

REVELATION TECHNOLOGIES

Dali User's Guide

Version 2.2

REVELATION TECHNOLOGIES

Dali User's Guide

Version 2.2
November 2006

Copyright © 2006 Revelation Technologies, LLC
All Rights Reserved

Contents

Installing and Configuring Dali	1
Installation	1
Configuration	1
Redistribution	1
Using Dali	1
Understanding Dali	1
Getting Started	2
Dali Attributes	6
Dali on the Web	8
Advanced Features	9
Deriving From DaliObject	9
Implementing IDaliObject	12
Using Connections and Transactions	13
Using Stored Procedures	13
Data Manager Options	16
Working with Multiple Databases	17
Concurrency	18
Singletons	18
Passing Data Between Tiers	19
Using Dali Without a Database	20
Implementing Typical Scenarios	21
Deep vs. Shallow Loading	21
Storing Objects as Blobs	27
Going on From Here	28

Installing and Configuring Dali

Installation

Dali is packaged as a .NET assembly. In addition to this assembly, the Dali installation package includes Dali Studio—a tool that can be used to generate data classes—as well as documentation and code samples. To install all of these, simply run the supplied Windows® Installer MSI package. The installer will ask you to choose an installation directory and will walk you through the installation. After installation, you will find three directories under your root installation directory. These are:

- **Bin:** Contains the Dali assembly (revelation.dali.dll). This is the file you will redistribute with your application. Also in this directory are DaliStudio.exe and the IntelliSense documentation file (revelation.dali.xml). This file provides information to Visual Studio. This file needs to be in the same directory as the assembly during development. It does not need to be (and should not be) distributed with the application.
- **Documentation:** Contains this User's Guide and other documentation about Dali.
- **Samples:** Contains sample source code that demonstrates how to use Dali.

The installer will add shortcuts to your Start menu for all of these.

Configuration

If you use Microsoft® Visual Studio® for your development you will need to add a reference to the Dali assembly within your project. With your project selected, select Project > Add Reference... from the menu. You should find Revelation.Dali.dll on the .NET tab. If not, click the Browse tab and locate the revelation.dali.dll file in the directory indicated above. Dali will now be available to your project.

Redistribution

You are free to distribute the Dali assembly (revelation.dali.dll) with your application, in accordance with the end user license agreement. How you distribute it is up to you. If you prefer the XCOPY approach, simply include the assembly with your other files. If you prefer to create a Windows® Installer package you may include the DLL in a generated MSI or CAB file. This document does not cover deployment strategies and techniques. For these, please consult the appropriate Microsoft® documentation. Note that this is the only file you may distribute with your application.

Using Dali

Understanding Dali

The first class in Dali you should get familiar with is the DataManager. This is the class that your application software interacts with to load and save data. The DataManager contains methods like Load() and Save() that take an object as an argument. The DataManager knows how to load and save the object's data. DataManager does not care what type of object you pass to it. The object does not need to have anything special in it. In the simplest case you can

use existing classes that you wrote before you even knew about Dali. `DataManager` does, however, need to know a little about your database. `DataManager` has three constructors, all of which take a `DaliProvider`-derived object as the first argument. `DaliProvider` is an abstract base class that all `DaliProviders` derive from. `DaliProviders` provide database-specific information and functionality to the `DataManager`. For now it is just important to note that Dali is supplied with three `DaliProvider` classes:

- `SqlDaliProvider` is used for connecting to Microsoft SQL Server™ databases
- `AccessDaliProvider` is used for connecting to Microsoft Access databases
- `DsDaliProvider` allows you to use a `DataSet` as the underlying database

If you are not using Microsoft SQL Server™ or Microsoft® Access you can create your own `DaliProvider` for almost any database fairly easily.

Using a `DataSet` as a data source enables Dali to be used in an n-tier environment. This is an advanced topic that will be discussed later. For now we'll keep it simple and assume Dali is talking directly to your database.

Getting Started

Let's get started using Dali. There are several constructors you may use to create a `DataManager`. The first argument is always an instance of a `DaliProvider`-derived class. The optional second argument is either a connection string or a connection object. The simplest way to create a `DataManager` is using a connection string. If you are using Microsoft SQL Server™ your code would look like this:

[Visual Basic]

```
Dim connectionString As String = "..."  
Dim provider As New SqlDaliProvider  
Dim dataManager As New DataManager (provider, connectionString)
```

[C#]

```
string connectionString = "...";  
SqlDaliProvider provider = new SqlDaliProvider();  
DataManager dataManager =  
    new DataManager (provider, connectionString);
```

If your program only needs one `DataManager` (as most do) you can keep a static (Shared) reference to it in the application. If you plan to do this, the `DataManager` class provides such a reference named `Default`. Rather than creating a `DataManager` instance as described above you can call the `SetDefaultDataManager()` method on the `DataManager` class, passing it the same arguments you would pass to the constructor:

[Visual Basic]

```
Dim connectionString As String = "..."  
Dim provider As New SqlDaliProvider
```

```
DataManager.SetDefaultDataManager (provider, connectionString)
```

[C#]

```
string connectionString = "...";  
SqlDaliProvider provider = new SqlDaliProvider();  
DataManager.SetDefaultDataManager (provider, connectionString);
```

You can then refer to the single default instance using the `Default` property. Whether you use the `DataManager`'s built-in `Default` or declare your own is up to you. If you use `Default`, just substitute `DataManager.Default` for all instances of `dataManager` in all following examples.

The easiest way to start using Dali is to use your existing business classes as-is. Without inheriting from a base class or implementing any special interfaces, Dali can associate your classes with tables in your database. All you need to do is pass an object to one of the data maintenance methods in a `DataManager`. The `DataManager` will automatically load the object from the appropriate table or store the object's data to that table, depending on the method invoked. Here's an example:

[Visual Basic]

```
Dim customer As New Customer  
customer.CustomerID = 10021  
customer.Name = "John Doe"  
customer.Address = "123 Poplar St."  
DataManager.Default.Save (customer)
```

[C#]

```
Customer customer = new Customer();  
customer.CustomerID = 10021;  
customer.Name = "John Doe";  
customer.Address = "123 Poplar St.";  
dataManager.Save (customer);
```

The customer object is now stored in your database. At this point you must be wondering how the `DataManager` knows into which table to store the customer data and also how it knows which fields in the object map to which columns in the table. Remember that this is the simplest way of using Dali. Therefore, the answer must also be simple: It looks for a table with the same name as the class, and columns with the same names as the object's fields or properties. So as long as the database contains a table named `Customer`, and the `Customer` table contains columns named `CustomerID`, `Name` and `Address`, the above code will work. In many cases, the simple nature of this may be just what you need. If you have the luxury of creating both the database and the application's data classes, simply choosing the same names for related items can give you this functionality automatically with no additional effort. Later we'll discuss ways to change this default naming mechanism. For now our goal is to get acquainted with the `DataManager`.

In addition to being able to save a new object as shown above, you can also load, update or delete an existing object just as easily. Here's an example:

[Visual Basic]

```
Dim customer As New Customer
customer.CustomerID = 10021
dataManager.Load (customer)
Console.WriteLine ("Address was {0}", customer.Address)
customer.Address = "246 Spruce St."
dataManager.Save (customer)
```

[C#]

```
Customer customer = new Customer();
customer.CustomerID = 10021;
dataManager.Load (customer);
Console.WriteLine ("Address was {0}", customer.Address);
customer.Address = "246 Spruce St.";
dataManager.Save (customer);
```

In this example we created a new Customer object and loaded it with data from the database for the customer having CustomerID = 10021. We then changed the address and saved this customer back to the database. In order for the DataManager to find the appropriate data we had to populate the field representing the primary key. In this case the CustomerID column in the database is the primary key. By setting the CustomerID field in the Customer object, the DataManager was able find the appropriate data to load into the Customer object. Notice that we used the same Save method to update that we used when we initially created the customer. The Save method can be used to insert new data or update existing data. This relieves your code from the responsibility of keeping track of whether the data already exists or not. If you prefer to be explicit you can call the Insert or Update methods on the DataManager instead. The Insert method will only insert new data. It will throw an exception if data with the same primary key has already been inserted. Similarly, the Update method will only update existing data. It will throw an exception if data with the specified primary key does not already exist in the database. In some cases you may want this explicit behavior.

Finally, to delete an object you would populate the primary key fields of the object and pass it to the Delete method on the DataManager, as shown below:

[Visual Basic]

```
Dim customer As New Customer
customer.CustomerID = 10021
dataManager.Delete (customer)
```

[C#]

```
Customer customer = new Customer();
customer.CustomerID = 10021;
dataManager.Delete (customer);
```

Typically, you would not create an object just for the purpose of deleting it. You would probably already have this object around when you decide to delete it. But if you don't already have an instance of the object there is a more efficient way to delete objects. This will be discussed later.

So far we've shown how you can retrieve a single object from the database using its primary key. But what if you don't know its primary key value? Or what if you want to load more than one object that meets certain criteria? For these you could use the Find method. The Find method will find all objects of the specified System.Type meeting the specified criteria, sorted in the specified order. It returns a strongly typed list of objects. Here's an example:

[Visual Basic]

```
Dim customers As List(Of Customer) = _
    dataManager.Find(Of Customer) ( _
        "Name LIKE 'John%'", "Name, Address")
```

[C#]

```
List<Customer> customers = dataManager.Find<Customer> (
    "Name LIKE 'John%'", "Name, Address");
```

In this example we've chosen to retrieve all Customer objects that have a Name that begins with John, sorted by Name, then Address. This method uses Generics to return a type-safe list of the appropriate type. The type specified (Customer) indicates the type of objects we'd like to find. This also gives the Find method a hint as to which table to look in. The first argument represents the selection criteria. This criteria is data source specific. For SQL compliant databases such as Microsoft SQL Server™ and Microsoft® Access, this is essentially the criteria from a SQL WHERE clause. If you are using a non-SQL data source you will need to use the selection criteria language native to that data source. The second argument indicates the order in which objects will be listed. This is also data source specific. For SQL compliant databases it is simply the names of the fields on which to sort, separated by commas. This is the same as the SQL ORDER BY clause. For non-SQL compliant data sources, again you will need to use the native sort language, if one is provided. Setting the criteria argument to null (Nothing in VB) or an empty string will return all objects of the specified type. Setting the order argument to null (Nothing in VB) or an empty string will return the objects in an order determined by the data source.

We mentioned earlier that you can delete objects without instantiating them. This is accomplished with another overload of the Delete method. This overload is similar to the Find method. It takes the type of object and the delete criteria. These are identical to the Find method. So, for example, the following code will delete all customers named John:

[Visual Basic]

```
Dim count As Integer = dataManager.Delete (Of Customer) ( _
    "Name LIKE 'John%'")
```

[C#]

```
int count = dataManager.Delete<Customer> (
    "Name LIKE 'John%'");
```

This Delete method's return value will tell us how many customers it deleted. You can also count the number of objects that meet specific criteria without deleting them by calling the Count method. This is called exactly the same as the Delete method.

[Visual Basic]

```
Dim count As Integer = dataManager.Count (Of Customer) ( _
    "Name LIKE 'John%'")
```

[C#]

```
int count = dataManager.Count<Customer> (
    "Name LIKE 'John%'");
```

<p>Note: If the underlying database supports parameterized queries, by default Dali will construct parameterized queries for these Find, Delete and Count methods. Parameterized queries are desirable as they reduce the possibility of SQL injection attacks.</p>
--

There is a lot more to the DataManager than what has been discussed so far. But this should be enough to get you started. In later sections we will discuss Dali's more advanced features.

Dali Attributes

Up until this point we've assumed that the data class name was the same as its associated table's name. We further assumed that the field and property names in the class were the same as the column names in the table. This is ok when you have the freedom to choose the names yourself. In many cases, however, a database administrator (DBA) chooses the database table and column names. Many times these names are not appropriate for class names. Many DBAs like to use underscores and all lower case letters in table and column names. This style is not consistent with common programming conventions. For example, "acme_customers" may be an appropriate name for a database table, but it is not a good name for a class that represents a customer. For this, "Customer" (singular form, proper case, no underscores) would be a better name. Fortunately, Dali allows you to name your classes differently than their associated tables. This is done by simply adding a DaliClass attribute to the class. The attribute indicates the name of the table that will store data from the class. Here's an example:

[Visual Basic]

```
<DaliClass (TableName := "acme_customers")> _
Public Class Customer
    ' class definition
End Class
```

[C#]

```
[DaliClass (TableName = "acme_customers")]
public class Customer
{
    // class definition
}
```

The above code instructs Dali to associate all operations on a Customer object with the acme_customers database table. Similarly, you can add DaliMember attributes to the fields and properties (data members) of the class to indicate which columns they are associated with:

[Visual Basic]

```
<DaliClass (TableName := "acme_customers")> _
Public Class Customer

    <DaliMember (ColumnName := "cust_id")> _
    Public CustomerID As Integer

    <DataMember (ColumnName := "cust_name")> _
    Public Name As String

End Class
```

[C#]

```
[DaliClass (TableName = "acme_customers")]
public class Customer
{
    [DaliMember (ColumnName = "cust_id")]
    public int CustomerID;

    [DataMember (ColumnName = "cust_name")]
    public string Name;
}
```

The code above indicates to Dali that the CustomerID field will be stored in the cust_id column and the Name field will be stored in the cust_name column.

Most databases (certainly all relation databases) have a concept of a primary key. This is one or more columns in a table that uniquely identify a row of data. As mentioned earlier, these primary key fields need to be populated prior to performing Load, Update and Delete

operations. Dali needs to know which of these fields comprise the primary key when accessing the database. Dali is able to detect primary keys in the database automatically. However, there may be cases where you (or your DBA) decide not to explicitly create primary keys. Instead you may use indexes and write your code such that certain columns act as the primary key even though the database does not recognize them as such. It is generally considered bad practice not to create a primary key. However, since it is permitted by databases, it is permitted by Dali. In these cases you can inform Dali that a data member should be treated as a primary key by using the `IsKey` attribute argument as follows:

[Visual Basic]

```
<DataMember (ColumnName := "cust_id", IsKey := True)> _
public int CustomerID
```

[C#]

```
[DataMember (ColumnName = "cust_id", IsKey = true)]
public int CustomerID;
```

When doing this, you need to be careful that the field or fields you designate as key fields do actually uniquely identify an object otherwise you may get unexpected results. If for instance you indicate that the `CustomerID` field is a key field as shown above, yet the table contains multiple rows with the same customer ID, `Load` will randomly load one of them, but `Delete` and `Update` will delete or update all of them. In most cases your tables should have a primary key. When this is the case, Dali will automatically detect this and act accordingly. It is only under rare circumstances that you should set the `IsKey` argument explicitly.

You can see from these examples just how easy it is to map your existing classes to database tables. The only code you need is a few additional attributes. Since the table and column mappings stay close to their associated code it is easy to keep these in sync if the database structure changes. It also improves readability by showing exactly where the data will come from. This historically has been one of the major disadvantages in conventional data access layers and other data mapping techniques.

Dali on the Web

Now that you know the basic functions of Dali, let's see how it can be used in ASP.NET. As you know, ASP.NET pages get compiled into classes. When a user hits a page, the worker process creates an instance of the page class. Your page class contains fields for text boxes, list boxes and other user-interface controls that display and collect data. Dali is designed to transfer data between objects and tables. You've already seen how data in a field in an object can get transferred to and from a column in a table. So why should Dali work any differently for Page objects? It doesn't. Dali recognizes the special types of fields used in web pages—namely objects derived from `System.Web.UI.Controls.WebControl` (such as `TextBox`, `Checkbox`, `DropDownList`, etc.) and knows how to set and get these values just as it does simple datatypes like integers and strings. What all this means to you is that you can treat your ASP.NET web page's class just like any other class and apply the same Dali principles to it. Just as before, if your page's code behind class has the same name as a database table, Dali will automatically know where to store its data. If not, you can apply a `DaliClass` attribute and set its `TableName`

property. If the controls on the page (TextBoxes, etc.) have the same names as their associated columns, Dali will automatically know where to save and retrieve their data. Otherwise you can add `DaliMember` attributes to them as well. All of the same rules apply. Here's a sample of the relevant parts of a Page class using Dali attributes:

[Visual Basic]

```
<DaliClass (TableName := "acme_customers")> _
Public Class Customer
    Inherits System.Web.UI.Page

    <DaliMember (ColumnName := "cust_id")> _
    Protected WithEvents CustomerID As DropDownList

    <DaliMember (ColumnName := "cust_name")> _
    Protected WithEvents Name As TextBox

End Class
```

[C#]

```
[DaliClass (TableName = "acme_customers")]
public class Customer : System.Web.UI.Page
{
    [DaliMember (ColumnName = "cust_id")]
    protected DropDownList CustomerID;

    [DaliMember (ColumnName = "cust_name")]
    protected TextBox Name;
}
```

In your page's Load event handler you could load the Page object by simply using the `DataManager`'s Load method. Similarly, a button's Clicked handler could call `DataManager`'s Save method, passing it the Page object. The web sample included with Dali provides a very simple working example of how to do this and more.

Advanced Features

Deriving From DaliObject

Dali provides a `DaliObject` abstract base class. This is an optional class from which you may derive your data classes. Although it is not necessary to derive your classes from `DaliObject`, doing so provides a lot of additional functionality. One very useful function added by `DaliObject` is events. When a class derives from `DaliObject` it inherits a collection of events that can be used to trap and handle various state changes and database interactions. For example, saving an object raises a Saving event when the save operation begins and a Saved event when the save operation completes. This is very useful for aggregated objects as will be demonstrated later. The `DaliObject` base class also provides state management to keep track of whether an object is new (hasn't been saved yet), deleted, modified or unchanged. It indicates this through its `DataState` property. It keeps track of whether it has been changed by providing

a SetModified method that you can call whenever you change an object's data. Let's start with this method.

The SetModified method in DaliObject is intended to be called either by the object modifying data, or by properties or methods on the DaliObject-derived class. The latter is preferred, since it is easier and less error-prone. Here's an example:

[Visual Basic]

```
Private name_As String
Public Property Name As String

    Set (ByVal value As String)
        SetModified ("Name", name_, value);
        name_ = value;
    End Set

    Get
        Return name_
    End Get

End Property
```

[C#]

```
public string _name;
public string Name
{
    set
    {
        SetModified ("Name", _name, value);
        _name = value;
    }
    get
    {
        return _name;
    }
}
```

The first argument indicates the name of the field or property being changed. In this case, it is the "Name" property. The second argument indicates the value *before* the change and the third indicates the *new* value. Whenever SetModified is called a Changed event is raised. This event passes a DaliObjectFieldChangedEventArgs object, which contains these three values. You may find it useful to trap this event and use the provided information to log the change that was made. If you don't see any use for this information, you can call the simpler overload of SetModified that takes no arguments. Calling either of the SetModified overloads will set the DataState of the object to Modified, with two exceptions. First, if the object is currently being loaded, properties on the object will need to be accessed. In this case we wouldn't want the status set to Modified. So during a load the state will not be changed. Second, if the object's

current state is New this means the data source has no knowledge of it. Therefore, setting it to Modified would not make sense. So in these two cases, SetModified will not change the DataState.

There is one major benefit to having DataState accurately indicate when an object is modified; it prevents unnecessary database operations. If you call the Save or Update method on the DataManager and it cannot determine whether an object has changed since it was loaded, it will save the object regardless. If the object has not changed, this wastes time and resources. However, if the state is determined to be Unchanged when the Update method is called, no update will even be attempted. Of course you could keep track of each object's state in your application and only call Update when necessary. But having Dali do this for you simplifies your code. Whether or not the DataState is used to determine if an object should be saved is a configurable option that will be discussed later.

Now let's look at the events provided by DaliObject. In addition to the Changed event described above, there are two events associated with every database operation. One event indicates the operation is about to begin (pre-events). The other indicates it has completed (post-events). The events are:

Pre-Event	Post-Event
Loading	Loaded
Saving	Saved
Updating	Updated
Inserting	Inserted
Deleting	Deleted

The pre-events pass a DaliObjectChangingEventArgs. This class contains an Abort property that any event handler can set to true to indicate that it wishes to abort the current operation. When this happens, an Aborted event is raised and no further event handlers are invoked for the event. The post-events pass a DaliObjectChangedEventArgs. This contains a ChangeStatus property that indicates the status of the operation (Success, Failed, Aborted or NoChange).

Since a Save operation can involve either an Update or an Insert, these events will also be raised when saving. If the DataManager is configured to use the DataState property, it will determine whether to Insert or Update based on this. In this case events will be raised in the following order: Saving, Updating, Updated, Saved (if the object is not a new object) or Saving, Inserting, Inserted, Saved (if the object is new). If the DataManager is configured *not* to use DataState the order will be: Saving, Updating, Updated, Saved (if the object is not new) or Saving, Updating, Updated, Inserting, Inserted, Saved (if the object is new). It raises the extra update events because it first attempts an update. If it couldn't find anything to update it then proceeds to perform an insert.

One more benefit of DaliObject derived classes is that DaliObject implements Load, Insert, Update, Save and Delete methods. These methods work just as their DataManager counterparts, except they work on the object itself. So your code will be a little cleaner:

[Visual Basic]

```
customer.Save ()
' instead of
dataManager.Save (customer)
```

[C#]

```
customer.Save ();
// instead of
dataManager.Save (customer);
```

This works because DaliObject holds a reference to the DataManager that loaded it. Thereafter, all data access methods will use the same DataManager. There are four ways the object can get a reference to a DataManager:

- Explicitly setting the DataManager property on the DaliObject derived class.
- Creating the object using the Create method on a DataManager.
- Loading the object using the DataManager.Load(obj) method.
- If none of the above, it will use the Default DataManager (described earlier) if available.

See the Dali Programmer's Reference for more information about these.

Implementing IDaliObject

Deriving from DaliObject certainly has its benefits. But what if you already have another base class you need to derive your class from? The .NET Framework does not permit multiple inheritance. So what do you do? You can implement the IDaliObject interface without extending DaliObject. There are two steps:

- Have your class implement IDaliObject. This is just a simple reference in the class definition.
- Create an instance of DaliObject within your class and create wrappers for all its methods, properties and events.

This may seem tedious but it's not too bad. What you may want to do though, is create one such class and have your class derive from it. It, of course, will derive from whatever base class you specified, while also implementing IDaliObject. Actually, the sample provided with Dali includes a class that does this. Look for FauxDaliObject.cs in the samples directory after installing Dali. This class implements IDaliObject and delegates to a real DaliObject instance. You can use the code in this example as-is and simply have it derive from your base class. See this sample class for more information.

Using Connections and Transactions

The DataManager supports both manual and automatic connections. In our previous examples we used automatic connections. When you create a DataManager giving it a connection string, the default behavior is to automatically create connections as needed. This means that each operation (Load, Update, Insert and Delete) will create a new connection. If you are using a data provider that supports connection pooling, such as the Microsoft SQL Server™ provider, this is the most scalable and efficient way to use connections. There are times, however, when you may already have a connection and you would like the DataManager to use it. Rather than providing the DataManager with a connection string, you can provide it with a connection object. The connection may be open or closed. If the connection is already open, the DataManager uses it and leaves it open. If it is closed the DataManager will automatically open and close it as needed.

Whether you provide a connection or a connection string you can specify whether the DataManager should wrap each method call in a transaction. Although an Update operation, for example, only updates one object, event handlers for that object's Updated event may update other objects. Those other objects' Updated event handlers may update still other objects. If you tell the DataManager to use transactions, all of these updates will be wrapped in a single transaction.

As of version 2.0 of the .NET Framework there is another means of handling transactions using a TransactionScope object. Using a TransactionScope is one of the easiest ways of managing transactions. All Dali operations may be contained within a TransactionScope block if desired. For more information on TransactionScope, please refer to the Microsoft .NET Framework documentation.

Using Stored Procedures

So far we have only discussed using Dali directly with tables. However, many developers prefer to use stored procedures to perform data access. In fact, some database administrators require that all data access is through stored procedures. Fortunately, Dali allows you to use stored procedures for all its loading, saving, deleting and finding operations.

Stored Procedures for Single-Object Operations

Dali gives you two ways to use stored procedures for single object operations. The first and simplest is to explicitly name the stored procedures you wish to use for each data class. DaliClassAttribute contains for properties for this named SelectProc, InsertProc, UpdateProc and DeleteProc. Setting any one or all of these to the name of a stored procedure tells Dali to use that stored procedure for performing the respective operation. For example:

[Visual Basic]

```
<DaliClass ( _
    TableName := "acme_customers", _
    SelectProc := "sp_select_customer", _
    DeleteProc := "sp_delete_customer")> _
Public Class Customer
    ' class definition
End Class
```

[C#]

```
[DaliClass (
    TableName = "acme_customers",
    SelectProc = "sp_select_customer",
    DeleteProc = "sp_delete_customer")]
public class Customer : System.Web.UI.Page
{
    // class definition
}
```

In the above example, the `DaliClassAttribute` instructs Dali to use the `sp_select_customer` stored procedure for loading a customer's data, the `sp_delete_customer` stored procedure for deleting a customer's data, and to go directly to the table for everything else. The presence of a stored procedure name in one of these properties overrides the default behavior of acting directly on the table. While this is the simplest way to get started it can have two undesirable effects. First, you need to explicitly specify the name of every stored procedure for every class. Second, if you plan on using your program with multiple databases and some of them do not support stored procedures, your code will fail on those. To get around these, you may use another argument to the `DaliClassAttribute`, `UseProcs`. When `UseProcs` is set to true and the database in use supports stored procedures, Dali will look for stored procedures that follow a specific naming convention. If `UseProcs` is false or the database in use does not support stored procedures, Dali falls-back to its default behavior and accesses the table directly. This is the most convenient way to use stored procedures, assuming you follow some form of naming convention for your stored procedures.

Stored Procedure Naming Convention

Dali is fairly flexible in adhering to your existing naming convention. By default it expects all stored procedure names to be in the form: `<TableName><StatementType>` where `<TableName>` is the name of the table and `<StatementType>` is either `Select`, `Insert`, `Update` or `Delete`. So, for example, your procedures for accessing the `Customers` table should be named: `CustomersSelect`, `CustomersInsert`, `CustomersUpdate` and `CustomersDelete`. You can easily change this default expectation by specifying your own templates. You may specify one template for all procedure types or separate templates for each statement type. These templates are specified in the `DataManagerOptions.StoredProcedure` object. The templates may contain the replaceable parameters `%TableName%` and `%StatementType%`. For example:

[Visual Basic]

```
dataManager.Options.StoredProcedure.DefaultTemplateString = _
    "sp_%TableName%_%StatementType%"
dataManager.Options.StoredProcedure.SelectTemplateString = _
    "sp_sel_%tablename%"
```

[C#]

```

dataManager.Options.StoredProcedure.DefaultTemplateString =
    "sp_%TableName%_%StatementType%";
dataManager.Options.StoredProcedure.SelectTemplateString =
    "sp_sel_%tablename%";

```

In this example, Dali will expect that for accessing data in the Customers table, the database contains stored procedures named: `sp_Customers_Insert`, `sp_Customers_Update`, `sp_Customers_Delete` and `sp_sel_customers`. Note that three of these follow the default template, while the last follows the specific Select template. Specific templates override the default template. Also note the casing. When specifying the replaceable parameters you may use mixed case, all lower case or all upper case. The resulting string will follow your casing for that replacement. Using all lower or all upper case creates names with all lower or upper case letters. Using mixed case leaves the table name in the form specified in the database and the statement type in mixed case. In the example, we used `%tablename%` to indicate we want the table name to be all lower case. Database element names are typically not case sensitive, but if yours is you can use this technique to match your procedure names' casing styles.

These templates should give you enough flexibility to match whatever naming convention you have. In cases where they don't you can still explicitly override procedure names in the `DaliClassAttribute`.

Stored Procedure Contents

Naming the procedures appropriately is the first step. The second is filling them with the proper statements. Like any program, Dali is expecting these procedures to perform a specific task. It further expects them to take specific parameters and possibly return a specific result. For example, when loading a customer the stored procedure that performs a `SELECT` is expected to take parameters for all primary key columns and return a result set containing all the columns used by the class. For example, if your `Customer` class represents the `acme_customers` table and the class contains fields representing the columns `cust_id` (key), `first_name` and `last_name`, your procedure definition should look something like this:

[SQL]

```

CREATE PROCEDURE acme_customers_select (
    @CustomerID Int
)
AS

    SELECT cust_id,
           first_name,
           last_name
    FROM acme_customers
    WHERE cust_id = @CustomerID

```

Note that the actual syntax may vary for your database and the procedure name will depend upon your name templates, as explained above. Other stored procedures will also need to expose an appropriate interface. The Update and Delete procedures pose a special problem in that different concurrency models require different interfaces and functionality. See the Dali Programmer's Reference for more information on creating appropriate stored procedures. Also, Dali Studio is capable of generating appropriate stored procedures for all concurrency models. Whether you choose to use it or not in your regular development, you may want to have it generate a few procedures using different concurrency models to use as examples.

Stored Procedures for Multiple-Object Operations

In addition to using stored procedures for single object operations, you may also use them in the Find and Delete methods when operating on multiple objects. These methods do not use the procedures described above, nor do they follow any naming conventions. You may specify any procedure name in the Find or Delete methods. See the Dali Programmer's Reference for more information about these.

Data Manager Options

When creating a DataManager you can specify certain options. These options are encapsulated in a DataManagerOptions object. This object contains all the configuration options for the DataManager. DataManager contains a reference to this object. You can either pass a DataManagerOptions object to it or set the properties in the contained object. As mentioned earlier, when deriving from DaliObject you have the option to use the DataState to determine whether an object needs to be inserted or updated or to do nothing. This option is set by the UseObjectState property. You can also specify whether the DataManager should use transactions by setting the UseTransactions property. If your database supports different transaction isolation levels, as Microsoft SQL Server™ does, you can set the IsolationLevel property to one of the enumerated constants in the System.Data.IsolationLevel enumeration.

The DataManagerOptions class also contains properties that determine whether or not you need to be explicit about indicating the classes and fields that will be mapped to the database. As mentioned earlier, no attributes are required. By default, DataManagers will look for a table with the same name as a class, and column names with the same names as the fields and properties. If you want to turn off this behavior you can set one of three settings. First, ExplicitTableNames can be set to indicate that a DaliClassAttribute with a TableName must be used on every class that will access data. If this property is set to true, attempting to load or save an object that does not have this attribute will throw an exception. Second, ExplicitMembers indicates whether all data members that are to be stored in the database need DataMemberAttributes. If this option is set and a data member does not have a DataMemberAttribute, it will be ignored in database operations. Finally, ExplicitColumnNames indicates whether all data members that are to be stored in the database need to have an explicit ColumnName specified in their DataMemberAttribute. If set, attempting to load or save an object that does not have explicit column names on all DataMemberAttributes will throw an exception. The following shows how to set various options:

[Visual Basic]

```
dataManager.Options.ExplicitMembers = True
dataManager.Options.ExplicitTableNames = True
dataManager.Options.ExplicitColumnNames = True
dataManager.Options.UseObjectState = True
dataManager.Options.UseTransactions = True
dataManager.Options.IsolationLevel = IsolationLevel.Snapshot
```

[C#]

```
dataManager.Options.ExplicitMembers = true;
dataManager.Options.ExplicitTableNames = true;
dataManager.Options.ExplicitColumnNames = true;
dataManager.Options.UseObjectState = true;
dataManager.Options.UseTransactions = true;
dataManager.Options.IsolationLevel = IsolationLevel.Snapshot;
```

Working with Multiple Databases

With Dali you are not locked in to one type of database. Dali's interface is very generic and not designed around any specific vendor's database. In most cases you can change databases by simply changing your DaliProvider and your connection string. The rest of your Dali-based code should continue to work without change. This makes Dali very attractive to those who know they will be changing databases in the future. It is also attractive to those writing software for clients using various databases.

There are very few differences in the way you work with different databases when using Dali. But while Dali does a good job of providing a consistent interface across all databases, there are some differences that are beyond Dali's control. If you are in a position where you know you must support multiple databases—now or in the future—there are a few things you need to keep in mind. First is in the criteria and sort expressions. These will be generally the same for most SQL compliant databases. However, there's a chance that your database's query language is not SQL compliant, so you may need to make some minor changes. Second, is in the IsolationLevel property of the DataManagerOptions. If you select an isolation level that is not supported by your database you may experience problems. Finally, there may be differences in features supported by your database. All of the features supported by Dali are encapsulated in the DaliProvider-derived classes. For example, some databases allow identifiers (table and column names) to contain spaces and other special characters. Some do not. Those that do typically use quoted identifiers. The SQL-92 standard specifies that databases should allow you to wrap your table and column names in quote characters as in: "My Table" whenever there is a space or special character in the name. Some databases use other characters. For example, Microsoft® Access uses square brackets as in: [My Table] to enclose identifiers. If you create a custom DaliProvider for your database you can specify which characters to use. But again, keep in mind that some databases do not allow special characters at all. If you create an application that must work with multiple databases, avoid using names that contain spaces and other special characters. You also need to be mindful that some databases do not support special data types like GUIDs. Dali will do its best to try to convert one data type to another if it is necessary. For example, if your class contains a Guid field, but your database does not support these, you can use a CHAR or VARCHAR instead. But if you are relying on the database to generate these values you may need to rewrite some of your code if you switch to a

database that can not generate these. The same is true of triggers. If your database supports triggers and your application relies on them you will need to redesign part of your application if you switch to a database that does not support them.

While Dali makes great strides in isolating your code from the underlying database there are some things beyond its control. So, in general, try not to rely on any special features of your database unless you know you will always be using that database.

Concurrency

This is a design decision that must be considered before developing any database-aware application. The question, simply stated, is: “what do you do if two users attempt to update the same data at the same time?” Should the second person’s changes overwrite the first or should the second person be warned that the data has changed? The answer is not always simple. It depends on the application and the specific situation. You will have to decide this. But whatever your decision, Dali supports concurrency checking in several different ways. This is one of the options in the `DataManagerOptions` object. You can set the `ConflictOptions` property to one of the enumerated constants in the `System.Data.ConflictOptions` enumeration. The three options are:

- `OverwriteChanges` – Each update overwrites any previous changes.
- `CompareAllSearchableValues` – Current values are compared to previous values. An exception is thrown if they differ.
- `CompareRowVersion` – If the table contains a `RowVersion` column (known as a `TimeStamp` in Microsoft® SQL Server™) its current value is compared to the previous value. An exception is thrown if they differ.

Singletons

What do you do if there is a possibility that the application may ask for an object that was previously requested and is still in memory? Perhaps you have a multithreaded application that allows multiple users access to the same objects. The first user loads an object and makes some changes. Before saving the object the second user loads the same object. Here, the term *same object* means having the same key value(s). Concurrency checking, as described above, can be used to detect this situation and throw an exception. But in some cases it would be nice to just let everyone know what’s happening rather than throwing the exception after it happens. Dali can easily be configured for *singleton* mode. Singleton mode means that if two or more users request the same object they all get a reference to the exact same instance of the object. Changes made by one user are immediately available to other users. That way there are no surprises when the object is saved. Depending on your needs, this is an option you may or may not want to exercise. If you do, here’s how:

The `DataManager` has a `Cache` property. This property can be set to hold an instance of an `IDictionary`. Remember that `Hashtable` implements `IDictionary`. So by simply setting this property to an instance of a `Hashtable` your `DataManager` will use it to cache its objects. When an object is requested, the `DataManager` will first look for it in its cache. If it finds it there it will return a reference to it. If not, it will retrieve it from the database. If more than one client application or more than one thread in the same application requests the same object they will both receive a reference to the same object. Pretty cool, huh? Well, don’t answer that yet.

Although this can give you the singleton behavior you may want, there is a problem. The problem is: when does the Hashtable get cleaned out? As long as the Hashtable contains a reference to an object it will never be garbage collected. If the application runs forever and never empties this cache, the program will appear to have a memory leak. You can create a scheme whereby objects get cleaned out regularly—perhaps every 30 minutes. But what if the object is still being referenced by a client when you dump the cache? That's probably not a good idea. Well, if you think about the problem long enough you'll probably come up with a solution that meets your needs. You can create any caching algorithm you'd like as long as it implements the IDictionary interface. If you'd like a good solution “out of the box” consider the WeakHashtable.

WeakHashtable is a class that is part of the Dali assembly. This class extends Hashtable by adding weak references. If you're not familiar with weak references, these are object references that do not inhibit garbage collection. Normally, if you have a reference to an object, the garbage collector will ignore it when looking for objects to dispose of. However, if you have a weak reference the garbage collector can take it. WeakHashtable is a hashtable that holds weak references to the objects in its collection. As long as a client application is holding a reference to an object in the WeakHashtable, the object will not be collected. When WeakHashtable is the only thing holding the reference, the object can be garbage collected.

So, to set the DataManager to use singletons, and use the WeakHashtable, simply set it as follows:

[Visual Basic]

```
dataManager.Cache = New WeakHashtable
```

[C#]

```
dataManager.Cache = new WeakHashtable();
```

Passing Data Between Tiers

So far, everything discussed applied to a two-tier model. In a two tier model, the database server is considered one tier and the application software is considered a second tier. Typically the database server resides on one computer and the application resides on another. In all the examples up until now, your executable code (with Dali) ran on one tier, and Dali connected directly to the database. The two-tier model is alright for small to medium sized applications. But in large scale enterprise applications a three-tier or n-tier (three or more tiers) model is usually desirable. In a three-tier model the application software executes on one tier and the database runs on another tier, just as in the two-tier model. The difference is that the data access code resides on yet another tier (sometimes called the middle-tier). The Microsoft® .NET Framework provides an excellent vehicle for passing data between tiers—the DataSet. A DataSet can contain data arranged in tables with rows and columns just like a relational database. It can also contain relations and primary and foreign keys just like a database. If it looks like a database and acts like a database, Dali can interact with it just as if it were a database. It was mentioned earlier that there is a third type of DaliAdapter, namely DsDaliAdapter designed just for this purpose. DsDaliAdapter also derives from DaliAdapter just as the others do. So it performs all the same functions. When creating a DataManager to use the DsDaliAdapter you use the simplest constructor that takes only the DaliAdapter argument

(there is no connection string or connection associated with a DataSet). However, the DsDaliAdapter needs to be constructed with an existing DataSet supplied to it. So, to construct a DataManager that will use a DataSet your code will look like this:

[Visual Basic]

```
Dim data As New DataSet
Dim provider As New DaliProvider (data)
Dim manager As New DataManager (provider)
```

[C#]

```
DataSet data = new DataSet();
DsDaliProvider provider = new DsDaliProvider (data);
DataManager manager = new DataManager (provider);
```

Typically, your data tier will request data from one or more tables in your database, load it into a DataSet, and pass this DataSet to the application tier, business object tier or presentation tier (e.g., web servers). The data will then be manipulated and/or displayed and possibly updated. After being updated, the DataSet can be passed back to the data-tier where DataAdapters store the changes back to the database. In this scenario, Dali is well suited for the application, business and presentation tiers. On the presentation tier (typically web servers) Dali can easily transfer form field data to and from the DataSet. On the business and application tiers Dali can store and load business objects to and from a DataSet just as easily as it would to the database. And since Dali works with DataSets using the same methods used for accessing a database, your code can start out with a two tier model and easily scale to three or more tiers when needed.

There are some things to be mindful of when using DataSets with Dali rather than communicating directly with the database. For one, you must keep in mind that (typically) the DataSet does not contain all the data from the database, but only a subset. So you may need to create a separate DataManager instance for each user or each browser session. Second, after Dali has updated (through the Insert(), Update() or Save() methods) you must remember that it thinks the DataSet is the database, so it thinks the database is up-to-date. At an appropriate time you must update the database using a data adapter's Update() method. Finally, if you are passing the DataSet between tiers for update, remember that the tier on which it is originating does not know that another tier has saved the changes, so you must explicitly call the AcceptChanges method if you will continue to use it.

Using Dali Without a Database

Another reason for using DataSets with Dali is to use the DataSet *instead of* a database. There may be cases where your needs are very simple. Maybe you only need to maintain a small amount of data. You plan to expand your needs some day, but you don't want to invest in a database right now. Or maybe you have some clients with millions of records and others with only dozens. You want to write the application once to cover all situations. Usually, this means forcing everyone to use a database whether they need it or not. With Dali you don't need to do that. Remember that the DataSet works just like an in-memory database to Dali. In fact, the DataManager does not know the difference between a DataSet and an enterprise database like Microsoft SQL Server™. It treats them both the same and gives you virtually the same

interface. Also remember that the DataSet can be saved to an XML file. This means you can take the entire contents of a DataSet and transfer it to a file. You can then later recreate that DataSet from the file. Using this technique you end up with a “poor man’s” database. Granted, there are significant limitations. For one, the entire data file needs to be loaded into memory, so you won’t want to store too much data. Also, DataSets do not support transactions. But if you can live with those limitations a DataSet database may be just what you need. Perhaps you’re distributing a trial version of your software. Rather than supplying a full database (potentially with licensing and installation issues) simply supply a DataSet file. You may find other uses for this as well. You will also find that querying a DataSet is surprisingly fast. Just remember to save your dataset to a file at key points in your application. To save a DataSet to a file:

[Visual Basic]

```
Dim data As New DataSet
' work with DataSet...
data.WriteXml ("MyData.xml", XmlWriteMode.WriteSchema)
```

[C#]

```
DataSet data = new DataSet();
// work with DataSet...
data.WriteXml ("MyData.xml", XmlWriteMode.WriteSchema);
```

To reload it:

[Visual Basic]

```
data.ReadXml ("MyData.xml")
```

[C#]

```
data.ReadXml ("MyData.xml");
```

Be sure to use the WriteSchema option when writing the XML. This option adds all the schema information (tables, columns, etc.) so the DataSet can be completely reconstructed.

Implementing Typical Scenarios

Deep vs. Shallow Loading

When loading an object you have one very important consideration: should you deep load or shallow load? This is analogous to deep vs. shallow copying when cloning objects. To understand this, consider a customer object that contains a collection of order objects. Each order object contains a collection of order detail objects. Each order detail contains a reference to the product object being ordered. Each product contains a reference to the catalog object in which it appears. When loading a customer object do you want to load just the customer object (a shallow load) or should you also load the orders (a deep load)? When loading an order object

do you want just the order or do you want its associated details as well? You need to ask yourself this question down to the last reference. If the answer is always yes you may run into a big problem. Loading a single customer will end up loading all of that customer's orders with full details and the entire product catalog as well! This is probably not what you want. What you need to decide up-front is how deep you need to go when initially loading objects of each class. Maybe you don't initially need to load the orders with the customers. Maybe you only want to load the orders when a request is made for one. In this case if they are never needed they never get loaded. This is known as lazy initialization. Or maybe you only want to load particular order objects on an as-needed basis, and then discard them when they are no longer needed. It is relatively easy to implement all of these scenarios. But the first step is to identify how deep you want to go.

Dali provides automatic support for the most common scenarios. Dali gives you the choice of three depths of loading, namely Shallow, Full and Deep. These are specified in the LoadDepth enumeration. These depths are described as follows:

- **Shallow:** Only the specified object is loaded. If the object contains references to other objects these will not be loaded. In the previous scenario, loading the Customer will not load Order objects.
- **Full:** The specified object as well as the objects it directly references will be loaded. This does not include objects referenced by those objects. In the previous scenario, loading a Customer object will load the related Order objects, but not the OrderDetail objects referenced by the Order objects.
- **Deep:** The specified object as well as all objects it references directly or indirectly will be loaded.

In order to use this automatic load feature the parent class must contain either a single reference or a collection reference to hold the child object(s). If a collection, it must implement IList. This member must also have a DaliMember attribute. The DaliMember attribute must set the ChildClass, ParentKeyFields and ChildKeyFields properties. The ChildClass indicates the Type of object(s) that will be referenced by the member. ParentKeyFields and ChildKeyFields are comma separated lists of member variables representing the foreign key values in the parent and child, respectively. Generally, a primary key contains only one column and that column is used as the foreign key from the child table. In this case, the ParentKeyFields and ChildKeyFields properties will only contain this single field name. In the case of a composite primary key containing more than one column, the names of the related fields can be separated by commas. The following example shows a Customer class that contains a collection of Order objects and a single reference to a Sales Rep (Employee) object:

[Visual Basic]

```

<DaliMember(TableName := "Customers")> _
Class Customer
{
  <DaliMember(ColumnName := "customer_id")> _
  Public CustomerID As Integer;

  <DaliMember(ColumnName := "sales_rep_id")> _
  Public SalesRepID As Integer;

  <DaliMember( _
    ChildClass := typeof(Phone), _
    ParentKeyFields := "CustomerID", _
    ChildKeyFields := "CustomerID" _
    LoadDepth := LoadDepth.Deep)>
  Public Orders As List(Of Order);

  <DaliMember( _
    ChildClass := typeof(Employee), _
    ParentKeyFields := "SalesRepID", _
    ChildKeyFields := "EmployeeID" _
    LoadDepth = LoadDepth.Deep)>
  Public SalesRep As Employee;
}

```

[C#]

```

[DaliMember(TableName = "Customers")]
class Customer
{
  [DaliMember(ColumnName = "customer_id")]
  public int CustomerID;

  [DaliMember(ColumnName = "sales_rep_id")]
  public int SalesRepID;

  [DaliMember(
    ChildClass = typeof(Phone),
    ParentKeyFields = "CustomerID",
    ChildKeyFields = "CustomerID",
    LoadDepth = LoadDepth.Deep)]
  public List<Order> Orders;

  [DaliMember(
    ChildClass = typeof(Employee),
    ParentKeyFields = "SalesRepID",
    ChildKeyFields = "EmployeeID",
    LoadDepth = LoadDepth.Deep)]
  public Employee SalesRep;
}

```

In the above example you will also note that we have specified that the Orders and SalesReps should be “deep” loaded. This means that when the Customer class is loaded, these members and any child objects they contain will be loaded. If not specified, the default behavior is to shallow load members, which means they are not loaded automatically when the parent is loaded.

When loading an object using the Load() method on DataManager or DaliObject you may override the behavior in the DaliMembers. The Load() methods take an optional second parameter that indicates the desired depth. For example, if a member specifies a Deep load in its DaliMember attribute, but Load() specifies Shallow or Full, the object will be shallow or full loaded instead. This combination of setting LoadDepth on the member and the Load() method provides a lot of flexibility and should cover most situations. However, it may not cover every situation you need. If more flexibility is needed you can use the various events to perform specific actions as appropriate for your situation. This is discussed below.

Saving objects also follows the same rules and uses the same attributes. If you’ve deep-loaded a customer, it follows that you probably want to deep-save it as well. In other words, when you save the Customer you would also like to save any changes or additions made to the Orders collection and the SalesRep object. To deep- or full-save, simply follow the same semantics as you do for loading. Either specify Deep or Full in the DaliMember attribute or specify these in the Save(), Insert() or Update() method.

Deleting objects poses a problem when considering deep-deleting. When saving, only the objects that are present in the child collection will be saved (inserted or updated). If there are any objects in the database that were not loaded into the collection for any reason, they will not be available in the child collection and therefore will not be saved. But since they’re already in the database, they don’t need to be saved. When deleting a parent object, it is possible that its child objects are not currently loaded. Perhaps you only did a shallow load. If you attempt to delete this parent, Dali will not be able to delete the child objects that are not present in the child collection. This will either cause a foreign key constraint violation (if you have constraint checking) or worse, orphaned child objects. Of course, having deep-load capabilities, Dali could simply locate all the child objects and delete them. It would also have to locate and delete all the children’s children and the children’s children’s children, and so on. With lots of descendents, this can cause performance problems, constructing and loading objects just for the sake of deleting them. Worse yet, it could end up deleting more than you expected. As it attempts to follow references you may end up with some surprises. Where data is concerned, we believe surprises should be avoided. For this reason, Dali does not provide deep-delete capability. If you really want this you can implement it yourself using events as described below. If possible, however, it would be much more efficient to use your database’s cascading delete capabilities if it provides this. Microsoft SQL Server™ provides this capability, as do many other databases. Check your database documentation for details on cascading deletes.

Using Events for Deep Loading and Saving

As noted above, events raised by DaliObject-derived objects can be used to perform deep loads, saves and deletes and allow for greater control over how these operations occur. The descriptions that follow indicate how you can get the basic deep load/save functionality using events. You can adapt these descriptions as needed to suit your individual needs.

First we'll start with the deep load. You have a Customer class that contains a collection of objects from the Order class. When you load a customer object you want all its associated order objects loaded and added to the Customer class's orders collection field. The best way to implement this is in the Customer class; specifically, in a Loaded event handler. Remember that the Loaded event is called immediately after an object is loaded from the database. So right after the customer is loaded is when you would want to load the orders. Within the Loaded event handler you would call the Find method on the DataManager to find all related orders. You would use the result of this call to populate the orders collection in the customer:

[Visual Basic]

```
Public Class Customer
    Dim orders As List(Of Order)
    Public Sub Customer_Loaded (Sender As Object, _
                               e As DaliObjectChangedEventArgs)

        orders = DataManager.Find(Of Order) ( _
            "CustID=" & CustID, "OrderID")
    End Sub
End Class
```

[C#]

```
public class Customer
{
    List<Order> orders;
    Public void Customer_Loaded (object sender,
                                DaliObjectChangedEventArgs e)
    {
        orders = DataManager.Find<Order> (
            "CustID=" + CustID, "OrderID");
    }
}
```

When inserting or updating the customer object you will likely want to also save any related order objects. This is easily done in the Updating and Inserting event handlers. What you would do in each of these is iterate the orders collection, saving each order object. Note that you would want to do this in both the Inserting and Updating event handlers. In the Inserting handler you would need to insert. In the Updating you would need to save. Note that you want to save rather than explicitly update because some of the orders may have just been added to the collection so they will need to be inserted. Here are examples of both:

[Visual Basic]

```
Public Sub Customer_Updated (sender As Object, _
    e As DaliObjectChangedEventArgs)

    For Each (ord As Order In orders)
        ord.Save 'add new ones, update existing ones
    End Sub
Public Sub Customer_Inserted (sender As Object,
    e As DaliObjectChangedEventArgs)

    For Each (ord As Order In orders)
        order.Insert 'new customer, all orders are new
    End Sub
```

[C#]

```
void Customer_Updated (object sender,
    DaliObjectChangedEventArgs e)
{
    foreach (Order order in orders_)
        order.Save(); //add new ones, update existing ones
}

void Customer_Inserted (object sender,
    DaliObjectChangedEventArgs e)
{
    foreach (Order order in orders_)
        order.Insert(); //new customer, all orders are new
}
```

In the second scenario we want to defer loading Order objects until the first one is requested. In this case we are still deep loading the Customer, but we do not want to deep load until we need to. To accomplish this we would create the orders collection as a property rather than a simple field. The get method of this property would have the code to load the orders. This code would be exactly the same as in the first scenario's Loaded event handler. By putting it in the get method of a property, no orders will be loaded until the orders property is accessed, thereby deferring loading until it is necessary. This technique, in general, is known as lazy initialization.

[Visual Basic]

```

Private orders_ As List (Of Order)
Private Property orders As List (Of Order)

    Get

        If (orders Is Nothing) Then
            orders = DataManager.Find (Of Order) ( _
                "CustID=" + CustID, "OrderID")
        End If

        Return orders

    End Get

End Property

```

[C#]

```

private List<Order> orders_;
private List<Order> orders
{
    get
    {
        if (orders_ == null)
            orders_ = DataManager.Find<Order> (
                "CustID=" + CustID, "OrderID");
        return orders_;
    }
}

```

In the third scenario we only want to load order objects one at a time as needed. We do not want to keep an entire collection of orders in the customer object. But when we need a particular order we'd like to be able to get it through the customer. The first thing we need to do is determine how the caller will ask for a particular order. If these will be requested by their primary key value you can simply create a method that calls the Load method on the DataManager. Most likely though, callers will not know the primary key value. It is more likely they will want to retrieve one or more orders based on some criteria. For this you can create a method that calls the Find method using the specified criteria. Another variation of this is if the application wants to load 10 orders at a time for displaying in a form or web page. For this you could create a pair of methods that return the next 10 and previous 10. The customer object could keep track of current position and order objects could be cached in a List or other collection as they are loaded.

There are many different scenarios and many different implementations for each. Dali offers a lot of flexibility in this area. So finding an implementation that suits your application's needs should not be difficult.

Storing Objects as Blobs

Up until this point we haven't talked about the various types of data you can store. In fact, Dali can store just about anything. Dali checks data types of both your fields and database columns.

It does automatic conversions wherever necessary. For instance, if the database stores a particular value as an int, but your class represents it as a decimal, Dali will automatically convert it as necessary. There are two cases where conversions don't take place. One is when a field is a byte array. For these, the database must have the column defined as some type of binary data field such as a varbinary or image. These are also known as Binary Large Objects, or, BLOBs. The second case is when you want to store an object other than a base data type. For example, let's say a Customer object contains an instance of an Address object. The address class has several fields including street address, city, region and postal code. For whatever reason, you decide you don't want to store the address in a separate table and you don't want the individual fields stored in the customer table. All you want to do is take the entire Address object and store it along with the customer. The way you would do this is by first serializing the Address object to a byte array, then storing the byte array in a binary column in the table. Dali makes it easy for you to do this. Just make sure the Address class is serializable and Dali will do the rest. To make a class serializable, add the `System.SerializableAttribute` to the class definition. When `DataManager` encounters an object that is not convertible to one of the base types and is not a byte array, it attempts to serialize it to a binary byte array and store it.

When loading data from the database into an object all the same conversions take place. In the case of binary data, if the field into which the data is being loaded is defined as a byte array, it will be loaded with the bytes from the database. If, however, the field is defined as anything other than a byte array, `DataManager` will deserialize the binary data back into an object and set the field to this object.

There are a couple things to be mindful of when storing objects this way. For one, if the object contains references to other objects, by default, those objects will be serialized also. So you could end up storing more than you intended. The solution here is to add the `System.NotSerializedAttribute` to those fields that you don't want included in the serialization. The second point to be aware of is that binary database data cannot be searched. Storing a customer's address this way means you cannot do queries, for instance, to return addresses with a specific postal code. SQL has no way of looking inside the binary data or reconstructing the object. So storing objects this way is only useful when you know you will never want to search the database directly based on its data.

Going on From Here

Hopefully this guide has given you enough information to start using Dali. There is a lot more to it than what is presented here. Review the reference documentation and the sample files for more information. Also visit the Revelation Technologies website at www.revtechnologies.com for the latest information.