\_Unsigned:9223372036854775808,

SimpleReal:3.14159,

MyReal32\_IEEE:6.02300e+23,

MyReal64\_IEEE:6.02312e+23

}

}

### Setting DATE and DATE\_TIME Attributes

ThingSpan supports creating schema class definitions that have DATE and DATE\_TIME attributes.

#### Setting DATE Attributes

Setting a DATE attribute requires that you create a com.objy.data.Date object and pass in the year, month, and day of month values. The com.objy.data.Date object can then be passed to the Variable .set(Object) method as shown below.

Variable vBirthdate = iPerson.getAttributeValue("DateOfBirth");

GregorianCalendar gCal = new GregorianCalendar();

gCal.setTime(dateOfBirth);

vBirthdate.set(new com.objy.data.Date(

gCal.get(Calendar.YEAR),

gCal.get(Calendar.MONTH),

gCal.get(Calendar.DAY\_OF\_MONTH)));

#### Setting DATE\_TIME Attributes

Setting a DATE\_TIME attribute requires that you create a com.objy.data.DateTime object, passing in a java.util.Date object to the constructor.

java.util.Date timestamp = new java.util.Date();

Variable vTimestamp = iPerson.getAttributeValue("Timestamp");

vTimestamp.set(new com.objy.data.DateTime(timestamp));

#### Lab03d – Setting DATE and DATE\_TIME Attributes

> Task :JavaULB\_Lab03:run3d

Running 'run3d'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab03/build/libs/

2019-03-26 16:55:50 INFO main Lab03d.java: Running Lab03d

2019-03-26 16:55:50 INFO main Lab03d.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot>

2019-03-26 16:55:50 INFO main Lab03d.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 16:55:50 INFO main Lab03d.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 16:55:50 INFO main Lab03d.java: Person class created in schema.

2019-03-26 16:55:52 INFO main Lab03d.java: iPerson OID: 3-3-1-4

2019-03-26 16:55:52 INFO main Lab03d.java: Person class created in schema.

2019-03-26 16:55:52 INFO main Lab03d.java: iPerson OID: 3-3-1-8

2019-03-26 16:55:52 INFO main Lab03d.java: Person class created in schema.

2019-03-26 16:55:52 INFO main Lab03d.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

#### Lab03d – Examining the Results

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot Lab03.boot

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

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

DO> from Person return \*;

{

Person

{

\_\_identifier\_\_:3-3-1-4,

FirstName:'John',

LastName:'Doe',

MiddleInitial:'Q',

DateOfBirth:2000-03-28,

Timestamp:2000-04-28 19:42:30

},

Person

{

\_\_identifier\_\_:3-3-1-8,

FirstName:'Mary',

LastName:'Smith',

MiddleInitial:'B',

DateOfBirth:1966-05-15,

Timestamp:1966-06-15 19:42:30

}

}

### Setting LIST Attributes

Setting LIST attributes is a common task especially when dealing with to-many relationships. Before we get to that, let’s just add some strings to a LIST attribute in a ThingSpan persistent object.

#### A List of Strings

As with setting the other attribute types, we have to get a Variable representation of the list attribute first. That Variable is shown below as vPhoneNumbers. Once we have that we get a ‘List’ representation of that Variable by calling “listValue()” on the Variable object. This gives us a com.objy.data.List object that we can interact with.

Now we need another Variable to represent the item being added to the list. We get this by simply instantiating a Variable object called vPN. We then iterate across our source data stored in the Java array phoneNumbers and for each phone number “pn” we set that value into the Varible vPN and add vPN to pnList.

We can reuse vPN but we should really call “.clear()” on it before we reuse it.

// Add the phone numbers to MyPhoneNumbers in the object.

Variable vPhoneNumbers = iPerson.getAttributeValue("MyPhoneNumbers");

com.objy.data.List pnList = vPhoneNumbers.listValue();

Variable vPN = new Variable();

for (String pn : phoneNumbers) {

vPN.set(pn);

pnList.add(vPN);

vPN.clear();

}

##### Lab03e – Setting LIST Attributes

> Task :JavaULB\_Lab03:run3e

Running 'run3e'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab03/build/libs/

2019-03-26 17:12:10 INFO main Lab03e.java: Running Lab03e

2019-03-26 17:12:10 INFO main Lab03e.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot>

2019-03-26 17:12:10 INFO main Lab03e.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 17:12:10 INFO main Lab03e.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-26 17:12:10 INFO main Lab03e.java: Person class created in schema.

2019-03-26 17:12:12 INFO main Lab03e.java: iPerson OID: 3-3-1-4

2019-03-26 17:12:12 INFO main Lab03e.java: Person class created in schema.

2019-03-26 17:12:12 INFO main Lab03e.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

##### Lab03e – Examining the Results

JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot Lab03.boot

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

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

DO> from Person return \*;

{

Person

{

\_\_identifier\_\_:3-3-1-4,

FirstName:'John',

LastName:'Doe',

MiddleInitial:'Q',

DateOfBirth:2000-03-28,

Timestamp:2000-04-28 19:58:50,

MyPhoneNumbers:LIST

{

'410-555-1234',

'410-555-9984'

}

}

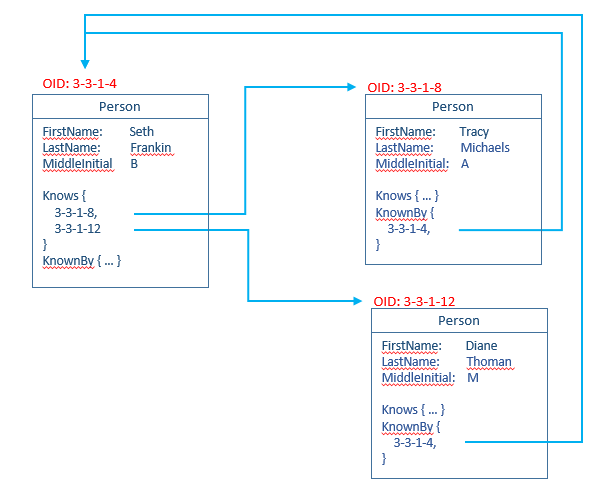
}

#### A List of References: To-Many

Relational databases use the concept of a JOIN to link objects together. In ThingSpan we simply create a reference or a list of references. In Section 6.5 we saw how to define to-one bidirectional relationship attributes and in Section 6.7 we saw how to define a to-many relationship attribute. Here we are going to populate a to-many relationship attribute.

We need to modify our Person class definition to have two new attributes. The first attribute is the outgoing attribute “Knows”. For a Person P, P.Knows will contain a list of OIDs of all of the Persons that P knows. The second attribute is the inverse attribute “KnownBy” and is the list of all of OIDs of Person objects that know P.

The structure is shown below.



Referring to Lab03f.java, let’s start by updating our schema creation function, createSchemaPerson(…).

We need to add the following lines of code to our createSchemaPerson(…) function:

// Create the "Knows" end of the bidirectional to-many reference.

cBuilder.addAttribute("Knows",

new ListSpecificationBuilder()

.setElementSpecification(

new ReferenceSpecificationBuilder()

.setReferencedClass("Person")

.setInverseAttribute("KnownBy")

.build())

.build());

// Create the "KnownBy" end of the bidirectional to-many reference.

cBuilder.addAttribute("KnownBy",

new ListSpecificationBuilder()

.setElementSpecification(

new ReferenceSpecificationBuilder()

.setReferencedClass("Person")

.setInverseAttribute("Knows")

.build())

.build());

Next, we will update our createPersonInstance(…) method. We are going to start by modifying the return type. We are going to update the method to return the OID of the Person object that was created. We’ll then use those OIDs to construct Knows/KnowBy relationships between some of the Person objects that we create.

Getting the OID from an Instance is easy enough:

// Using the cPerson Class object, create a Person Instance.

Instance iPerson = Instance.createPersistent(cPerson);

// Get and retain the OID of the Instance object we just created.

String personOID = iPerson.getObjectId().toString();

We simply return the value of String personOID to the calling method and then use that construct the Knows/KnownBy relationship.

The core of the “establishKnows(…)” method is straight forward. As with previous samples, the following code will be enclosed in a TransactionScope block.

// Lookup the Person associated with fromOID.

Instance iPersonFrom = Instance.lookup(ObjectId.fromString(fromOID));

// Lookup the Person associated with toOID.

Instance iPersonTo = Instance.lookup(ObjectId.fromString(toOID));

// Get the Knows list from iPersonFrom.

Variable vKnows = iPersonFrom.getAttributeValue("Knows");

com.objy.data.List knowsList = vKnows.listValue();

// vKnownEntry represents the Person that is "Known"

Reference refPersonTo = new Reference(iPersonTo);

Variable vPersonTo = new Variable(refPersonTo);

// Add the the vPersonTo entry to the knowsList.

knowsList.add(vPersonTo);

##### Lab03f – Adding References to a List Attribute

> Task :JavaULB\_Lab03:run3f

Running 'run3f'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab03/build/libs/

2019-03-29 15:59:39 INFO main Lab03f.java: Running Lab03f

2019-03-29 15:59:39 INFO main Lab03f.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot>

2019-03-29 15:59:39 INFO main Lab03f.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-29 15:59:40 INFO main Lab03f.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

2019-03-29 15:59:40 INFO main Lab03f.java: Person class created in schema.

2019-03-29 15:59:42 INFO main Lab03f.java: iPerson OID: 3-3-1-4

2019-03-29 15:59:42 INFO main Lab03f.java: Person class created in schema.

2019-03-29 15:59:42 INFO main Lab03f.java: iPerson OID: 3-3-1-8

2019-03-29 15:59:42 INFO main Lab03f.java: Person class created in schema.

2019-03-29 15:59:42 INFO main Lab03f.java: iPerson OID: 3-3-1-12

2019-03-29 15:59:43 INFO main Lab03f.java: Person class created in schema.

2019-03-29 15:59:43 INFO main Lab03f.java: Person class created in schema.

2019-03-29 15:59:43 INFO main Lab03f.java: Person class created in schema.

2019-03-29 15:59:43 INFO main Lab03f.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs\Lab03.boot

##### Lab03f – Examining the Results

Using DO Runner you can examine the results.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot Lab03.boot

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

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

DO> from ooObj return \*;

{

Person

{

\_\_identifier\_\_:3-3-1-4,

FirstName:'Seth',

LastName:'Franklin',

This is the first Person that was created and has a “Knows” list that points to two other objects.

MiddleInitial:'B',

DateOfBirth:0001-01-01,

Knows:LIST

{

3-3-1-8,

3-3-1-12

},

KnownBy:null

},

Person

{

\_\_identifier\_\_:3-3-1-8,

FirstName:'Tracy',

LastName:'Michaels',

This is the second Person that was created and has a “KnownBy” list that points to Person 3-3-1-4.

MiddleInitial:'A',

DateOfBirth:0001-01-01,

Knows:null,

KnownBy:LIST

{

3-3-1-4

}

},

Person

{

\_\_identifier\_\_:3-3-1-12,

FirstName:'Diane',

LastName:'Thoman',

This is the third Person that was created and has a “KnownBy” list that points to Person 3-3-1-4.

MiddleInitial:'M',

DateOfBirth:0001-01-01,

Knows:null,

KnownBy:LIST

{

3-3-1-4

}

}

}

So now we should stop and make an observation about the code and the resulting relationships. The code that created the relationships only added entries to the “Knows” list. We didn’t write any code to set the value of the inverse relationship in the “KnownBy” list. Because we defined the relationship “Knows” and its inverse “KnownBy” relationship when we created the schema definition, ThingSpan automatically fills in the inverse value when we set one end of the relationship. If we add an entry to the “Knows” list, ThingSpan will fill in the inverse value in the “KnownBy” list and if we add an entry to a “KnownBy” list ThingSpan will fill in the inverse value in the “Knows” list.

This automatic updating works for both adding and removing entries from a defined relationship list. When removing a value from a list attribute that represents one end of a relationship, ThingSpan will remove the corresponding entry from the inverse object.

ThingSpan takes this one step further. If one of the objects is deleted ThingSpan will updates all of the inverse objects and remove the deleted object’s OID from all inverse lists that point back to the object being deleted. This behavior maintains the referential integrity of the data stored in the ThingSpan repository.

##### Lab03g - Removing References from a List Attribute

To remove a specific entry from a List of References attribute, we need to iterate over the list, get the Variable for each entry and then get the Reference from the Variable. Once we have the reference we can call “reference.getObjectId().toString()” to get the OID and we can compare the java.util.String representations of the OIDs to see if they are equal. If they are equal we can call knowsList.remove(i) to remove that entry.

// Lookup the Person associated with fromOID.

Instance iPersonFrom = Instance.lookup(ObjectId.fromString(fromOID));

// Lookup the Person associated with toOID.

Instance iPersonTo = Instance.lookup(ObjectId.fromString(toOID));

// Get the Knows list from iPersonFrom.

Variable vKnows = iPersonFrom.getAttributeValue("Knows");

com.objy.data.List knowsList = vKnows.listValue();

// vKnownEntry represents the Person that is "Known"

Reference rPersonTo = new Reference(iPersonTo);

Variable vKnowsEntry;

boolean found = false;

for (int i = 0; i < knowsList.size(); i++) {

vKnowsEntry = knowsList.get(i);

Reference rPersonToCheck = vKnowsEntry.referenceValue();

// Check to see if the referenced objects are the same by OID.

if(rPersonToCheck

.getObjectId()

.toString()

.equals(rPersonTo.getObjectId().toString())) {

logger.info("Removing " + rPersonToCheck.getObjectId().toString()

+ " from " + fromOID);

knowsList.remove(i);

found = true;

break;

}

vKnowsEntry.clear();

}

### Working with Map Attributes

### Working with Embedded Objects

Section 6.8 discussed how to create schema for embedded classes as well as how to create schema for class definitions that contained attributes that were instances of embeddable classes. Here we will extend that example and actually create instance objects that contain embedded objects.

com.objy.data.Class cAddress = com.objy.data.Class.lookupClass("Address");

Instance iAddress = Instance.createPersistent(cAddress);

Variable vStreet1 = iAddress.getAttributeValue("Street1");

vStreet1.set(street1);

Variable vStreet2 = iAddress.getAttributeValue("Street2");

vStreet2.set(street2);

Variable vCity = iAddress.getAttributeValue("City");

vCity.set(city);

Variable vState = iAddress.getAttributeValue("State");

vState.set(state);

Variable vZIP = iAddress.getAttributeValue("ZIP");

vZIP.set(zip);

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

// Set the attributes of the embedded GeoLocation object

// First, get the attribute as a Variable.

Variable vGeoLoc = iAddress.getAttributeValue("GeoLocation");

// Second, get the Instance value of that Variable.

Instance iGeoLoc = vGeoLoc.instanceValue();

// Set the attributes of the GeoLocation instance object.

Variable vLat = iGeoLoc.getAttributeValue("Latitude");

vLat.set(lat);

Variable vLon = iGeoLoc.getAttributeValue("Longitude");

vLon.set(lon);

#### Lab03h – Creating an Instance Object with an Embedded Object Attribute

This is the command to recreate the federation and clean, build, and run the Lab03h sample:

PS D:\Root\Workspaces\JavaULBTraining> .\gradlew.bat -p JavaULB\_Lab03 recreateFD clean build run3h

#### Lab03h – Examining the Results

Use DO Runner to examine the results.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab03\data\dbs> objy DO -boot Lab03.boot

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.8.0

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

DO> from Address return \*;

{

Address

{

\_\_identifier\_\_:3-3-1-4,

Street1:'407 Main St.',

Street2:'Apartment 3',

City:'Mt. Pleasant',

State:'MI',

ZIP:'48858',

GeoLocation:GeoLocation

{

Latitude:43.6012,

Longitude:-84.7764

}

}

}

# Object Lookup and Queries

At this point you should be comfortable with com.objy.data.Class, Instance, Attribute, and Variable. We are now going to put these to use in looking up objects and running queries on the database.

## Object Lookup by OID

The simplest way to retrieve an object from a ThingSpan federation is to already know its OID and to simply use the OID to look up the object with that OID. We don’t expect users to provide us with the OIDs of the objects they want to find but in your own code it is very common to be working with OIDs and have to retrieve an object using the OID.

Let’s create a Person object.

// Lookup the Person class from the schema in the ThingSpan federation.

com.objy.data.Class cPerson

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

// Using the cPerson Class object, create a Person Instance.

Instance iPerson = Instance.createPersistent(cPerson);

oid = iPerson.getObjectId().toString();

The last line of code in the sample above calls the “.getObjectId()” method to get an ObjectId object and then calls “.toString()” on that. ~~A quick note: The ObjectId object is only valid within the TransactionScope so we can’t simply return it; we need to convert it to a String which is remains valid outside of the TransactionScope.~~ We can later use that String representation of the OID to look up the object we’ve just created.

At this point it is appropriate to say a few words about OIDs. A ThingSpan OID in String-form has the form: “#D-C-P-S”, where:

* “D” is a number representing the database where the object can be found.
* “C” represents the container within the database.
* “P” represents the page in the container and
* “S” represents the slot that points to the object data on page.

In an actual OID each of these will have numeric values, so an OID might be “#2-4-3-17” which refers to database 2, container 4, page 3, and slot 17. The OID represents the unique “logical” address of an object. It is a “logical” address because each of these values is used to look up the related physical component to actually find the object.

Now that we have an OID, we can look up the related object. To lookup the object we will use the static method, Instance.lookup(ObjectId) which will give us an Instance object representation that we can read or update attributes on.

// Using the cPerson Class object, create a Person Instance.

Instance iPerson = Instance.lookup(ObjectId.fromString(oid));

logger.info("iPerson OID: " + iPerson.getObjectId().toString());

// We access the value of each attribute in the Instance using

// a Variable that we 'associate' with each attribute.

Variable vFirstName = iPerson.getAttributeValue("FirstName");

logger.info(oid + " Person.FirstName: "

+ vFirstName.stringValue());

Variable vMiddleInitial = iPerson.getAttributeValue("MiddleInitial");

logger.info(oid + " Person.MiddleInitial: "

+ vMiddleInitial.stringValue());

Variable vLastName = iPerson.getAttributeValue("LastName");

logger.info(oid + " Person.LastName: "

+ vLastName.stringValue());

The highlighted line above does two things. First, it creates an ObjectId from the String value oid and second, it performs the lookup on that ObjectId. The result is that we end up with an Instance object representation of the object with the given oid.

The remaining lines of code above show how to extract known attributes from the Instance object.

This code is presented in Lab04a.

### Lab04a – Looking Up an Object Using the OID

This is the command to recreate the federation and clean, build, and run the Lab03h sample:

PS D:\Root\Workspaces\JavaULBTraining> .\gradlew.bat -p JavaULB\_Lab04 recreateFD clean build run4a

> Task :JavaULB\_Lab04:run4a

Running 'run2a'

LIB\_DIR = D:/Root/Workspaces/JavaULBTraining/JavaULB\_Lab04/build/libs/

log4j:ERROR Could not find value for key log4j.appender.FILE

log4j:ERROR Could not instantiate appender named "FILE".

2019-04-03 16:18:18 INFO main Lab04a.java: Running Lab04a

2019-04-03 16:18:18 INFO main Lab04a.java: bootFile: <D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab04\data\dbs\Lab04.boot>

2019-04-03 16:18:18 INFO main Lab04a.java: Boot file is valid: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab04\data\dbs\Lab04.boot

2019-04-03 16:18:18 INFO main Lab04a.java: Connected to ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab04\data\dbs\Lab04.boot

2019-04-03 16:18:19 INFO main Lab04a.java: Person class created in schema.

2019-04-03 16:18:20 INFO main Lab04a.java: iPerson OID: 3-3-1-4

2019-04-03 16:18:20 INFO main Lab04a.java: Person class created in schema.

2019-04-03 16:18:20 INFO main Lab04a.java: iPerson OID: 3-3-1-4

2019-04-03 16:18:20 INFO main Lab04a.java: 3-3-1-4 Person.FirstName: John

2019-04-03 16:18:20 INFO main Lab04a.java: 3-3-1-4 Person.MiddleInitial: Q

2019-04-03 16:18:20 INFO main Lab04a.java: 3-3-1-4 Person.LastName: Doe

2019-04-03 16:18:20 INFO main Lab04a.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab04\data\dbs\Lab04.boot

### Lab04a – Examining the Results

Use DO Runner to examine the results.

PS D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab04\data\dbs> objy DO -boot Lab04.boot

Objectivity/DB (TM) Execute DO Statement(s), Version: 12.9 develop Apr 2 2019

Copyright (c) Objectivity, Inc 2012, 2019. All rights reserved.

DO> from Person return \*;

{

Person

{

\_\_identifier\_\_:3-3-1-4,

FirstName:'John',

LastName:'Doe',

MiddleInitial:'Q'

}

}

This is how you do a simple lookup using the OID.

## Queries

ThingSpan provides the “Declarative Object” or DO query language that we will introduce in this section. DO is a cross between SQL and Cypher and allows you to execute many different kinds of queries with many different kinds of result structures. We are going to start off with simple queries and increase the complexity as we move forward.

In order to run a DO query in JavaULB you will need to use the com.objy.data.Statement class. The Statement class has a “.execute(String)” method that takes a properly formatted DO query as a String and executes it returning a Variable that contains the results of your query.

Here are the core steps in running a query:

1. We have to get a Language object that represents the DO query language.

Language doLang = LanguageRegistry.lookupLanguage("DO");

1. Create our Statement with the Language object and the String representing the text of the query.

Statement statement = new Statement(doLang, doQuery);

1. Call .execute on our statement and get back a Variable that represents the results of the query.

Variable vStatementExecute = statement.execute();

1. Create an Iterator on the vStatementExecute.sequenceValue().

java.util.Iterator<Variable> it

= vStatementExecute.sequenceValue().iterator();

1. Iterate over the result objects.

while (it.hasNext()) {

// Get the next result from the iterator as a Variable.

Variable vResult = it.next();

// Get the Instance representation of the Variable.

Instance iPerson = vResult.instanceValue();

// Do something with the Instance.

displayPersonInstance(iPerson);

}

### Lab04b – Running a DO Query in JavaULB

Lab04b.java puts the above steps together into a program that demonstrates some simple DO queries.

PS D:\Root\Workspaces\JavaULBTraining> .\gradlew.bat -p JavaULB\_Lab04 recreateFD clean build run4b

> Task :JavaULB\_Lab04:run4b

Running 'run4b'

***Not all output is shown…***

2019-04-04 18:43:42 INFO main Lab04b.java: Persons created: 900

========================================================

QUERY: FROM Person return \*

--------------------------------------------------------

FirstName LastName MiddleName

STRING STRING STRING

--------------- --------------- ---------------

Ivonne Amo Valentina

Dion Rosenbeck Lashaun

***Not all output is shown…***

Wm Solinski Aubrey

Vernetta Hajduk Pennie

Kellie Hodor Anthony

Darren Vecino Dannie

--------------------------------------------------------

Result Count: 1000

========================================================

========================================================

QUERY: FROM Person WHERE LastName =~ 'M.\*' return \*

--------------------------------------------------------

FirstName LastName MiddleName

STRING STRING STRING

--------------- --------------- ---------------

Dona Manthei Jesusa

Dee Milo Theodora

***Not all output is shown…***

Eleni Manista Carletta

Chang Mcgurr Denver

Mackenzie Maglione Aura

--------------------------------------------------------

Result Count: 87

========================================================

========================================================

QUERY: FROM Person WHERE LastName =~ 'M.\*' AND FirstName =~ 'T.\*' return \*

--------------------------------------------------------

FirstName LastName MiddleName

STRING STRING STRING

--------------- --------------- ---------------

Tanesha Maffit Lakisha

Tennie Montero Wendolyn

Thurman Mcguirl Lenard

Tomasa Morisey Lovie

Taina Mcgee Margeret

Tracy Maddin Odell

--------------------------------------------------------

Result Count: 6

========================================================

2019-04-04 18:43:42 INFO main Lab04b.java: Disconnected from ThingSpan federation: D:\Root\Workspaces\JavaULBTraining\JavaULB\_Lab04\data\dbs\Lab04.boot

The core of Lab04b.java is the query(String doQuery) method. It is called three times with three different queries.

String doQuery1 = "FROM Person return \*";

query(doQuery1);

String doQuery2 = "FROM Person WHERE LastName =~ 'M.\*' return \*";

query(doQuery2);

String doQuery3

= "FROM Person "

+ "WHERE LastName =~ 'M.\*' "

+ "AND FirstName =~ 'T.\*' "

+ "return \*";

query(doQuery3);

Let’s discuss these queries for just a minute.

The first query returns all Person objects in the database.

**A Note about Object Orientation…**

ThingSpan is an object-oriented platform so this query will return anything that is either exactly a Person object as well as any object that is a type derived from Person. If your federation has a schema definition for Employee where Employee inherits from Person, all Employee objects will be returned by any query that returns all Person objects because Employee objects **are** Person objects.

The second query is exactly the same as the first query except that it adds a predicate: We only want Person objects (and sub-type objects) where the LastName field starts with the letter ‘M’ followed by anything. The “=~” is the pattern-match operator and ‘M.\*” is the pattern we want to match. If you look at the first have of the results, shown above, you can see the tail-end of the results that match this query. The LastName attributes all begin with “M” and the FirstName attributes begin with a variety of letters.

The third query adds an additional qualifier to the second query. Here, we are looking for Person objects whose LastName begins with “M” and whose FirstName begins with “T” and we found four objects that match that criteria.

### An Introduction to Projections

A projection is a temporary ad-hoc object containing the attributes specified by the RETURN clause of the query. The asterisk “\*” is a special case when used in the RETURN clause as it indicates to include all of the attributes from the result object.

In each of the queries above we always returned all of the attributes in each Person object that matched our predicate, that is to say that we didn’t just return the FirstName or the LastName attribute of the object, we returned the entire object.

#### A Simple Projection

What if we only wanted the LastName? The query is easy enough, we simply replace the “\*” with “LastName” as shown below:

FROM Person return Lastname;

**Note**: The semicolon is only used in DO Runner and not when issuing the query in JavaULB.

Here, we are not going to get back an entire Person, we are going to get back a **projection**. In our case we will get back a single projection for each Person object and each projection object contains just a LastName attribute.

If we run the query in DO Runner we see the projection representation:

DO> from Person return LastName;

{

Person\_Projection

{

\_\_identifier\_\_:3-3-1-4,

LastName:'Amo'

},

Person\_Projection

{

\_\_identifier\_\_:3-3-1-8,

LastName:'Rosenbeck'

},

In JavaULB we will still iterate over the results, getting a Variable for each result (the projection) and from each of those Variables we can get an Instance. It’s just that the Instance now represents an ad-hoc projection object that only contains the attributes that were specified in the query, i.e. the LastName attribute.

There are two ways to approach results processing. The first way requires us to have knowledge of the attributes that are going to be getting back in the projection Instance so we know exactly what to pull out. The second way is to interrogate the projection Instance to see what attributes it contains.

Let’s look at how to dynamically process an Instance.

Lab04c.java contains a method that dynamically processes the available attribute.

private void displayInstance(Instance ix) {

com.objy.data.Class cx = ix.getClass(true);

StringBuilder sb = new StringBuilder();

for (int i = 0; i < cx.getNumberOfAttributes(); i++) {

Attribute at = cx.getAttribute(i);

Variable v = ix.getAttributeValue(at.getName());

// We have to get the logical type of the attribute.

LogicalType lt = at.getAttributeValueSpecification()

.getFacet()

.getLogicalType();

// LogicalType contain an enumeration for the

// different types.

switch (lt) {

case STRING:

sb.append(String.format("%-15s ",

v.stringValue()));

break;

case REFERENCE:

sb.append(String.format("%-15s ",

v.referenceValue().getObjectId().toString()));

break;

default:

sb.append(String.format("%-15s ",

"Not Handled"));

}

}

print(sb.toString());

}

This method is greatly oversimplified because it only handles two types: STRING and REFERENCE. You will need to add additional cases to handle each of the attribute types you need to process in the objects you query. The class LogicalType contains the enumeration of all of the types that are available in the database.

#### OIDs in Projections

When you run a query using DO Runner, you get an attribute “\_\_identifier\_\_” whose value is the OID of object from which the other attributes are being returned. When you run the same query in JavaULB you don’t get a convenient attribute that contains the OID. That’s okay, it’s easy enough to modify our query to get the OID.

The following query will return a projection that contains two attributes: LastName and OID:

FROM Person WHERE LastName =~ “M.\*” RETURN LastName, $$ID as OID

Here, we are using the special $$ID which represents the OID of the object and we are using the “as” clause to store the $$ID value in the resulting projection with the attribute name “OID”.

Now we can pull the attributes out of the resulting projection Variable.

LastName is a STRING so you can use the .stringValue() method to extract it from the projection:

String s = variable.stringValue()

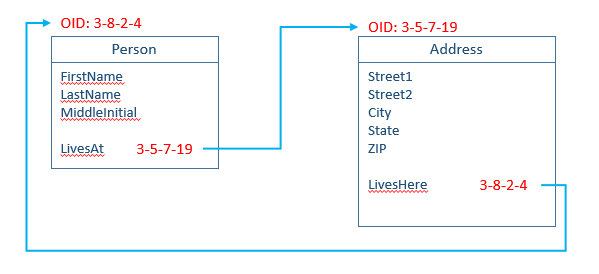
The OID value is of type REFERENCE, so to extract the OID from the projection you can use:

String oidStr = variable.referenceValue().getObjectId().toString()

Here, we get the reference value from the Variable and then convert it to an ObjectID and then we convert that to a String. The resulting String representation of the OID will be a Java String of the form: “3-3-4-171”.

#### Using Dot Notation to Traverse Relationship

The DO query language allows us to use a dot notation to traverse relationships in our objects. Consider a Person class that contains a reference relationship “LivesAt” to objects of type Address and the Address class defines a list of references “LivesHere” to Person objects.



Now what we want to do is to create a single projection that includes elements from both the Person objects and the related Address objects. It’s actually easy. If we want the FirstName and LastName from Person as well as the City and State that each person lives in, all we have to do is use the dot notation on the LivesAt attribute in Person. We can say LivesAt.City and LivesAt.State to get those attributes on the referenced object.

Example:

FROM Person

RETURN FirstName, LastName,

LivesAt.City as City,

LivesAt.State as State;

In DO Runner, the results will be:

DO> FROM Person RETURN FirstName, LastName, LivesAt.City as City, LivesAt.State as State;

{

Person\_Projection

{

\_\_identifier\_\_:3-3-1-4,

FirstName:'Sherley',

LastName:'Crego',

City:'Plymouth',

State:'CA'

},

Person\_Projection

{

\_\_identifier\_\_:3-3-1-14,

FirstName:'Patrick',

LastName:'Isam',

City:'Putnam',

State:'OK'

},

#### Lab04d – Using Dot Notation to Traverse Relationship

Lab04d demonstrates using the dot notation in a DO query using JavaULB.

PS D:\Root\Workspaces\JavaULBTraining> .\gradlew.bat -p JavaULB\_Lab04 clean build recreatefd run4d

> Task :JavaULB\_Lab04:run4d

Running 'run4d'

2019-04-08 10:40:41 INFO main Lab04d.java: Persons created: 900

***Not all output is shown…***

========================================================

QUERY: FROM Person WHERE LastName =~ 'M.\*' AND FirstName =~ 'T.\*' RETURN LastName, LivesAt.City as City, LivesAt.State as State

--------------------------------------------------------

LastName City State

STRING STRING STRING

--------------- --------------- ---------------

Miene Thorne Bay AK

Mehling Nauvoo IL

Mestemacher Spruce MI

Mccune Whiting VT

Makino Monroe LA

--------------------------------------------------------

Result Count: 5

========================================================

========================================================

QUERY: FROM Person WHERE LastName =~ 'M.\*' AND FirstName =~ 'T.\*' RETURN LastName, FirstName, LivesAt.City as City, LivesAt.State as State

--------------------------------------------------------

LastName FirstName City State

STRING STRING STRING STRING

--------------- --------------- --------------- ---------------

Miene Toccara Thorne Bay AK

Mehling Tabitha Nauvoo IL

Mestemacher Teofila Spruce MI

Mccune Trinidad Whiting VT

Makino Treena Monroe LA

--------------------------------------------------------

Result Count: 5

========================================================

========================================================

QUERY: FROM Person WHERE LastName =~ 'M.\*' AND FirstName =~ 'T.\*' RETURN $$ID as oid, LastName, FirstName,

LivesAt.City as City, LivesAt.State as State

--------------------------------------------------------

oid LastName FirstName City State

REFERENCE STRING STRING STRING STRING

--------------- --------------- --------------- --------------- ---------------

3-3-1-4 Miene Toccara Thorne Bay AK

3-3-2-19 Mehling Tabitha Nauvoo IL

3-3-7-87 Mestemacher Teofila Spruce MI

3-3-10-92 Mccune Trinidad Whiting VT

3-3-41-66 Makino Treena Monroe LA

--------------------------------------------------------

Result Count: 5

========================================================

The dot notation can also be used in developing WHERE clauses.

FROM Person WHERE LivesAt.City =~ '^Sp.\*'

RETURN $$ID as oid, LastName, FirstName,

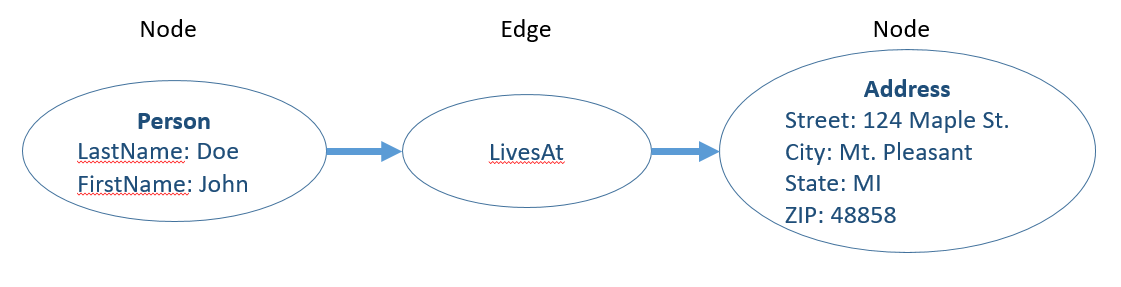
LivesAt.City as City,

LivesAt.State as State;

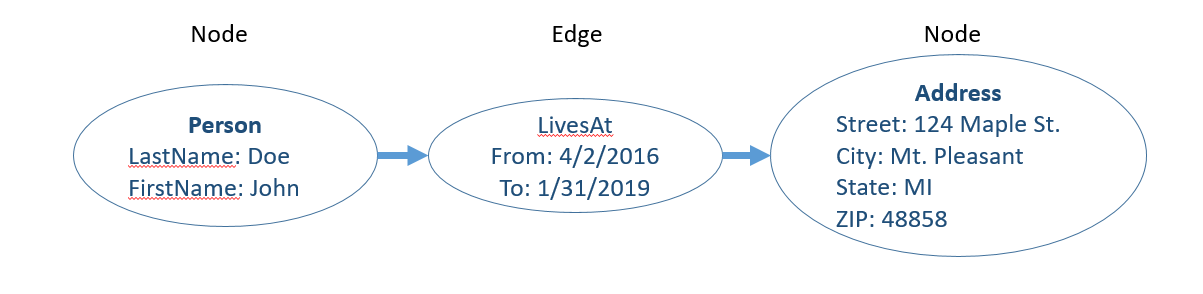
# Graphs

A graph is formally defined as a set containing a set of vertices or nodes and a set of edges. ThingSpan supports this definition by defining both nodes and edges as objects and using reference attributes to connect node objects to one or more edge objects and each edge object is connected to exactly two node objects. (There are very abstract cases where edge objects can be connected to more than two node objects but we will not cover those cases here.)

We can represent a graph as a simple diagram.



In previous labs, when we were working with the concept of LivesAt we created a LivesAt attribute in the Person class. That was fine but that representation did not allow us to associate attributes with the concept of LivesAt. By adjusting our diagram slightly and introducing the concept of a LivesAt edge object, we can now put attributes on that edge object. We can now capture when the person lived at an address by putting From and To date attributes on the LivesAt edge class definition.



This gives us much greater flexibility and fidelity. Let’s look at how to implement this type of graph structure.

## Creating Schema for a Graph

When defining the schema to represent a graph we will define our classes exactly as we have in the past except that on the Person object definition we will define an attribute that holds the outgoing reference to the LivesAt edge and on the Address object we will define an attribute that holds the outgoing reference to the LivesAt edge.

# Session Logging

SessionLogging.setLoggingOptions(SessionLogging.LogAll,

"D:/Root/temp");