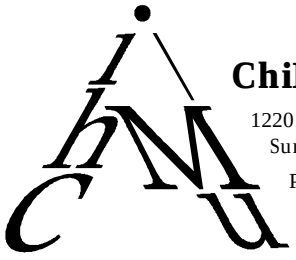
Learning FORM

R1.0



ChiMu Corporation

1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089

Phone: 408 734-9068
Email: info@chimu.com

www.chimu.com

Learning FORM

Table of Contents

Example Listing

1 Overview

This document explains what FORM is and how to use it. It is the main document for understanding the FORM framework and it also provides many examples of how to use FORM. These examples provide one of the integrating threads to Learning FORM. Another integrating thread is the growth from basic and general FORM concepts to more advanced FORM capabilities. When you have finished with this document you will have a comprehensive understanding of FORM's concepts, how to use the FORM API to accomplish specific results, and an overall approach for structuring programs that use FORM.

1.1 Requirements

Learning FORM assumes that you are familiar with Java, Relational Databases, and with the issues and concepts for object to relational mapping. The material in the Supplemental Reading section below or in the References section can help you become more familiar with these concepts and tools.

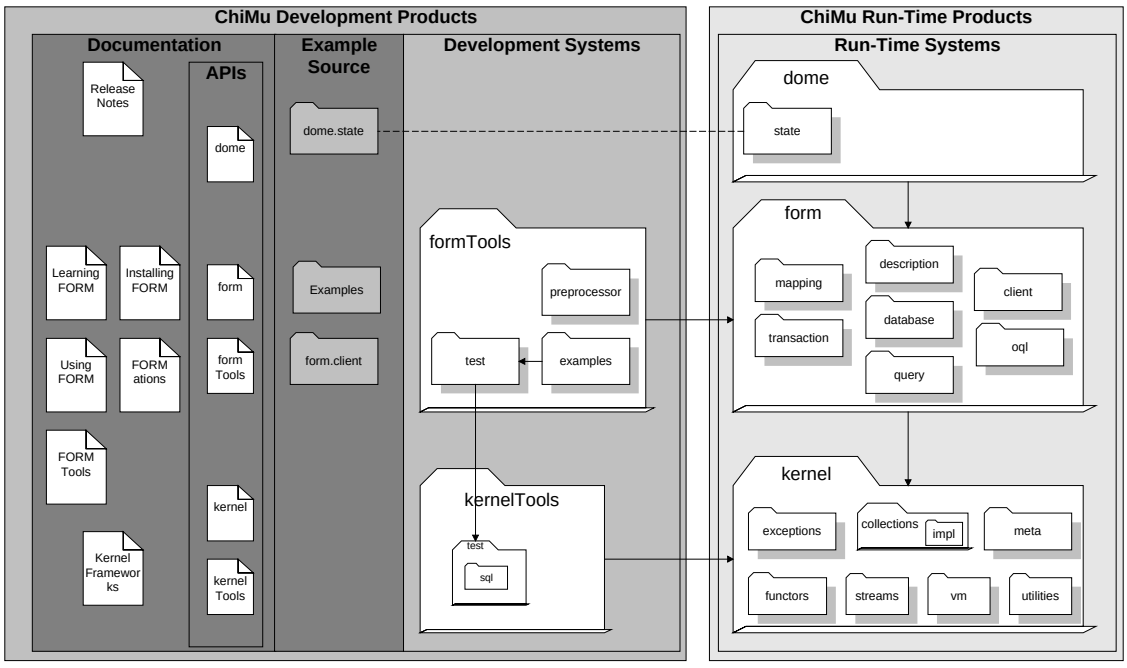
You may also want to have FORM installed and the FORM examples available. Installing FORM is described in *Getting Started with FORM*.

1.2 FORM Components

FORM has a large amount of functionality and is organized into several major subsystems:

form	The outer package for FORM and the Orm class
form.mapping	ObjectMapper functionality
form.database	Database layer and JDBC enhancements
form.query	QueryDescriptions
form.oql	OQL
form.transaction	TransactionCoordinators
form.description	Database and ObjectMapper descriptions
form.client	Example abstract classes for client DomainObjects

Each of these subsystems will be covered in the upcoming chapters as you learn more about FORM functionality. FORM also relies on the subsystems within ChiMu's Kernel frameworks both for implementation and for FORMs API. The Kernel frameworks are discussed in [Kernel].



1.3 Supplemental Reading

The following documents are supplemental to this Learning FORM document. They can either be read prior to this document or concurrently as topic areas come up.

Introduction to FORM provides an overview of FORM capabilities, concepts, and terms.

ChiMu Java Development Standards describes the standards for ChiMu Java development, which includes all aspects of FORM. Reading the standards document can help you to more rapidly understand the FORM API and the examples given within this document. Many of the design approaches (e.g., separating interfaces from implementation classes) are assumed and not explained within the current document.

Foundations of Object-Relational Mapping describes general concepts and approaches for object-relational mapping. Although more general than FORM, these concepts can be helpful for getting a broad picture of what FORM is trying to accomplish and some of its possible approaches.

FORM Tools describes the Database testing and other FORM Tools.

FORM API Documentation is provided in Javadoc format.

2 Introduction

FORM is a framework for Object-Relational mapping. Object-Relational mapping is the process of transforming information between an object model and a relational storage model. FORM is designed such that high-quality object models written in Java can be stored using high-quality relational models accessible through JDBC. FORM models your Domain Object Model, your Relational Scheme, and the mapping between them. It provides great flexibility in how object models are transformed into relational models, so the optimal approach for your specific needs can be chosen. FORM's mapping descriptions also provide a loose coupling between the two models so changes within them are better confined.

FORM is primarily a very flexible mapping engine. This means many different approaches are possible when using FORM. This document will assume a specific approach to using FORM for consistency and because that approach works very well for most applications. Other approaches are also possible and may solve particular applications needs better. These will be discussed in other documents.

2.1 Chapter Overview

This chapter will introduce FORM through the simplest example: retrieving a single domain object from the database. We will first describe the model for the example and then show a test "application" for that domain model. This will lead into the discussion of FORM's concepts, how to configure FORM, and how to run FORM.

2.2 Domain Model-1

The domain model for this example has a single Type, 'Person', which is implemented by the class 'PersonClass'. We also have a single database table named 'Person'.

Person

Person is a type, which is represented in Java as an interface.

```
public interface Person {
    //(P)****************** Displaying *************************

        /**
         * Provide a simple string representation of a person
         */
    public String toString();

        /**
         * Give some info about the person
         */
    public String info();

    //(P)******************** Asking *************************

    public String name();
    public String email();
    public int height();

    //(P)******************* Altering ************************

    public void setName(String name);
    public void setEmail(String email);
    public void setHeight(int height);
}
```

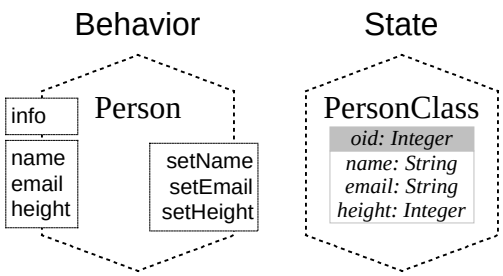
PersonClass

PersonClass implements the Person interface: this is PersonClass's primary responsibility as a Domain Class. PersonClass will also implement the responsibilities required to work with FORM and to help save Person objects to the database. The code for PersonClass is given in the appendix.

Object Model

The object model for Person is very simple. In terms of behavior, there are several asking and displaying messages and a few messages that change the state of a Person object.

For storage of state, there are three instance variables corresponding to basic attributes. There is also an object identifier to keep track of which database object this object replicates (this will be discussed shortly).



Database Scheme-1

The database schema for storing people is also very simple. The Person table has 4 columns with the column 'id' as the primary key column. Some of the column naming is different from the object model attribute names (frequently, database naming standards are different from object model naming standards). FORM can easily handle these differences.

Person

ID : int
Person_Name : varchar
Email : varchar
Height : int

2.3 Example: PersonRetrieval_1

Our first example application illustrates using FORM to retrieve a single Person.

```
public class Ex_PersonRetrieval_1 extends FormExampleAbsClass {

    public void run (Connection jdbcConnection) {
        //-------Configuration--------
        Orm orm = FormPack.newOrm();
        DatabaseConnection dbConnection =
                FormPack.newDatabaseConnection(jdbcConnection);

            orm.addInfoClass_withDb(
                    PersonClass_FormInfo.class, dbConnection);
        orm.doneSetup();

        //----------Running-----------
        ObjectRetriever personRetriever = orm.retrieverNamed("Person");

        Person person = (Person) personRetriever.findAny();
        outputStream.println(person);
        outputStream.println(person.info());
    }
}
```

The example contains two sections to handle the two different aspects for using FORM. The first section configures FORM to describe the Object-Relational mapping; and the second section runs FORM to retrieve and display the domain objects.

Running the Example

All examples subclass from FormExampleAbsClass, which allows them to be run easily. For information on setting up the example environment for your system and running test programs see the FORM Installation document and the FORM Tools document. The following will assume you are using a JdbcOdbc driver and that the database name for the first domain model is ExampleScheme1. Then you can run the first example through the command line with:

```
> FormDatabaseTester ExampleScheme1 scheme1.Ex1_PersonRetrieval_1
```

Which should print:

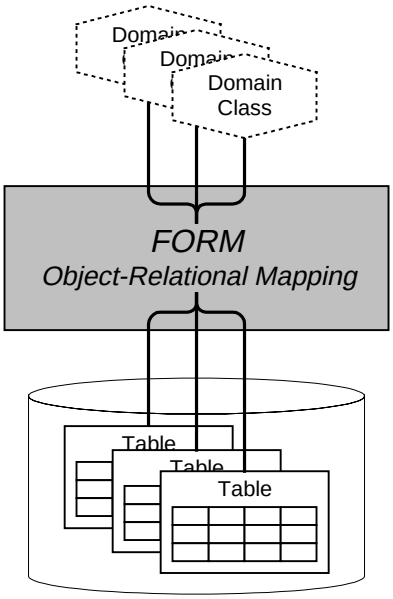
```
Test: ---- Ex1_PersonRetrieval_1 ----
<Person#28 Harry Houdini>
Harry Houdini is 66 inches tall and can be contacted at hhoudini@org.com
```

The first line is execution information from the DatabaseRunner. The second line shows the "developer's information string"[1] and identifies the person object. The third line shows information that is more informative and user-friendly.

You can also run all the examples through the FormExampleRunner GUI application.

2.4 Configuring FORM

Configuring FORM means describing how domain objects are translated to and from the database scheme. This must be done before you can use FORM to retrieve objects or store objects to the database.



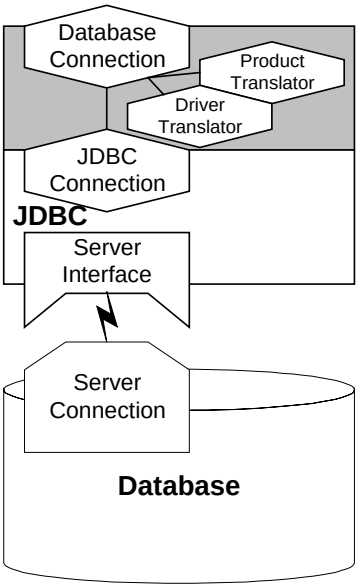
For FORM to do object-relational mapping it will need to understand the database, the domain model, and the mapping between them. FORM learns about the database scheme by asking the database. FORM learns about the domain model by information provided by the domain classes. Finally, FORM learns about the mapping by you telling it how you would like the translation done. FORM provides a great amount of flexibility in mapping and makes common mappings easy to build and maintain.

To describe the object-relational mapping we need two root objects: a DatabaseConnection for describing the database and an Orm for describing the mapping. These are both created from FormPack, the outermost package library for FORM.

DatabaseConnection

A **DatabaseConnection** is a connection to a particular database. This is similar to a JDBC connection except a DatabaseConnection:

- Knows the scheme of the database.
- Knows about different database products (Oracle, Sybase, DB2, MS SqlServer), database drivers, and the capabilities available to those products and drivers.



You create a DatabaseConnection through FormPack and supply a JDBC connection.

```
DatabaseConnection dbConnection =
   FormPack.newDatabaseConnection(
   jdbcConnection );
```

The DatabaseConnection will determine the database product type and the driver type from the metadata available through the JDBC connection[2]. FORM also learns about the database scheme through the JDBC connection[3].

---

[1] It is useful to have a developer information string used during tracing and debugging that is separate from an end-user display string.

[2] The Database product can also be set explicitly if FORM does not recognize it through the driver.

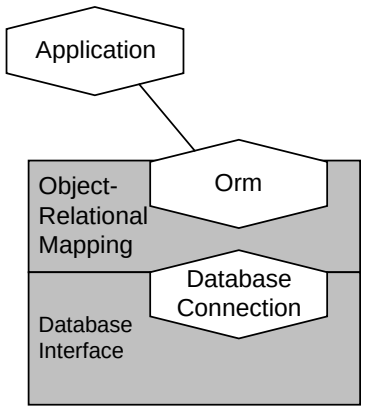
[3] FORM's database model can also includes virtual columns (called CompoundColumns) which are not part of the database itself. These must be explicitly mentioned during configuration.

Orm

An **Orm** is responsible for translating between the database information and the object model. It is *the* Object-Relational Mapper, but primarily serves as a coordinator among all the objects doing the real work. An Orm is created from FormPack.

```
Orm orm = FormPack.newOrm();
```

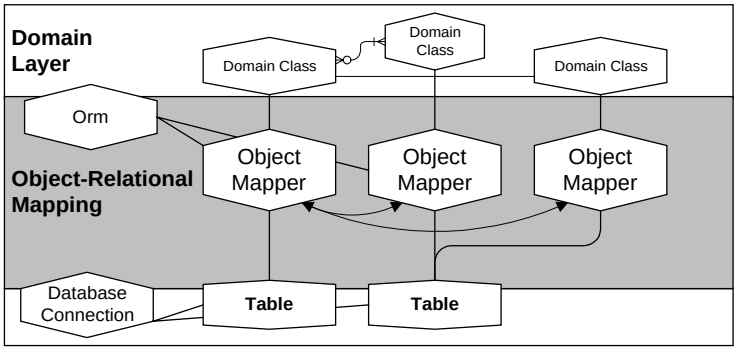
The Orm will be needed throughout FORM configuration. After configuration the Orm is not directly used, but you will need to hold onto it for the duration of the application's interaction with FORM. We can accomplish this through a static variable or through a longer-lived object (e.g., the application object) holding onto the Orm.



Conceptually the Object-Relational Mapping layer sits on top of the FORM database interface layer. For most applications, there will be only one Orm and one DatabaseConnection and a one-to-one correspondence between them.

ObjectMappers

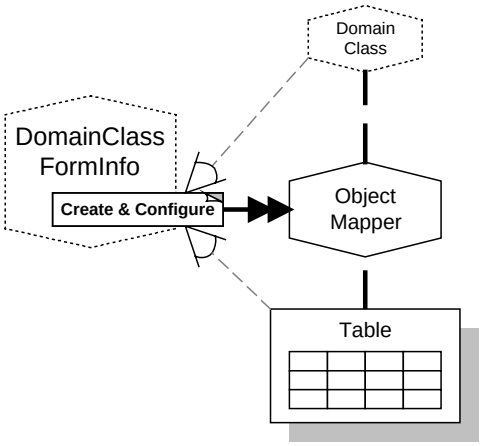
Now that we have an Orm and a DatabaseConnection, we can specify the details of how to map domain objects to the database. An Orm uses **ObjectMappers** to convert, store, and retrieve objects of particular classes into the information in database tables. To specify the details of an object-relational mapping we need to create and configure ObjectMappers. In PersonRetrieval_1 we delegate this responsibility to a FormInfo object[4].



[4] Note again that there are many different ways to configure FORM and a particular Orm.

FormInfos

A **FormInfo**[5] object is responsible for understanding how to map a particular domain class to the database. A FormInfo creates and configures one or more ObjectMappers that will handle the specifics of (usually) a single class. By using many FormInfos together to configure an Orm, we can provide a complete mapping of the domain to the database.



We will discuss implementing and generating FormInfo classes in the next chapter.

To use a FormInfo you need to create a FormInfo object, tell it what Orm and DatabaseConnection to use, and then add it to the orm so the Orm will call it at the appropriate moments to create an configure ObjectMappers.
This whole process of creating and adding a FormInfo object looks something like this:

```
FormInfo formInfo = new PersonClass_FormInfo();
formInfo.initOrm_db(orm,dbConnection);
orm.addInfo(formInfo);
```

Because the above sequence is very common, Orms provide a shortcut method to do all three steps:

```
orm.addInfoClass_withDb(PersonClass_FormInfo.class, dbConnection);
```

This method is what PersonRetrieval_1 uses.

Completing the Orm setup

After all the ObjectMappers have been created, the Orm must be told that it has completed the configuration and that it should prepare for running. This is also the time when the mapping is validated for any obvious errors. These tasks are accomplished through

```
orm.doneSetup();
```

This pattern of using #doneSetup to indicate the completion of configuration will be used throughout ChiMu's Java code.

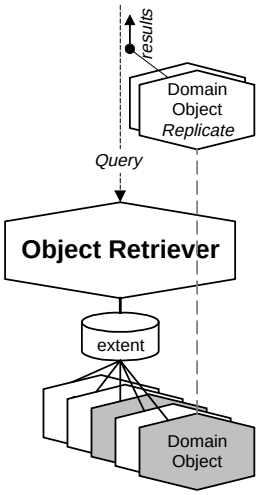
2.5 Running with FORM

After the mapping has been configured we can start retrieving objects from the database. First we need to get an ObjectRetriever to be able to retrieve some objects from the database (later we will learn about other ways to retrieve objects).

[5] FormInfos use a naming convention and design similar to Java Beans' BeanInfo classes.

ObjectRetrievers

An **ObjectRetriever** retrieves domain objects from the database. An ObjectRetriever knows of an extent of objects from that database and can retrieve objects from that extent which satisfy a client's particular criteria.



The retrieved objects are called **replicates** because they are replicated from the objects[6] that exist on the server. The database has the central "true" objects and each client application[7] has only temporary copies (replicates) of those true domain objects.

The ObjectRetriever accomplishes one-half of the task of synchronizing between these two "spaces" of objects: it moves information from the database objects to the replicate objects on the client.

The **ObjectMapper** that we created during configuration is a subtype of ObjectRetriever and provides complete retrieval capabilities. ObjectMappers also accomplish the other half of the synchronization task: the can move information from the client replicates to the database objects.

During the Orm configuration we created an ObjectMapper/Retriever named "Person" and we can get this retriever by name from the orm:

```
ObjectRetriever personRetriever = orm.retrieverNamed("Person");
```

Retrieving Objects

Next we can use the personRetriever to retrieve some Person objects. The simplest retrieval is to ask the retriever for any object from the extent of objects on the database:

```
Object person = personRetriever.findAny();
```

This will return either an Object from the personMapper's extent (i.e. the database table) or 'null' if the extent is empty. We can then print out the returned object (or null). Alternatively we could have asked for ***all*** the objects from the extent:

```
Collection people = personRetriever.selectAll();
```

We could then print the whole collection.

2.6 Example: PersonRetrieval_2

As another simple example, PersonRetrieval_2 selects all the people from the extent and then print information about some of them. There is a small change to the configuration part of the example: the configuration itself is done in ExampleAbsClass (the parent class to the example) through the call 'createAndConfigureOrm'. This simply removes redundant code from all the examples: All the examples in a given 'scheme' have the same domain model and the same mapping.

```
public class Ex_PersonRetrieval_2 extends ExampleAbsClass {

    public void run (Connection jdbcConnection) {

        createAndConfigureOrm(jdbcConnection);

        //----------Running-----------

        ObjectRetriever personRetriever = orm.retrieverNamed("Person");

        Sequence people = (Sequence) personRetriever.selectAll();
        outputStream.println(people);
```

[6] Relational databases do not have real objects that you can interact with, but they have enough information to know that particular object's exist. This could be considered a virtual object or an **object shadow**. See [Foundations] for more information.

[7] Even if that "client" is another server, such as an application server.

```
        int size = people.size();
        for (int i = 0; i<size; i++) {
            Person person = (Person) people.atIndex(i);
            if (person.name().length() > 15) {
                outputStream.println(person.info());
            }
        }
    }
}
```

2.7 Object Typing

FORM is designed to handle any type of domain model, and in Java it accomplishes this by being very general with regards to declared types. Most of FORM deals with objects of type 'Object'. If you need more domain specific behavior you will have to "cast" (really "type-check") the results of FORM functions to the domain object type you expect. For our example, if you need to treat the object returned by the ObjectRetriever as a Person you must perform a "(Person)" cast.

```
        Person person = (Person) personMapper.findAny();
        outputStream.println(person.name());
```

Because the ObjectRetriever is known to only deal with Persons, the type-check will always succeed. In other cases you may need to check the type of the object before casting it.

2.8 Summary

This chapter introduced FORM by the simple single-class PersonRetrieval example. FORM must be configured before being run, and this configuration uses an Orm, a DatabaseConnection, and multiple FormInfo objects that create ObjectMappers and ObjectRetrievers. After configuration is complete you can ask an ObjectRetriever to retrieve either a single object or multiple objects.

This example only showed the most basic FORM functionality. FORM also supports general object queries, attributes, associations, and heterogeneous retrievals. All of these will be covered in upcoming chapters. But this chapter and the next will show many of the major components to FORM, which support FORM's more advanced capabilties.

3 Creating an Object-Relational Mapping

In the previous chapter we introduced Orms, DatabaseConnections, ObjectMappers, Domain Classes, and FormInfos to show how these classes can work together to retrieve domain objects. What we will cover now is how to specify the details of the object to relational mapping. FormInfo objects create ObjectMappers to accomplish this detailed mapping.

3.1 ObjectMappers and FormInfos

ObjectMappers are one of the core concepts in FORM. They are responsible for mapping an individual domain class to a database table[8]. Each ObjectMapper only manages domain objects for a single class, but all the ObjectMappers collaborating together can retrieve and save the entire object model, or any part of it. Building ObjectMappers is broken into three stages. First the ObjectMapper is created as part of an Orm and attached to the appropriate tables. Second, the ObjectMapper is configured to describe the details of the mapping. Third, the ObjectMapper is linked with any interested DomainClasses. A FormInfo object provides a **MapperBuilder** that supports these three stages through the methods #createMappers, #configureMappers, and #configureCompleted. Each of the following sections will describe a stage to building an ObjectMapper and show how a FormInfo object implements the stage.

3.2 Creating an ObjectMapper

ObjectMappers are created through the Orm with the method #newObjectMapperNamed_table:

```
ObjectMapper mapper =
        orm.newObjectMapperNamed_table(“MapperName”,aTable);
```

All ObjectMappers and ObjectRetrievers have a unique name within a given Orm, which can be later used to find them programmatically and to refer to them in OQL queries. Creating the ObjectMapper automatically registers it with orm. The one piece of information we have not identified yet is the table.

Tables

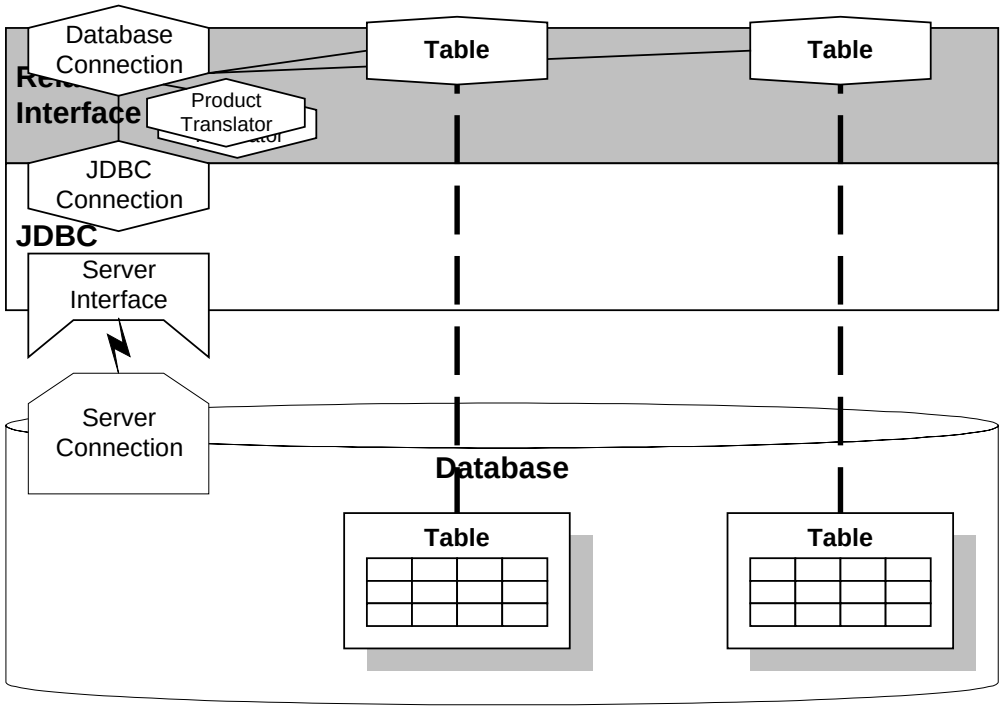
A **Table** represents a connection to a particular table on the database (so it could be thought of as a “TableConnection”). To get a table connection you need the database connection and the table’s name on the database.

```
Table table = dbConnection.table("Person");
```

A Table is not valid unless the database has a table with the same name as the parameter.

[8] Actually neither of these cardinalities are mandatory. An ObjectMapper can map to multiple tables and for multiple classes. For the moment we will ignore these more complex variations.

Together the DatabaseConnection and the Tables provide a representation of the database within the client application. This is FORM's database interface layer. The database interface layer is directly above the JDBC layer, which handles lower level issues like client to database communication.



FormInfo createMappers

A FormInfo object must create mappers in the method #createMappers. The implementation of this method in com.chimu.form.client.DomainObject_1_AbsClass_FormInfo (a superclass of PersonClass_FormInfo) looks like this:

```
    /**
     * Create and add to the Orm all mappers used by this builder.
     * During this stage you can not assume any other mappers exist.
     */
public void createMappers() {
    //...Dupplicate Mapper Error Checking
    myTable  = dbConnection.table(tableName());
    myMapper = orm.newObjectMapperNamed_table(mapperName(),myTable);
}
```

So if #tableName and #mapperName return 'Person' (which they do for the PersonClass_FormInfo object), we have a mapper creation identical to:

```
    Table table = dbConnection.table("Person");
    ObjectMapper mapper =
           orm.newObjectMapperNamed_table("MapperName",table);
```

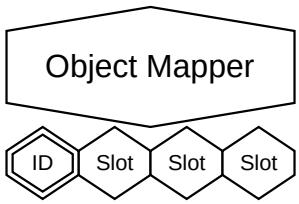
Because PersonClass has a standard mapper creation it can simply use the inherited method's implementation instead of adding its own.

3.3 Configuring an ObjectMapper

After we have created an ObjectMapper we need to configure it with the details of how to create objects of a particular class and how those objects' identity and attributes are connected to the information in the table. Constructors support creating objects and ObjectMapper Slots describe the detailed mapping of an object's identity and attributes.

Slots

Slot is the second core class in FORM (with ObjectMapper being the first). Whereas an ObjectMapper describes the mapping of a Class, a Slot describes the mapping of an Attribute. A Slot represents an attribute independently of whether it is stored in the object (as an instance variable) or stored on the database (as a column value). Slots work cooperatively with an ObjectMapper to describe the complete mapping of a class.



There are a number of different types of slots to handle different types of information sources. We only need two types of slots, DirectSlots and IdentitySlots to completely describe the state information of a Person.

Direct Slots

A **DirectSlot** moves values from instance variables directly to and from database column values. Direct slots are only useable with SimpleObjects that can be stored directly in databases: Strings, Numbers, Dates, Boolean, and binary data. To create a DirectSlot requires specifying the attribute name (e.g. "name") and the database column name (e.g. "person_name"). If the database column name is left off, it is assumed to be the same as the slot name.

```
mapper.newDirectSlot_column("name","person_name");
mapper.newDirectSlot("email");
```

Identity Slots

An **IdentitySlot** specifies the attribute that determines the real identity of an object; i.e., the identity for the object that is stored on the database instead of the identity of the local replicate of that object. This slot produces the unique[9] **IdentityKey** which links the local replicated object to the server object. An ObjectMapper must have one and only one IdentitySlot. An IdentitySlot is created just like a DirectSlot.

```
mapper.newIdentitySlot_column("oid", "id");
```

Slot Value Types

Slots default to a type of Object for SlotValues. This allows any type of data to pass through the slot and it is up to the application to be compatible with the database and JDBC driver[10]. Certain databases (e.g. MS SQL Server) support a large number of JDBC data-types so specifying the type of the Slot is frequently unnecessary. Other databases have fewer data-types (e.g. Oracle) so specifying the type helps transform data to the most convenient form. Finally, some applications may want to use a Java type that is different from the default database SQL type or maintain database independence.

To isolate your application from database specific and JDBC driver specific behavior you can tell FORM what Java type to use for a Slot and Table column. All the Java types useable in JDBC 'get<Type>' methods are available[11] and the type is specified by passing in an appropriate Class object. For the above examples we would use:

```
mapper.newDirectSlot_column_type("name","person_name",String.class);
mapper.newDirectSlot_type("email",String.class);
mapper.newIdentitySlot_column_type("oid", "id",Integer.class);
```

Constructors

FORM will automatically build DomainObject replicates whenever it retrieves a database object that is not already in the application. All the state information for these replicates is coming from the database, so the

[9] Unique at least to the class, but possibly unique on a larger scale like for all server objects or all objects on all servers.
[10] Supposedly JDBC drivers should be able to do a number of datatype conversions, but this ability is non-existent for the general setObject, getObject in current (and probably future) JDBC Drivers.
[11] Plus java.util.Date

replicate construction process is different from normal domain construction. Normal construction should be complete: the resulting domain object should be a valid, useable domain object. Instead, FORM wants a "raw" constructor, which does nothing other than allocate the object. Any extra construction behavior is just overhead because FORM will overwrite the information when it copies the database information into the domain object[12].

FORM Constructors

FORM uses a constructor that takes a single CreationInfo argument. This specific argument isolates the **FORM constructor** from your normal constructors and is also used to optimize the creation process. The declaration of a FORM constructor is

```
    protected PersonClass(com.chimu.form.mapping.CreationInfo cInfo) {
        super(cInfo);
    }
```

The FORM constructors will call up the superclass hierarchy until they get to the root class (usually one of the supplied DomainObject_n_AbsClass). Because Java does not inherit constructors, each class in the superclass hierarchy must declare a constructor even if they have nothing to add to the construction process.

The full path name to CreationInfo is only required in the specification if you do not import "com.chimu.form.mapping.*" or CreationInfo specifically. The subsequent examples will assume this import.

Creation Functions

FORM needs to be told how to call the constructor on your domain class. FORM uses a specific CreationFunction functor to encapsulate this information[13]. The **CreationFunction** declaration is

```
    static protected CreationFunction form_creationFunction() {
        return new CreationFunction() {
            public MappedObject valueWith(CreationInfo cInfo){
                return new EmployeeClass(cInfo);
            }
        };
    }
```

This function is similar to a Function1Arg functor [14], but it is FORM specific because it takes a CreationInfo as its argument and returns a MappedObject (we will discuss MappedObjects in the next section) as its result. The function simply calls the FORM constructor.

Finally, the CreationFunction is used when configuring a mapper

```
        mapper.setupCreationFunction(form_creationFunction());
```

FormInfo configureMappers

A FormInfo object configures mappers in the method #configureMappers. During this method it should set up the creation function, specify an identity slot, and add attribute slots to one or more ObjectMappers. The full configuration process for Person in the FormInfo files looks like:

```
        /**
         * Configure the created mappers.
         */
    public void configureMappers() {
        setupCreationFunction();
        myMapper.newIdentitySlot_column("oid",oidColumnName());

        myMapper.newDirectSlot_column_type("name","person_name",
           String.class);
        myMapper.newDirectSlot_type("email",String.class);
        myMapper.newDirectSlot_type("height",Integer.class);
```

[12] For more information, see the next chapter's section 'Initializing the State of a Mapped Object'.
[13] FORM could have used the reflection capabilities of Java to identify and later call the constructor, but this would cause a significant amount of runtime overhead and would reduce configuring flexibility.
[14] See ChiMu's Kernel Frameworks

```
    }
```

FormInfo Superclasses

All of this could be specified in the PersonClass_FormInfo class but usually there is consistency among DomainObjects, so they all will (for example) use the same IdentitySlot column and have the same creation function. This means abstract classes (DomainObjectAbsClass_FormInfo in the scheme1 examples and the FORM provided com.chimu.form.client.DomainObject_1_AbsClass_FormInfo) can provide the common behavior and subclasses only have to provide their class specific information. The configureMapper method in PersonClass_FormInfo looks like this:

```
    public void configureMappers() {
        super.configureMappers();

        myMapper.newDirectSlot_column_type("name","person_name",
           String.class);
        myMapper.newDirectSlot_type("email",String.class);
        myMapper.newDirectSlot_type("height",Integer.class);
    }
```

The call to 'super' allows the superclasses to set up the CreationFunction and IdentitySlot.

FORM Preprocessor

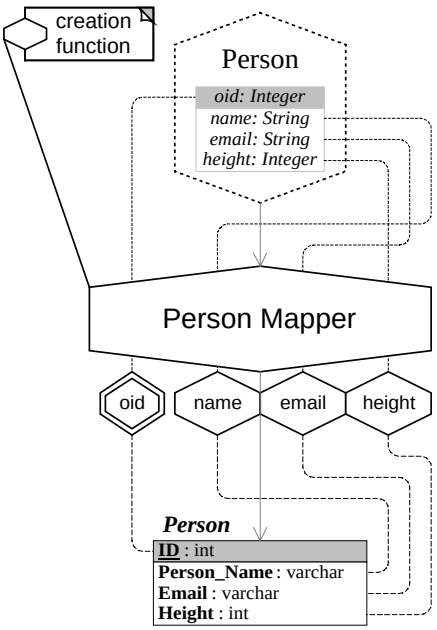
The FORM Preprocessor is a tool that helps prepare Domain classes to be used with FORM. It can annotate a DomainClass with constructor information and can generate "guesses" at the appropriate FormInfo classes. The FORM Preprocessor is described in [Tools] and more of the Domain class annotation information will be discussed in the chapter on "Domain Object Services".

3.4 Linking ObjectMappers

The last stage of building an ObjectMapper is to hook it up to classes that may want to refer to it. This is primarily used for allowing objects to save themselves (which will be discussed in a later chapter), but can also be used for other services. DomainObject_1_AbsClass has a method #linkMapper_toClass which allows a Class to remember its particular ObjectMapper. The DomainObject_1_AbsClass_FormInfo file calls this method during #configureCompleted to finish the building process. This can be overridden to support extended functionality.

3.5 The Completed Person ObjectMapper

We can now look at the whole Person ObjectMapper resulting from the configuration. The mapper has four slots: one ("oid") represents the server identity of a person, and the other three are basic attributes. These slots are connected to the appropriate columns of the database table and will be used for initializing and extracting the state information from the Person object. The Person mapper also holds onto a functor that creates a "blank" person object whenever it is called.



The PersonMapper knows of the Person table and will use that table to retrieve and write objects to the database. The Person class knows of the PersonMapper so that person objects can find their mapper when they need to interact with it.

A later chapter will discuss how information is moved from the mapper and its slots into a particular object instance (in this case, a person). The next section after the example will describe how to use an abstract class to

simply and standardize configuration, but the resulting Person Mapper will not change.

3.6 New Examples: Using Slots in Retrievals

Now that we have specified what slots exist on a Person, we can use that information in our requests to the ObjectRetriever. Object Retrievers support the protocol to ask for objects with a particular slot value:

```
    Object findWhereSlotNamed_equals(String slotName, Object value);
    Collection selectWhereSlotNamed_equals(String slotName, Object
           value);
```

Example: PersonRetrieval_3

So we can ask for the person named “Patricio Steel”

```
public class Ex1_PersonRetrieval_3 extends ExampleAbsClass {
    public void run (Connection jdbcConnection) {

        createAndConfigureOrm(jdbcConnection);
        ObjectRetriever personRetriever = orm.retrieverNamed("Person");

        Person person = (Person)
           personRetriever.findWhereSlotNamed_equals("name", "Patricio
           Steel");
        outputStream.println(person.info());
    }
}
```

Example: PersonRetrieval_4

Or for all the people 5’6” tall

```
public class Ex1_PersonRetrieval_4 extends ExampleAbsClass {
    public void run (Connection jdbcConnection) {

        createAndConfigureOrm(jdbcConnection);
        ObjectRetriever personRetriever = orm.retrieverNamed("Person");

        Collection people =
           personRetriever.selectWhereSlotNamed_equals("height", new
           Integer(66));
        outputStream.println(people);
    }
}
```

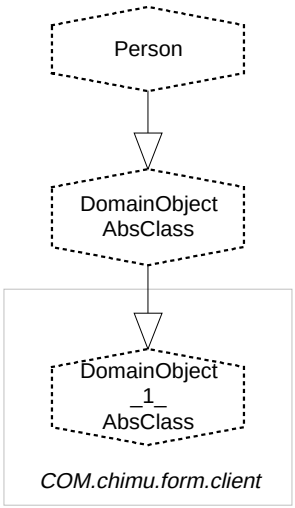
3.7 General Domain Interfaces and Abstract Classes

Domain classes usually have a common set of core responsibilities and core behavior. Domain interfaces declare what core responsibilities your domain objects support and domain abstract classes implement common core behavior. These are very useful, especially as you build larger domain models.

FORM does not require using abstract classes but it does provide them to make working with FORM easier. These can be used with your own abstract classes through inheritance or code merging (if you would otherwise need multiple inheritance).

FORM Provided Abstract Classes

FORM provides several abstract classes with slightly different behaviors to handle the most common cases an application may have. These classes are in “com.chimu.form.client”. The easiest and most flexible approach for your application would be to create an abstract class that inherits from one of the FORM provided abstract classes. Your domain objects can then inherit from your abstract class and if your needs (for all domain objects) change in the future you can simply modify your abstract class.



This same approach can be used for the FormInfo classes extending from com.chimu.form.client.DomainObject_1_AbsClass_FormInfo.

Client defined Abstract Classes

All of the FORM provided abstract classes use the same API available to any FORM client. The source is included in the distribution. If you require behavior different from that provided by the FORM provided abstract classes you can copy and modify the abstract classes to add your functionality. You can also use this approach if you have another framework that has a mandatory inherited root class.

Domain Interface for PersonRetrieval

The PersonRetrieval example does not provide a general DomainObject interface. This is solely because it is a single class example and separating the interface seems overkill. Later domain models will separate out general protocol (like #info) which should be available for all domain classes into a DomainObject interface that documents and commits to the common functionality.

3.8 Putting it all Together

This section collects the major code pieces discussed in this chapter so you can see them all together. The full source for PersonRetrieval and the domain model is available in the example source files.

PersonRetrieval_4

```
public class Ex1_PersonRetrieval_4 extends ExampleAbsClass {
    public void run (Connection jdbcConnection) {
        createAndConfigureOrm(jdbcConnection);
        ObjectRetriever personRetriever = orm.retrieverNamed("Person");

        Collection people =
           personRetriever.selectWhereSlotNamed_equals("height", new
           Integer(63));
        outputStream.println(people);
    }
}
```

ExampleAbsClass

```
public abstract class ExampleAbsClass
            extends FormDatabaseTestAbsClass {
    protected void createAndConfigureOrm(Connection jdbcConnection) {
        createOrm(jdbcConnection);
            orm.addInfoClass_withDb(
                PersonClass_FormInfo.class,
                dbConnection);
        orm.doneSetup();
    }
}
```

PersonClass_FormInfo

```
public class PersonClass_FormInfo extends DomainObjectAbsClass_FormInfo
            {
    public void configureMappers() {
        super.configureMappers();

        myMapper.newDirectSlot_column_type("name","person_name",
           String.class);
        myMapper.newDirectSlot_type("email",String.class);
        myMapper.newDirectSlot_type("height",Integer.class);
    }

    public Class myClass() {
       return PersonClass.class;
    }
}
```

DomainObjectAbsClass_FormInfo

```
public class DomainObjectAbsClass_FormInfo extends
            DomainObject_1_AbsClass_FormInfo {
//...
    protected String oidColumnName() {
        return "id";
    }
}
```

DomainObject_1_AbsClass_FormInfo

```
public abstract class DomainObject_1_AbsClass_FormInfo
             implements FormInfo, MapperBuilder {

//...
    public void createMappers() {
        //...
        myTable  = dbConnection.table(tableName());
        myMapper =
           orm.newObjectMapperNamed_table(mapperName(),myTable);
    }

//...
    public void configureMappers() {
        setupCreationFunction();
        myMapper.newIdentitySlot_column("oid",oidColumnName());
    }

//...
    public void configureCompleted() {
        linkClassToMapper();
    }
//...
}
```

3.9 Summary

The PersonRetrieval examples used FORM to map a single class (Person) to the database. This was accomplished by the PersonClass_FormInfo object creating and configuring an ObjectMapper and its slots to map to the database Table that stored Person information. After configuration was completed we could ask simple questions of the ObjectMapper to retrieve objects.

The FORM classes introduced were Orm, DatabaseConnection, Table, ObjectMapper, ObjectRetriever, and Slot. These are core classes in FORM, especially ObjectMapper and Slot. The chapter also described how to delegate the Orm configuration among FormInfos that are each responsible for a single class.

FORM has a programmatic interface. This interface is designed to be concise yet flexible. The program structure that was shown in these examples is recommended for structuring your Orm configuration, but you

can also choose other approaches to match your needs. Likewise, FORM provides abstract classes that implement functionality your program will need, but you can also choose to have your own abstract classes extend or replace the FORM provided classes.

4 Introduction to Queries

FORM provides full query capabilities that allow you to easily evaluate both simple and sophisticated queries on your database domain model. FORM's query model is an object-extended relational model, which allows you to benefit from the simplicity and expressiveness of both object-orientation and predicate logic. FORM provides both a programmatic interface to queries and a query language (OQL) interface. These have identical capabilities, so the programmer can choose the appropriate interface for the given task.

This chapter provides an introduction to all the major concepts in the Query system and in FORM's OQL. Queries over associated objects will be covered in a later chapter, but that functionality will require only a couple new concepts. For query examples we can use the same simple Person model from the previous chapters. The configuration will be the same as in the PersonRetrieval examples, which results in a single retriever/mapper being passed to the main application.

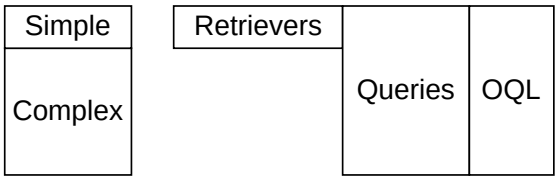
```
ObjectRetriever personRetriever = orm.retrieverNamed("Person");
```

The first part of this chapter covers programmatic functionality and the second part covers FORM's OQL.

4.1 Queries

Queries are requests to retrieve server objects matching specified criteria. Queries work "on top of" your object model, using information about your objects to answer questions. FORM provides queries as an adjunct to automatic association traversal, which will be discussed in an upcoming chapter. Queries can be used to retrieve "original" objects from an extent, to provide optimizations for complex relationships, or to support general query needs for the end-user.

FORM provides three ways to express queries: ObjectRetriever queries, QueryDescription queries, and OQL queries. Of these options, ObjectRetriever provide only simple queries where as QueryDescriptions and OQL can handle all types of queries.



4.2 ObjectRetriever Queries

ObjectRetrievers have protocol for performing simple queries, which we have seen in the previous chapters. The two basic queries are:

```
personRetriever.findAny();
personRetriever.selectAll();
```

These return an object or all objects from the retriever's extent, respectively. This shows a first classification of queries. Queries are divided into two different categories based on the return value. A '**select**' query returns a collection of 0 or more objects, and a '**find**' query returns either a single object or null. The 'find' protocol is primarily for clarity and caller convenience (getting a single object instead of a collection) since it is subsumed in the functionality of the 'select' queries[15]. For our simple queries #findAny is much more logical than the possible #selectAny.

An ObjectRetriever can also retrieve objects based on a particular slot and slot value.

```
personRetriever.findWhereSlotNamed_equals("name", "Jane Doe")
```

All the above ObjectRetriever queries are really shorthand methods for the more extensive query functionality available through QueryDescriptions,

[15] It can also be faster to execute a 'find' in certain cases because FORM need return only a single value whether more than one might match.

4.3 QueryDescriptions and QueryVars

The query subsystem allows you to build complex queries based on multiple types of objects and multiple conditions. FORM's query subsystem is based on the concepts in OQL[16] and relational calculus[17,18]. It is has a simple, declarative model which enables you to retrieve desired objects by specifying what objects you wish to consider and how to restrict which of those objects will be selected. The major classes in the query subsystem are QueryDescription, QueryVar, and Condition.

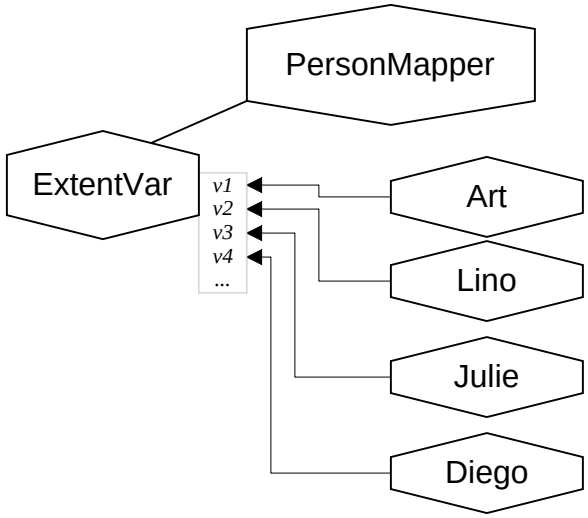
QueryDescriptions

A **QueryDescription** (frequently just called a query) collects a description of the desired query before executing it. A query description specifies what objects should be considered, what condition(s) are used to restrict which objects are chosen, and which of the chosen objects will be returned from the query. QueryDescriptions will be further described in the following sections as we add the other concepts needed for a query. The following creates a new query description:

```
QueryDescription query = orm.newQueryDescription();
```

QueryVars

The most important concept in the query subsytem is the query variable (QueryVar). A **QueryVar** is a variable that ranges over a collection of domain objects. At any given time in the evaluation of the query, the QueryVar will hold onto a single object, but throughout the whole evaluation the QueryVar will selectively take on all its possible values.



The simplest QueryVar is an extent variable (ExtentVar). An ExtentVar ranges over the objects in the extent of an ObjectRetriever. An ExtentVar defined over PersonMapper would first hold onto the object for Art, then Lino, then Julie, then Diego, and so on for all the objects that exist at the time of the query evaluation.

We can create an ExtentVar from a queryDescription by supplying an ObjectRetriever to range over.

```
query.newExtentVar(personRetriever);
```

Query evaluation

After building a QueryDescription we can evaluate it and get the results. This is done similarly to querying an ObjectRetriever. We can retrieve all the objects that match the query:

```
query.selectAll()
```

Or we can retrieve a single object that matches the query:

```
query.findAny()
```

There are also special methods for counting and testing whether any objects are in the query.

---

[16] See [Cattell+ 97], Chapter 4 for an overview of OQL.
[17] See [Date 95] for a (thorough) introduction to relational calculus.
[18] You may notice the omission of SQL from this list. The "concepts" in SQL are all from relational calculus, and SQL provides syntax to express those concepts. Unfortunately, SQL "… is filled with numerous restrictions, *ad hoc* constructs, and annoying special rules." [Date+H 97: Page 7]. Also, many of the concepts from relational calculus are distorted by these peculiarities of SQL so they are better simply avoided.

Example: Query_1

Using the query concepts we have so far, we can perform simple queries through query descriptions instead of the ObjectRetriever query protocol. The following retrieves all the people from the personRetriever extent.

```
public void run (Connection jdbcConnection) {
    //-------Configuration--------
    //...
    //-------Creating Query--------
    QueryDescription query = orm.newQueryDescription();
    QueryVar personVar = query.newExtentVar(personRetriever);

    //----------Running-----------
    Collection persons = query.selectAll();
    outputStream.println(persons);
 }
```

The results of the above:

```
    Collection persons = query.selectAll();
```

will be identical to:

```
    Collection persons = personRetriever.selectAll()
```

For this particular example the query description was simply extra effort, but it will be able to provide much more functionality with the capabilities described in the following section.

4.4 SlotVars and Results

ExtentVars freely range over the values from an extent of objects. Some QueryVars are not "free", but instead are derived from the value of another QueryVar. The first example of a DerivedVar is a SlotVar.

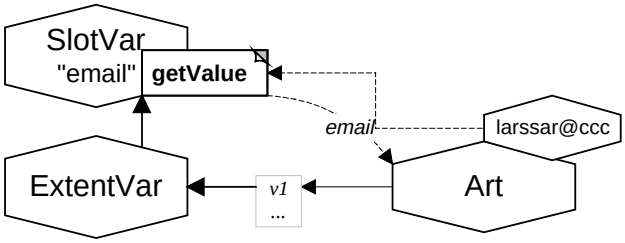
SlotVars

SlotVars determine their value during query evaluation by traversing:

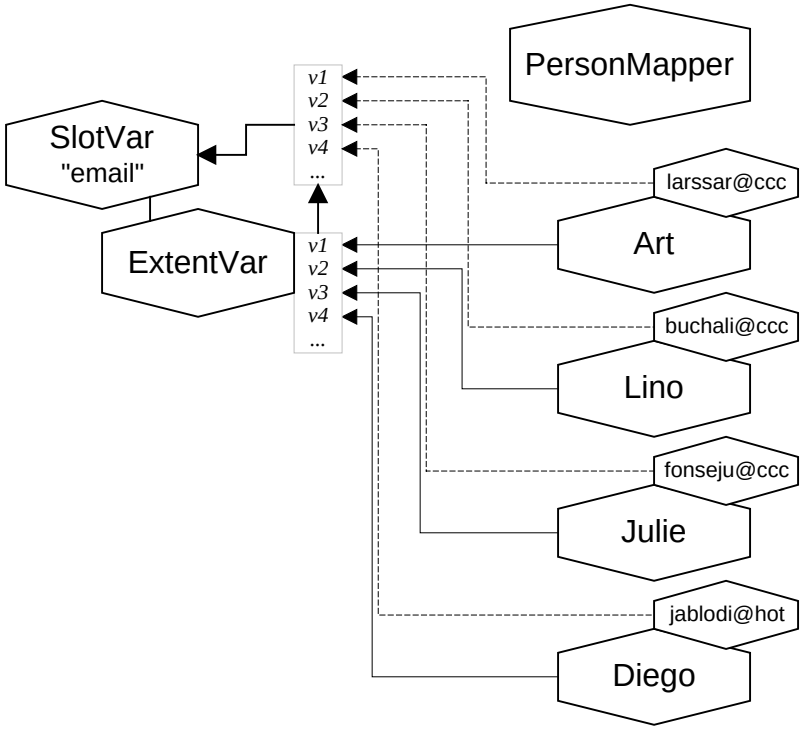
- from the object which is the value of the SlotVar's starting QueryVar
- through a named Slot
- to the value of the Slot for that object

For example, we have an ExtentVar that ranges over People and a SlotVar that is derived from the ExtentVar through the slot "email". The first value of the SlotVar is derived from the following flow:

- ExtentVar equals the *Art*
- Get *Art*'s email
- SlotVar equals "larssar@ccc"



The following diagram shows more values for our example:



We create a SlotVar through a QueryVar with the name of the Slot to traverse.

```
QueryVar personVar = query.newExtentVar(personRetriever);
QueryVar personEmailVar = personVar.slotVarNamed(“email”);
```

Query Results

A query evaluation always returns the values of a single QueryVar. If you do not explicitly specify what variable to return for a query, then the query will use the first ExtentVar that was created. If a different QueryVar is desired, then you can specify it:

```
query.setResultsVar(personEmailVar);
```

This says to build the results of this query from the values of personEmailVar.

Example: Query_2

Using SlotVars and a specified query result, we can build a query that asks, “What are the email addresses of People?”

```
public void run (Connection jdbcConnection) {
    //-------Configuration--------

    //-------Creating Query--------
    QueryDescription query = orm.newQueryDescription();
    QueryVar personVar = query.newExtentVar(personRetriever);

    QueryVar personEmailVar = personVar.slotNamed("email");
    query.setResultsVar(personEmailVar);

    //----------Running-----------
    Collection names = query.selectAll();
    outputStream.println(names);
 }
```

This would return all the e-mail values for people, without any duplicates.

Duplicates

By default, the result of a Query is a Set, which will not have duplicates. This is the correct behavior in terms of relational theory: if a value is in the result set, it means that the value satisfies a truth statement. Repeating any given value would be ambiguous (i.e., does this value really, really satisfies the truth statement?). Because of this, duplicates are ignored and discarded. For example, we previously asked the question "What values are known to be the email address of a person?" (This is the correct phrasing of a query. Our previous phrasing was a common, but misleading, shorthand.) Among the result set our example returned, we find entries such as "larssar", "buchali", "fonseju". It would not be correct to return the answer "fonseju" twice, because we already answered the question.[19].

Unfortunately, many relational databases do not default to this behavior and return duplicates in the result. It would be best to let FORM act correctly and discard duplicates. But if there are performance problems with database queries, you can tell a particular query to include duplicates.

```
query.makeResultsNotDistinct();
```

4.5 Query Conditions

A **Condition** allows you to restrict the results of a query: It determines which values are acceptable and which values should be filtered out. The simplest condition is an equality test:

```
Constant anEmail = query.newConstant("hhoudini");
Condition personNameCondition = query.newEqualTo(personEmailVar,
       anEmail);
```

The above code contains two features. It has a **Constant**, which holds onto a value that will not change during the query evaluation. It also has a Condition that will return "True" when the current value of personEmailVar is equal to the personEmail constant. Condition creation should be read with an 'is' just after the 'new', so this is a 'personEmailVar _isEqualTo_ personEmail' condition.

Query Condition

Creating a condition is separate from assigning a condition to a query. To assign a condition, use:

```
query.setCondition(personNameCondition);
```

which specifies that the query will only return results for which 'personNameCondition' is true.

Example: Query_3

```
public void run (Connection jdbcConnection) {
    //-------Configuration--------

    //-------Creating Query--------
    QueryDescription query = orm.newQueryDescription();
    QueryVar personVar = query.newExtentVar(personRetriever);

    QueryVar personEmailVar = personVar.slotNamed("email");
    query.setResultsVar(personEmailVar);

    Constant anEmail = query.newConstant("hhoudini");
    Condition personNameCondition =
       query.newEqualTo(personEmailVar, anEmail);

    query.setCondition(personNameCondition);

    //----------Running-----------
    Collection people = query.selectAll();
    outputStream.println(people);
}
```

[19] See [Date].

Compounding conditions

Though a query can only have one single condition, this condition can be built from other conditions. We can 'and' and 'or' other conditions together using:

```
    Condition c3 = query.newAnd(c1,c2);
```

or

```
    Condition c3 = query.newOr(c1,c2);
```

Example: Query_4

So if we want to find all the People who either have an email address of "hhoudini" or have heights between 5' 5" and 6', we could use the following query:

```
public void run (Connection jdbcConnection) {
        //-------Configuration--------
…….
        //-------Creating Query--------
        QueryDescription query = orm.newQueryDescription();
        QueryVar personVar = query.newExtentVar(personRetriever);
        QueryVar personEmailVar = personVar.slotNamed("email");
        QueryVar personHeightVar = personVar.slotNamed("height");

        Constant anEmail = query.newConstant("hhoudini");
        Constant height = query.newConstant(new Integer(72));  //height
           is in inches
        Constant height2 = query.newConstant(new Integer(65));
           //height is in inches

        Condition c1 = query.newLessThan(personHeightVar , height);
        Condition c2 = query.newGreaterThan(personHeightVar , height2);
        Condition c3 = query.newAnd(c1 , c2 );
        Condition c4 = query.newEqualTo(personEmailVar, anEmail);
        Condition c5 = query.newOr(c3 , c4 );
        query.setCondition(c5);

        //----------Running----------
        Collection people = query.selectAll();
        outputStream.println(people);
    }
```

At this point our code is becoming a bit verbose and difficult to follow. With QueryDescriptions we have the functionality that allows arbitrary programmer flexibility, but it is provided as a programming API and if the program can not be more organized it may become too complicated. There are two solutions to this: (1) factor the program better or (2) use a more concise query interface (OQL). The next section will show an example of better factoring, and OQL is covered in the next chapter.

4.6 Query Factoring

Example Query4 became a bit complex because we needed to declare and organize a structure with multiple lines of code. The advantage of a programming API is we can easily factor the query to make it easier to manage. For example, our height (or more general) comparison could be put into a single routine:

```
    Condition newConditionFor_whereHeight_isBetween_and (
           QueryDescription query, QueryVar heightVar, int height1, int
           height2 ) {
        Constant cHeight1 = query.newConstant(new Integer(height1));
        Constant cHeight2 = query.newConstant(new Integer(height2));
        Condition c1 = query.newGreaterThan(heightVar, cHeight1 );
        Condition c2 = query.newLessThan(heightVar, cHeight2 );
        return query.newAnd(c1 , c2 );
    }
```

This (and removing some excess temporary variables) allows use to shrink the main body of the routine to:

```
        //-------Creating Query--------
        QueryDescription query = orm.newQueryDescription();
        QueryVar personVar = query.newExtentVar(personRetriever);
```

```
Condition c1 = newConditionFor_whereHeight_isBetween_and
   (query, personVar.slotNamed(“height”), 65, 72);
Condition c2 = query.newEqualTo(
                  personVar.slotNamed("email"),
                  query.newConstant("hhoudini") );
query.setCondition(query.newOr(c1 , c2 ));
```

Enabling this type of programmatic simplification is part of why FORM provides the QueryDescription API and not just OQL alone.

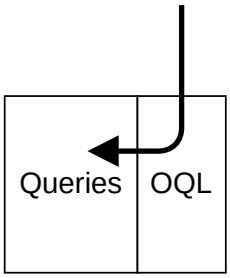
4.7 Summary

ObjectRetrievers provide simple query capabilities and QueryDescriptions provide sophisticated query capabilities. QueryDescriptions use QueryVars, Extents, Conditions, and ResultVars to describe the desired query. This allows flexibility and expressiveness through the programmatic API.

At times, QueryDescriptions may not provide the simplest interface for writing queries, and in these cases FORM’s OQL can be used instead.

5 OQL: FORM's Object Query Language

The previous sections described FORM's query engine and how to construct query descriptions, query variables, and conditions directly against the query API. These are the objects that FORM uses internally, so this is the most direct and programmatically flexible approach. The direct API can also become very wordy and indirect making it harder for the program developer and maintainers to understand.



FORM also offers an Object Query Language (**OQL**) subsystem that takes queries as strings and converts them into the query constructs of the main query engine. This query language is more concise and human readable than the programmatic interface, so some queries may be much easier to visualize and form correctly through OQL. It can also be used to provide end users with ad-hoc querying capabilities.

The query engine and OQL have identical capabilities and identical core concepts. OQL requires a few extra concepts because OQL needs to handle the relationships between the QueryString and actual objects.

All the examples in this chapter will also be based on the simple Person model from the previous chapters. The configuration will be the same as in the PersonRetrieval examples, which creates a single mapper named "Person". Although the name of the mapper was not important for the previous functionality, it is used with OQL.

Reviewing the important parts of the Person configuration, our main program called:

```
PersonClass.createTheMapper(orm,dbConnection);
```

And PersonClass#createTheMapper created the ObjectMapper as follows:

```
orm.newObjectMapperNamed_table("Person",table);
```

5.1 Example: OqlQuery_1

The simplest OQL query retrieves a single person from the Extent of Person.

```
OqlQuery oql = orm.newQuery("FROM Person person");
QueryDescription query = oql.query();
Object person = query.findAny();

outputStream.println(person);
```

This is exactly the same query as Query_1 and PersonRetrieval_1. The only difference is we have expressed the query concisely in a String and then converted that String into a QueryDescription (using the 'query()' message). From that point on we have a QueryDescription just like what was discussed in the previous chapter. OQL is just a simpler way to create QueryDescriptions.

5.2 OqlQuery

An **OqlQuery** knows how to create a QueryDescription from an OQL String. This is composed of two parts: (1) understanding the syntax of OQL and (2) resolving string references into the actual Objects that should participate in the query.

The syntax of OQL is very simple and similar to SQL. The full syntax is described in ???, but the examples in this chapter will show most of the syntax as well. The overall structure to an OQL query is:

```
[ SELECT variable ]
  FROM   variableDeclaration (, variableDeclaration)*
[ WHERE  condition ]
```

Where the '[' and ']' mark optional clauses and the '(' and ')*' indicates optional repetitions.

In OqlQuery1, we have a string "FROM Person person". This is the "**Variable Declaration Clause**": It names variables that the query requires and describes how those variables get their values. A Variable Declaration is the OQL approach to creating and remembering QueryVars. The simplest QueryVar is an ExtentVar, and the expresion "Person person" creates an ExtentVar named "person" which ranges over the extent of the "Person" ObjectMapper. How this reference is resolved will be discussed momentarily.

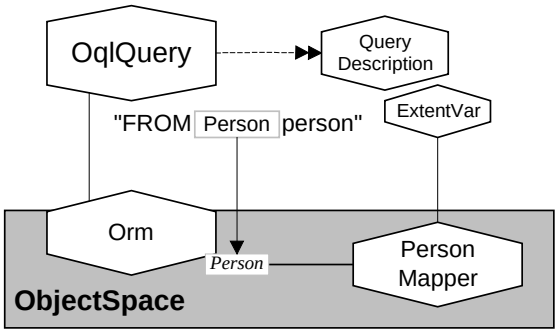
The expression "FROM …" may seem a bit short for a Query, but FORM's OQL queries have similar defaults to creating QueryDescriptions. If we define a single ExtentVar variable, OQL assumes that the query wishes to return that variable. We could explicitly mention to return the person variable, and this might look a bit more familiar "SELECT person FROM Person person". The results would be identical.

In addition to understanding the syntax of OQL, an OqlQuery must be able to resolve various string references into ObjectRetreivers, DomainObjects, slots, and other Objects that need to be passed into the query. All these types of objects will be discussed in the upcoming sections. For our example the OqlQuery must be able to resolve the "Person" refererence into the Person ObjectMapper. It does this by communicating with the Orm it was created from. OqlQueries are created from the Orm that you wish the query to apply to. This Orm provides the context (or **ObjectSpace**[20]) for the query: what Extents are available to be queried upon.

The OQL String is evaluated in the context of the Orm to bind Extent References (like "Person") to the ObjectRetrievers (like the Person ObjectMapper) that represent these Extents. When we created the person mapper for the Orm we gave it a name:

```
orm.newObjectMapperNamed_table(
          "Person", table );
```

This name[21] is used for the Extent Reference lookup and "Person person" is converted into "newExtentVar("person", PersonMapper)".



So translating our OQL query to the equivalent direct QueryDescription construction:

```
OqlQuery oql = orm.newQuery("FROM Person person");
```

is equivalent to

```
QueryDescription query = orm.newQueryDescription();
QueryVar personVar = query.newExtentVar(personMapper);
```

5.3 OQL to QueryDescription Equivalencies

Most of OQL maps directly onto the terms from the FORM QueryDescriptions and OQL has similar syntax to what is available in other query languages. As an example, the following is the most complex query covered in the Introduction to Queries chapter:

```
FROM Person person
WHERE   person.height > 65 AND person.height < 72
     OR person.email = 'hhoudini'
```

As this example shows OQL can just be much more concise than QueryDescriptions. The functionality is the same and each of the following subsections covers these equivalencies for the concepts we have seen so far.

---

[20] See [Foundations].

[21] Note that the name of the ObjectMapper or ObjectRetriever that was given to the Orm is all that matters, not the class, interface, or database table name, even though these will frequently be similarly named. The name of the ObjectMapper is the name of the Object Extent.

SlotVars

A SlotVar is created using either '.' or '->' on a QueryVariable. So the expression "SELECT Person.email FROM Person person" is the same as QueryExample2:

```
QueryVar personVar = query.newExtentVar(personMapper);

QueryVar personEmailVar = personVar.slotNamed("email");
query.setResultsVar(personEmailVar);
```

Literal Constants

A literal constant is an object that the OqlQuery can create directly from parsing the OQL String. These include integers, floats, strings, and booleans. All of these have the same format as for Java literal values, for example Strings are double quoted and boolean 'true' and 'false' can be typed directly and are case insensitive. FORM's OQL standard also allows you to use single quotes for Strings (the type Character is not used in OQL). The following table shows some example equivalents between OQL literals and the QueryDescription constant creation.

OQL	QueryDescription
"String"	query.newConstant("String")
'String'	query.newConstant("String")
true	query.newConstant(Boolean.TRUE);
FALSE	query.newConstant(Boolean.FALSE);
3456	query.newConstant(new Integer(3456));
34.56	query.newConstant(new Float(34.56));

Complex constants (e.g. a Person object) or constants whose value is determined at runtime can be inserted into OQL queries using bound parameters. This will be discussed in a later section. An example using literal constant will be in the next section after we have introduced conditions.

Conditions and Expressions

OQL supports all the condition and expression operators used with QueryDescriptions. These have the standard Java syntax and some alternate (and more common in query languages) syntax as shown in the following table:

Java	Alt	Meaning	Query Description
==	=	Equals	newEquals
!=	<>	Not Equals	newNotEquals
<		Less Than	newLessThan
<=		Less Than Equal To	newLessThanEqualTo
>		Greater Than	newGreaterThan
>=		Greater Than Equal To	newGreaterThanEqualTo
&	and	And	newAnd
|	or	Or	newOr

5.4 Example: OqlQuery_2

Now we can return to the previous chapter's example in OQL. It only uses literal constants (integers and a string) and two slot variables ('height' and 'email'). The pure OQL would look like the following.

```
FROM Person person
WHERE   person.height > 65 AND person.height < 72
     OR person.email = 'hhoudini'
```

Putting this into a program requires creating an OqlQuery with the above in a String form. Because Java does not allow line ends in the middle of a string we will have to break the above into multiple concatenated strings. The full example is:

```
   public void run (Connection jdbcConnection) {
        //-------Configuration--------
…….
        //-------Creating Query--------
        OqlQuery oql = orm.newOqlQuery(
                "FROM Person person "+
                "WHERE   person.height > 65 AND person.height < 72 "+
                "     OR person.email = 'hhoudini'"
            );

        //----------Running-----------
        Collection people = oql.query().selectAll();
        outputStream.println(people);
    }
```

5.5 Bound Values

Bound values solve two problems: (1) how to delay putting a value into a query until it is known, and (2) how to put a complex object into a query. Both of these are solved by the same approach, a **BoundValue** marker for an arbitrary object is placed into the OQL query string and latter this marker is bound to an Object. This value can be more complex than an OQL literal could be and it does not need to be bound until before building the QueryDescription.

Taking the above example and placing BoundValue markers into it:

```
    FROM Person person
    WHERE   person.height > :minHeight AND person.height < :maxHeight
         OR person.email = :emailAddress
```

we have exactly the same query except we have not specified some constants for height and emailAddress.

To bind the markers we can use:

```
    void bindName_toValue(String name, Object value)
```

Example: OqlQuery_3

Using the BoundValue markers, our example program becomes:

```
   public void run (Connection jdbcConnection) {
        //-------Configuration--------
…….
        //-------Creating Query--------
        OqlQuery oql = orm.newOqlQuery(
                "FROM Person person "+
                "WHERE       person.height > :minHeight "+
                "        AND person.height < :maxHeight "+
                "     OR person.email = :emailAddress "
            );

        oql.bindName_toValue("minHeight",new Integer(65));
        oql.bindName_toValue("maxHeight",new Integer(72));
        oql.bindName_toValue("emailAddress", "hhoudini");

        //----------Running-----------
        Collection people = oql.query().selectAll();
        outputStream.println(people);
    }
```

This adds three lines but now we can easily allow the height and email address to vary at runtime.

Complex Object Values

Another reason to use BoundValues in OQL queries is to insert a complex non-literal object like a Person into the query. Because our scheme does not have associations we will give a somewhat unrealistic example for the moment and then give a better example in a future chapter after we have associations. Say we wanted to find if a person actually is on the server. We do not care about the attributes being the same, we want to see if the actual object is the same. So we might want to ask:

```
    FROM Person person
    WHERE person = :personToFind
```

If this returns an object than the ‘personToFind’ is on the server (and because FORM manages object identity, the returned object will be identical to the ‘personToFind’ object).

5.6 Combining with QueryDescriptions

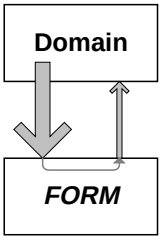
OQL queries can also be combined with functionality that uses the QueryDescription API.

5.7 Summary

FORM’s OQL allows specifying complex queries using a concise string format. OQL is much easier to read and write for queries that are mostly known at compile time. OQL uses all the same concepts as QueryDescriptions: QueryVars, SlotVars, constants, and conditions. OQL adds the concept of BoundValues to handle arbitrary runtime objects that need to be placed in the OQL query.

These two chapters on queries have been introductory but have discussed all the major query topics. FORM provides a complete query model that can be accessed either via an API or via OQL. After we add associations the query system will become very expressive and flexible. It will be able to traverse over several extents and impose restrictions on multiple related objects. This will be discussed in the “Queries-2” chapter below.

6 Domain Object Services to FORM



FORM has the several responsibilities: querying and retrieving objects from the database, saving objects to the database, and translating between the object model and the database scheme. FORM can handle most of each responsibility with the information given to it during configuration, but FORM must interact with domain objects at run-time. DomainObjects need to provide services to FORM during program execution for FORM to do its job.

This chapter will discuss the responsibilities DomainObjects must implement to work with FORM and it will introduce how FORM Tools (the FORM Preprocessor and FORM Abstract Classes) help implement those responsibilities.

6.1 FORM to Domain Object Communication

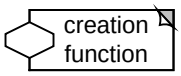
Most of the communication with FORM is in one direction: you configure FORM and you ask FORM to retrieve and modify objects in the database. There are also times when FORM needs to communicate back to your domain. FORM will communicate for

- Construction
  - Construct a DomainObject as a proxy of a database object
- Identification
  - Determine whether an object is a MappedObject
  - Get the ObjectMapper associated with a MappedObject
  - Retrieve the identity key of a MappedObject
  - Give a MappedObject a database identity key
- Manipulation and Inquiry
  - Put state information from the database into a MappedObject
  - Extract state information out of a MappedObject to save it to the database

Your domain must be able to handle these communication requests so FORM can do its job.

Constructing DomainObjects

As we stated in Chapter 2, FORM will automatically build DomainObject replicates whenever it retrieves a database object that is not already in the application. This is accomplished through the creation function given to the ObjectMapper during configuration. This creation function will call the FORM constructor and build a "blank" domain object.



Determining whether an Object is a MappedObject

FORM determines whether a given object is "owned" by FORM by checking whether the object implements the com.chimu.form.MappedObject interface. This interface also defines all the responsibilities your domain object must support to work with FORM.

Implementing the MappedObject interface is usually accomplished by inheriting from one of the DomainObject__AbsClass supplied with FORM. All of the MappedObject methods are implemented in the abstract class, but each domain subclass will override certain methods (i.e. initState and extractState) to include specific subclass information.

```
public interface MappedObject {

    public ObjectMapper  form_objectMapper();

    public Object        form_identityKey();
    public void          form_initIdentity(Object identityKey);
```

```
    public void          form_initState(KeyedArray slotValues);
    public void          form_extractStateInto(KeyedArray slotValues);

}
```

The “form_” prefix on the methods is used to prevent collisions between the FORM MappedObject interface and other methods that are defined in your class. It also makes it apparent that only FORM should be calling these methods and not other clients of your domain class. This is important because Java requires that all methods in an interface are public, so other classes will be able to see and potentially access these methods incorrectly.

Each of the methods in the MappedObject interface corresponds directly to each of the responsibilities that come after “Determining whether an object is a mapped object”. The subsequent sections will discuss them.

Getting the ObjectMapper for a MappedObject

A domain object is always associated with an ObjectMapper, which is responsible for saving and retrieving that object.

```
    public ObjectMapper  form_objectMapper();
```

The standard implementation is to have all the objects of a particular class belong to the same ObjectMapper and to implement this by using the Kernel class information registry.

FORM Support

The above approach is what DomainObject_1_AbsClass provides support for. Each subclass has to link their class object with the desired objectMapper (usually in the “createTheMapper” method) using:

```
        linkClass_toMapper(myClass(),mapper);
```

The rest is handled by the abstract class’s implementation.

Getting the IdentityKey of a MappedObject

All domain objects are either associated with an object on the database or are “new” and only exist in this client application. If a MappedObject is associated with a server object, it will have the IdentityKey of that server object; otherwise, it will have ‘null’ as its IdentityKey value. FORM asks this information via the #form_identityKey method. The usual approach to remembering the database object’s IdentityKey is to dedicate an instance variable to holding the key.

FORM does not care what the actual class of the IdentityKey is. The only requirement is that the identityKey implements a value-oriented #equals. Equals is declared and implemented in Object, but Object has an identity based equals, so whatever subclass is used for the IdentityKey must have overridden Object’s implementation. Other than sending #equals, FORM treats the identityKey as if it were completely opaque.

FORM Support

DomainObject_1_AbsClass has an instance variable named “oid” which is typed as an Object. It also implements the #form_identityKey() method

```
    /*friend:FORM*/ public Object form_identityKey() {
        return this.oid;
    }
```

The only requirement for subclasses is to build an IdentityKey slot with the appropriate database column. This can be accomplished using “buildOidSlotInto_columnName” (defined in DomainObject_1_AbsClass) during mapper configuration.

Giving a MappedObject a Database Identity

If a MappedObject is retrieved from the database, FORM will set the object's identity key immediately after construction. Similarly, as soon as a locally created domain object is written to the database, FORM will set the object's identity key. Both of these operations are accomplished through

```
public void          form_initIdentity(Object identityKey);
```

The implementation of this method should record the identityKey for the current object.

FORM Support

DomainObject_1_AbsClass implements #form_initIdentity to set the 'oid' instance variable.

```
/*friend:FORM*/ public void form_initIdentity(Object identityKey) {
    this.oid = identityKey;
}
```

Initializing the State of a MappedObject

A MappedObject that is retrieved or refreshed from the database will have its state initialized from the information on the database. This is accompolished by

```
public void          form_initState(KeyedArray slotValues);
```

SlotValues can be accessed either by slot name (the preferred approach) or by slot ordinal position. The initState method should call the superclass's initState method and then set all the instance variables from the slots for this particular subclass. An example implementation looks like

```
protected void form_initState(KeyedArray slotValues) {
    super.form_initState(slotValues);
    this.name = (String) slotValues.atKey("name");
}
```

FORM Support

The FORM Preprocessor can generate an initState method that sets all the instance variables of that class and then call the superclass's #initState method. This is discussed further in the next section.

Extracting the State of a MappedObject

When a MappedObject is written to the database, the ObjectMapper will call #form_extractState to ask the object to record all the non-transient information into the slotValues.

```
public void          form_extractStateInto(KeyedArray slotValues);
```

Each class should implement the extractState method to first call the superclass's extractState method and then set the values for the slots with instance variables in this particular subclass. An example implementation looks like

```
protected void form_extractStateInto(KeyedArray slotValues) {
    super.form_extractStateInto(slotValues);

    values.atKey_put("name",this.name);
}
```

FORM Support

The FORM Preprocessor can generate an extractState method that will record all the local instance variables.

6.2 FORM Preprocessor

The FORM Preprocessor is part of the FORM Tool set. It is not a required tool: you can implement all the domain object responsibilities described above by hand. But in most cases, the implementation of the above responsibilities is based only on information that is already available in the Java source files. The **FORM Preprocessor** analyzes the source file and then automatically generates the source for

#form_initState, #form_extractState, #form_creationFunction, and the #myClass function. This saves you time and also guarantees consistency whenever the preprocessor is run.

For complete information on the FORM Preprocessor, please refer to the FORM Tools document.

6.3 PersonClass Implementation

PersonClass needed to implement all the responsibilities listed above. It did this by inheriting from DomainObject_1_AbsClass (through DomainObjectAbsClass), and through FORM Preprocessor generated methods. The preprocessor generated methods included #form_initState, #form_extractStateInto, and #form_creationFunction:

```
    /*friend:FORM*/ public void form_initState(
            com.chimu.kernel.collections.KeyedArray slotValues) {
        super.form_initState(slotValues);

        this.name = (String) slotValues.atKey("name");
        this.email = (String) slotValues.atKey("email");
        try{ this.height = ((Integer) slotValues.atKey("height")).intValue();
            } catch (Exception e) {this.height=0;};
    }
```

```
    /*friend:FORM*/ public void form_extractStateInto(
            com.chimu.kernel.collections.KeyedArray slotValues) {
        super.form_extractStateInto(slotValues);

        slotValues.atKey_put("name",this.name);
        slotValues.atKey_put("email",this.email);
        slotValues.atKey_put("height",new Integer(this.height));
    }
```

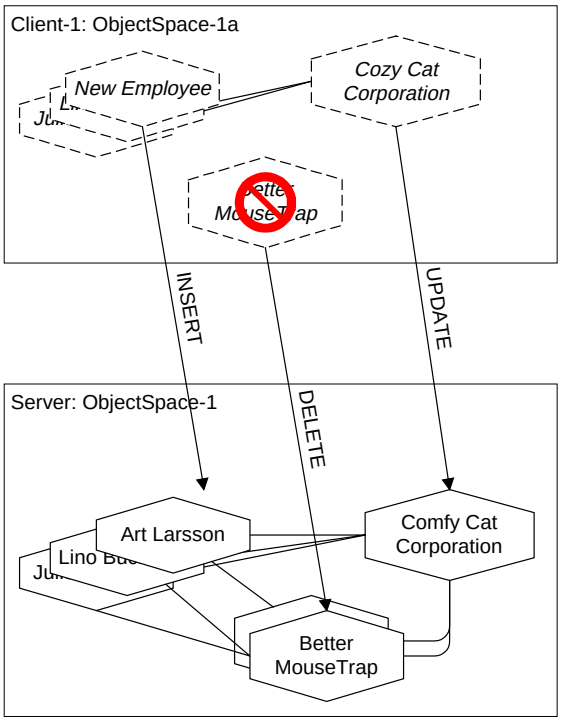
```
    static protected com.chimu.form.mapping.CreationFunction
            form_creationFunction() {
        return new com.chimu.form.mapping.CreationFunction() {
             public com.chimu.form.mapping.MappedObject valueWith(
            com.chimu.form.mapping.CreationInfo cInfo){
                 return new PersonClass();
             }
        };
    }
```

6.4 Summary

FORM requires several responsibilities from domain objects so it can communicate with them during program execution. These services help FORM to construct, identify, inquire, and manipulate the domain objects. All these service are very simple and they can be implemented easily using DomainObject__AbsClass and the FORM Preprocessor.

7 Inserts, Updates, and Deletes

In the previous chapters we discussed how to retrieve objects from the database into the client. This is only one direction of information flow: making the client know the database state. We also need to have functionality that will allow a client to change the database: to make the database know the client's new state. These state changes include: **inserts**, telling the database a new object was created; **updates**, telling the database an old object was modified; and **deletes**, telling the database an old object has been removed from the model. All of these changes are really "requests" because our client is not alone and other clients may have made changes to the server that conflict with our desired changes[22].



This chapter will describe how FORM can automatically manage these change operations and what information you can configure FORM with to work with your database. FORM needs to be flexible when changing the database state: the database is in control, it can be set to manage changes to its data (triggers and stored procedures), and each database has different functionality. FORM needs to be able to work with the database for you to best model you information and best use you database's features.

7.1 Domain Model-1b

This chapter will continue to use the same basic Person Domain Model and Database Scheme for examples with only a couple changes. The only external change between DomainModel-1 and DomainModel-1b is the addition of the DomainObject interface that was suggested in an earlier chapter. The Person interface now inherits from DomainObject which provides Person with the new methods of #write and #delete as well as defining #info and #copy more generally. The implementation in PersonClass is modified to support identity generation, which will be discussed in an upcoming section.

Database Scheme-1b

The database scheme for Person was not changed for this DomainModel-1b but we added a new table to the scheme to support identity generation: IdentityGenerators. This will help with a database independent solution to identity. A database specific approach could use other product specific features and may not require this table. This will be discussed in an upcoming section.

7.2 Example: Update_1

The first example will be updating an object. In this example, we retrieve a person object from the Person Extent. After making some modifications to the state of the object, we write the object back to the database. Because the object was replicated from the database the #write message send will cause an update of the database information and not the creation of a new object.

[22] If sufficient locking has been done (or we are the only active client) and we haven't violated the servers integrity rules then the request will almost certainly be accepted. The important point is that the server is an active participant and gets the final say on what is acceptable for its ObjectSpace.

```
   public void run (Connection jdbcConnection) {
        //-------Configuration--------
…….
        //-------Creating Query--------

        Person aPerson = (Person) personRetriever.findAny();
        outputStream.println(aPerson.info());
        aPerson.setHeight(60);

        aPerson.write();

        outputStream.println(aPerson.info());
    }
```

Does a domain object (like Person) know how to write itself to the database? The answer with FORM is no and yes. FORM uses ObjectMappers to accomplish writing objects to the database, but this functionality should usually be known only within you domain object. It is encapsulated within the implementation of the object.

A domain object should be responsible for managing all aspects of its own state: it should know how to delete, update, and insert itself into permanent storage. FORM supports this model through abstract class functionality that is available to your domain object. These write methods immediately delegate to the ObjectMapper to do the actual save operation, but that implementation is encapsulated and irrelevant to the client of the domain object.

7.3 DomainObject Interface and DomainObject__AbsClass

If you decide to have the domain object be responsible for writing and deleting itself, FORM provides the two components that define and implement the needed protocol, the DomainObject interface and the DomainObject abstract class that you have seen repeatedly.

DomainObject Interface

In an earlier chapter we mentioned that you might want to have a general DomainObject interface that describes the responsibilities which all domain objects will support. FORM provides a DomainObject interface (in the com.chimu.form.client package) that defines some FORM related public protocol that is useful to have on all DomainObjects. The two methods important to our current discussion are #write and #delete.

```
        /**
         * Write will either insert or update the object
         * to the database depending on the whether it
         * is a new object or a database replicate.
         */
    public void write();

        /**
         * Delete removes the domain object from the database.
         */
    public void delete();
```

By extending or implementing the FORM provided DomainObject interface you allow other objects to call #write and #delete on your domain objects. The most flexible approach would be to extend this interface in your own DomainObject interface

```
public interface DomainObject
                    extends com.chimu.form.client.DomainObject_1 {
}
```

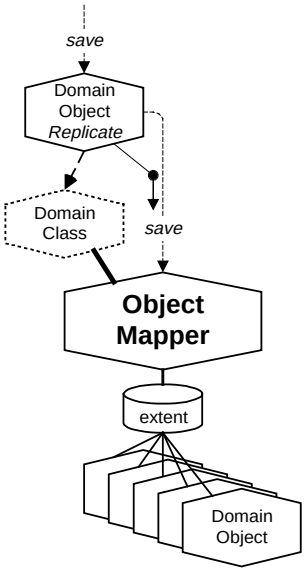
Which will allow you to add other interface methods that your application requires from all of its domain objects. For our example that might be the #info method on Person.

DomainObject__AbsClass

Now that we have committed to supporting #write and #delete, we have to implement them. The standard com.chimu.form.client.DomainObject_1_AbsClass implements these methods to do the delegation to the ObjectMapper, so if you extend from it the behavior is automatically available.

7.4 ObjectMappers as ObjectSavers

Now that the domain object has delegated to the ObjectMapper we need describe the ObjectMapper protocol. So far we have used ObjectMappers only as ObjectRetrievers. ObjectMappers are also "ObjectSavers". They handle writing and deleting individual objects for a particular Domain Class into the database extent. Each DomainObject replicate will have a single ObjectMapper for its entire life and this mapper will be responsible for synchronizing the replicate's state with the database state.



An ObjectMapper provides the protocol: #insertObject, #updateObject, and #deleteObject that change the state of the server appropriately. A DomainObject would know whether it needs to insert or update depending on whether it is linked to a server object by an IdentityKey. So these two operations were simplified into a single #write on the domain object interface. The #delete method corresponds directly to a #deleteObject for the ObjectMapper.

FORM provides basic capabilities for communicating state changes to the server and works with the server to provide referential integrity for those changes. [23]

7.5 Example: Delete_1

```
Person aPerson = (Person) personRetriever.findAny();
aPerson.delete();
```

7.6 Example: Insert_1

To insert an object in the database we simply create a new object and then write it.

```
String  name = "Mary Houdini";
String  email = "houdima";
int     height = 60;

Person newPerson = new PersonClass(name, email, height);

newPerson.write();
```

Although the process of writing a new object is simple, FORM needs extra information to know how to accomplish the server object creation. All database objects must have an identity, and FORM must know how to generate the identity of a newly created Server object. There are many mechanisms that can be used to create Server objects, which each have different characteristics for performance and responsibility distribution.

7.7 Types of Identity Generation

There are several ways an object's IdentityKey can be generated which depend on the timing of the generation and whether the client or the server is responsible for the generation.

[23] You should see the chapter "Extending FORM" for discussion of more sophisticated state management.

During-Insert, Server Generated

The server could automatically generate a unique IdentityKey when a new row is inserted. This could be done by a special data type (e.g., Unique Identifier Columns for MS SQL Server) or by database logic (e.g., row level triggers in Oracle). For each of these cases the identity is generated during the insert process and has to be retrieved either at that same time or very shortly there after.

Pre-Insert, Server

Another option is that the server can generate IdentityKeys at command and the client application should take the generated key and use it in the to be inserted object. There are two variations within this generation approach. The server must have enough programming capability (e.g. stored procedures or read triggers) or a particular feature (e.g., sequences) to automatically generate new unique values.

Pre-Insert, Client-Server

A third option is that the server provides a central key repository but that the client has the logic for fetching *and* updating the key value. This requires less sophistication by the server but means clients must control locking to make sure the integrity of unique values is maintained.

Pre-Insert, Client

The final option we will discuss is that the client can generate a key completely on its own. This is still done before inserting into the database, but the server does not need to be involved until the insert itself.

7.8 Example Identity Generation

Pre-Insert generation is more flexible and database independent than the other approaches, so we will chose it for the examples. You can generate the identity in many ways and decide whether the client or the server will do the work. In exchange for this flexibility you must tell FORM how to generate an IdentityKey. This is accomplished with:

```
mapper.setupPreInsertIdentityGenerator(identityGenerator);
```

Where identityGenerator implements the interface “com.chimu.kernel.streams.Generator”. This interface is very simple, it is similar to an Enumeration, and FORM will simply call #nextValue each time it needs a value from the generator. For our example we assume a simple, database independent approach to generating the IdentityKey: we will record the IdentityKey information in a database table and then use and update that table whenever a new object is saved.

To support this new configuration information we will need to augment our DomainObject_FormInfo. We can specify to use ‘PreInsertIdentityGeneration’ in the #configureMappers method.

```
public void configureMappers() {
    super.configureMappers();

    myMapper.setupPreInsertIdentityGenerator(identityGenerator());
}
```

Next we need to provide the identityGenerator used in the setup method. Again, for our example we will assume a client-server approach where a database table will hold IdentityKey counters and the client will fetch and update the counter as needed. This approach is very simple, works with any database, is guaranteed to be unique with proper locking, and can be tuned to require few database hits. FORM provides the following function to create this type of generator:

DatabaseSupportPack

```
public Generator createStandardTableGenerator(
                        Table table, Object generatorIdentifier,
                        String counterColumn,
                        int blockSize);
```

This requires a Table that has an IdentityKey column and one other column that will hold numeric counter values. Our example table definition is shown to the right.

GeneratorCounters

ID: varchar
Counter: int

So all we need is a method that builds and remembers the generator. The following method defines the two-column table, decides on the block size, creates the generator, and remembers it in a static variable.

```
static public void buildIdentityGenerator(DatabaseConnection
            dbConnection) {
        if (identityGenerator != null) return;

        Table table = dbConnection.table("GeneratorCounters");
            Column primaryKey = table.newBasicColumnNamed("ID");
            table.setupPrimaryKeyColumn(primaryKey);
            table.newBasicColumnNamed("COUNTER");
        table.doneSetup();

        identityGenerator =
           DatabaseSupportPack.createStandardTableGenerator(table,...);
    }
```

We store a single generator in a static variable because all objects will be using the same generator. This means objects will have both class-unique and globally unique identity keys. So all we need now is to return the created identityGenerator when asked and to have the main program build the generator during configuration.

```
    static protected Generator identityGenerator() {
        if (identityGenerator == null) throw new RuntimeException("Need
           to build the identity generator");
        return identityGenerator;
    }
```

```
        /**
         * prepareForMapping lets DomainObjectAbsClass know about the
           database
         * connection an prepare for having ObjectMappers created
         */
    static public void prepareForMapping(Orm orm, DatabaseConnection
           dbConnection) {
        buildIdentityGenerator(dbConnection);
    }
```

ExampleAbsClass

```
    protected void createAndConfigureOrm(Connection jdbcConnection) {
        createOrm(jdbcConnection);
            DomainObjectAbsClass_FormInfo.prepareForMapping(
                   orm,dbConnection);
            orm.addInfoClass_withDb(PersonClass_FormInfo.class,
                   dbConnection);
        orm.doneSetup();
    }
```

7.9 Example: Write_1

After the information about Identity is added to the configuration, both Inserts and Updates happen automatically when an object is sent #write. An Update occurs if the object already exists, and an Insert occurs if the object is new:

```
        Person newPerson = new PersonClass();
        newPerson.setName("Kity Purrbox");
        newPerson.setEmail("purrboki");
        newPerson.setHeight(34);

        Person updatePerson = (Person) personMapper.findAny();
        updatePerson.setHeight(60);

        updatePerson.write();  //inserts to the database, since it's
           new.
```

```
        newPerson.write();  //updates the database, since the row
           already exists.
```

7.10 Summary

This chapter described how a client can change the state of the database through inserts, updates, and deletes. All of these are the responsibility of DomainObjects, which work through the ObjectMapper to effect the change. Deletes and Updates have default behavior, which requires no extra configuration beyond the mapping information for "retrieval". Inserts require telling FORM how you want server IdentityKeys generated, but after that FORM will automatically generate an IdentityKey when it is needed.

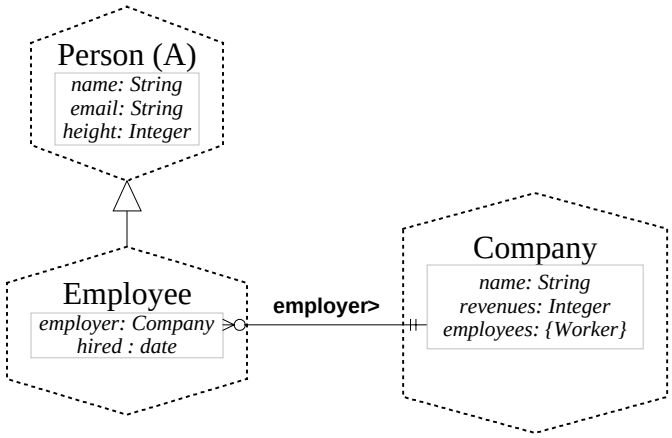
8 Associations

This chapter introduces how FORM maps associations between classes into relationships between tables. These are similar but not identical concepts because there are many variations on association and relationship semantics and the relational model provides flexibility in implementing an association which has more to do with performance than modeling semantics. FORM tries to generalize the different types of associations (as well as attributes) into the unifying concept of a Slot and then provides flexibility in how that general AssociationSlot is implemented on the database.

The previous chapters kept to a single class so the discussion would be simpler and because the concepts are independent of the complexity of the model. But it is now time to start building a real DomainModel with many related classes. As each sophistication is added, the example DomainModel will be enhanced. As this chapter focuses on the different types of associations that can be used, new classes will be included that require the type of association.

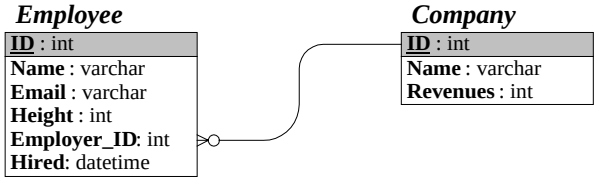
8.1 Domain Model-2

The first new domain model will focus on two new types, 'Employee' and 'Company', which have an 'employee-employer' association. Although we could model and implement these types "from scratch", we will instead take advantage of the similarity between our new Employee and Person from our Domain Model-1. Employee and Employee class will extend from Person and PersonAbsClass (renamed because it is now an abstract class) respectively.



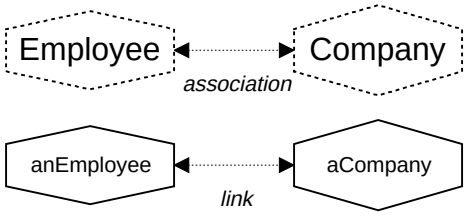
Database Model

The new database model uses two tables to describe the two classes, and it records the employee-employer relationship by a foreign key embedded in Employee.



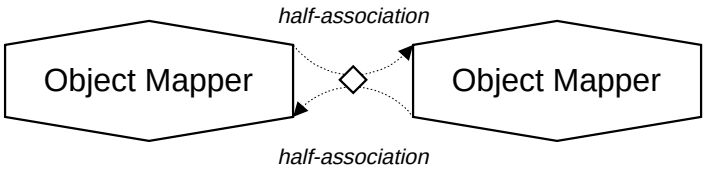
8.2 Associations

An association is a defined relationship between two[24] object types. An association defines the possibility of two objects (one of each type) being "linked" to each other. This is only appropriate between two objects that both have identities: A Person can be associated with a Company, but a Person holds onto its email address String as a direct value, not an association.



8.3 Association Slots

FORM uses Slots to describe associations. A Slot represents a stored attribute of an Object whether it is a basic attribute or an association attribute. DirectSlots are used for basic attributes and **AssociationSlots** are used for associations. Each AssociationSlot describes the association from only one of the two linked-object's point of view. Together the two AssociationSlots and the two ObjectMapper's know of the full relationship.



Properties of Association Slots

Each associationSlots has a **partner**, the ObjectMapper on the other side of the association.

AssociationSlots are **traversable**: you can move from one side of the association to the value or values on the other side. Some AssociationSlots have a cardinality of at most "one" and will return either an object or a null as their traversal value. Other AssociationSlots have a cardinality of possibly "many" and will return a collection as their traversal value.

AssociationSlots can contain their partner, in which case the partner will be deleted when the object is deleted.

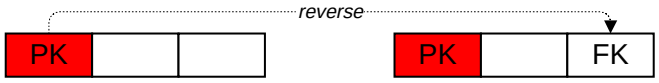
Association Slot Types

There are three different types of AssociationSlots to handle the three different ways an association can be stored on a relational database:setting up the mapping to a particular database requires accounting for the different ways associations can be stored. However, from an external point of view (i.e. the domain) these AssociationSlots will appear identical; their only differences will be from restrictions in cardinalities.

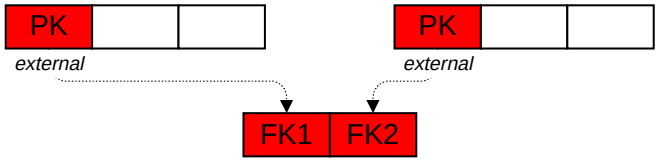
ForwardAssociation — Your table has a foreign key that points to your partner's table's primary key

[24] Or more, but for the discussion we will assume only two.

ReverseAssociation — Your partner's table has a foreign key that points to your table's primary key



ExternalAssociation — A third table contains two foreign keys that correspondingly point to your primary key and your partner's primary key



Each of these association forms can generally record similar information, but they have different cardinality constraints, performance characteristics, storage costs, and updating isolation.

ForwardAssociation	0..N	0 | 1	Restricted cardinality. Good storage if nulls are not allowed (in which case the cardinality is 0..N to 1)
ReverseAssociation	0 | 1	0..N	Restricted cardinality. No storage costs on our table.
ExternalAssociation	0..M	0..N	Most general cardinality. Requires an extra table. Good in terms of storage. Good for most types of updating. Has performance implications; requires an extra join.

Association Slot Pairings

Forward and Reverse Associations are paired: If one side of an association has a ForwardAssociation then the partner will have a ReverseAssociation. ExternalAssociations are paired with themselves: If one side of an association has an ExternalAssociation then the partner will also have an ExternalAssociation.

8.4 Example: EmployeeRetrieval_1

The EmployeeRetrieval example does not appear to be very different from our original PersonRetrieval query. We changed from Person and PersonClass to Employee and EmployeeClass. We also use the method #employer which is available to an employee. Other than those changes, the code is the same.

```
public class EmployeeRetrieval_1 extends FormDatabaseTestAbsClass {

    public void run (Connection jdbcConnection) {
        //-------Configuration--------
        //-------Running--------
        Employee employee = (Employee) employeeRetriever.findAny();
                outputStream.println(employee.info() + " is employed by
           " + employee.employer().name());
    }
}
```

Although the main application is almost identical, there are significant changes within the domain classes themselves. In Domain Model-2, employees are related to companies (an employee knows who his/her employer is and vice-versa). This association between the EmployeeClass and CompanyClass needs to be configured into the ObjectMappers. Because of the cardinality and the desired database model, this association is configured with a ForwardAssocationSlot for the Employee mapper and a ReverseAssociationSlot for the Company mapper.

8.5 Forward Associations in a Mapping

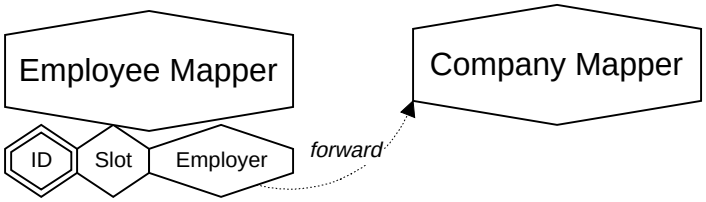
To create an association requires two mappers that are to be associated[25]. For Employee, the Company mapper must be available during configuration and for Company the Employee mapper must be available. This circularity is resolved by using multiple stages. First, all ObjectMappers/ObjectRetrievers are created (in FormInfo#createMappers), and then they are all configured (in FormInfo#configureMappers). The createMappers is the same with associations as it was without them for Person. But the configureMappers method for EmployeeClass_FormInfo now uses association slots:

```
public void configureMappers() {
   super.configureMappers();

    myMapper.newForwardSlot_column_partner("employer",
            "employer_ID",
            orm.retrieverNamed("Company"));
    myMapper.newDirectSlot_column_type("hiredDate",
       "hired_date",Date.class);
}
```

Because EmployeeClass_FormInfo inherits from PersonAbsClass_FormInfo, it can first take advantage of the abstract classes configuration of basic slots just by calling "super.configureMappers()". Then it configuration the mapper with Employee specific information. The interesting new slot is the "employer" slot, which is a ForwardAssociationSlot.

A **ForwardAssociationSlot** is created like a DirectSlot but takes one more parameter: the corresponding partner for the association. In this example, the slot is named "employer" and a foreign key to the employer primary key is stored on the database in the column "employer_ID". The association with CompanyClass is through the companyMapper, which we can retrieve by asking the 'orm'. We need not specify anything else because the companyMapper itself knows the rest: in which table companies are stored and how their identities are determined.



The reason we used a ForwardAssociation is solely because of how the association was stored on the database, as a foreign key pointing to a Company primary key. This does not impact the user of the Slot (it is just like any other AssociationSlot), but it does constrain cardinality: a Person can have at most one employer at any given time, and a company can have many employees at any given time because it uses a ReverseAssocation.

8.6 Reverse Associations in a Mapping

Company can not have a ForwardAssociation to Employee because the database scheme does not support a ForwardAssociation (no foreign key in the Company table). In addition, each Company must be able to have multiple employees so we cannot change the database scheme. Instead Company will use a **ReverseAssociationSlot** that links back to the EmployeeClass's ForwardAssociationSlot.

The following code for CompanyClass_FormInfo shows the configuration of the company mapper.

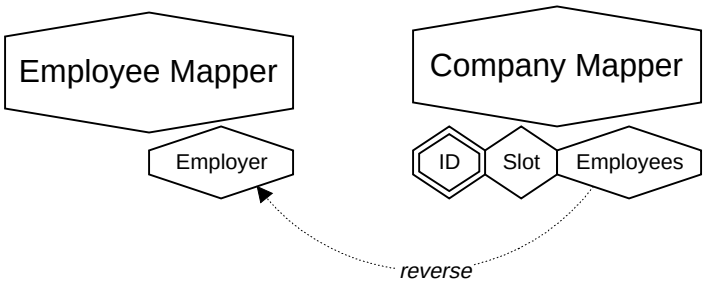
```
public void configureMappers() {
   super.configureMappers();

    myMapper.newDirectSlot_type("name",String.class);
    myMapper.newDirectSlot_type("revenue",BigDecimal.class);
    myMapper.newReverseSlot_partner_partnerSlot("employees",
            orm.retrieverNamed("Employee"),"employer");
```

[25] In some cases the two mappers can be the identical mapper (e.g., a parent-to-child relationship), but usually they involve a different type of mapper.

```
    }
```

Creating a ReverseAssociationSlot is similar to a ForwardAssociationSlot but has two differences. First, it does not need a database column because a ReverseAssociation does not store a foreign key value in the table to find its partners: it uses its own the primary key value. Second, a ReverseAssociation requires the name of the ForwardAssociationSlot with which it is paired. Knowing the name of its ForwardAssociation allows it to determine the rest of the relationship information (including the fact of it being an employer-employee relationship). For our example, the name of the slot is "employees", the partner is employeeMapper, and the partner's Forward slot name is "employer".



The "reverseness" of the slot is obvious from its construction. The ReverseAssocationSlot can only be created by reversing the information from a ForwardAssocationSlot: employees are determined by employer.

Cardinality

There is no inherent cardinality limit for a ReverseAssociation, so we can choose whether a Company should have, at most, a single employee or many employees. By default it will be many employees and the "employees" slot will return a collection. If we wanted to change it we would use:

```
        mapper.newReverseSlot_partner_partnerSlot("employees",
                employeeMapper,
                "employer"
            ).setupIsOneToOne();
```

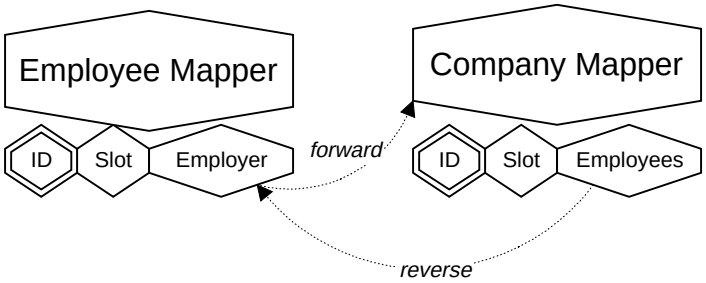
The #setupIsOneToOne() method would make the "employees" slot (better named "employee" after the change) return a single object, or null.

8.7 Example: CompanyRetrieval_1

```
        Company company = (Company) companyRetriever.findAny();
        Collection employees = company.employees();
        outputStream.println(company.info() + " employs " +
           employees.size()+" employees");
```

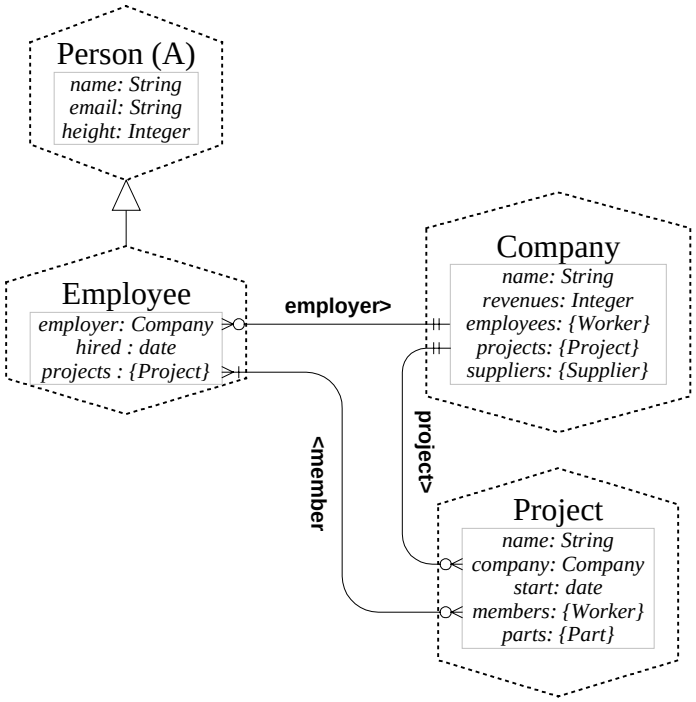
Asymmetry between Forward and Reverse Associations

A slight asymmetry exists between the Forward and Reverse associations. A Forward association does not require a reverse association to function properly, but a Reverse association requires a Forward association. Having the association does not mean you have to use it (a Person may not care who their employer is), but the mapping information must be specified.



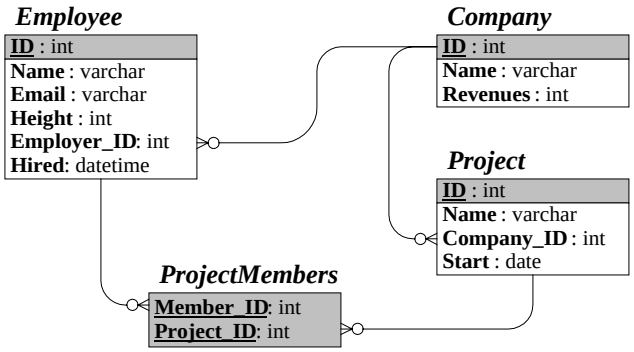
8.8 Domain Model-3

To cover the last type of association we will use a new domain model that includes Projects in addition to Companies and Employees. A given company can have multiple projects, which are staffed by employees from that company. An employee can be on zero, one, or multiple projects at any given time. We model this by having Employee hold a collection of Projects, Projects holds a collection of employees, and Company contains a collection of Projects.



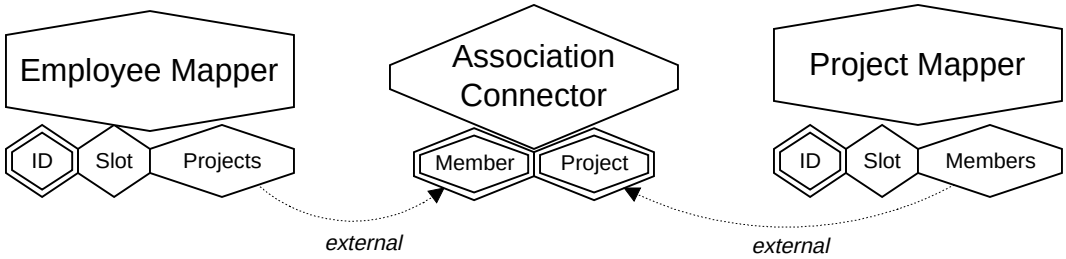
Database Model

The database model is similar to the object model, but the many-to-many association between Employee and Project is captured through an external association table. For each project member a row is added to the ProjectMembers tables.



8.9 External Associations

To represent the Employee-Project association we use an **ExternalAssociationSlot**. An ExternalAssociationSlot looks almost the same as a ReverseAssociationSlot except that instead of hooking directly to the partner mapper, it hooks to an intermediate AssociationConnector.



To create an external slot you specify the association connector, which association element you are, and (optionally) which association partner you want to traverse to. In the case of a binary association you can leave off the partner's slot name because it can be derived.

For our new EmployeeClass_FormInfo, the configuration has changed to:

```
    public void configureMappers() {
        super.configureMappers();

        myMapper.newDirectSlot_column_type("hiredDate",
           "hired_date",Date.class);

        myMapper.newForwardSlot_column_partner("employer",
                "employer_ID",
                orm.retrieverNamed("Company"));

        myMapper.newExternalSlot_connector_mySlot("projects",
                orm.associationConnectorNamed("projectAndEmployee"),
                "employee");
    }
```

Again, the external association slot looks identical to a reverse association slot. The only difference is that an association connector is used instead of the partner mapper. The naming still looks reversed: 'projects' connects to the slot 'employee' in the connector.

On the other side of this association, ProjectClass_FormInfo will also have an external association slot, which mirrors the EmployeeClass_FormInfo's external association slot.

```
    public void configureMappers() {
        super.configureMappers();

        myMapper.newDirectSlot_type("name",String.class);
        myMapper.newDirectSlot_column_type("startDate",
           "start_date",Date.class);

        myMapper.newForwardSlot_column_partner("company",
                "company_ID",
```

```
                orm.retrieverNamed("Company"));

        myMapper.newExternalSlot_connector_mySlot("employees",
                orm.associationConnectorNamed("projectAndEmployee"),
                "project");
    }
```

Now we only have to describe what an AssociationConnector is.

Association Connectors

AssociationConnectors are used to describe an external association table within FORM. Connectors know how to interpret the columns on the association table and they provide the placeholders for ObjectMappers to connect to. Creating AssociationConnectors is very similar to creating ObjectMappers. You first create a connector with a database table and then you configure the connector to have slots for each of the association partners.

```
    public void createMappers() {
        orm.newAssociationConnectorNamed_table("projectAndEmployee",
           dbConnection.table("ProjectMembers"));
    }

    public void configureMappers() {
        AssociationConnector connector =
           orm.associationConnectorNamed("projectAndEmployee");

        connector.newConnectorSlot_column("employee", "employee_ID");
        connector.newConnectorSlot_column("project", "project_ID");
    }
```

Who is responsible for creating the AssociationConnector? Where does it belong? We could put it in either of the domain classes (EmployeeClass_FormInfo or ProjectClass_FormInfo) or we could place it outside of both classes and group multiple associations together. For the example domain model we choose the later: all AssociationsConnectors (in this case one) are placed in AssociationsFormInfo. This allows us to see all the associations in one place and makes the fetching of the association identical for both of the associated classes.

Example Configuration

The configuration for Scheme3 includes the new class ‘Project’ and the extra AssociationsFormInfo configuration information.

```
    createOrm(jdbcConnection);
        DomainObjectAbsClass_FormInfo.prepareForMapping(orm,dbConnection);
        orm.addInfoClass_withDb(EmployeeClass_FormInfo.class,dbConnection);
        orm.addInfoClass_withDb(CompanyClass_FormInfo.class,dbConnection);
        orm.addInfoClass_withDb(ProjectClass_FormInfo.class,dbConnection);
        orm.addInfoClass_withDb(AssociationsFormInfo.class,dbConnection);
    orm.doneSetup();
```

All the examples can now take advantage of the fuller model, which uses n-m cardinality ExternalAssociations.

8.10 Summary

Associations are one of the core concepts to object-oriented modeling, so it is crucial that they can be stored in a relational database. Relational databases provide three different approaches for recording associations between entities: Forward Associations, Reverse Associations, and External Associations. FORM can use all of these storage approaches to model an object association, and is only restricted by the cardinality constraints of the different approaches. Forward and Reverse associations are complementary and can model N-1 and 1-N associations respectively. External associations can model N-M associations and use an AssociationConnector to “hook up” between the two classes participating in the association. All associations give our Domain Model much greater expressiveness for queries, which we can now revisit in the next chapter.

9 Queries-2

Now that we have multiple associated classes in our domain model we can perform more sophisticated queries. This chapter will describe the fuller capabilities of FORM queries when used with association Slots.

9.1 Association Slots

In many ways association slots are just like any other slot. We can use them to create QueryVars which are then used slot in slot conditions and return values. For example, we can do a search for all employees that are employed by a particular company by the following[26]:

```
QueryDescription query = orm.newQueryDescription();
QueryVar employeeVar= query.newExtentVar(employeeMapper);

query.setCondition ( query.newEqualTo(
        employeeVar.slotNamed("employer"),
        query.newConstant(company)
     ));

//----------Running-----------
Collection employees = query.selectAll();
```

There are two distinctive properties of association slot variables. (1) Associations lead to other mapped objects with their own slots. These slots can also be traversed in an endless chain of associated objects. (2) Associations can lead to multiple partner objects, so for a given variable value (e.g. a company) multiple partner values could be available to the associated variable (e.g. all the employees for the company).

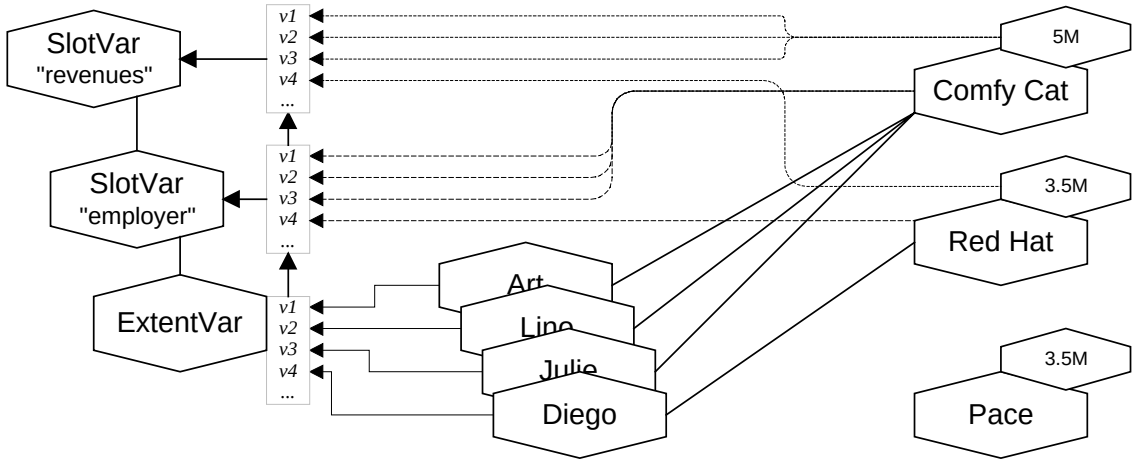
Multiple Traversals

As mentioned above, an association slot query variable leads to another mapped object, which has its own slots. This means you can form a chain of slot traversals. To find all the employees that work for a company with revenues greater than 2,000,000 we would use a query of:

```
"SELECT employee
 FROM Employee employee
 WHERE employee.employer.revenues > 2,000,000"
```

This query would iterate through all the employees in the EmployeeMapper extent. For each employee it would find that employee's employer, and then find that employer's revenues. This value would be compared to the constant value specified.

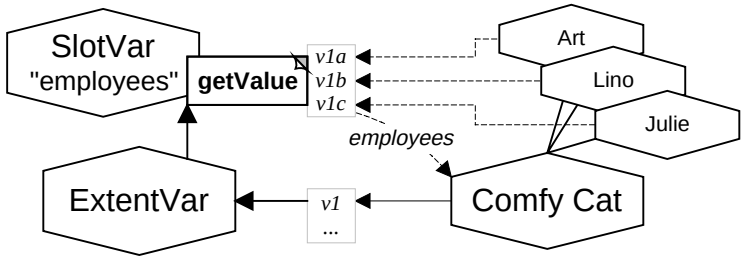
employee.employer.revenues



[26] You are more likely to just ask the company for its employees "company.employees()" in this particular case.

Multiple Partners

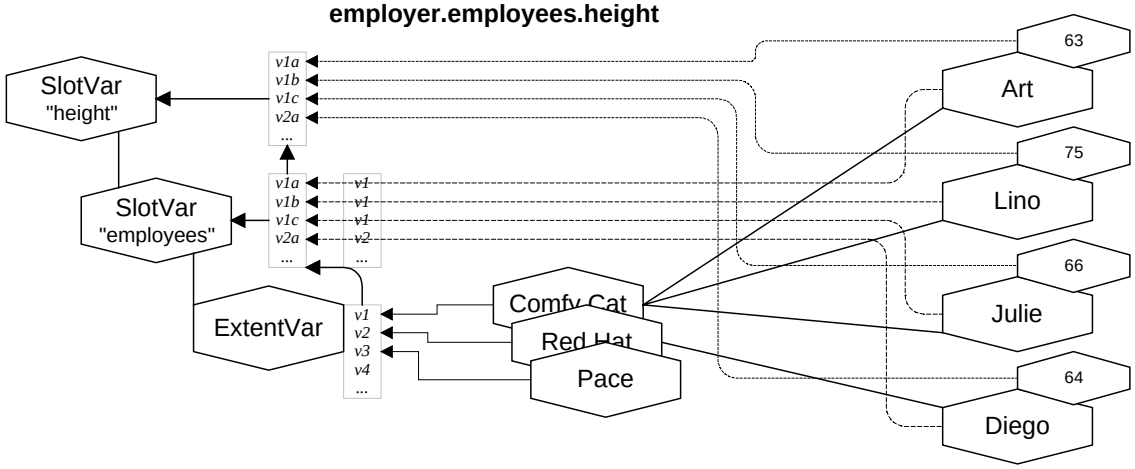
In the above example the association query variable behaved just like a basic attribute slot variable. For any given subject variable value there is only one derived slot variable value. This is not always the case with associations. An association can have multiple partners, which means that the derived variable will take on multiple values for any given subject value. For example, the "employees" slot variable will take on three values when the employer variable is equal to 'Comfy Cat'.



So if we ask the question

```
"SELECT employer
 FROM Employer employer
 WHERE employer.employees.height > 63"
```

we will look at each of the three employees of Comfy Cat before deciding whether Comfy Cat qualifies, and then the one employee of Red Hat to decide whether Red Hat qualifies, and then ignore Pace because Pace has no employees, and so on through all the companies and all the employees associated with those companies.



9.2 Connected and Unconnected Query Variables

When building complex queries involving association variables you need to consider whether you want to have two independently varying variables or a single variable. Take for example:

```
"SELECT employer
 FROM Employer employer
 WHERE employer.employees.height > 63
   AND employer.employees.name = Art"
```

We might assume this query would not return any companies because Art is not taller than '63' inches, but it would instead return 'Comfy Cat'. Each time you refer to "employer.employees" you are *creating a new query variable* that varies over multiple employees. The above query is asking for "an employer who has an employee who is taller than 63 inches and has an employee who is named 'Art' ", it is not asking for "an employer who has an employee who is taller than 63 inches and is named 'Art' ". To ask that question we need to reuse the same query variable for both conditions.

Explicitly Named OQL Query Variables

We have already seen explicitly named query variables in OQL: the Extent variables are explicit. We can also explicitly name derived query variables by putting them in the FROM clause of the OQL query. If our above query example was changed to:

```
"SELECT employer
 FROM Employer employer, employer.employees employee
 WHERE employee.height > 63
   AND employee.name = Art"
```

It would now return the answer we originally expected: no companies. If we wanted to produce exactly the same query as the original but with explicitly named variables we would instead use:

```
"SELECT employer
 FROM Employer employer, employer.employees employee1,
       employer.employees employee2
 WHERE employee1.height > 63
   AND employee2.name = Art"
```

Which makes it more visible why 'Comfy Cat' satisfies this query.

Reusing Query Variables in the Query API

When using the direct QueryDescription API we have the same issues as for OQL and similar solutions. If we want to have two independent QueryVars we create them with two different 'slotNamed' messages. If instead we wish to have the same QueryVar in multiple places, we just have to save the QueryVar in a variable and reuse it.

10 Inserts, Updates, and Deletes – 2

Now that we have covered associated objects we can consider the issues for saving multiple and related objects to the database. These issues can be divided into general issues involving transactions and issues with handling associations among objects.

10.1 Transactions

Transactions provide the ability to have multiple database changes succeed or fail as one unit. For example, we can decide to have all the changes made to a particular "window" (a user's conceptual unit of work) count as a single unit. To make this happen we will start a transaction when the window has opened and will not commit the changes until the user has finished and said "OK" or "Cancel".

Java's JDBC and most databases can work in either of two transactional modes: Either there are no explicit transactions and each database statement will be committed automatically, or all database changes are accumulated within a transaction until an explicit commit is made. In the previous chapters we have been assuming statement level transactions, so the FORM client would not have to do anything special. If a client wants to use a larger unit of granularity they can either control the JDBC Connection's transaction directly or use FORM's TransactionCoordinator to help with the process. All of FORM's transaction functionality is in the 'com.chimu.form.transaction' package.

TransactionCoordinator

A **TransactionCoordinator** has very simple functionality: it takes a block of code (a Procedure0Arg) and executes the code within a database transaction. The coordinator handles beginning and committing a transaction and will automatically rollback the database if any errors occur. The TransactionCoordinator allows you to specify transactions by grouping code together that should be executed in that transaction and you only have to deal with the application recovery if a **TransactionException** is thrown.

You get a TransactionCoordinator from a DatabaseConnection using #transactionCoordinator()[27]. You then tell the Coordinator to execute a block of code using #executeProcedure(Procedure0Arg). The #executeProcedure method may throw a TransactionException which must be caught and can be used to determine what failed in the transaction.

```
TransactionCoordinator tc = dbConnection.transactionCoordinator();

try {
    tc.executeProcedure(aProcedure);
} catch (TransactionException e) {
    // need to recover from failure
}
```

The method #executeProcedure takes a Procedure0Arg as an argument. Sometimes this procedure will only be needed in the specific context, so an anonymous class can be used and the specifics of the transaction operation "inlined". For example:

```
TransactionCoordinator tc = dbConnection.transactionCoordinator();

try {
    tc.executeProcedure(
        new Procedure0Arg() {public void execute(){
            anEmployee.write();
            anotherEmployee.delete();
        }}
    );
} catch (TransactionException e) {
    // need to recover from failure
}
```

In other cases the transaction procedure may be very complicated and the Procedure0Arg object will be a full named class with multiple methods.

[27] By default a given DatabaseConnection has only a single coordinator because it has only a single JDBC Connection.

Controlling Transaction Isolation

By default a TransactionCoordinator uses the current transaction isolation level of the JDBC Connection[28]. If you want a different isolation level within the transaction you can tell the Coordinator what it should be before starting a transaction. The method #setupTransactionIsolationLevel takes a JDBC transaction level that will be enforced during the transaction.

```
TransactionCoordinator tc = dbConnection.newTransactionCoordinator(
          jdbcConnection);
tc.setupTransactionIsolationLevel(
          Connection.TRANSACTION_READ_UNCOMMITTED);
```

If the transaction level is not available on the particular database product than an exception will be thrown at 'setup'. You can verify that the database supports a particular isolation level using:
DatabaseConnection#supportsTransactionIsolationLevel(int isolationLevel)

TransactionCoordinator stages

After the transaction has completed, the JDBC Connection will be returned to its pre-transaction state. The series of stages for the TransactionCoordinator are:

1. Record the state of the JDBC Connection
2. Prepare the JDBC Connection's IsolationLevel and begin a transaction
3. Execute the transaction procedure and catch any errors
4. Either rollback or commit depending on whether there were errors
5. Restore the state of the JDBC Connection to (1)

TransactionException

A TransactionException is thrown whenever a transaction has failed. Failures are primarily caused by other, lower-level exceptions, and the low-level exception may be needed to understand what recovery should be attempted. To provide this recoverability, TransactionExceptions are "WrappingExceptions", they may wrap another exception which is the real cause of the failure. See the API for "kernel.RuntimeWrappedException" and the documentation in [Kernel] for more information.

Domain Model-3b

The object model and relational model in Scheme3b are exactly the same as those in Scheme3. The difference between the two sets of examples is that Scheme3b uses explicit TransactionCoordinator transactions. To do this we modified the #createandConfigureOrm() function in ExampleAbsClass to create the TransactionCoordinator:

```
    protected void createAndConfigureOrm(Connection jdbcConnection) {
        createOrm(jdbcConnection);
            createTransactionCoordinator();
            DomainObjectAbsClass_FormInfo.prepareForMapping(orm,
                  dbConnection);
            orm.addInfoClass_withDb(EmployeeClass_FormInfo.class,
                  dbConnection);
            //...
        orm.doneSetup();
    }
```

We also added the procedure to create and remember the TransactionCoordinator:

```
    protected void createTransactionCoordinator() {
        tc = dbConnection.newTransactionCoordinator();
        tc.setupTransactionIsolationLevel(
            Connection.TRANSACTION_READ_UNCOMMITTED);
    }
```

[28] If you have not specified a TransactionIsolation level for the JDBC connection it will have been set to a database and driver specific default value.

Example: EmployeeInsert_1

Now, if we want to create and add a new Employee within a transaction we can do the following:

```
final Employee newEmployee = new EmployeeClass();
newEmployee.setName("NewEmployee");

try {
    tc.executeProcedure(
        new Procedure0Arg() {public void execute(){
            newEmployee.write();
        }}
    );
} catch (TransactionException e) {
    outputStream.println("Error enoucntered.  Could not write
       employee: "+ newEmployee + " to the database");
}
```

All the operations we are interested in doing as one unit should be placed within the same procedure object and we can guarantee that they will all be written successfully together or the database will be unchanged.

Example: EmployeeInsert_2

For example, instead of writing a single DomainObject at a time we can write two together:

```
protected void writeBoth(final DomainObject object1, final
        DomainObject object2) {
    try {
        tc.executeProcedure(
            new Procedure0Arg() {public void execute(){
                if (object1 != null) object1.write();
                if (object2 != null) object2.write();
            }}
        );
    } catch (TransactionException e) {
        throw new RuntimeWrappedException("Could not write both
       "+object1+" and "+object2,e);
    };
}
```

And then we will know that either both employees get written or neither will:

```
Employee newEmployee = new EmployeeClass();
newEmployee.setName("NewEmployee");
Employee newEmployee2 = (Employee) newEmployee.copy();

writeBoth(newEmployee, newEmployee2);
```

Using the “final” keyword

You will notice that the keyword “final” has appeared in these examples. This is because of a restriction with Java inner classes: Java inner classes can only refer to variables in the local method if those variables are declared “final” (i.e. unchanging). This can be quite annoying but is relatively easy to work around. Usually the most convenient is to have a special method that has final parameters like the ‘writeBoth’ method.

Commits and Identity Generation within Transactions

If we are using transactions to make sure we do changes as a single unit, we need to make sure we do not ‘commit’ during the transaction itself. The core of FORM does not do any implicit commits and is designed to be used within a transaction.

The one place implicit commits do occur in FORM is within the ‘database.support’ package for Table based identity generation. If you are using client based identity generation[29] with a central table, FORM locks the table to prevent two clients from accidentally using the same numbers. On some databases unlocks can only

[29] We strongly recommend server based identity generation to prevent ‘client caching’ mistakes and to put this central logic in one place. The FORM examples do not use server generation so they can work with all databases JDBC supports (least common denominator).

occur by committing the transaction, so the identity generator may produce an implicit commit to unlock the central resource table.

There are several alternatives available. The simplest option is to use an IdentityGenerator that does not produce the extra commits and assumes that an outer transaction will commit and release the locks. This will impede concurrency because the central table will be locked for longer. But if transactions are short, the delay may be acceptable. You must also be careful to reset the IdentityGenerator if a rollback occurs, since the generator will be out of synch with the database.

A second option is to have the IdentityGenerator use its own DatabaseConnection and JDBC Connection so it will have an independent transaction to work with and will only lock the table for the shortest possible time. This is the more resilient and higher concurrency alternative.

The Scheme-3b examples choose a 'withinTransaction' generator and DomainObjectAbsClass_FormInfo was changed to use it:

```
    static protected void buildIdentityGenerator(DatabaseConnection
            dbConnection) {
        Table table = (Table) dbConnection.table("GeneratorCounters");
            Column primaryKey = table.newBasicColumnNamed(
                                        "counter_id");
            table.setupPrimaryKeyColumn(primaryKey);
            table.newBasicColumnNamed_type("counter_value",
                                        identityColumnType);
        table.doneSetup();

        identityGenerator = DatabaseSupportPack.
           newIntegerTable_row_column_block_WithinTransactionGenerator(
                    table,"GEN1","counter_value",
                    5);
    }
```

Future Enhancements to FORM Transactions

Future versions will provides multiple-block execution and the ability to cause the model itself to rollback as needed. To take advantage of the rollback capability will require enhancements to the object model itself, which can be provided by Dome or your own framework.

Dome: Units of Work

FORM provides a transactional interface that is based on the JDBC transactional capabilities. A desirable layer above this is to support optimistic units of work where a number of changes can be done to your domain model "off-line" (outside of a transaction) and then written to the database as one unit. FORM does not handle this because it is a more general problem than a relational mapping problem: it is a general domain model enhancement for storage whether to a Relational Database or Object Database or other type of medium. This is instead part of Dome level functionality and you can see the Dome description for more information if you desire Unit of Work functionality.

Other Alternatives: Semantic Transactions and Workflow

If the transaction support of JDBC, FORM, and your database are not sufficient for your needs, then you can provide application level transactions that record state changes within you domain model itself.

10.2 Associated Objects in Inserts, Updates, and Deletes

FORM provides some automatic behavior with regard to writing associated objects to the database. FORM's behavior is currently oriented to the correct behavior for a relational database and does not attempt to support business rules for associations. These can be implemented through database procedures and triggers, or through business logic in your domain model. Support for complete traversal specification will be in future releases of FORM and will be coupled with Dome's association specifications.

FORM will automatically force the generation of an identify key for a new object that is referred to via a ForwardAssociation or ExternalAssociation of another object. This is done in a two-pass process to prevent excessive chaining and loops: the object is first saved with minimal information to generate a valid object identity and the object is then updated with all its association information.

10.3 Automatic Refreshing

FORM provides the ability to specify whether particular slots or the whole objects must be refreshed after being written to the database. This is especially appropriate behavior if the database has triggers, timestamp columns, or other value generating mechanisms. For example, we can specify that a particular slot will need to be refreshed because the database generates an "updated-on" value when an object is written to the database. If we do not refresh the object, we will have a 'stale' value for that particular slot.

Refreshing can occur either at the slot level or on the whole object level. Slot level control is accessible through Slot#setupRefreshAfterInsert (Update, or InsertAndUpdate). Likewise, you can specify that a whole object should be refreshed through ObjectMapper#setupRefreshAfterInsert.

10.4 Optimistic Locking

Optimistic locking allows you to have high concurrency, and be protected against one change overwriting another. This is accomplished by having one or more slots that have "version" indicators for an object. If the version upon writing is different from the version read, then someone else has changed the object in the interim. The transaction then aborts and the application is allowed to deal with the conflict.

To specify that a slot should be part of the OptimisticLock, you use Slot#setOptimisticLockSlot(). The main requirement for a Slot to be used as an OptimisticLock is that the SlotValues can be compared for "equality". Simply, two equal objects must respond 'True' to #equals. Be careful of certain objects that seem to be values actually use Identity for comparison. For example, Binary arrays use identity and not "equality" for equals. If this is the case, convert the value to another datatype (e.g. String) or use a more sophisticated comparison function through setOptimisticLockSlot_predicate(,).

10.5 Identity Cache Management

FORM keeps track of the identity of DomainObjects through an IdentityCache. Ideally, IdentityCaches should be "weak": they should "hold onto" an object only if the application is still using it. This is accomplished using WeakReferences and FORM's IdentityCaches are designed to work with WeakReferences.

Weak Cache Management

Weak Cache Management means you do not have to do anything to manage FORM's identity caches. If you are using a DomainObject (e.g. by displaying it in a window), FORM will know its identity. If you stop referring to that DomainObject, FORM will not prevent it from being garbage collected. When the object is garbage collected, FORM will clear it from its identity cache. If the object needs to be referenced again, FORM will re-fetch it from the database. This behavior is what most applications need to limit memory use.

Unfortunately, the current Java specification (1.1) does not include WeakReferences and not all VMs support it. ChiMu's Kernel frameworks currently support WeakReferences on the SUN JVM and the Microsoft's JView. If you are using a different VM weak cache management will not be enabled and you will have to use explicit cache management[30].

[30] Also notify us of the VM you are using so we can either add it to our WeakReference support or put pressure on the VM maker to support WeakReferences.

Explicit Cache Management

You can also explicitly control FORM's identity caches. You would want to do this if your Java VM does not support WeakReferences or if you do not want to rely on the VM's garbage collector to throw away the old objects. If you clear FORM's cache, FORM will fetch new objects even if an existing object with the same identity already exists.

Explicit cache management can be accomplished for the Orm, for individual ObjectMappers, or for individual objects. The different explicit cache management levels allow you to decide the granularity of 'freeing' objects from the Orm. The most important consideration is that you must be careful not to free objects that are still being modified by other parts of your application or you will have two local objects that both represent the same server object.

To clear the caches for an entire Orm or ObjectMapper, use:

```
orm.clearIdentityCaches()
objectMapper.clearIdentityCache();
```

At the individual Object level you can add and remove objects from an ObjectMapper with the methods:

```
objectMapper.addToIdentityCache(domainObject);
objectMapper.removeFromIdentityCache(domainObject);
```

10.6 Other Control Functionality

More of FORM controllable functionality will only be mentioned and not discussed in this document. This functionality includes controlling whether a Slot should be written to the database (e.g. disabling writes for server generated timestamps) and specifying the cardinality of an association (e.g. from 1-many to 1-1). See the FORM API and the examples for more information.

11 Conversions and Transformation Slots

FORM supports both simple conversions and sophisticated transformations between database values and objects. Slots can specify a basic Java type that they want to treat the database column as, and FORM will use JDBC to convert between the Java type and the actual database representation. For more sophisticated transformations you can use Transformation slots, which can do arbitrary transformations between slot values and column values. Transformation Slots can do sophisticated lookups, can have different transformations for different databases, and can embed complex objects into one or more columns.

11.1 Simple Data Type Conversions

The simplest way to cause a conversion is to specify the Java data type expected of a column when defining a Slot. This tells FORM that you want to define a 'Virtual' column in terms of a Java data type that is independent of the database storage type.

11.2 Transformation Slots

Transformation slots are a specific type of slot that can do more general transformations. They have two independent functors: one to decode from the database column to the slot value, and the other to encode from the slot value to the database column. If either of these is 'null' then the value is transferred directly for that process. Both the encoder and the decoder are Function1Args: they each take in an Object and return an Object.

DomainModel-4

The examples in Scheme 4 through Scheme 4d illustrate different types of transformation slots and their usage. The object and/or relational models are slightly different in each scheme, either in data type definition or in the addition of attributes for a particular object.

11.3 Transforming Basic Data Types

Often, we run into situations where a database column needs one data type, but the object model needs another type for the corresponding slot. Sometimes these can be supported through simple conversions, but in other cases there is no default conversion and we will need to transform the data ourselves.

Mapping between Numerical Data Types

One example of using Transformation Slots for translating data types is in mapping different numerical specifications between the object model and the relational data model.

Sometimes, we may need to store numbers in a different format from what we use in the object model. For example, in our object model, a Project keeps track of its profit margin. Because we like to be very precise about how much we are earning or losing, we have decided that this attribute needs to be a decimal data type with precision. The database data type will be a precise decimal. Decimals are represented in Java as BigDecimals, but using BigDecimals for numeric operations may be too slow to be acceptable. We decide to use a double instead for the profit margin attribute in the Java object model. To accomplish this we can use a Transformation Slot to convert the value when the attribute is being stored/retrieved.

Example: NumericTransform_1

In order to build a transformation slot, we first need to define the encoding and decoding functors for the transformation. In the case of converting between BigDecimal and Double, the conversion functors are already defined in ChiMu Kernel Utilities TranslationLib package (to circumvent the Java BigDecimal error), we need not redefine them. Both functors are of type Fucntion1Arg, as specified in the TransformationSlot constructor.

In the ProjectClass_FormInfo class, we define the transformation slot as follows:

```
myMapper.newTransformSlot_column_type_decoder_encoder("profitMargin",
            "profit_margin", Double.class,
            TranslationLib.numberToDoubleFunction(),
           TranslationLib.numberToBigDecimalFunction());
```

Note that we can still specify the basic Java datatype conversion we expect from the database. Otherwise we could have different transformations for different storage formats.

11.4 Database dependent Transformations

Another usage of the Transformation Slot is in translating Java data types to different data types depending on what the database product supports. Boolean data type is a good example of a data type that is supported differently on each of the databases: MSSQL has the data type of BIT, but for Informix we may have to use a CHAR(1). These are significantly different encoding functions, but the Java data type (boolean) is the same for each.

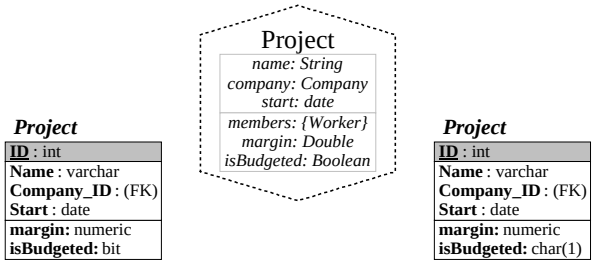
Example: BooleanTransform_1

In our object model, we will define that the project knows whether it has been included in the overall company budget or not. We amend our model by including an attribute of 'isBudgeted' in the Project. In our data model, we shall first assume that we will map the Boolean column into a Char(1) column, with 'F' denoting 'false', and 'T' denoting 'true'.

To map this attribute into the database, we specify a Transformation Slot for the 'isBudgeted' attribute. We then specify the encoding and decoding functors. Since this transformation will probably occur fairly often, we abstracted the slot adding function and put the functionality in the common super class so that all subclasses needing to map the Boolean data type will have access to it.

DomainModel-4

For the DomainModel-4b scheme, we added the attribute for 'isBudgeted' to the Project class. The ObjectModel is simple: just add a Boolean attribute. The relational data model is also simple, but it will depend on what type of database we have. Either the Project table will use a 'bit' or a 'char(1)' depending on what the database supports.



To support the potential for different transformations, we will create a procedure that creates a boolean slot in a given mapper. This we place in DomainObjectAbsClass_FormInfo:

```
    static protected void
           buildBooleanSlotInto_named_column(ObjectMapper mapper,
           String name, String columnName) {
        mapper.newTransformSlot_column_decoder_encoder(name,columnName,
            stringToBoolean,
            booleanToString
        );
    }
```

To support this procedure we include the encoding and the decoding functors:

```
    static protected final String STRING_FALSE = "F";
    static protected final String STRING_TRUE  = "T";

    static protected Function1Arg booleanToString = new Function1Arg()
           {public Object valueWith(Object arg1) {
            if ( arg1 == null) {
                return null;
            } else if (((Boolean) arg1).booleanValue() ) {
```

```
                return STRING_TRUE;
            } else {
                return STRING_FALSE;
            }
        }};

    static protected Function1Arg stringToBoolean = new Function1Arg()
           {public Object valueWith(Object arg1) {
            if (arg1 == null) {
                return null;
            } else {
                return new Boolean(arg1.equals(STRING_TRUE));
            }
        }};
```

Now ProjectClass_FormInfo class can simply call the buildBooleanSlot.

```
    buildBooleanSlotInto_named_column(myMapper, "isBudgeted",
           "budgeted");
```

Example: ProjectInsert_1

The ProjectInsert_1 example for Scheme Four retrieves a project, copies it, make modification, and write the copy to the database. The ‘profitMargin’ attribute and the ‘isBudgeted’ attribute are automatically converted to the correct data types via the Transformation Slots.

```
    public void run (Connection jdbcConnection) {

        createAndConfigureOrm(jdbcConnection);
        ObjectMapper projectMapper = orm.mapperNamed("Project");
        ObjectMapper employeeMapper = orm.mapperNamed("Employee");

        Project project = (Project) projectMapper.findAny();
        Employee employee = (Employee) employeeMapper.findAny();

        outputStream.println(project.info());
        Project newProject = (Project) project.copy();

        newProject.setName("Testing");
        newProject.setIsBudgeted(true);

        Sequence aCollection = CollectionsImplPack.newSequence();
        aCollection.add(employee);
        newProject.setEmployees(aCollection);

        newProject.write();
        outputStream.println("A new project named " + newProject.name()
           + " has been added.");
        outputStream.println(newProject.info());

    }
```

12 Embedded Objects

Object models can have attributes that are not simple objects (e.g. Money or Length, instead of Strings, Numbers, or Booleans), but which are nonetheless basic attributes: The attribute object is completely contained in and is private to another object. You can implement these attribute objects in FORM in a couple ways: (1) You can use the same mechanisms as described for mapped domain classes and provide the attribute object with its own table, public identity, and associations. If you take this approach you must constrain your business rules to prevent multiple objects from referring to the same attribute object. Or (2) you can **embed** the attribute object within the row describing the containing object. This prevents multiple objects from referring to the same attribute object without using a supplemental business rule. It can also improve performance when the attribute is frequently needed because it does not require an association traversal.

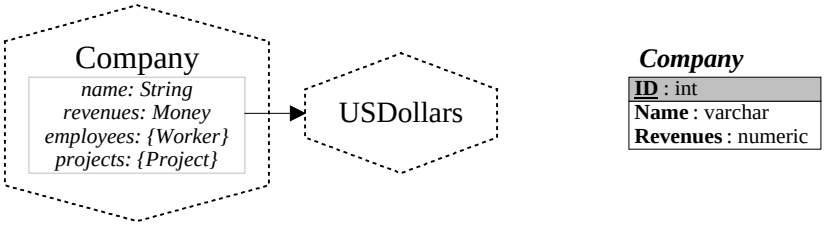
To implement an Embedded Object with FORM we use the same TransformationSlot capabilities as described in the previous section. There are two main differences. First, the embedded object's class will be responsible for knowing how to decode and encode the embedded objects. The containing class will know about the slots and the columns, but it delegates to the embedded class for the required transformation. Second, an embedded object will usually require multiple columns to store the necessary information

12.1 Simple Embedded Object: US Dollars

Given new requirements, our object model now requires substantially complex behavior in calculating our revenue: the behavior of currency interchange and monetary accuracy. We could have Company know how to work with revenue as well as money, but this will not be very good design: Company will be overburdened and the implementation will not be reusable. It would be better to define a new Type called Money and provide classes that implement Money. The Money classes know the rules of monetary calculations. Company then holds on to some Money and can rely on the Money object's functionality instead of reimplementing it.

DomainModel-4b

This is a simple example of an embedded object. Our relational model has not changed. Revenue is still mapped as a single column to hold the amount of US Dollars as a number. But when we instanciate a Company, it will not have revenue as a simple number, it will instead have revenue as a Money (USDollars) object that was created using the database number. The addition to our new domain model and the corresponding database structure are:



Transformation slots will handle the differences between these two models.

Money

A Money object is a ValueObject: it is Immutable and all operations return new Money objects as their result. Money has the following interface:

```
import java.math.BigDecimal;

public interface Money {

    //(P)******************** Asking *************************
    public Currency   currency();
    public BigDecimal amount();
```

```
    //(P)******************** Conversion **********************
    public Money toCurrency(String newCurrency);
    public Money toUsd();

    //(P)******************** Calculations ************************
    public Money add(Money money);
    public Money subtract(Money money);
    public Money multiply(double scaling);
    public Money divide(double scaling);
}
```

US Dollars

US Dollars is one type of Money. Using the approach we stated above, we define the encoding and decoding responsibilities in the USDollarClass.

```
    static public Function1Arg encodeFunctor() {
        return new Function1Arg() {public Object valueWith(Object arg1)
            {
             if (arg1 == null) return null;
             Money someMoney = (Money) arg1;
             return someMoney.amount();
        }};
    }

    static public Function1Arg decodeFunctor() {
        return new Function1Arg() {public Object valueWith(Object arg1)
            {
             if (arg1 == null) return null;
             BigDecimal amount = (BigDecimal) arg1;
             return new USDollarClass(amount);
        }};
    }
```

The arguments of the functors are converted into a type of Money (for use in the domain), or are converted into a BigDecimal (for storage in the database).

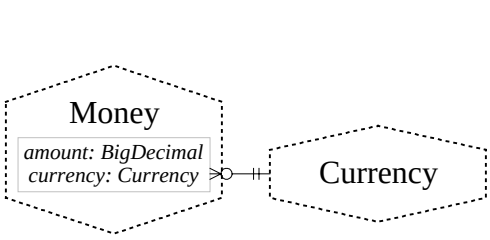
In Company, we simply define a TransformationSlot that directs the functor arguments to USDollarClass.

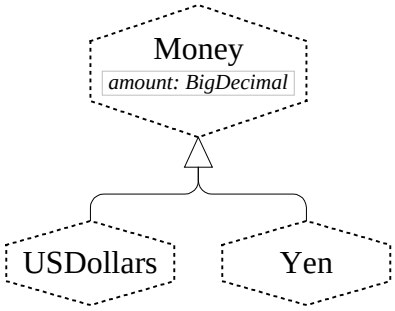
```
    myMapper.newTransformSlot_column_type_decoder_encoder("revenue",
           "revenue", BigDecimal.class, USDollarClass.decodeFunctor(),
           USDollarClass.encodeFunctor());
```

12.2 Complex Embedded Object: Money

Instead of assuming that all companies do business in US Dollars, we will now allow them to do business in any currency they desire. This means 'revenue' must become a more sophisticated object and must know both its amount (still a fixed decimal) and its currency (one of USD, YEN, DEM, etc.).

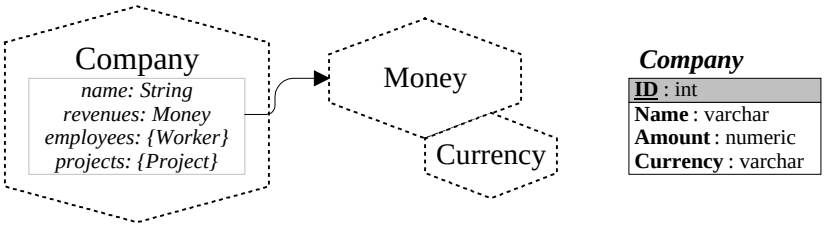
The object model is the same as before, except Money now has more than one type: instead of just representing US currency it represents all the currencies of the world.





DomainModel-4c

The addition for our new domain model and the corresponding database structure are:



Mapping Money

To store a Money object now requires two columns on the database table, one each for the currency and the amount. A Transformation Slot can only work with a single column at any given time. To implement a more complex transformation that requires multiple columns, we must build a virtual column that is composed of multiple basic columns. A **compound column** represents a virtual column that aggregates multiple basic columns into a new column. To support our Money object we will need to create a compound column that combines "currency" and "amount" into a virtual "revenue" column.

```
Table table = dbConnection.table("Company");
    table.newBasicColumnNamed_type("revenue", BigDecimal.class);
    table.newCompoundColumnNamed("revenuePair", "revenue_cur",
       "revenue");
```

We first define a basic column and specify the data type for the column. Then we define the compound column. The compound column consists of the name of the compound column, and a sequence of database columns to which the compound column will map.

We can now proceed to treat the embedded Money just like the embedded USDollars. The CompanyClass will map the "revenue" slot onto the "revenue" compound column by asking the MoneyClass for the proper transformations.

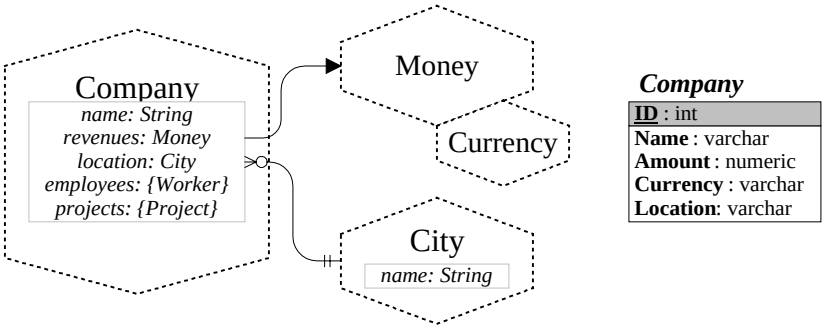
```
myMapper.newTransformSlot_column_type_decoder_encoder("revenue",
       "revenuePair", BigDecimal.class, MoneyClass.decodeFunctor(),
       MoneyClass.encodeFunctor());
```

12.3 Mapping a look up object

Sometimes, objects may not be mapped to the database. Instead, it could come from other sources, such as legacy data set, or stored in memory. Transformation Slots can perform these more general lookups to a different source of data.

DomainModel-4d

The final change to the domain model to support a location looks like this:



Mapping a City

In our domain model, a Company is located in a City. The City will not be stored in the database. Instead, it is created 'on the fly' based on the name, which is stored as a single column in the database. We use the Transformation Slot to handle retrieving the column and looking up the city in memory, when necessary.

We modify the mapper configuration to include city by adding the following Transformation Slot:

```
    myMapper.newTransformSlot_column_decoder_encoder("city", "city",
           CityClass.columnToCityFunctor(),
           CityClass.cityToColumnFunctor());
```

The actual transformation functors

```
    //(P)******************** Functors ************************

    static public Function1Arg cityToColumnFunctor() {
        return new Function1Arg() {public Object valueWith(Object arg1)
            {//<DontAutoUnstub>
             return nameOfCity((CityClass) arg1);
        }};
    }

    static public Function1Arg columnToCityFunctor() {
        return new Function1Arg() {public Object valueWith(Object arg1)
            {//<DontAutoUnstub>
             return getCity((String) arg1);
        }};
    }
```

The two functors return objects that will satisfy either the relational or object model.

```
        /*
            Answers a city given the name of the city
        */
    static protected CityClass getCity(String cityName) {
//        outputStream.println("This is aString " + aString);

        Object anObject = cities.detect(includesNameFunctor(cityName));
        if (anObject != null) return (CityClass) anObject;

        CityClass newCity = new CityClass();
        newCity.setName(cityName);
        cities.add(newCity);
        return newCity;
    }
```

The CityClass keeps track of a list of all the cities that has been instantiated thus far. If there was already one with the same name, the CityClass will return the instance with that name, instead of creating a new one every time.

13 Advanced Mappers and Retrievers

There are special situations where an ordinary object mapper cannot handle. In FORM, more types of mappers and retrievers exist and are utilized in FORM for these special situations.

This chapter will describe the more advanced types of Mappers and Retrievers.

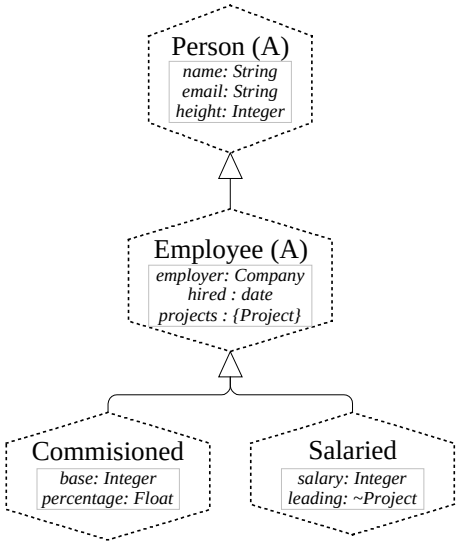
13.1 Distinguishing Object Mappers

In object-oriented modeling, we often discern entities (objects) by behaviors rather attributes. Therefore, objects with similar attributes but with different behaviors are often modeled as distinct classes. But since these classes do share similar attributes, we often may choose to store them on the same table with a identifying column what would discern these objects, for expediency.

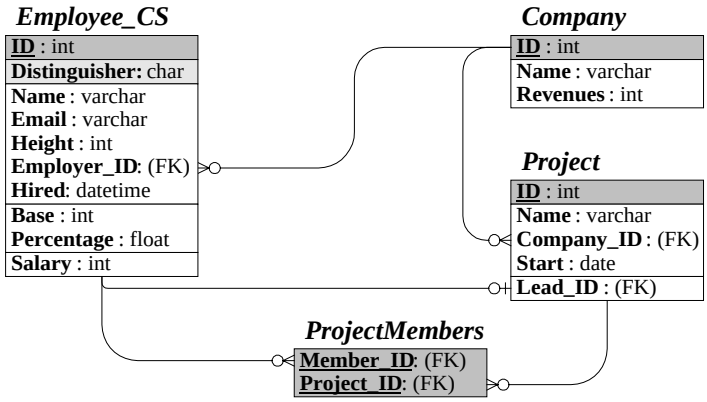
In FORM, we use a DistinguishingObjectMapper to retrieve and store this category of objects. DistinguishingObjectMapper has most of the same API's as a regular object mapper, with the exception that it contains a special slot that tells it how to distinguish objects of its types from the database table. DistinguishingObjectMapper can also save its object state along with the appropriate identifier value.

To illustrate how this is accomplished, we now amend our domain model to include more specific types of employees: SalariedEmployees and CommissionedEmployees. Both types of employees have similar attributes, but each has its own way of calculating an employee's annual salary. Salaried Employees have a fixed annual salary; Commissioned Employees have a base plus some percentage thereof.

The Employee is now an abstract class with two concrete subclasses: SalariedEmployee and CommissionedEmployee. Since both classes share similar attributes and are often queried as a collective unit, we store both types of employees on the same table. To discern between two types of objects, we add a column that would demarcate the two object types.



Our relational data model now looks like:



The EmployeeAbsClass_FormInfo

```
public class EmployeeAbsClass_FormInfo extends PersonAbsClass_FormInfo
            {
    public void configureMappers() {
       super.configureMappers();

       myMapper.newForwardSlot_column_partner("employer",
            "employer_ID",
            orm.retrieverNamed("Company"));
       myMapper.newDirectSlot_column_type("hiredDate", "hired_date",
           Date.class);
       myMapper.newExternalSlot_connector_mySlot("projects",
                orm.associationConnectorNamed("projectAndEmployee"),
                "employee");
    }

   public Class myClass() {
       return EmployeeAbsClass.class;
    }
}
```

We then create the two concrete form info classes that will create the respective distinguishing mappers and assign them to the Orm. Both SalariedEmployeeClass_FormInfo and CommissionedEmployeeClass_FormInfo classes would need to overwrite the inherited method of #createMappers(), as well as #configMapper(), since the #createMapper() assumes the creation of an ObjectMapper instead of a DistinguishingObjectMapper, as is required here.

Salaried Employee Form Info:

```
public class SalariedEmployeeClass_FormInfo extends
           EmployeeAbsClass_FormInfo {

    public void createMappers() {
        if (myMapper != null) throw new RuntimeException("Mapper
           creation called multiple times");
        DistinguishingObjectMapper myOrmMapper =
                (DistinguishingObjectMapper)
           orm.mapperNamed_orNull(mapperName());
        if (myOrmMapper != null) throw new RuntimeException("Multiple
           mappers with the same name");
        myTable  = dbConnection.table("employee");
        myMapper = orm.newDistinguishingObjectMapperNamed_table(
           "SalariedEmployee",myTable);
    }

    public void configureMappers() {
        super.configureMappers();
        final DistinguishingObjectMapper mapper =
           (DistinguishingObjectMapper) myMapper;
```

```
        mapper.newDistinguishingSlot("distinguisher", "distinguisher",
           "s");
        mapper.newDirectSlot_column_type("salary", "annual_salary",
           Integer.class);
    }

    public Class myClass() {
        return SalariedEmployeeClass.class;
    }
}
```

A new slot called DistinguishingSlot is added specifying the slot name, slot column name for the slot that will differentiate the object's type. We assign all SalariedEmployee rows to have a value of "s" in the "distinguisher" column.

The CommissionedEmployeeClass_FormInfo, similarly, looks like:

```
public class CommissionedEmployeeClass_FormInfo extends
           EmployeeAbsClass_FormInfo {

    static private Class identityColumnType = Integer.class;

    public void createMappers() {
        if (myMapper != null) throw new RuntimeException("Mapper
           creation called multiple times");
        DistinguishingObjectMapper myOrmMapper =
                (DistinguishingObjectMapper)
           orm.mapperNamed_orNull(mapperName());
        if (myOrmMapper != null) throw new RuntimeException("Multiple
           mappers with the same name");
        myTable  = dbConnection.table("employee");
        myTable.newBasicColumnNamed_type("id",identityColumnType);
        myMapper = orm.newDistinguishingObjectMapperNamed_table(
           "CommissionedEmployee",myTable);
    }

    public void configureMappers() {
        super.configureMappers();
        final DistinguishingObjectMapper mapper =
           (DistinguishingObjectMapper) myMapper;
        mapper.newDistinguishingSlot("distinguisher", "distinguisher",
           "c");
        mapper.newDirectSlot_column_type("base", "base",
           Integer.class);
        mapper.newDirectSlot_column_type("percentage", "percentage",
           Float.class);
    }

    public Class myClass() {
        return CommissionedEmployeeClass.class;
    }
}
```

where the "distinguisher" column value is "c".

In the ExampleAbsClass, we register these additional form info classes with the orm:

```
    protected void createAndConfigureOrm(Connection jdbcConnection) {
        createOrm(jdbcConnection);
        DomainObjectAbsClass_FormInfo.prepareForMapping(orm,dbConnection);
           orm.addInfoClass_withDb(CommissionedEmployeeClass_FormInfo.class,
           dbConnection);
           orm.addInfoClass_withDb(SalariedEmployeeClass_FormInfo.class,
           dbConnection);
           orm.addInfoClass_withDb(CompanyClass_FormInfo.class,dbConnection);
           orm.addInfoClass_withDb(ProjectClass_FormInfo.class,dbConnection);
           orm.addInfoClass_withDb(AssociationsFormInfo.class,dbConnection);
        orm.doneSetup();
    }
```

We now can save and retrieve both types of objects to and from the database.

Example: CommissionedRetrieval_1

CommissionedRetrieval_1 shows how a DistinguishingObjectMapper is created and used:

```
public void run (Connection jdbcConnection) {

    //*********************************************************
    //(P)*************** Configuration ***********************
    //*********************************************************
    createAndConfigureOrm(jdbcConnection);
        ObjectMapper mapper = orm.mapperNamed("CommissionedEmployee");

    //*********************************************************
    //(P)***************** Running ***************************
    //*********************************************************
        Employee employee = (Employee) mapper.findAny();

        outputStream.println(employee.info());
    }
}
```

This query selects an instance of Commissioned Employee and displays its information. Since DistinguishingObjectMapper is a kind of ObjectMapper, #mapperNamed(aString) method will return the appropriate object for the purpose of this query.

13.2 CollectiveObjectRetrievers

CollectiveObjectRetriever is somewhat complimentary of DistinguishingObjectMapper. While DistinguishingObjectMapper maps a specific type of object, the CollectiveObjectRetriever can retrieves all types objects from one single table. And depending on what DistinguishingObjectMapper the retriever knows about, it can restore the objects to its appropriate type. The CollectiveObjectRetriever achieves this feat by 'collecting' the appropriate DistinguishingObjectMappers it is allowed to use, and associate itself with a specific table.

The CollectiveObjectRetriever can be defined anywhere. In our scheme, we create and configure our CollectiveObjectRetriever in the class EmployeeClass_FormInfo. EmployeeClass_FormInfo class is responsible for retrieving all instances of employees, whether they are Salaried Employees or Commissioned Employees. Since this class does not need to implement many of the required interface of object mappers, we simply extend the basic SimpleFormInfoAbsClass.

```
public class Employee_FormInfo extends SimpleFormInfoAbsClass {

    protected CollectiveObjectRetriever myRetriever;
    public void createMappers() {
        if (myRetriever != null) throw new RuntimeException("Mapper
           creation called multiple times");
        CollectiveObjectRetriever myOrmRetriever =
                (CollectiveObjectRetriever)
           orm.retrieverNamed_orNull("Employee");
        if (myOrmRetriever != null) throw new
           RuntimeException("Multiple retrievers with the same name");
        myRetriever =
           orm.newCollectiveObjectRetrieverNamed("Employee");
    }

    public void configureMappers() {

           myRetriever.addDistinguishingMapper((DistinguishingObjectMap
           per) orm.mapperNamed("SalariedEmployee"));

           myRetriever.addDistinguishingMapper((DistinguishingObjectMap
           per) orm.mapperNamed("CommissionedEmployee"));
    }
    public void configureCompleted() {
    }

    public Class myClass() {
        return Employee.class;
    }
}
```

Now, the CollectiveObjectRetriever knows about the two different DistinguishingObjectMapper that this retriever can use to restore the objects it retrieves: CommissionedEmployee and SalariedEmployee.

Finally, in the ExampleAbsClass, we register the class that will create and configure the mapper:

```
protected void createAndConfigureOrm(Connection jdbcConnection) {
   createOrm(jdbcConnection);
   DomainObjectAbsClass_FormInfo.prepareForMapping(orm,dbConnection);
       orm.addInfoClass_withDb(CommissionedEmployeeClass_FormInfo.class,db
       Connection);
       orm.addInfoClass_withDb(SalariedEmployeeClass_FormInfo.class,dbConn
       ection);
        orm.addInfoClass_withDb(Employee_FormInfo.class, dbConnection);
       orm.addInfoClass_withDb(CompanyClass_FormInfo.class,dbConnection);
       orm.addInfoClass_withDb(ProjectClass_FormInfo.class,dbConnection);
       orm.addInfoClass_withDb(AssociationsFormInfo.class,dbConnection);
    orm.doneSetup();
}
```

Example: EmployeeRetrieval_2

EmployeeRetrieval_2 illustrates how a CollectiveObjectRetriever is used to retrieve an instance from the Employee table, and cast the retrieved object into the specified type for the retriever.

```
//*********************************************************
//(P)*************** Configuration ***********************
//*********************************************************

    createAndConfigureOrm(jdbcConnection);
    ObjectRetriever retriever = orm.retrieverNamed("Employee");

//*********************************************************
//(P)***************** Running ***************************
//*********************************************************

   Collection employees = (Collection) retriever.selectAll();

   for (Enumeration enum = employees.elements();
       enum.hasMoreElements();) {
        Employee employee = (Employee) enum.nextElement();
        outputStream.println(employee.info());
    };
```

Example: EmployeeInsert_2

In this example, we insert a new Commissioned employee, and since the scheme is described and defined, FORM will insert the row with the appropriate “distinguisher” column value into the database.

```
//*********************************************************
//(P)******************* Running *************************
//*********************************************************

Company aCompany = (Company) companyMapper.findAny();
    Employee salariedEmployee =
        new SalariedEmployeeClass("Dr. Watson",
       "watson@privateeye.com", 68, aCompany,
        new java.sql.Timestamp((new Date()).getTime()),
        new Integer(80000));
    salariedEmployee.write();
    outputStream.println("Writing new salaried employee " +
        salariedEmployee.info());

    Float percentage = new Float("0.2345");
    Employee commissionedEmployee =
        new CommissionedEmployeeClass("Sherlock Holmes",
      "sherlock@privateeye.com", 72, aCompany,
        new java.sql.Timestamp((new Date()).getTime()),
        new Integer(50000), percentage);
    commissionedEmployee.write();
```

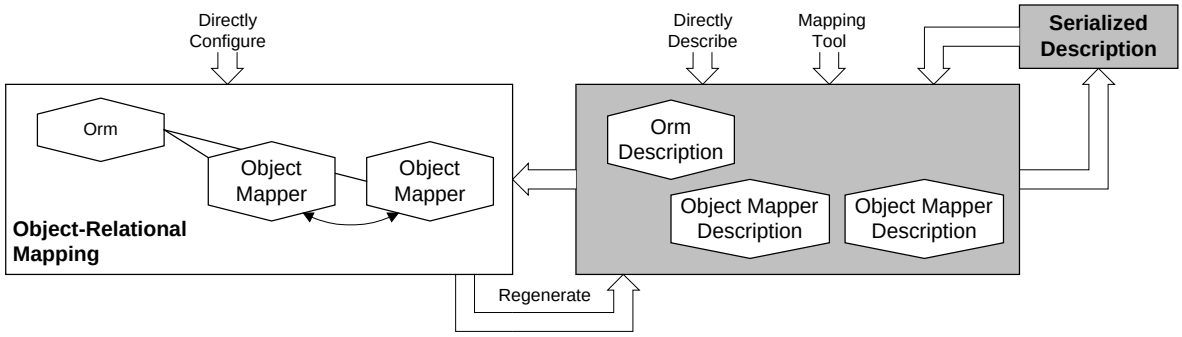
```
        outputStream.println("Writing new commissioned employee " +
           commissionedEmployee.info());
```

The Distinguishing mappers for each type of employees will automatically insert the appropriate value into the Dinstuinguisher column.

14 Descriptions

FORM currently uses programmatic configuration: code written by the FORM client or generated from FORM tools creates and configures the mapping information FORM uses at runtime. This allows precise control of the FORM engine and supports very flexible, context-sensitive configuration.

In future releases FORM will also support configuration solely based on data descriptions. Although less flexible, description based configuration supports easier reading & generation of mapping specifications (by a UI Tool) and supports using serialization instead of using more sophisticated class files. FORM will support both approaches, so developers can decide which is the most convenient for their project (or even on a class by class basis).



14.1 Database Descriptions

The current release of FORM includes one portion of the Description package: it includes DatabaseDescriptions, which allow you to record information about a particular database for later use. For example, you can use the recorded information to configure FORM instead of the default approaches of using direct configuration or using database metadata. This tool can be used to provide either database independence or database specific configuration with higher performance.

The type of information in a description can be classified into two types.

6. Information about Tables, Columns, and Column Java Data Types, which should be independent of the specific database product.
7. Information about Column sql data types, length, and nullability, which could be database product specific.

In the second case, there is a performance as well as convenience factor: FORM will not need to query the database for meta-information when it connects to it. Instead it will receive that information from the deserialized description. Because FORM did not need to talk to the server, this process can be much quicker for some applications. In exchange you run the risk that the database has changed and FORM does not know about the problem because it did not get the meta information from the database.

To create a database description you can either build the Description directly or “regenerate” the description from a completely configured FORM database connection. The following example will assume you regenerate the description, see the form.description API for how to generate a description directly. After completing the setup of the orm and databaseConnection, create a DatabaseDescription that is initialized from the databaseConnection information.

```
Orm orm = FormPack.newOrm();
DatabaseConnection dbConnection = FormPack.newDatabaseConnection(
            jdbcConnection
        );
//...
orm.doneSetup();
```

```
    DatabaseDescription dbDesc = DescriptionPack.
           newDatabaseDescriptionFrom(dbConnection);
```

With a DatabaseDescription in hand, you can then serialize it to a file like any other object. The following is an incomplete example of serialization:

```
    //... serialize the description
    FileOutputStream   fos = new FileOutputStream(fileName);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(dbDesc);
```

After the DatabaseDescription has been serialized, future executions of an application can decide to use the serialized information or not. If you decide to use the serialized description you first need to deserialize it:

```
    //... deserialize the description
    FileInputStream   fis = new FileInputStream(fileName);
    ObjectInputStream ois = new ObjectInputStream(fis);
    DatabaseDescription dbDesc = (DatabaseDescription)
           ois.readObject();
```

Than you can use the description in the DatabaseConnection creation process:

```
    DatabaseConnection dbConnection = FormPack.newDatabaseConnection(
                jdbcConnection
            );

    dbDesc.buildDatabaseConnection(dbConnection);
```

This will build the dbConnection with any information that is in dbDesc, which will include the database specific sql type information. If you instead use:

```
    dbDesc.buildDatabaseConnection_noDbInfo(dbConnection);
```

None of the Database Specific column information will be put into the databaseConnection. In this case FORM will query the database metadata, which will guarantee the scheme information to be correct.

In either of these cases, the description may provide more or less information than FORM needs. FORM will use its standard behavior (query the database) to find out any further information that was not provided by the description.

DomainModel 5b

The only difference between DomainModel-5 and DomainModel-5b is in generating and reusing DatabaseDescriptions from previous runs. For DomainModel-5b, ExampleAbsClass has three simple changes, first it inherits from ExampleDescriptionAbsClass to get the functionality to save and read DatabaseDescriptions from Files. Second, it builds the dbConnection from information stored in a saved DatabaseDescription if a description file exists. Third, after the orm and dbConnection have been setup it will create and save a DatabaseDescription if one does not exist already.

```
public abstract class ExampleAbsClass extends ExampleDescriptionAbsClass {
    protected void createAndConfigureOrm(Connection jdbcConnection) {
        createOrm(jdbcConnection);
            buildConnection_fromDescription(dbConnection);
            DomainObjectAbsClass_FormInfo.prepareForMapping(orm,dbConnection);
//...
            orm.addInfoClass_withDb(AssociationsFormInfo.class,dbConnection);
        orm.doneSetup();
        writeDescription(dbConnection);
    }
}
```

This provides a simple caching model for the database information, and if you do a trace of DomainModel-5b examples you will notice that they only query the database for metadata once instead of multiple times.

14.2 Summary

DatabaseDescriptions allow you to record information about a database to be used in configuring FORM more rapidly for the same database or to move one database model from one database to another.

DatabaseDescriptions are fully serializable so they can be saved and restored whereever serialized data is used.

In future releases, FORM will provide a similar model for creating OrmDescriptions which can be generated and read either programmatically or through UI Tools.

15 References

15.1 ChiMu Corporation References

Architecture *A Good Architecture for Object-Relational Mapping*

Examples *Examples of FORM*

FORM Tools *FORM Tools.*

Foundations *Foundations of Object-Relational Mapping*

Introduction *Introduction to FORM*

Kernel *Kernel Frameworks*

Learning *Learning FORM*

Standards *Java Development Standards*

Starting *Getting started with FORM*

15.2 General References

Cattell+ 97 R.G.G. Cattell and Douglas K. Barry, Editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, San Francisco, 1997.

Date 95 C.J. Date. *An Introduction to Database Systems*. Addison-Wesley, Reading, MA, 1995.

Date 95b C.J. Date. *Relational Database Writings 1991- 1994.* Addison-Wesley, Reading, MA, 1995.

Date 97 C.J. Date with Hugh Darwen. *A Guide to the SQL Standard, Fourth Edition.* Addison-Wesley, Reading, MA, 1997.

16 Appendix-A OQL Syntax

This appendix contains the complete syntax for FORM's OQL. FORM uses a parser that understands Unicode characters. These terminal values are shown with the '\u' Unicode escape sequence. All of OQL's reserved words (see section "Reserved Words and Literals") are case insensitive.

16.1 Non-Terminals

```
Query ::= ( SelectQuery | FromQuery) <EOF>
SelectQuery ::= SelectClause FromClause ( WhereClause )?
FromQuery ::= FromClause ( WhereClause )?
SelectClause ::= <SELECT> ExpressionList
NamedVariableClause ::= <FROM> NamedVariableList
WhereClause ::= <WHERE> Condition
NamedVariableList ::= NamedVariable ( <COMMA> NamedVariable )*
NamedVariable ::= MessageSequence ( ( <AS> )? Name )?
ExpressionList ::= LabeledExpression ( <COMMA> LabeledExpression )*
LabeledExpression ::= Expression ( <AS> Name )?
Expression ::= InclusiveOrCondition
Condition ::= InclusiveOrCondition
InclusiveOrCondition ::= ExclusiveOrCondition ( <OR> ExclusiveOrCondition )*
ExclusiveOrCondition ::= AndCondition ( <XOR> AndCondition )*
AndCondition ::= ComparisonCondition ( <AND> ComparisonCondition )*
ComparisonCondition ::= AdditiveExpression ( ( <EQ> | <EQ2> | <NE> | <NE2> | <LT> |
                        <GT> | <LE> | <GE> ) AdditiveExpression )?
AdditiveExpression ::= MultiplicativeExpression ( ( <PLUS> | <MINUS> )
                       MultiplicativeExpression )*
MultiplicativeExpression ::= UnarySignExpression ( ( <TIMES> | <DIVIDE> | <REM> )
                             UnarySignExpression )*
UnarySignExpression ::= ( <PLUS> | <MINUS> ) UnaryNotCondition
                      | UnaryNotCondition
UnaryNotCondition ::= ( <TILDE> | <BANG> ) PrimaryPrefix
                    | PrimaryPrefix
PrimaryPrefix ::= Literal
                | MessageSequence
                | <LPAREN> Expression <RPAREN>
```

```
Literal ::= ( ( <INTEGER_LITERAL> ) )
          | ( ( <FLOATING_POINT_LITERAL> ) )
          | ( ( <STRING_LITERAL> ) )
          | ( ( <STRING_LITERAL2> ) )
          | BooleanLiteral
          | NullLiteral

BooleanLiteral ::= <TRUE>
                 | <FALSE>

NullLiteral ::= <NULL>

MessageSequence ::= Name ( Message )*

Name ::= ( ( <IDENTIFIER> ) )
       | UnboundName
       | UnboundIndex

UnboundName ::= <COLON> <IDENTIFIER>

UnboundIndex ::= <DOLLAR> <INTEGER_LITERAL>

Message ::= ( <DOT> | <RARROW> ) <IDENTIFIER> ( Arguments )?

Arguments ::= <LPAREN> ( ArgumentList )? <RPAREN>

ArgumentList ::= Expression ( <COMMA> Expression )*
```

16.2 Terminals

WHITE SPACE

```
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

COMMENTS

```
{
  <SINGLE_LINE_COMMENT: "//"
          (~["\n","\r"])*
          ("\n"|"\r"|"\r\n")>
| <MULTI_LINE_COMMENT: "/*"
          (~["*"])* "*" ("*" |
          (~["*","/"] (~["*"])*
          "*"))* "/">
}
```

OPERATORS

```
{
  < BANG:  "!" >
| < TILDE: "~" >

| < EQ:  "=" >
| < EQ2: "==" >
| < NE:  "!=" >
| < NE2: "<>" >

| < GT: ">" >
| < LT: "<" >
| < LE: "<=" >
| < GE: ">=" >

| < PLUS:   "+" >
| < MINUS:  "-" >
| < TIMES:  "*" >
| < DIVIDE: "/" >
| < REM:    "%" >

| < AND: "&" | "AND">
| < OR:  "|" | "OR" >
| < XOR: "^" | "XOR">
}
```

SEPARATORS

```
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < COMMA:  "," >
| < DOT:    "." >
| < RARROW: "->" >
}
```

SPECIAL

```
{
  < DOLLAR: "$" >
  < COLON:  ":" >
}
```

RESERVED WORDS AND LITERALS

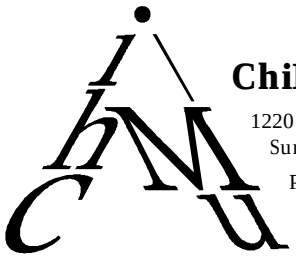
```
{
  < IN:     "in" >
| < WHERE:  "where" >
| < AS:     "as" >
| < SELECT: "select" >
| < FROM:   "from" >
| < NULL:   "null" >
| < TRUE:   "true" >
| < FALSE:  "false" >
}
```

IDENTIFIERS

```
{
  < IDENTIFIER: <LETTER>
          (<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}
```

LITERALS

```
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL>
          (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL>
          (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"]
          (["0"-"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"]
          (["0"-"9","a"-"f","A"-
          "F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-
          "7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-
          "9"])* (<EXPONENT>)?
          (["f","F","d","D"])?
      | "." (["0"-"9"])+
          (<EXPONENT>)?
          (["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT>
          (["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)?
          ["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-
          "])? (["0"-"9"])+ >
|
  < STRING_LITERAL2:
      "'"
      (   (~["'","\\"])
        | ("\\"
            (
          ["n","t","b","r","f","\\"
          ,"'","\""]
            | ["0"-"7"] ( ["0"-
          "7"] )?
            | ["0"-"3"] ["0"-"7"]
          ["0"-"7"]
            )
          )
      )*
      "'"
  >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\"])
        | ("\\"
            (
          ["n","t","b","r","f","\\"
          ,"'","\""]
            | ["0"-"7"] ( ["0"-"7"]
          )?
            | ["0"-"3"] ["0"-"7"]
          ["0"-"7"]
            )
          )
      )*
      "\""
  >
}
```



ChiMu Corporation

1220 N. Fair Oaks Ave, #1314
Sunnyvale, CA 94089

Phone: 408 734-9068
Email: info@chimu.com

www.chimu.com