NHibernate - Relational Persistence for Idiomatic .NET

NHibernate Reference Documentation

4.1

Table of Contents

Preface

Working with object-oriented software and a relational database can be cumbersome and time consuming in today's enterprise environments. NHibernate is an object/relational mapping tool for .NET environments. The term object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema.

NHibernate not only takes care of the mapping from .NET classes to database tables (and from .NET data types to SQL data types), but also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and ADO.NET.

NHibernate's goal is to relieve the developer from 95 percent of common data persistence related programming tasks. NHibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the .NET-based middle-tier. However, NHibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

If you are new to NHibernate and Object/Relational Mapping or even .NET Framework, please follow these steps:

- Read Chapter 1, *Quickstart with IIS and Microsoft SQL Server* for a 30 minute tutorial, using Internet Information Services (IIS) web server.
- Read Chapter 2, *Architecture* to understand the environments where NHibernate can be used.
- Use this reference documentation as your primary source of information. Consider reading *Hibernate in Action* (java http://www.manning.com/bauer/) or *NHibernate in Action* (http://www.manning.com/kuate/) or *NHibernate 3.0 Cookbook* (https://www.packtpub.com/nhibernate-3-0-cookbook/book) or *NHibernate 2 Beginner's Guide* (http://www.packtpub.com/nhibernate-2-x-beginners-guide/book)if you need more help with application design or if you prefer a step-by-step tutorial. Also visit http://nhibernate.sourceforge.net/NHibernateEg/ for NHibernate tutorial with examples.
- FAQs are answered on the NHibernate users group.
- The Community Area on the NHibernate website is a good source for design patterns and various integration solutions (ASP.NET, Windows Forms).

If you have questions, use the NHibernate user forum. We also provide a JIRA issue trackings system for bug reports and feature requests. If you are interested in the development of NHibernate, join the developer mailing list. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

Chapter 1. Quickstart with IIS and Microsoft SQL Server

1.1. Getting started with NHibernate

This tutorial explains a setup of NHibernate 4.1 within a Microsoft environment. The tools used in this tutorial are:

- Microsoft Internet Information Services (IIS) - web server supporting ASP.NET.
- Microsoft SQL Server (at least, 2005, but can be 2008, 2008 R2, 2012 or 2014) - the database server. This tutorial uses the desktop edition (SQL Express), a free download from Microsoft. Support for other databases is only a matter of

changing the NHibernate SQL dialect and driver configuration.
- Microsoft Visual Studio .NET (at least, 2005, but can be 2008, 2010, 2012 or 2013, including the Express and Community editions) - the development environment.

First, we have to create a new Web project. We use the name `QuickStart`, the project web virtual directory will `http://localhost/QuickStart`. Download NHibernate, either from SourceForge (http://sourceforge.net/projects/nhibernate/) or NuGet (`Install-Package NHibernate`). In the project, add a reference to `NHibernate.dll` (NuGet already does this). Visual Studio will automatically copy the library and its dependencies to the project output directory. If you are using a database other than SQL Server, add a reference to its driver assembly to your project.

We now set up the database connection information for NHibernate. To do this, open the file `Web.config` automatically generated for your project and add configuration elements according to the listing below:

```
                                <?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <!-- Add this element -->
    <configSections>
        <section
            name="hibernate-configuration"
            type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate"
        />
    </configSections>

    <!-- Add this element -->
    <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
        <session-factory>
            <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
            <property name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>
            <property name="connection.connection_string">Server=localhost\SQLEXPRESS;initial catalog=quickstart;Integrated
            <mapping assembly="QuickStart" />
        </session-factory>
    </hibernate-configuration>

    <!-- Leave the system.web section unchanged -->
    <system.web>
        ...
    </system.web>
</configuration>
```

The `<configSections>` element contains definitions of sections that follow and handlers to use to process their content. We declare the handler for the configuration section here. The `<hibernate-configuration>` section contains the configuration itself, telling NHibernate that we will use a Microsoft SQL Server 2005 database and connect to it through the specified connection string. The dialect is a required setting, databases differ in their interpretation of the SQL "standard". NHibernate will take care of the differences and comes bundled with dialects for several major commercial and open source databases.

An `ISessionFactory` is NHibernate's concept of a single datastore, multiple databases can be used by creating multiple XML configuration files and creating multiple `Configuration` and `ISessionFactory` objects in your application.

The last element of the `<hibernate-configuration>` section declares `QuickStart` as the name of an assembly containing class declarations and mapping files. The mapping files contain the metadata for the mapping of the POCO class to a database table (or multiple tables). We'll come back to mapping files soon. Let's write the POCO class first and then declare the mapping metadata for it.

1.2. First persistent class

NHibernate works best with the Plain Old CLR Objects (POCOs, sometimes called Plain Ordinary CLR Objects) programming model for persistent classes. A POCO has its data accessible through the standard .NET property mechanisms, shielding the internal representation from the publicly visible interface:

```
namespace QuickStart
{
    public class Cat
    {
        private string id;
        private string name;
        private char   sex;
        private float  weight;

        public Cat()
        {
        }

        public virtual string Id
        {
            get { return id; }
            set { id = value; }
        }

        public virtual string Name
        {
            get { return name; }
            set { name = value; }
        }

        public virtual char Sex
        {
            get { return sex; }
            set { sex = value; }
        }

        public virtual float Weight
        {
            get { return weight; }
            set { weight = value; }
        }
    }
}
```

NHibernate is not restricted in its usage of property types, all .NET types and primitives (like `string`, `char` and `DateTime`) can be mapped, including classes from the `System.Collections` namespace. You can map them as values, collections of values, or associations to other entities. The `Id` is a special property that represents the database identifier (primary key) of that class, it is highly recommended for entities like a `Cat`. NHibernate can use identifiers only internally, without having to declare them on the class, but we would lose some of the flexibility in our application architecture.

No special interface has to be implemented for persistent classes nor do we have to subclass from a special root persistent class. NHibernate also doesn't use any build time processing, such as IL manipulation, it relies solely on .NET reflection and runtime class enhancement. So, without any dependency in the POCO class on NHibernate, we can map it to a database table.

For the above mentioned runtime class enhancement to work, NHibernate requires that all public properties of an entity class are declared as `virtual`.

1.3. Mapping the cat

The `Cat.hbm.xml` mapping file contains the metadata required for the object/relational mapping. The metadata includes declaration of persistent classes and the mapping of properties (to columns and foreign key relationships to other entities) to database tables.

Please note that the `Cat.hbm.xml` should be set to an embedded resource.

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    namespace="QuickStart" assembly="QuickStart">

    <class name="Cat" table="Cat">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by NHibernate with the UUID pattern. -->
        <id name="Id">
            <column name="CatId" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex" />
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -->
        <property name="Name">
            <column name="Name" length="16" not-null="true" />
        </property>
        <property name="Sex" />
        <property name="Weight" />
    </class>

</hibernate-mapping>
```

Every persistent class should have an identifer attribute (actually, only classes representing entities, not dependent value objects, which are mapped as components of an entity). This property is used to distinguish persistent objects: Two cats are equal if `catA.Id.Equals(catB.Id)` is true, this concept is called *database identity*. NHibernate comes bundled with various identifer generators for different scenarios (including native generators for database sequences, hi/lo identifier tables, and application assigned identifiers). We use the UUID generator (only recommended for testing, as integer surrogate keys generated by the database should be prefered) and also specify the column `CatId` of the table `Cat` for the NHibernate generated identifier value (as a primary key of the table).

All other properties of Cat are mapped to the same table. In the case of the Name property, we mapped it with an explicit database column declaration. This is especially useful when the database schema is automatically generated (as SQL DDL statements) from the mapping declaration with NHibernate's *SchemaExport* tool. All other properties are mapped using NHibernate's default settings, which is what you need most of the time. The table Cat in the database looks like this:

```
 Column |     Type     | Modifiers
--------+--------------+----------------------
 CatId  | char(32)     | not null, primary key
 Name   | nvarchar(16) | not null
 Sex    | nchar(1)     |
 Weight | real         |
```

You should now create the database and this table manually, and later read Chapter 20, *Toolset Guide* if you want to automate this step with the SchemaExport tool. This tool can create a full SQL DDL, including table definition, custom column type constraints, unique constraints and indexes. If you are using SQL Server, you should also make sure the ASPNET user has permissions to use the database.

1.4. Playing with cats

We're now ready to start NHibernate's ISession. It is the *persistence manager* interface, we use it to store and retrieve Cats to and from the database. But first, we've to get an ISession (NHibernate's unit-of-work) from the ISessionFactory:

```
ISessionFactory sessionFactory =
            new Configuration().Configure().BuildSessionFactory();
```

An ISessionFactory is responsible for one database and may only use one XML configuration file (Web.config or hibernate.cfg.xml). You can set other properties (and even change the mapping metadata) by accessing the Configuration *before* you build the ISessionFactory (it is immutable). Where do we create the ISessionFactory and how can we access it in our application?

An ISessionFactory is usually only built once, e.g. at startup inside Application_Start event handler. This also means you should not keep it in an instance variable in your ASP.NET pages, but in some other location. Furthermore, we need some kind of *Singleton*, so we can access the ISessionFactory easily in application code. The approach shown next solves both problems: configuration and easy access to a ISessionFactory.

We implement a NHibernateHelper helper class:

```
using System;
using System.Web;
using NHibernate;
using NHibernate.Cfg;

namespace QuickStart
{
    public sealed class NHibernateHelper
    {
        private const string CurrentSessionKey = "nhibernate.current_session";
        private static readonly ISessionFactory sessionFactory;

        static NHibernateHelper()
        {
            sessionFactory = new Configuration().Configure().BuildSessionFactory();
        }

        public static ISession GetCurrentSession()
        {
            HttpContext context = HttpContext.Current;
            ISession currentSession = context.Items[CurrentSessionKey] as ISession;

            if (currentSession == null)
            {
                currentSession = sessionFactory.OpenSession();
                context.Items[CurrentSessionKey] = currentSession;
            }

            return currentSession;
        }

        public static void CloseSession()
        {
            HttpContext context = HttpContext.Current;
            ISession currentSession = context.Items[CurrentSessionKey] as ISession;

            if (currentSession == null)
            {
                // No current session
                return;
            }

            currentSession.Close();
            context.Items.Remove(CurrentSessionKey);
        }

        public static void CloseSessionFactory()
        {
            if (sessionFactory != null)
            {
                sessionFactory.Close();
            }
        }
    }
}
```

This class does not only take care of the `ISessionFactory` with its static attribute, but also has code to remember the `ISession` for the current HTTP request.

An `ISessionFactory` is threadsafe, many threads can access it concurrently and request `ISession`s. An `ISession` is a non-threadsafe object that represents a single unit-of-work with the database. `ISession`s are opened by an `ISessionFactory` and are closed when all work is completed:

```
ISession session = NHibernateHelper.GetCurrentSession();

ITransaction tx = session.BeginTransaction();

Cat princess = new Cat();
princess.Name = "Princess";
princess.Sex = 'F';
princess.Weight = 7.4f;

session.Save(princess);
tx.Commit();

NHibernateHelper.CloseSession();
```

In an `ISession`, every database operation occurs inside a transaction that isolates the database operations (even read-only operations). We use NHibernate's `ITransaction` API to abstract from the underlying transaction strategy (in our case, ADO.NET transactions). Please note that the example above does not handle any exceptions.

Also note that you may call `NHibernateHelper.GetCurrentSession();` as many times as you like, you will always get the current `ISession` of this HTTP request. You have to make sure the `ISession` is closed after your unit-of-work completes, either in `Application_EndRequest` event handler in your application class or in a `HttpModule` before the HTTP response is sent. The nice side effect of the latter is easy lazy initialization: the `ISession` is still open when the view is rendered, so NHibernate can load unitialized objects while you navigate the graph.

NHibernate has various methods that can be used to retrieve objects from the database. The most flexible way is using the Hibernate Query Language (HQL), which is an easy to learn and powerful object-oriented extension to SQL:

```
ITransaction tx = session.BeginTransaction();

IQuery query = session.CreateQuery("select c from Cat as c where c.Sex = :sex");
query.SetCharacter("sex", 'F');
foreach (Cat cat in query.List<Cat>())
{
    Console.Out.WriteLine("Female Cat: " + cat.Name);
}

tx.Commit();
```

NHibernate also offers an object-oriented *query by criteria* API that can be used to formulate type-safe queries as well as a strongly-typed LINQ API, which translates internally to HQL. NHibernate of course uses `IDbCommand` s and parameter binding for all SQL communication with the database. You may also use NHibernate's direct SQL query feature or get a plain ADO.NET connection from an `ISession` in rare cases.

1.5. Finally

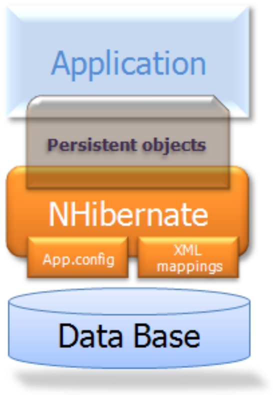
We only scratched the surface of NHibernate in this small tutorial. Please note that we don't include any ASP.NET specific code in our examples. You have to create an ASP.NET page yourself and insert the NHibernate code as you see fit.

Keep in mind that NHibernate, as a data access layer, is tightly integrated into your application. Usually, all other layers depend on the persistence mechanism. Make sure you understand the implications of this design.

Chapter 2. Architecture

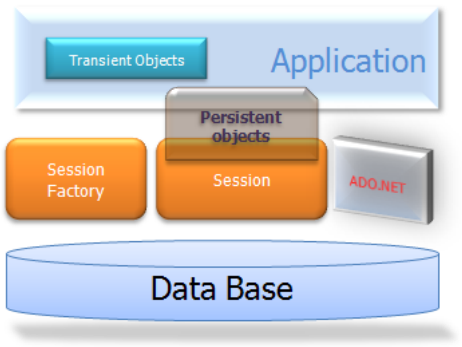
2.1. Overview

A (very) high-level view of the NHibernate architecture:

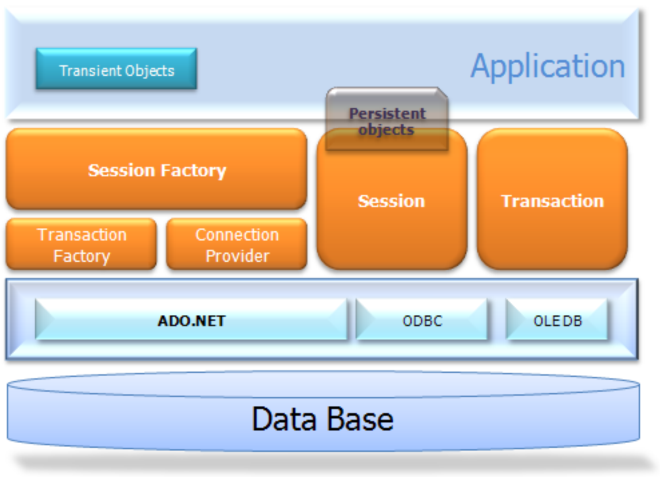


This diagram shows NHibernate using the database and configuration data to provide persistence services (and persistent objects) to the application.

We would like to show a more detailed view of the runtime architecture. Unfortunately, NHibernate is flexible and supports several approaches. We will show the two extremes. The "lite" architecture has the application provide its own ADO.NET connections and manage its own transactions. This approach uses a minimal subset of NHibernate's APIs:



The "full cream" architecture abstracts the application away from the underlying ADO.NET APIs and lets NHibernate take care of the details.



Heres some definitions of the objects in the diagrams:

ISessionFactory (`NHibernate.ISessionFactory`)

A threadsafe (immutable) cache of compiled mappings for a single database. A factory for `ISession` and a client of `IConnectionProvider`. Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level.

ISession (`NHibernate.ISession`)

A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps an ADO.NET connection. Factory for `ITransaction`. Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.

Persistent Objects and Collections

Short-lived, single threaded objects containing persistent state and business function. These might be ordinary POCOs, the only special thing about them is that they are currently associated with (exactly one) `ISession`. As soon as the `Session` is closed, they will be detached and free to use in any application layer (e.g. directly as data transfer objects to and from presentation).

Transient Objects and Collections

Instances of persistent classes that are not currently associated with a `ISession`. They may have been instantiated by the application and not (yet) persisted or they may have been instantiated by a closed `ISession`.

ITransaction (`NHibernate.ITransaction`)

(Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. Abstracts application from underlying ADO.NET transaction. An `ISession` might span several `ITransaction`s in some cases.

IConnectionProvider (`NHibernate.Connection.IConnectionProvider`)

(Optional) A factory for ADO.NET connections and commands. Abstracts application from the concrete vendor-specific implementations of `IDbConnection` and `IDbCommand`. Not exposed to application, but can be extended/implemented by the developer.

IDriver (`NHibernate.Driver.IDriver`)

(Optional) An interface encapsulating differences between ADO.NET providers, such as parameter naming conventions and supported ADO.NET features.

ITransactionFactory (`NHibernate.Transaction.ITransactionFactory`)

(Optional) A factory for `ITransaction` instances. Not exposed to the application, but can be extended/implemented by the developer.

Given a "lite" architecture, the application bypasses the `ITransaction`/`ITransactionFactory` and/or `IConnectionProvider` APIs to talk to ADO.NET directly.

2.2. Instance states

An instance of a persistent classes may be in one of three different states, which are defined with respect to a *persistence context*. The NHibernate `ISession` object is the persistence context:

transient

The instance is not, and has never been associated with any persistence context. It has no persistent identity (primary key value).

persistent

The instance is currently associated with a persistence context. It has a persistent identity (primary key value) and, perhaps, a corresponding row in the database. For a particular persistence context, NHibernate *guarantees* that persistent identity is equivalent to CLR identity (in-memory location of the object).

detached

The instance was once associated with a persistence context, but that context was closed, or the instance was serialized to another process. It has a persistent identity and, perhaps, a corrsponding row in the database. For detached instances, NHibernate makes no guarantees about the relationship between persistent identity and CLR identity.

2.3. Contextual Sessions

Most applications using NHibernate need some form of "contextual" sessions, where a given session is in effect throughout the scope of a given context. However, across applications the definition of what constitutes a context is typically different; and different contexts define different scopes to the notion of current.

Starting with version 1.2, NHibernate added the `ISessionFactory.GetCurrentSession()` method. The processing behind `ISessionFactory.GetCurrentSession()` is pluggable. An extension interface (`NHibernate.Context.ICurrentSessionContext`) and a new configuration parameter (`hibernate.current_session_context_class`) have been added to allow pluggability of the scope and context of defining current sessions.

See the API documentation for the `NHibernate.Context.ICurrentSessionContext` interface for a detailed discussion of its contract. It defines a single method, `CurrentSession()`, by which the implementation is responsible for tracking the current contextual session. Out-of-the-box, NHibernate comes with several implementations of this interface:

- `NHibernate.Context.CallSessionContext` - current sessions are tracked by `CallContext`. You are responsible to bind and unbind an `ISession` instance with static methods of class `CurrentSessionContext` .
- `NHibernate.Context.ThreadStaticSessionContext` - current session is stored in a thread-static variable. This context only supports one session factory. You are responsible to bind and unbind an `ISession` instance with static methods of class `CurrentSessionContext`.
- `NHibernate.Context.WebSessionContext` - stores the current session in `HttpContext`. You are responsible to bind and unbind an `ISession` instance with static methods of class `CurrentSessionContext`.
- `NHibernate.Context.WcfOperationSessionContext` - current sessions are tracked by WCF `OperationContext`. You need to register the `WcfStateExtension` extension in WCF. You are responsible to bind and unbind an `ISession` instance with static methods of class `CurrentSessionContext`.
- `NHibernate.Context.ManagedWebSessionContext` - current sessions are tracked by `HttpContext`. Removed in NHibernate 4.0 - `NHibernate.Context.WebSessionContext` should be used instead. You are responsible to bind and unbind an `ISession` instance with static methods on this class, it never opens, flushes, or closes an `ISession` itself.

The `hibernate.current_session_context_class` configuration parameter defines which `NHibernate.Context.ICurrentSessionContext` implementation should be used. Typically, the value of this parameter would just name the implementation class to use (including the assembly name); for the out-of-the-box implementations, however, there are corresponding short names: "call", "thread_static", "web" and "wcf_operation", respectively.

Chapter 3. ISessionFactory Configuration

Because NHibernate is designed to operate in many different environments, there are a large number of configuration parameters. Fortunately, most have sensible default values and NHibernate is distributed with an example `App.config` file (found in `src\NHibernate.Test`) that shows the various options. You usually only have to put that file in your project and customize it.

3.1. Programmatic Configuration

An instance of `NHibernate.Cfg.Configuration` represents an entire set of mappings of an application's .NET types to a SQL database. The `Configuration` is used to build an (immutable) `ISessionFactory`. The mappings are compiled from various XML mapping files.

You may obtain a `Configuration` instance by instantiating it directly. Heres an example of setting up a datastore from mappings defined in two XML configuration files:

```
Configuration cfg = new Configuration()
    .AddFile("Item.hbm.xml")
    .AddFile("Bid.hbm.xml");
```

An alternative (sometimes better) way is to let NHibernate load a mapping file from an embedded resource:

```
Configuration cfg = new Configuration()
    .AddClass(typeof(NHibernate.Auction.Item))
    .AddClass(typeof(NHibernate.Auction.Bid));
```

Then NHibernate will look for mapping files named `NHibernate.Auction.Item.hbm.xml` and

NHibernate.Auction.Bid.hbm.xml embedded as resources in the assembly that the types are contained in. This approach eliminates any hardcoded filenames.

Another alternative (probably the best) way is to let NHibernate load all of the mapping files contained in an Assembly:

```
Configuration cfg = new Configuration()
    .AddAssembly( "NHibernate.Auction" );
```

Then NHibernate will look through the assembly for any resources that end with .hbm.xml. This approach eliminates any hardcoded filenames and ensures the mapping files in the assembly get added.

If a tool like Visual Studio .NET or NAnt is used to build the assembly, then make sure that the .hbm.xml files are compiled into the assembly as Embedded Resources.

A Configuration also specifies various optional properties:

```
IDictionary props = new Hashtable();
...
Configuration cfg = new Configuration()
    .AddClass(typeof(NHibernate.Auction.Item))
    .AddClass(typeof(NHibernate.Auction.Bind))
    .SetProperties(props);
```

A Configuration is intended as a configuration-time object, to be discarded once an ISessionFactory is built.

3.2. Obtaining an ISessionFactory

When all mappings have been parsed by the Configuration, the application must obtain a factory for ISession instances. This factory is intended to be shared by all application threads:

```
ISessionFactory sessions = cfg.BuildSessionFactory();
```

However, NHibernate does allow your application to instantiate more than one ISessionFactory. This is useful if you are using more than one database.

3.3. User provided ADO.NET connection

An ISessionFactory may open an ISession on a user-provided ADO.NET connection. This design choice frees the application to obtain ADO.NET connections wherever it pleases:

```
IDbConnection conn = myApp.GetOpenConnection();
ISession session = sessions.OpenSession(conn);

// do some data access work
```

The application must be careful not to open two concurrent ISessions on the same ADO.NET connection!

3.4. NHibernate provided ADO.NET connection

Alternatively, you can have the ISessionFactory open connections for you. The ISessionFactory must be provided with ADO.NET connection properties in one of the following ways:

- Pass an instance of IDictionary mapping property names to property values to Configuration.SetProperties().
- Add the properties to a configuration section in the application configuration file. The section should be named nhibernate and its handler set to System.Configuration.NameValueSectionHandler.
- Include <property> elements in a configuration section in the application configuration file. The section should be named hibernate-configuration and its handler set to NHibernate.Cfg.ConfigurationSectionHandler. The XML namespace of the section should be set to urn:nhibernate-configuration-2.2.
- Include <property> elements in hibernate.cfg.xml (discussed later).

If you take this approach, opening an ISession is as simple as:

```
ISession session = sessions.OpenSession(); // open a new Session
// do some data access work, an ADO.NET connection will be used on demand
```

All NHibernate property names and semantics are defined on the class `NHibernate.Cfg.Environment`. We will now describe the most important settings for ADO.NET connection configuration.

NHibernate will obtain (and pool) connections using an ADO.NET data provider if you set the following properties:

Table 3.1. NHibernate ADO.NET Properties

Property name	Purpose
`connection.provider_class`	The type of a custom `IConnectionProvider` implementation. eg. `full.classname.of.ConnectionProvider` if the Provider is built into NHibernate, or `full.classname.of.ConnectionProvider, assembly` if using an implementation of `IConnectionProvider` not included in NHibernate. The default is `NHibernate.Connection.DriverConnectionProvider`.
`connection.driver_class`	The type of a custom `IDriver`, if using `DriverConnectionProvider`. `full.classname.of.Driver` if the Driver is built into NHibernate, or `full.classname.of.Driver, assembly` if using an implementation of IDriver not included in NHibernate. This is usually not needed, most of the time the `dialect` will take care of setting the `IDriver` using a sensible default. See the API documentation of the specific dialect for the defaults.
`connection.connection_string`	Connection string to use to obtain the connection.
`connection.connection_string_name`	The name of the connection string (defined in `<connectionStrings>` configuration file element) to use to obtain the connection.
`connection.isolation`	Set the ADO.NET transaction isolation level. Check `System.Data.IsolationLevel` for meaningful values and the database's documentation to ensure that level is supported. eg. `Chaos, ReadCommitted, ReadUncommitted, RepeatableRead, Serializable, Unspecified`
`connection.release_mode`	Specify when NHibernate should release ADO.NET connections. See Section 11.7, "Connection Release Modes". eg. `auto` (default) | `on_close` | `after_transaction` Note that this setting only affects `ISession`s returned from `ISessionFactory.OpenSession`. For `ISession`s obtained through `ISessionFactory.GetCurrentSession`, the `ICurrentSessionContext` implementation configured for use controls the connection release mode for those `ISession`s. See Section 2.3, "Contextual Sessions".
`command_timeout`	Specify the default timeout of `IDbCommands` generated by NHibernate.
`adonet.batch_size`	Specify the batch size to use when batching update statements. Setting this to 0 (the default) disables the functionality. See Section 19.6, "Batch updates".

This is an example of how to specify the database connection properties inside a `web.config`:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
        <configSections>
                <section name="hibernate-configuration"
                                 type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
        </configSections>

        <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
                <session-factory>
                        <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
                        <property name="connection.connection_string">
                                Server=(local);initial catalog=theDb;Integrated Security=SSPI
                        </property>
                        <property name="connection.isolation">ReadCommitted</property>
                </session-factory>
        </hibernate-configuration>

    <!-- other app specific config follows -->
</configuration>
```

NHibernate relies on the ADO.NET data provider implementation of connection pooling.

You may define your own plugin strategy for obtaining ADO.NET connections by implementing the interface `NHibernate.Connection.IConnectionProvider`. You may select a custom implementation by setting `connection.provider_class`.

3.5. Optional configuration properties

There are a number of other properties that control the behaviour of NHibernate at runtime. All are optional and have reasonable default values.

System-level properties can only be set manually by setting static properties of `NHibernate.Cfg.Environment` class or be defined in the `<nhibernate>` section of the application configuration file. These properties cannot be set using `Configuration.SetProperties` or be defined in the `<hibernate-configuration>` section of the application configuration file.

Table 3.2. NHibernate Configuration Properties

Property name	Purpose
`dialect`	The classname of a NHibernate `Dialect` - enables certain platform dependent features. eg. `full.classname.of.Dialect, assembly`
`default_schema`	Qualify unqualified tablenames with the given schema/tablespace in generated SQL. eg. `SCHEMA_NAME`
`max_fetch_depth`	Set a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A `0` disables default outer join fetching. eg. recommended values between `0` and `3`
`use_reflection_optimizer`	Enables use of a runtime-generated class to set or get properties of an entity or component instead of using runtime reflection (System-level property). The use of the reflection optimizer inflicts a certain startup cost on the application but should lead to better performance in the long run. You can not set this property in `hibernate.cfg.xml` or `<hibernate-configuration>` section of the application configuration file. eg. `true` | `false`
`bytecode.provider`	Specifies the bytecode provider to use to optimize the use of reflection in NHibernate. Use `null` to disable the optimization completely, `lcg` to use lightweight code generation. eg. `null` | `lcg`
`cache.provider_class`	The classname of a custom `ICacheProvider`. eg. `classname.of.CacheProvider, assembly`
`cache.use_minimal_puts`	Optimize second-level cache operation to minimize writes, at the cost of more frequent reads (useful for clustered caches). eg. `true` | `false`
`cache.use_query_cache`	Enable the query cache, individual queries still have to be set cacheable. eg. `true` | `false`
`cache.query_cache_factory`	The classname of a custom `IQueryCacheFactory` interface, defaults to the built-in `StandardQueryCacheFactory`. eg. `classname.of.QueryCacheFactory, assembly`
`cache.region_prefix`	A prefix to use for second-level cache region names. eg. `prefix`
`query.substitutions`	Mapping from tokens in NHibernate queries to SQL tokens (tokens might be function or literal names, for example). eg. `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`query.linq_provider_class`	The classname of a custom LINQ provider class, one that implements `INhQueryProvider`. The default is `DefaultQueryProvider` eg. `true` | `false`
`show_sql`	Write all SQL statements to console. eg. `true` | `false`
`hbm2ddl.auto`	Automatically export schema DDL to the database when the `ISessionFactory` is created. With `create-drop`, the database schema will be dropped when the `ISessionFactory` is closed explicitly. eg. `create` | `create-drop`
`hbm2ddl.keywords`	Automatically import `reserved/keywords` from the database when the `ISessionFactory` is created. none : disable any operation regarding RDBMS KeyWords keywords : imports all RDBMS KeyWords where the `Dialect` can provide the implementation of `IDataBaseSchema`. auto-quote : imports all RDBMS KeyWords and auto-quote all table-names/column-names . eg. `none` | `keywords` | `auto-quote`

`use_proxy_validator`	Enables or disables validation of interfaces or classes specified as proxies. Enabled by default. eg. `true` | `false`
`transaction.factory_class`	The classname of a custom `ITransactionFactory` implementation, defaults to the built-in `AdoNetWithDistributedTransactionFactory`. eg. `classname.of.TransactionFactory, assembly`
`default_flush_mode`	The default `FlushMode`, defaults to `Unspecified` eg. `Unspecified` | `Never` | `Commit` | `Auto` | `Always`

3.5.1. SQL Dialects

You should always set the `dialect` property to the correct `NHibernate.Dialect.Dialect` subclass for your database. This is not strictly essential unless you wish to use `native` or `sequence` primary key generation or pessimistic locking (with, eg. `ISession.Lock()` or `IQuery.SetLockMode()`). However, if you specify a dialect, NHibernate will use sensible defaults for some of the other properties listed above, saving you the effort of specifying them manually.

Table 3.3. NHibernate SQL Dialects (`dialect`)

RDBMS	Dialect	Remarks
DB2	`NHibernate.Dialect.DB2Dialect`	
DB2 for iSeries (OS/400)	`NHibernate.Dialect.DB2400Dialect`	
Ingres	`NHibernate.Dialect.IngresDialect`	
PostgreSQL	`NHibernate.Dialect.PostgreSQLDialect`	
PostgreSQL 8.1	`NHibernate.Dialect.PostgreSQL81Dialect`	This dialect supports `FOR UPDATE NOWAIT` available in PostgreSQL 8.1.
PostgreSQL 8.2	`NHibernate.Dialect.PostgreSQL82Dialect`	This dialect supports `IF EXISTS` keyword in `DROP TABLE` and `DROP SEQUENCE` available in PostgreSQL 8.2.
MySQL 3 or 4	`NHibernate.Dialect.MySQLDialect`	
MySQL 5	`NHibernate.Dialect.MySQL5Dialect`	
Oracle	`NHibernate.Dialect.Oracle8iDialect`	
Oracle 9i	`NHibernate.Dialect.Oracle9iDialect`	
Oracle 10g, Oracle 11g	`NHibernate.Dialect.Oracle10gDialect`	
Oracle 12c	`NHibernate.Dialect.Oracle12cDialect`	
Sybase Adaptive Server Enterprise 15	`NHibernate.Dialect.SybaseASE15Dialect`	
Sybase Adaptive Server Anywhere 9	`NHibernate.Dialect.SybaseASA9Dialect`	
Sybase SQL Anywhere 10	`NHibernate.Dialect.SybaseSQLAnywhere10Dialect`	
Sybase SQL Anywhere 11	`NHibernate.Dialect.SybaseSQLAnywhere11Dialect`	
Microsoft SQL Server 7	`NHibernate.Dialect.MsSql7Dialect`	
Microsoft SQL Server 2000	`NHibernate.Dialect.MsSql2000Dialect`	
Microsoft SQL Server 2005	`NHibernate.Dialect.MsSql2005Dialect`	
Microsoft SQL Server 2008	`NHibernate.Dialect.MsSql2008Dialect`	
Microsoft SQL Server 2012	`NHibernate.Dialect.MsSql2012Dialect`	
Microsoft SQL Server Compact Edition	`NHibernate.Dialect.MsSqlCeDialect`	
Microsoft SQL Server Compact Edition 4	`NHibernate.Dialect.MsSqlCe40Dialect`	
Firebird	`NHibernate.Dialect.FirebirdDialect`	Set `driver_class` to `NHibernate.Driver.FirebirdClientDriver` for Firebird ADO.NET provider 2.0.
SQLite	`NHibernate.Dialect.SQLiteDialect`	Set `driver_class` to `NHibernate.Driver.SQLite20Driver` for System.Data.SQLite provider for .NET 2.0.

Additional dialects may be available in the NHibernate.Dialect namespace.

3.5.2. Outer Join Fetching

If your database supports ANSI or Oracle style outer joins, *outer join fetching* might increase performance by limiting the number of round trips to and from the database (at the cost of possibly more work performed by the database itself). Outer join fetching allows a graph of objects connected by many-to-one, one-to-many or one-to-one associations to be retrieved in a single SQL `SELECT`.

By default, the fetched graph when loading an objects ends at leaf objects, collections, objects with proxies, or where circularities occur.

For a *particular association*, fetching may be configured (and the default behaviour overridden) by setting the `fetch` attribute in the XML mapping.

Outer join fetching may be disabled *globally* by setting the property `max_fetch_depth` to `0`. A setting of `1` or higher enables outer join fetching for one-to-one and many-to-one associations which have been mapped with `fetch="join"`.

See Section 19.1, "Fetching strategies" for more information.

In NHibernate 1.0, `outer-join` attribute could be used to achieve a similar effect. This attribute is now deprecated in favor of `fetch`.

3.5.3. Custom `ICacheProvider`

You may integrate a process-level (or clustered) second-level cache system by implementing the interface `NHibernate.Cache.ICacheProvider`. You may select the custom implementation by setting `cache.provider_class`. See the Section 19.2, "The Second Level Cache" for more details.

3.5.4. Query Language Substitution

You may define new NHibernate query tokens using `query.substitutions`. For example:

```
query.substitutions true=1, false=0
```

would cause the tokens `true` and `false` to be translated to integer literals in the generated SQL.

```
query.substitutions toLowercase=LOWER
```

would allow you to rename the SQL `LOWER` function.

3.6. Logging

NHibernate logs various events using Apache log4net.

You may download log4net from `http://logging.apache.org/log4net/`. To use log4net you will need a `log4net` configuration section in the application configuration file. An example of the configuration section is distributed with NHibernate in the `src/NHibernate.Test` project.

We strongly recommend that you familiarize yourself with NHibernate's log messages. A lot of work has been put into making the NHibernate log as detailed as possible, without making it unreadable. It is an essential troubleshooting device. Also don't forget to enable SQL logging as described above (`show_sql`), it is your first step when looking for performance problems.

3.7. Implementing an `INamingStrategy`

The interface `NHibernate.Cfg.INamingStrategy` allows you to specify a "naming standard" for database objects and schema elements.

You may provide rules for automatically generating database identifiers from .NET identifiers or for processing "logical" column and table names given in the mapping file into "physical" table and column names. This feature helps reduce the verbosity of the mapping document, eliminating repetitive noise (`TBL_` prefixes, for example). The default strategy used by NHibernate is quite minimal.

You may specify a different strategy by calling `Configuration.SetNamingStrategy()` before adding mappings:

```
ISessionFactory sf = new Configuration()
```

```
    .SetNamingStrategy(ImprovedNamingStrategy.Instance)
    .AddFile("Item.hbm.xml")
    .AddFile("Bid.hbm.xml")
    .BuildSessionFactory();
```

`NHibernate.Cfg.ImprovedNamingStrategy` is a built-in strategy that might be a useful starting point for some applications.

3.8. XML Configuration File

An alternative approach is to specify a full configuration in a file named `hibernate.cfg.xml`. This file can be used as a replacement for the `<nhibernate;>` or `<hibernate-configuration>` sections of the application configuration file.

The XML configuration file is by default expected to be in your application directory. Here is an example:

```
<?xml version='1.0' encoding='utf-8'?>
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">

    <!-- an ISessionFactory instance -->
    <session-factory>

        <!-- properties -->
        <property name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>
        <property name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
        <property name="connection.connection_string">Server=localhost;initial catalog=nhibernate;User Id=;Password=</proper
        <property name="show_sql">false</property>
        <property name="dialect">NHibernate.Dialect.MsSql2000Dialect</property>

        <!-- mapping files -->
        <mapping resource="NHibernate.Auction.Item.hbm.xml" assembly="NHibernate.Auction" />
        <mapping resource="NHibernate.Auction.Bid.hbm.xml" assembly="NHibernate.Auction" />

    </session-factory>

</hibernate-configuration>
```

Configuring NHibernate is then as simple as

```
ISessionFactory sf = new Configuration().Configure().BuildSessionFactory();
```

You can pick a different XML configuration file using

```
ISessionFactory sf = new Configuration()
    .Configure("/path/to/config.cfg.xml")
    .BuildSessionFactory();
```

Chapter 4. Persistent Classes

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). Persistent classes have, as the name implies, transient and also persistent instance stored in the database.

NHibernate works best if these classes follow some simple rules, also known as the Plain Old CLR Object (POCO) programming model.

4.1. A simple POCO example

Most .NET applications require a persistent class representing felines.

```
using System;
using System.Collections.Generic;

namespace Eg
{
    public class Cat
    {
        long id; // identifier

        public virtual long Id
        {
            get { return id; }
            protected set { id = value; }
        }
        public virtual string Name { get; set; }
        public virtual Cat Mate { get; set; }
        public virtual DateTime Birthdate { get; set; }
        public virtual float Weight { get; set; }
        public virtual Color Color { get; set; }
        public virtual ISet<Cat> Kittens { get; set; }
        public virtual char Sex { get; set; }

        // AddKitten not needed by NHibernate
        public virtual void AddKitten(Cat kitten)
        {
            kittens.Add(kitten);
        }
    }
}
```

There are four main rules to follow here:

4.1.1. Declare properties for persistent fields

`Cat` declares properties for all the persistent fields. Many other ORM tools directly persist instance variables. We believe it is far better to decouple this implementation detail from the persistence mechanism. NHibernate persists properties, using their getter and setter methods.

Properties need *not* be declared public - NHibernate can persist a property with an `internal`, `protected`, `protected internal` or `private` visibility.

As shown in the example, both automatic properties and properties with a backing field are supported.

4.1.2. Implement a default constructor

`Cat` has an implicit default (no-argument) constructor. All persistent classes must have a default constructor (which may be non-public) so NHibernate can instantiate them using `Activator.CreateInstance()`.

4.1.3. Provide an identifier property (optional)

`Cat` has a property called `Id`. This property holds the primary key column of a database table. The property might have been called anything, and its type might have been any primitive type, `string` or `System.DateTime`. (If your legacy database table has composite keys, you can even use a user-defined class with properties of these types - see the section on composite identifiers below.)

The identifier property is optional. You can leave it off and let NHibernate keep track of object identifiers internally. However, for many applications it is still a good (and very popular) design decision.

What's more, some functionality is available only to classes which declare an identifier property:

- Cascaded updates (see "Lifecycle Objects")
- `ISession.SaveOrUpdate()`

We recommend you declare consistently-named identifier properties on persistent classes.

4.1.4. Prefer non-sealed classes and virtual methods (optional)

A central feature of NHibernate, *proxies*, depends upon the persistent class being non-sealed and all its public methods, properties and events declared as virtual. Another possibility is for the class to implement an interface that declares all public members.

You can persist `sealed` classes that do not implement an interface and don't have virtual members with NHibernate, but you won't be able to use proxies - which will limit your options for performance tuning.

4.2. Implementing inheritance

A subclass must also observe the first and second rules. It inherits its identifier property from `Cat`.

```
using System;
namespace Eg
{
    public class DomesticCat : Cat
    {
        public virtual string Name { get; set; }
    }
}
```

4.3. Implementing `Equals()` and `GetHashCode()`

You have to override the `Equals()` and `GetHashCode()` methods if you intend to mix objects of persistent classes (e.g. in an `ISet<T>`).

This only applies if these objects are loaded in two different `ISession`s, as NHibernate only guarantees identity (`a == b` , the default implementation of `Equals()`) inside a single `ISession`!

Even if both objects `a` and `b` are the same database row (they have the same primary key value as their identifier), we can't guarantee that they are the same object instance outside of a particular `ISession` context.

The most obvious way is to implement `Equals()`/`GetHashCode()` by comparing the identifier value of both objects. If the value is the same, both must be the same database row, they are therefore equal (if both are added to an `ISet<T>`, we will only have one element in the `ISet<T>`). Unfortunately, we can't use that approach. NHibernate will only assign identifier values to objects that are persistent, a newly created instance will not have any identifier value! We recommend implementing `Equals()` and `GetHashCode()` using *Business key equality*.

Business key equality means that the `Equals()` method compares only the properties that form the business key, a key that would identify our instance in the real world (a *natural* candidate key):

```
public class Cat
{

    ...
    public override bool Equals(object other)
    {
        if (this == other) return true;

        Cat cat = other as Cat;
        if (cat == null) return false; // null or not a cat

        if (Name != cat.Name) return false;
        if (!Birthday.Equals(cat.Birthday)) return false;

        return true;
    }

    public override int GetHashCode()
    {
        unchecked
        {
            int result;
            result = Name.GetHashCode();
            result = 29 * result + Birthday.GetHashCode();
            return result;
        }
    }

}
```

Keep in mind that our candidate key (in this case a composite of name and birthday) has to be only valid for a particular comparison operation (maybe even only in a single use case). We don't need the stability criteria we usually apply to a real primary key!

4.4. Dynamic models

Note that the following features are currently considered experimental and may change in the near future.

Persistent entities don't necessarily have to be represented as POCO classes at runtime. NHibernate also supports dynamic models (using `Dictionaries` of `Dictionary`s at runtime) . With this approach, you don't write persistent classes, only mapping files.

By default, NHibernate works in normal POCO mode. You may set a default entity representation mode for a particular `ISessionFactory` using the `default_entity_mode` configuration option (see Table 3.2, “NHibernate Configuration Properties”.

The following examples demonstrates the representation using `Maps` (Dictionary). First, in the mapping file, an `entity-name` has to be declared instead of (or in addition to) a class name:

```
<hibernate-mapping>
```

```
    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

Note that even though associations are declared using target class names, the target type of an associations may also be a dynamic entity instead of a POCO.

After setting the default entity mode to `dynamic-map` for the `ISessionFactory`, we can at runtime work with `Dictionaries` of `Dictionaries`:

```
using(ISession s = OpenSession())
using(ITransaction tx = s.BeginTransaction())
{
    // Create a customer
    var frank = new Dictionary<string, object>();
    frank["name"] = "Frank";

    // Create an organization
    var foobar = new Dictionary<string, object>();
    foobar["name"] = "Foobar Inc.";

    // Link both
    frank["organization"] =  foobar;

    // Save both
    s.Save("Customer", frank);
    s.Save("Organization", foobar);

    tx.Commit();
}
```

The advantages of a dynamic mapping are quick turnaround time for prototyping without the need for entity class implementation. However, you lose compile-time type checking and will very likely deal with many exceptions at runtime. Thanks to the NHibernate mapping, the database schema can easily be normalized and sound, allowing to add a proper domain model implementation on top later on.

Entity representation modes can also be set on a per `ISession` basis:

```
using (ISession dynamicSession = pocoSession.GetSession(EntityMode.Map))
{
    // Create a customer
    var frank = new Dictionary<string, object>();
    frank["name"] = "Frank";
    dynamicSession.Save("Customer", frank);
    ...
}
// Continue on pocoSession
```

Please note that the call to `GetSession()` using an `EntityMode` is on the `ISession` API, not the `ISessionFactory`. That way, the new `ISession` shares the underlying ADO connection, transaction, and other context information. This means you don't have tocall `Flush()` and `Close()` on the secondary `ISession`, and also leave the transaction and connection handling to the primary unit of work.

4.5. Tuplizers

`NHibernate.Tuple.Tuplizer`, and its sub-interfaces, are responsible for managing a particular representation of a piece of data, given that representation's `NHibernate.EntityMode`. If a given piece of data is thought of as a data structure, then a tuplizer is the thing which knows how to create such a data structure and how to extract values from and inject values into such a data structure. For example, for the POCO entity mode, the correpsonding tuplizer knows how create the POCO through its constructor and how to access the POCO properties using the defined property accessors. There are two high-level types of Tuplizers, represented by the `NHibernate.Tuple.Entity.IEntityTuplizer` and `NHibernate.Tuple.Component.IComponentTuplizer` interfaces. `IEntityTuplizer`s are responsible for managing the above mentioned contracts in regards to entities, while `IComponentTuplizer`s do the same for components.

Users may also plug in their own tuplizers. Perhaps you require that a `System.Collections.IDictionary` implementation other than `System.Collections.Hashtable` be used while in the dynamic-map entity-mode; or perhaps you need to define a different proxy generation strategy than the one used by default. Both would be achieved by defining a custom tuplizer implementation. Tuplizers definitions are attached to the entity or component mapping they are meant to manage. Going back to the example of our customer entity:

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>


public class CustomMapTuplizerImpl : NHibernate.Tuple.Entity.DynamicMapEntityTuplizer
{
    // override the BuildInstantiator() method to plug in our custom map...
    protected override IInstantiator BuildInstantiator(NHibernate.Mapping.PersistentClass mappingInfo)
    {
        return new CustomMapInstantiator(mappingInfo);
    }

    private sealed class CustomMapInstantiator : NHibernate.Tuple.DynamicMapInstantiator
    {
        // override the generateMap() method to return our custom map...
        protected override IDictionary GenerateMap()
        {
            return new CustomMap();
        }
    }
}
```

4.6. Lifecycle Callbacks

Optionally, a persistent class might implement the interface `ILifecycle` which provides some callbacks that allow the persistent object to perform necessary initialization/cleanup after save or load and before deletion or update.

The NHibernate `IInterceptor` offers a less intrusive alternative, however.

```
public interface ILifecycle
{                                                                   (1)
        LifecycleVeto OnSave(ISession s);                           (2)
        LifecycleVeto OnUpdate(ISession s);                         (3)
        LifecycleVeto OnDelete(ISession s);                         (4)
        void OnLoad(ISession s, object id);
}
```

(1) `OnSave` - called just before the object is saved or inserted

(2) `OnUpdate` - called just before an object is updated (when the object is passed to `ISession.Update()`)

(3) `OnDelete` - called just before an object is deleted

(4) `OnLoad` - called just after an object is loaded

`OnSave()`, `OnDelete()` and `OnUpdate()` may be used to cascade saves and deletions of dependent objects. This is an alternative to declaring cascaded operations in the mapping file. `OnLoad()` may be used to initialize transient properties of the object from its persistent state. It may not be used to load dependent objects since the `ISession` interface may not be invoked from inside this method. A further intended usage of `OnLoad()`, `OnSave()` and `OnUpdate()` is to store a reference to the current `ISession` for later use.

Note that `OnUpdate()` is not called every time the object's persistent state is updated. It is called only when a transient object is passed to `ISession.Update()`.

If `OnSave()`, `OnUpdate()` or `OnDelete()` return `LifecycleVeto.Veto`, the operation is silently vetoed. If a `CallbackException` is thrown, the operation is vetoed and the exception is passed back to the application.

Note that `OnSave()` is called after an identifier is assigned to the object, except when native key generation is used.

4.7. IValidatable callback

If the persistent class needs to check invariants before its state is persisted, it may implement the following interface:

```
public interface IValidatable
{
        void Validate();
}
```

The object should throw a `ValidationFailure` if an invariant was violated. An instance of `Validatable` should not change its state from inside `Validate()`.

Unlike the callback methods of the `ILifecycle` interface, `Validate()` might be called at unpredictable times. The application should not rely upon calls to `Validate()` for business functionality.

Chapter 5. Basic O/R Mapping

5.1. Mapping declaration

Object/relational mappings are defined in an XML document. The mapping document is designed to be readable and hand-editable. The mapping language is object-centric, meaning that mappings are constructed around persistent class declarations, not table declarations.

Note that, even though many NHibernate users choose to define XML mappings by hand, a number of tools exist to generate the mapping document, including NHibernate.Mapping.Attributes library and various template-based code generators (CodeSmith, MyGeneration).

Let's kick off with an example mapping:

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Eg"
    namespace="Eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="Id" column="uid" type="Int64">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="Char"/>
                <property name="BirthDate" type="Date"/>
                <property name="Color" not-null="true"/>
                <property name="Sex" not-null="true" update="false"/>
                <property name="Weight"/>
                <many-to-one name="Mate" column="mate_id"/>
                <set name="Kittens">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat" discriminator-value="D">
                        <property name="Name" type="String"/>
                </subclass>
        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

We will now discuss the content of the mapping document. We will only describe the document elements and attributes that are used by NHibernate at runtime. The mapping document also contains some extra optional attributes and elements that affect the database schemas exported by the schema export tool. (For example the `not-null` attribute.)

5.1.1. XML Namespace

All XML mappings should declare the XML namespace shown. The actual schema definition may be found in the `src\nhibernate-mapping.xsd` file in the NHibernate distribution.

Tip: to enable IntelliSense for mapping and configuration files, copy the appropriate `.xsd` files as part of any project in your solution, (`Build Action` can be "None") or as "Solution Files" or in your `"Lib"` folder and then add it to the `Schemas` property of the xml file. You can copy it in `<VS installation directory>\Xml\Schemas`, take care because you will have to deal with different version of the xsd for different versions of NHibernate.

5.1.2. hibernate-mapping

This element has several optional attributes. The `schema` attribute specifies that tables referred to by this mapping belong to the named schema. If specified, tablenames will be qualified by the given schema name. If missing, tablenames will be unqualified. The `default-cascade` attribute specifies what cascade style should be assumed for properties and collections which do not specify a `cascade` attribute. The `auto-import` attribute lets us use unqualified class names in the query language, by default. The `assembly` and `namespace` attributes specify the assembly where persistent classes are located and the namespace they are declared in.

```
          <hibernate-mapping                        (1)
        schema="schemaName"                         (2)
        default-cascade="none|save-update"          (3)
        auto-import="true|false"                    (4)
        assembly="Eg"                               (5)
        namespace="Eg"                              (6)
        default-access="field|property|field.camecase(7)..."
        default-lazy="true|false"
 />
```

(1) `schema` (optional): The name of a database schema.

(2) `default-cascade` (optional - defaults to `none`): A default cascade style.

(3) `auto-import` (optional - defaults to `true`): Specifies whether we can use unqualified class names (of classes in this mapping) in the query language.

(4)
(5) `assembly` and `namespace`(optional): Specify assembly and namespace to assume for unqualified class names in the mapping document.

(6) `default-access` (optional - defaults to property): The strategy NHibernate should use for accessing a property value

(7) `default-lazy` (optional - defaults to `true`): Lazy fetching may be completely disabled by setting default-lazy="false".

If you are not using `assembly` and `namespace` attributes, you have to specify fully-qualified class names, including the name of the assembly that classes are declared in.

If you have two persistent classes with the same (unqualified) name, you should set `auto-import="false"`. NHibernate will throw an exception if you attempt to assign two classes to the same "imported" name.

5.1.3. class

You may declare a persistent class using the `class` element:

```
<class
        name="ClassName"                            (1)
        table="tableName"                           (2)
        discriminator-value="discriminator_value"   (3)
        mutable="true|false"                        (4)
        schema="owner"                              (5)
        proxy="ProxyInterface"                      (6)
        dynamic-update="true|false"                 (7)
        dynamic-insert="true|false"                 (8)
        select-before-update="true|false"           (9)
        polymorphism="implicit|explicit"            (10)
        where="arbitrary sql where condition"       (11)
        persister="PersisterClass"                  (12)
        batch-size="N"                              (13)
        optimistic-lock="none|version|dirty|all"    (14)
        lazy="true|false"                           (15)
        abstract="true|false"                       (16)
/>
```

(1) `name`: The fully qualified .NET class name of the persistent class (or interface), including its assembly name.

(2) `table`(optional - defaults to the unqualified class name): The name of its database table.

(3) `discriminator-value` (optional - defaults to the class name): A value that distiguishes individual subclasses, used for polymorphic behaviour. Acceptable values include `null` and `not null`.

(4) `mutable` (optional, defaults to `true`): Specifies that instances of the class are (not) mutable.

(5) `schema` (optional): Override the schema name specified by the root `<hibernate-mapping>` element.

(6) `proxy` (optional): Specifies an interface to use for lazy initializing proxies. You may specify the name of the class itself.

(7) `dynamic-update` (optional, defaults to `false`): Specifies that `UPDATE` SQL should be generated at runtime and contain only those columns whose values have changed.

(8) `dynamic-insert` (optional, defaults to `false`): Specifies that `INSERT` SQL should be generated at runtime and contain only the columns whose values are not null.

(9) `select-before-update` (optional, defaults to `false`): Specifies that NHibernate should *never* perform an SQL `UPDATE` unless it is certain that an object is actually modified. In certain cases (actually, only when a transient object has been associated with a new session using `update()`), this means that NHibernate will perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required.

(10) `polymorphism` (optional, defaults to `implicit`): Determines whether implicit or explicit query polymorphism is used.

(11) `where` (optional) specify an arbitrary SQL `WHERE` condition to be used when retrieving objects of this class

(12) `persister` (optional): Specifies a custom `IClassPersister`.

(13) `batch-size` (optional, defaults to `1`) specify a "batch size" for fetching instances of this class by identifier.

(14) `optimistic-lock` (optional, defaults to `version`): Determines the optimistic locking strategy.

(15) `lazy` (optional): Lazy fetching may be completely disabled by setting `lazy="false"`.

(16) `abstract` (optional): Used to mark abstract superclasses in `<union-subclass>` hierarchies.

It is perfectly acceptable for the named persistent class to be an interface. You would then declare implementing classes of that interface using the `<subclass>` element. You may persist any inner class. You should specify the class name using the standard form ie. `Eg.Foo+Bar, Eg`. Due to an HQL parser limitation inner classes can not be used in queries in NHibernate 1.0.

Changes to immutable classes, `mutable="false"`, will not be persisted. This allows NHibernate to make some minor performance optimizations.

The optional `proxy` attribute enables lazy initialization of persistent instances of the class. NHibernate will initially return proxies which implement the named interface. The actual persistent object will be loaded when a method of the proxy is invoked. See "Proxies for Lazy Initialization" below.

Implicit polymorphism means that instances of the class will be returned by a query that names any superclass or implemented interface or the class and that instances of any subclass of the class will be returned by a query that names the class itself. *Explicit* polymorphism means that class instances will be returned only be queries that explicitly name that class and that queries that name the class will return only instances of subclasses mapped inside this `<class>` declaration as a `<subclass>` or `<joined-subclass>`. For most purposes the default, `polymorphism="implicit"`, is appropriate. Explicit polymorphism is useful when two different classes are mapped to the same table (this allows a "lightweight" class that contains a subset of the table columns).

The `persister` attribute lets you customize the persistence strategy used for the class. You may, for example, specify your own subclass of `NHibernate.Persister.EntityPersister` or you might even provide a completely new implementation of the interface `NHibernate.Persister.IClassPersister` that implements persistence via, for example, stored procedure calls, serialization to flat files or LDAP. See `NHibernate.DomainModel.CustomPersister` for a simple example (of "persistence" to a `Hashtable`).

Note that the `dynamic-update` and `dynamic-insert` settings are not inherited by subclasses and so may also be specified on the `<subclass>` or `<joined-subclass>` elements. These settings may increase performance in some cases, but might actually decrease performance in others. Use judiciously.

Use of `select-before-update` will usually decrease performance. It is very useful to prevent a database update trigger being called unnecessarily.

If you enable `dynamic-update`, you will have a choice of optimistic locking strategies:

- `version` check the version/timestamp columns
- `all` check all columns
- `dirty` check the changed columns
- `none` do not use optimistic locking

We *very* strongly recommend that you use version/timestamp columns for optimistic locking with NHibernate. This is the optimal strategy with respect to performance and is the only strategy that correctly handles modifications made outside of the session (ie. when `ISession.Update()` is used). Keep in mind that a version or timestamp property should never be null, no matter what `unsaved-value` strategy, or an instance will be detected as transient.

Beginning with NHibernate 1.2.0, version numbers start with 1, not 0 as in previous versions. This was done to allow using 0 as `unsaved-value` for the version property.

5.1.4. subselect

An alternative to mapping a class is to map a *query*. For that, we use the `<subselect>` element, which is mutually exclusive with `<class>`, `<subclass>`, `<joined-subclass>` and `<union-subclass>`. The content of the `subselect` element is a SQL query:

```
<subselect>
    SELECT cat.ID, cat.NAME, cat.SEX, cat.MATE FROM cat
</subselect>
```

Normally, when mapping a query using `subselect` you will want to mark the class as not mutable (`mutable="false"`), unless you specify custom SQL for performing the UPDATE, DELETE and INSERT operations.

Also, it makes sense to force synchronization of the tables affected by the query, using one or more `<synchronize>` entries:

```
<subselect>
    SELECT cat.ID, cat.NAME, cat.SEX, cat.MATE FROM cat
</subselect>
<syncronize table="cat"/>
```

5.1.5. id

Mapped classes *must* declare the primary key column of the database table. Most classes will also have a property holding the unique identifier of an instance. The `<id>` element defines the mapping from that property to the primary key column.

```
<id
        name="PropertyName"                      (1)
        type="typename"                          (2)
        column="column_name"                     (3)
        unsaved-value="any|none|null|id_value"   (4)
        access="field|property|nosetter|ClassName(5)">

        <generator class="generatorClass"/>
</id>
```

(1) `name` (optional): The name of the identifier property.

(2) `type` (optional): A name that indicates the NHibernate type.

(3) `column` (optional - defaults to the property name): The name of the primary key column.

(4) `unsaved-value` (optional - defaults to a "sensible" value): An identifier property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session.

(5) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

If the `name` attribute is missing, it is assumed that the class has no identifier property.

The `unsaved-value` attribute is almost never needed in NHibernate 1.0.

There is an alternative `<composite-id>` declaration to allow access to legacy data with composite keys. We strongly discourage its use for anything else.

5.1.5.1. generator

The required `generator` names a .NET class used to generate unique identifiers for instances of the persistent class.

The generator can be declared using the `<generator>` child element. If any parameters are required to configure or initialize the generator instance, they are passed using `<param>` elements.

```
<id name="Id" type="Int64" column="uid" unsaved-value="0">
        <generator class="NHibernate.Id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

If no parameters are required, the generator can be declared using a `generator` attribute directly on the `<id>` element, as follows:

```
<id name="Id" type="Int64" column="uid" unsaved-value="0" generator="native" />
```

All generators implement the interface `NHibernate.Id.IIdentifierGenerator`. This is a very simple interface; some applications may choose to provide their own specialized implementations. However, NHibernate provides a range of built-in implementations. There are shortcut names for the built-in generators:

`increment`

generates identifiers of any integral type that are unique only when no other process is inserting data into the same table. *Do not use in a cluster.*

`identity`

supports identity columns in DB2, MySQL, MS SQL Server and Sybase. The identifier returned by the database is converted to the property type using `Convert.ChangeType`. Any integral property type is thus supported.

`sequence`

uses a sequence in DB2, PostgreSQL, Oracle or a generator in Firebird. The identifier returned by the database is converted to the property type using `Convert.ChangeType`. Any integral property type is thus supported.

`hilo`

uses a hi/lo algorithm to efficiently generate identifiers of any integral type, given a table and column (by default `hibernate_unique_key` and `next_hi` respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database. *Do not use this generator with a user-supplied connection.*

You can use the "where" parameter to specify the row to use in a table. This is useful if you want to use a single tabel for your identifiers, with different rows for each table.

`seqhilo`

uses a hi/lo algorithm to efficiently generate identifiers of any integral type, given a named database sequence.

`uuid.hex`

uses `System.Guid` and its `ToString(string format)` method to generate identifiers of type string. The length of the string returned depends on the configured `format`.

`uuid.string`

uses a new `System.Guid` to create a `byte[]` that is converted to a string.

`guid`

uses a new `System.Guid` as the identifier.

`guid.comb`

uses the algorithm to generate a new `System.Guid` described by Jimmy Nilsson in the article http://www.informit.com/articles/article.asp?p=25862.

`native`

picks `identity`, `sequence` or `hilo` depending upon the capabilities of the underlying database.

`assigned`

lets the application to assign an identifier to the object before `Save()` is called.

`foreign`

uses the identifier of another associated object. Usually used in conjunction with a `<one-to-one>` primary key association.

5.1.5.2. Hi/Lo Algorithm

The `hilo` and `seqhilo` generators provide two alternate implementations of the hi/lo algorithm, a favorite approach to identifier generation. The first implementation requires a "special" database table to hold the next available "hi" value. The second uses an Oracle-style sequence (where supported).

```
<id name="Id" type="Int64" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="Id" type="Int64" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Unfortunately, you can't use `hilo` when supplying your own `IDbConnection` to NHibernate. NHibernate must be able to fetch the "hi" value in a new transaction.

5.1.5.3. UUID Hex Algorithm

```
<id name="Id" type="String" column="cat_id">
        <generator class="uuid.hex">
            <param name="format">format_value</param>
            <param name="separator">separator_value</param>
        </generator>
</id>
```

The UUID is generated by calling `Guid.NewGuid().ToString(format)`. The valid values for `format` are described in the MSDN documentation. The default `separator` is - and should rarely be modified. The `format` determines if the configured `separator` can replace the default separator used by the `format`.

5.1.5.4. UUID String Algorithm

The UUID is generated by calling `Guid.NewGuid().ToByteArray()` and then converting the `byte[]` into a `char[]`. The `char[]` is returned as a `String` consisting of 16 characters.

5.1.5.5. GUID Algorithms

The `guid` identifier is generated by calling `Guid.NewGuid()`. To address some of the performance concerns with using Guids as primary keys, foreign keys, and as part of indexes with MS SQL the `guid.comb` can be used. The benefit of using the `guid.comb` with other databases that support GUIDs has not been measured.

5.1.5.6. Identity columns and Sequences

For databases which support identity columns (DB2, MySQL, Sybase, MS SQL), you may use `identity` key generation. For databases that support sequences (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) you may use `sequence` style key generation. Both these strategies require two SQL queries to insert a new object.

```
<id name="Id" type="Int64" column="uid">
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
</id>
```

```
<id name="Id" type="Int64" column="uid" unsaved-value="0">
        <generator class="identity"/>
</id>
```

For cross-platform development, the `native` strategy will choose from the `identity`, `sequence` and `hilo` strategies, dependent upon the capabilities of the underlying database.

5.1.5.7. Assigned Identifiers

If you want the application to assign identifiers (as opposed to having NHibernate generate them), you may use the `assigned` generator. This special generator will use the identifier value already assigned to the object's identifier property.

Be very careful when using this feature to assign keys with business meaning (almost always a terrible design decision).

Due to its inherent nature, entities that use this generator cannot be saved via the ISession's SaveOrUpdate() method. Instead you have to explicitly specify to NHibernate if the object should be saved or updated by calling either the `Save()` or `Update()` method of the ISession.

5.1.5.8. Enhanced identifier generators

Starting with NHibernate release 3.3.0, there are 2 new generators which represent a re-thinking of 2 different aspects of identifier generation. The first aspect is database portability; the second is optimization Optimization means that you do not have to query the database for every request for a new identifier value. These two new generators are intended to take the place of some of the named generators described above, starting in 3.3.x. However, they are included in the current releases and can be referenced by FQN.

The first of these new generators is `NHibernate.Id.Enhanced.SequenceStyleGenerator` (short name `enhanced-sequence`) which is intended, firstly, as a replacement for the `sequence` generator and, secondly, as a better portability generator than `native`. This is because `native` generally chooses between `identity` and `sequence` which have largely different semantics that can cause subtle issues in applications eyeing portability. `NHibernate.Id.Enhanced.SequenceStyleGenerator`, however, achieves portability in a different manner. It chooses between a table or a sequence in the database to store its incrementing values, depending on the capabilities of the dialect being used. The difference between this and `native` is that table-based and sequence-based storage have the same exact semantic. In fact, sequences are exactly what NHibernate tries to emulate with its table-based generators. This generator has a number of configuration parameters:

- `sequence_name` (optional, defaults to `hibernate_sequence`): the name of the sequence or table to be used.
- `initial_value` (optional, defaults to `1`): the initial value to be retrieved from the sequence/table. In sequence creation terms, this is analogous to the clause typically named "STARTS WITH".
- `increment_size` (optional - defaults to `1`): the value by which subsequent calls to the sequence/table should differ. In sequence creation terms, this is analogous to the clause typically named "INCREMENT BY".
- `force_table_use` (optional - defaults to `false`): should we force the use of a table as the backing structure even though the dialect might support sequence?
- `value_column` (optional - defaults to `next_val`): only relevant for table structures, it is the name of the column on the table which is used to hold the value.
- `prefer_sequence_per_entity` (optional - defaults to `false`): should we create separate sequence for each entity that share current generator based on its name?
- `sequence_per_entity_suffix` (optional - defaults to `_SEQ`): suffix added to the name of a dedicated sequence.
- `optimizer` (optional - defaults to `none`): See Section 5.1.5.8.1, “Identifier generator optimization”

The second of these new generators is `NHibernate.Id.Enhanced.TableGenerator` (short name `enhanced-table`), which is intended, firstly, as a replacement for the `table` generator, even though it actually functions much more like `org.hibernate.id.MultipleHiLoPerTableGenerator` (not available in NHibernate), and secondly, as a re-implementation of `org.hibernate.id.MultipleHiLoPerTableGenerator` (not available in NHibernate) that utilizes the notion of pluggable optimizers. Essentially this generator defines a table capable of holding a number of different increment values simultaneously by using multiple distinctly keyed rows. This generator has a number of configuration parameters:

- `table_name` (optional - defaults to `hibernate_sequences`): the name of the table to be used.
- `value_column_name` (optional - defaults to `next_val`): the name of the column on the table that is used to hold the value.
- `segment_column_name` (optional - defaults to `sequence_name`): the name of the column on the table that is used to hold the "segment key". This is the value which identifies which increment value to use.
- `segment_value` (optional - defaults to `default`): The "segment key" value for the segment from which we want to pull increment values for this generator.
- `segment_value_length` (optional - defaults to `255`): Used for schema generation; the column size to create this segment key column.
- `initial_value` (optional - defaults to `1`): The initial value to be retrieved from the table.
- `increment_size` (optional - defaults to `1`): The value by which subsequent calls to the table should differ.

- `optimizer` (optional - defaults to `??`): See Section 5.1.5.8.1, “Identifier generator optimization”.

5.1.5.8.1. Identifier generator optimization

For identifier generators that store values in the database, it is inefficient for them to hit the database on each and every call to generate a new identifier value. Instead, you can group a bunch of them in memory and only hit the database when you have exhausted your in-memory value group. This is the role of the pluggable optimizers. Currently only the two enhanced generators (Section 5.1.5.8, “Enhanced identifier generators” support this operation.

- `none` (generally this is the default if no optimizer was specified): this will not perform any optimizations and hit the database for each and every request.
- `hilo`: applies a hi/lo algorithm around the database retrieved values. The values from the database for this optimizer are expected to be sequential. The values retrieved from the database structure for this optimizer indicates the "group number". The `increment_size` is multiplied by that value in memory to define a group "hi value".
- `pooled`: as with the case of `hilo`, this optimizer attempts to minimize the number of hits to the database. Here, however, we simply store the starting value for the "next group" into the database structure rather than a sequential value in combination with an in-memory grouping algorithm. Here, `increment_size` refers to the values coming from the database.
- `pooled-lo`: similar to `pooled`, except that it's the starting value of the "current group" that is stored into the database structure. Here, `increment_size` refers to the values coming from the database.

5.1.6. composite-id

```
<composite-id
        name="PropertyName"
        class="ClassName"
        unsaved-value="any|none"
        access="field|property|nosetter|ClassName">

        <key-property name="PropertyName" type="typename" column="column_name"/>
        <key-many-to-one name="PropertyName class="ClassName" column="column_name"/>
        ......
</composite-id>
```

For a table with a composite key, you may map multiple properties of the class as identifier properties. The `<composite-id>` element accepts `<key-property>` property mappings and `<key-many-to-one>` mappings as child elements.

```
<composite-id>
        <key-property name="MedicareNumber"/>
        <key-property name="Dependent"/>
</composite-id>
```

Your persistent class *must* override `Equals()` and `GetHashCode()` to implement composite identifier equality. It must also be marked with the `Serializable` attribute.

Unfortunately, this approach to composite identifiers means that a persistent object is its own identifier. There is no convenient "handle" other than the object itself. You must instantiate an instance of the persistent class itself and populate its identifier properties before you can `Load()` the persistent state associated with a composite key. We will describe a much more convenient approach where the composite identifier is implemented as a saparate class in Section 7.4, “Components as composite identifiers”. The attributes described below apply only to this alternative approach:

- `name` (optional, required for this approach): A property of component type that holds the composite identifier (see next section).
- `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.
- `class` (optional - defaults to the property type determined by reflection): The component class used as a composite identifier (see next section).

5.1.7. discriminator

The `<discriminator>` element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy and declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. A restricted set of types may be used: `String`, `Char`, `Int32`, `Byte`, `Short`, `Boolean`, `YesNo`, `TrueFalse`.

```
<discriminator
```

```
        column="discriminator_column"  (1)
        type="discriminator_type"      (2)
        force="true|false"             (3)
        insert="true|false"            (4)
        formula="arbitrary SQL expressi(5)on"
/>
```

(1) column (optional - defaults to class) the name of the discriminator column.

(2) type (optional - defaults to String) a name that indicates the NHibernate type

(3) force (optional - defaults to false) "force" NHibernate to specify allowed discriminator values even when retrieving all instances of the root class.

(4) insert (optional - defaults to true) set this to false if your discriminator column is also part of a mapped composite identifier.

(5) formula (optional) an arbitrary SQL expression that is executed when a type has to be evaluated. Allows content-based discrimination.

Actual values of the discriminator column are specified by the discriminator-value attribute of the <class> and <subclass> elements.

The force attribute is (only) useful if the table contains rows with "extra" discriminator values that are not mapped to a persistent class. This will not usually be the case.

Using the formula attribute you can declare an arbitrary SQL expression that will be used to evaluate the type of a row:

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="Int32"/>
```

5.1.8. version (optional)

The <version> element is optional and indicates that the table contains versioned data. This is particularly useful if you plan to use *long transactions* (see below).

```
<version
        column="version_column"                          (1)
        name="PropertyName"                              (2)
        type="typename"                                  (3)
        access="field|property|nosetter|ClassName"       (4)
        unsaved-value="null|negative|undefined|value"    (5)
        generated="never|always"                         (6)
/>
```

(1) column (optional - defaults to the property name): The name of the column holding the version number.

(2) name: The name of a property of the persistent class.

(3) type (optional - defaults to Int32): The type of the version number.

(4) access (optional - defaults to property): The strategy NHibernate should use for accessing the property value.

(5) unsaved-value (optional - defaults to a "sensible" value): A version property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session. (undefined specifies that the identifier property value should be used.)

(6) generated (optional - defaults to never): Specifies that this version property value is actually generated by the database. See the discussion of Section 5.5, “Generated Properties”.

Version numbers may be of type Int64, Int32, Int16, Ticks, Timestamp, or TimeSpan (or their nullable counterparts in .NET 2.0).

5.1.9. timestamp (optional)

The optional <timestamp> element indicates that the table contains timestamped data. This is intended as an alternative to versioning. Timestamps are by nature a less safe implementation of optimistic locking. However, sometimes the application might use the timestamps in other ways.

```
<timestamp
        column="timestamp_column"           (1)
        name="PropertyName"                 (2)
        access="field|property|nosetter|Clas(3)sName"
        unsaved-value="null|undefined|value"(4)
        generated="never|always"            (5)
/>
```

(1) column (optional - defaults to the property name): The name of a column holding the timestamp.

name: The name of a property of .NET type DateTime of the persistent class.

(2)

(3) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

(4) `unsaved-value` (optional - defaults to `null`): A timestamp property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from transient instances that were saved or loaded in a previous session. (`undefined` specifies that the identifier property value should be used.)

(5) `generated` (optional - defaults to `never`): Specifies that this timestamp property value is actually generated by the database. See the discussion of Section 5.5, “Generated Properties”.

Note that `<timestamp>` is equivalent to `<version type="timestamp">`.

5.1.10. property

The `<property>` element declares a persistent property of the class.

```
<property
        name="propertyName"                 (1)
        column="column_name"                (2)
        type="typename"                     (3)
        update="true|false"                 (4)
        insert="true|false"                 (4)
        formula="arbitrary SQL expression"  (5)
        access="field|property|ClassName"   (6)
        optimistic-lock="true|false"        (7)
        generated="never|insert|always"     (8)
        lazy="true|false"                   (9)
/>
```

(1) `name`: the name of the property of your class.

(2) `column` (optional - defaults to the property name): the name of the mapped database table column.

(3) `type` (optional): a name that indicates the NHibernate type.

(4) `update, insert` (optional - defaults to `true`) : specifies that the mapped columns should be included in SQL `UPDATE` and/or `INSERT` statements. Setting both to `false` allows a pure "derived" property whose value is initialized from some other property that maps to the same column(s) or by a trigger or other application.

(5) `formula` (optional): an SQL expression that defines the value for a *computed* property. Computed properties do not have a column mapping of their own.

(6) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

(7) `optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.

(8) `generated` (optional - defaults to `never`): Specifies that this property value is actually generated by the database. See the discussion of Section 5.5, “Generated Properties”.

(9) `lazy` (optional - defaults to `false`): Specifies that this property is lazy. A lazy property is not loaded when the object is initially loaded, unless the fetch mode has been overriden in a specific query. Values for lazy properties are loaded when any lazy property of the object is accessed.

typename could be:

- The name of a NHibernate basic type (eg. `Int32, String, Char, DateTime, Timestamp, Single, Byte[], Object, ...`).
- The name of a .NET type with a default basic type (eg. `System.Int16, System.Single, System.Char, System.String, System.DateTime, System.Byte[], ...`).
- The name of an enumeration type (eg. `Eg.Color, Eg`).
- The name of a serializable .NET type.
- The class name of a custom type (eg. `Illflow.Type.MyCustomType`).

Note that you have to specify full *assembly-qualified* names for all except basic NHibernate types (unless you set `assembly` and/or `namespace` attributes of the `<hibernate-mapping>` element).

NHibernate supports .NET 2.0 `Nullable` types. These types are mostly treated the same as plain non-`Nullable` types internally. For example, a property of type `Nullable<Int32>` can be mapped using `type="Int32"` or `type="System.Int32"`.

If you do not specify a type, NHibernate will use reflection upon the named property to take a guess at the correct NHibernate type. NHibernate will try to interpret the name of the return class of the property getter using rules 2, 3, 4 in that order. However, this is not always enough. In certain cases you will still need the `type` attribute. (For example, to distinguish between `NHibernateUtil.DateTime` and `NHibernateUtil.Timestamp`, or to specify a custom type.)

The `access` attribute lets you control how NHibernate will access the value of the property at runtime. The value of the `access` attribute should be text formatted as `access-strategy.naming-strategy`. The `.naming-strategy` is not always required.

Table 5.1. Access Strategies

Access Strategy Name	Description
`property`	The default implementation. NHibernate uses the get/set accessors of the property. No naming strategy should be used with this access strategy because the value of the `name` attribute is the name of the property.
`field`	NHibernate will access the field directly. NHibernate uses the value of the `name` attribute as the name of the field. This can be used when a property's getter and setter contain extra actions that you don't want to occur when NHibernate is populating or reading the object. If you want the name of the property and not the field to be what the consumers of your API use with HQL, then a naming strategy is needed.
`nosetter`	NHibernate will access the field directly when setting the value and will use the Property when getting the value. This can be used when a property only exposes a get accessor because the consumers of your API can't change the value directly. A naming strategy is required because NHibernate uses the value of the `name` attribute as the property name and needs to be told what the name of the field is.
`ClassName`	If NHibernate's built in access strategies are not what is needed for your situation then you can build your own by implementing the interface `NHibernate.Property.IPropertyAccessor`. The value of the `access` attribute should be an assembly-qualified name that can be loaded with `Activator.CreateInstance(string assemblyQualifiedName).`

Table 5.2. Naming Strategies

Naming Strategy Name	Description
`camelcase`	The `name` attribute is converted to camel case to find the field. `<property name="FooBar" ... >` uses the field `fooBar`.
`camelcase-underscore`	The `name` attribute is converted to camel case and prefixed with an underscore to find the field. `<property name="FooBar" ... >` uses the field `_fooBar`.
`camelcase-m-underscore`	The `name` attribute is converted to camel case and prefixed with the character `m` and an underscore to find the field. `<property name="FooBar" ... >` uses the field `m_fooBar`.
`lowercase`	The `name` attribute is converted to lower case to find the Field. `<property name="FooBar" ... >` uses the field `foobar`.
`lowercase-underscore`	The `name` attribute is converted to lower case and prefixed with an underscore to find the Field. `<property name="FooBar" ... >` uses the field `_foobar`.
`pascalcase-underscore`	The `name` attribute is prefixed with an underscore to find the field. `<property name="FooBar" ... >` uses the field `_FooBar`.
`pascalcase-m`	The `name` attribute is prefixed with the character `m` to find the field. `<property name="FooBar" ... >` uses the field `mFooBar`.
`pascalcase-m-underscore`	The `name` attribute is prefixed with the character `m` and an underscore to find the field. `<property name="FooBar" ... >` uses the field `m_FooBar`.

5.1.11. many-to-one

An ordinary association to another persistent class is declared using a `many-to-one` element. The relational model is a many-to-one association. (It's really just an object reference.)

```
<many-to-one
        name="PropertyName"                                (1)
        column="column_name"                               (2)
        class="ClassName"                                  (3)
        cascade="all|none|save-update|delete|delete-orphan|(4)all-delete-orphan"
        fetch="join|select"                                (5)
        update="true|false"                                (6)
        insert="true|false"                                (6)
        property-ref="PropertyNameFromAssociatedClass"     (7)
        access="field|property|nosetter|ClassName"         (8)
        unique="true|false"                                (9)
        optimistic-lock="true|false"                       (10)
        not-found="ignore|exception"                       (11)
/>
```

(1) `name`: The name of the property.

(2) `column` (optional): The name of the column.

(3) `class` (optional - defaults to the property type determined by reflection): The name of the associated class.

(4) `cascade` (optional): Specifies which operations should be cascaded from the parent object to the associated object.

(5) `fetch` (optional - defaults to `select`): Chooses between outer-join fetching or sequential select fetching.

(6) `update, insert` (optional - defaults to `true`) specifies that the mapped columns should be included in SQL `UPDATE` and/or `INSERT` statements. Setting both to `false` allows a pure "derived" association whose value is initialized from some other property that maps to the same colum(s) or by a trigger or other application.

(7) `property-ref`: (optional) The name of a property of the associated class that is joined to this foreign key. If not specified, the primary key of the associated class is used.

(8) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

(9) `unique` (optional): Enable the DDL generation of a unique constraint for the foreign-key column.

(10) `optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, dertermines if a version increment should occur when this property is dirty.

(11) `not-found` (optional - defaults to `exception`): Specifies how foreign keys that reference missing rows will be handled: `ignore` will treat a missing row as a null association.

The `cascade` attribute permits the following values: `all`, `save-update`, `delete`, `none`. Setting a value other than `none` will propagate certain operations to the associated (child) object. See "Lifecycle Objects" below.

The `fetch` attribute accepts two different values:

- `join` Fetch the association using an outer join
- `select` Fetch the association using a separate query

A typical `many-to-one` declaration looks as simple as

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

The `property-ref` attribute should only be used for mapping legacy data where a foreign key refers to a unique key of the associated table other than the primary key. This is an ugly relational model. For example, suppose the `Product` class had a unique serial number, that is not the primary key. (The `unique` attribute controls NHibernate's DDL generation with the SchemaExport tool.)

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

Then the mapping for `OrderItem` might use:

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

This is certainly not encouraged, however.

5.1.12. one-to-one

A one-to-one association to another persistent class is declared using a `one-to-one` element.

```
<one-to-one
        name="PropertyName"                                (1)
        class="ClassName"                                  (2)
        cascade="all|none|save-update|delete|delete-orphan|(3)all-delete-orphan"
        constrained="true|false"                           (4)
        fetch="join|select"                                (5)
        property-ref="PropertyNameFromAssociatedClass"     (6)
        access="field|property|nosetter|ClassName"         (7)
/>
```

(1) `name`: The name of the property.

(2) `class` (optional - defaults to the property type determined by reflection): The name of the associated class.

(3) `cascade` (optional) specifies which operations should be cascaded from the parent object to the associated object.

(4) `constrained` (optional) specifies that a foreign key constraint on the primary key of the mapped table references the table of the associated class. This option affects the order in which `Save()` and `Delete()` are cascaded (and is also used by the schema export tool).

(5) `fetch` (optional - defaults to `select`): Chooses between outer-join fetching or sequential select fetching.

(6) `property-ref`: (optional) The name of a property of the associated class that is joined to the primary key of this class. If not specified, the primary key of the associated class is used.

(7) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

There are two varieties of one-to-one association:

- primary key associations
- unique foreign key associations

Primary key associations don't need an extra table column; if two rows are related by the association then the two table rows share the same primary key value. So if you want two objects to be related by a primary key association, you must make sure that they are assigned the same identifier value!

For a primary key association, add the following mappings to `Employee` and `Person`, respectively.

```
<one-to-one name="Person" class="Person"/>
```

```
<one-to-one name="Employee" class="Employee" constrained="true"/>
```

Now we must ensure that the primary keys of related rows in the PERSON and EMPLOYEE tables are equal. We use a special NHibernate identifier generation strategy called `foreign`:

```
<class name="Person" table="PERSON">
    <id name="Id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">Employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="Employee"
        class="Employee"
        constrained="true"/>
</class>
```

A newly saved instance of `Person` is then assigned the same primar key value as the `Employee` instance refered with the `Employee` property of that `Person`.

Alternatively, a foreign key with a unique constraint, from `Employee` to `Person`, may be expressed as:

```
<many-to-one name="Person" class="Person" column="PERSON_ID" unique="true"/>
```

And this association may be made bidirectional by adding the following to the `Person` mapping:

```
<one-to-one name="Employee" class="Employee" property-ref="Person"/>
```

5.1.13. natural-id

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

Even though we recommend the use of surrogate keys as primary keys, you should still try to identify natural keys for all entities. A natural key is a property or combination of properties that is unique and non-null. If it is also immutable, even better. Map the properties of the natural key inside the `<natural-id>` element. NHibernate will generate the necessary unique key and nullability constraints, and your mapping will be more self-documenting.

We strongly recommend that you implement `Equals()` and `GetHashCode()` to compare the natural key properties of the entity.

This mapping is not intended for use with entities with natural primary keys.

- `mutable` (optional, defaults to `false`): By default, natural identifier properties as assumed to be immutable (constant).

5.1.14. component, dynamic-component

The `<component>` element maps properties of a child object to columns of the table of a parent class. Components may, in turn, declare their own properties, components or collections. See "Components" below.

```
<component
        name="PropertyName"                                (1)
        class="ClassName"                                  (2)
        insert="true|false"                                (3)
        upate="true|false"                                 (4)
        access="field|property|nosetter|ClassName"         (5)
        optimistic-lock="true|false">                      (6)

        <property ...../>
        <many-to-one .... />
        ........
```

```
</component>
```

(1) `name`: The name of the property.

(2) `class` (optional - defaults to the property type determined by reflection): The name of the component (child) class.

(3) `insert`: Do the mapped columns appear in SQL `INSERT`s?

(4) `update`: Do the mapped columns appear in SQL `UPDATE`s?

(5) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

(6) `optimistic-lock` (optional - defaults to `true`): Specifies that updates to this component do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.

The child `<property>` tags map properties of the child class to table columns.

The `<component>` element allows a `<parent>` subelement that maps a property of the component class as a reference back to the containing entity.

The `<dynamic-component>` element allows an `IDictionary` to be mapped as a component, where the property names refer to keys of the dictionary.

5.1.15. properties

The `<properties>` element allows the definition of a named, logical grouping of the properties of a class. The most important use of the construct is that it allows a combination of properties to be the target of a `property-ref`. It is also a convenient way to define a multi-column unique constraint. For example:

```
<properties
      name="logicalName"                                  (1)
      insert="true|false"                                 (2)
      update="true|false"                                 (3)
      optimistic-lock="true|false"                        (4)
      unique="true|false">                                (5)

      <property .../>
      <many-to-one .../>
      ........
</properties>
```

(1) `name`: the logical name of the grouping. It is *not* an actual property name.

(2) `insert`: do the mapped columns appear in SQL `INSERT`s?

(3) `update`: do the mapped columns appear in SQL `UPDATE`s?

(4) `optimistic-lock` (optional - defaults to `true`): specifies that updates to these properties either do or do not require acquisition of the optimistic lock. It determines if a version increment should occur when these properties are dirty.

(5) `unique` (optional - defaults to `false`): specifies that a unique constraint exists upon all mapped columns of the component.

For example, if we have the following `<properties>` mapping:

```
<class name="Person">
      <id name="personNumber" />
      <properties name="name" unique="true" update="false">
          <property name="firstName" />
          <property name="lastName" />
          <property name="initial" />
      </properties>
</class>
```

You might have some legacy data association that refers to this unique key of the `Person` table, instead of to the primary key:

```
<many-to-one name="owner" class="Person" property-ref="name">
        <column name="firstName" />
        <column name="lastName" />
        <column name="initial" />
</many-to-one>
```

The use of this outside the context of mapping legacy data is not recommended.

5.1.16. subclass

Finally, polymorphic persistence requires the declaration of each subclass of the root persistent class. For the (recommended) table-per-class-hierarchy mapping strategy, the `<subclass>` declaration is used.

```
<subclass
        name="ClassName"                              (1)
        discriminator-value="discriminator_value"     (2)
        proxy="ProxyInterface"                        (3)
        lazy="true|false"                             (4)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <property .... />
        <properties .... />
        .....
</subclass>
```

(1) `name`: The fully qualified .NET class name of the subclass, including its assembly name.

(2) `discriminator-value` (optional - defaults to the class name): A value that distiguishes individual subclasses.

(3) `proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.

(4) `lazy` (optional, defaults to `true`): Setting `lazy="false"` disables the use of lazy fetching.

Each subclass should declare its own persistent properties and subclasses. `<version>` and `<id>` properties are assumed to be inherited from the root class. Each subclass in a hierarchy must define a unique `discriminator-value`. If none is specified, the fully qualified .NET class name is used.

For information about inheritance mappings, see Chapter 8, *Inheritance Mapping*.

5.1.17. joined-subclass

Alternatively, a subclass that is persisted to its own table (table-per-subclass mapping strategy) is declared using a `<joined-subclass>` element.

```
<joined-subclass
        name="ClassName"                    (1)
        proxy="ProxyInterface"              (2)
        lazy="true|false"                   (3)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

        <property .... />
        <properties .... />
        .....
</joined-subclass>
```

(1) `name`: The fully qualified class name of the subclass.

(2) `proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.

(3) `lazy` (optional): Setting `lazy="true"` is a shortcut equivalent to specifying the name of the class itself as the `proxy` interface.

No discriminator column is required for this mapping strategy. Each subclass must, however, declare a table column holding the object identifier using the `<key>` element. The mapping at the start of the chapter would be re-written as:

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Eg"
    namespace="Eg">

        <class name="Cat" table="CATS">
                <id name="Id" column="uid" type="Int64">
                        <generator class="hilo"/>
                </id>
                <property name="BirthDate" type="Date"/>
                <property name="Color" not-null="true"/>
                <property name="Sex" not-null="true"/>
                <property name="Weight"/>
                <many-to-one name="Mate"/>
                <set name="Kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                    <key column="CAT"/>
                        <property name="Name" type="String"/>
                </joined-subclass>
        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>
```

```
</hibernate-mapping>
```

For information about inheritance mappings, see [Chapter 8, *Inheritance Mapping*]().

5.1.18. union-subclass

A third option is to map only the concrete classes of an inheritance hierarchy to tables, (the table-per-concrete-class strategy) where each table defines all persistent state of the class, including inherited state. In NHibernate, it is not absolutely necessary to explicitly map such inheritance hierarchies. You can simply map each class with a separate `<class>` declaration. However, if you wish use polymorphic associations (e.g. an association to the superclass of your hierarchy), you need to use the `<union-subclass>` mapping.

```
<union-subclass
        name="ClassName"                    (1)
        table="tablename"                   (2)
        proxy="ProxyInterface"              (3)
        lazy="true|false"                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <property .... />
        <properties .... />
        .....
</union-subclass>
```

(1) `name`: The fully qualified class name of the subclass.

(2) `table`: The name of the subclass table.

(3) `proxy` (optional): Specifies a class or interface to use for lazy initializing proxies.

(4) `lazy` (optional, defaults to `true`): Setting `lazy="false"` disables the use of lazy fetching.

No discriminator column or key column is required for this mapping strategy.

For information about inheritance mappings, see [Chapter 8, *Inheritance Mapping*]().

5.1.19. join

Using the `<join>` element, it is possible to map properties of one class to several tables, when there's a 1-to-1 relationship between the tables.

```
<join
        table="tablename"                        (1)
        schema="owner"                           (2)
        fetch="join|select"                      (3)
        inverse="true|false"                     (4)
        optional="true|false">                   (5)

        <key ... />

        <property ... />
        ...
</join>
```

(1) `table`: The name of the joined table.

(2) `schema` (optional): Override the schema name specified by the root `<hibernate-mapping>` element.

(3) `fetch` (optional - defaults to `join`): If set to `join`, the default, NHibernate will use an inner join to retrieve a `<join>` defined by a class or its superclasses and an outer join for a `<join>` defined by a subclass. If set to `select` then NHibernate will use a sequential select for a `<join>` defined on a subclass, which will be issued only if a row turns out to represent an instance of the subclass. Inner joins will still be used to retrieve a `<join>` defined by the class and its superclasses.

(4) `inverse` (optional - defaults to `false`): If enabled, NHibernate will not try to insert or update the properties defined by this join.

(5) `optional` (optional - defaults to `false`): If enabled, NHibernate will insert a row only if the properties defined by this join are non-null and will always use an outer join to retrieve the properties.

For example, the address information for a person can be mapped to a separate table (while preserving value type semantics for all properties):

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

This feature is often only useful for legacy data models, we recommend fewer tables than classes and a fine-grained domain model. However, it is useful for switching between inheritance mapping strategies in a single hierarchy, as explained later.

5.1.20. map, set, list, bag

Collections are discussed later.

5.1.21. import

Suppose your application has two persistent classes with the same name, and you don't want to specify the fully qualified name in NHibernate queries. Classes may be "imported" explicitly, rather than relying upon `auto-import="true"`. You may even import classes and interfaces that are not explicitly mapped.

```
<import class="System.Object" rename="Universe"/>
```

```
<import
        class="ClassName"              (1)
        rename="ShortName"             (2)
/>
```

(1) `class`: The fully qualified class name of any .NET class, including its assembly name.

(2) `rename` (optional - defaults to the unqualified class name): A name that may be used in the query language.

5.2. NHibernate Types

5.2.1. Entities and values

To understand the behaviour of various .NET language-level objects with respect to the persistence service, we need to classify them into two groups:

An *entity* exists independently of any other objects holding references to the entity. Contrast this with the usual .NET model where an unreferenced object is garbage collected. Entities must be explicitly saved and deleted (except that saves and deletions may be *cascaded* from a parent entity to its children). This is different from the ODMG model of object persistence by reachability - and corresponds more closely to how application objects are usually used in large systems. Entities support circular and shared references. They may also be versioned.

An entity's persistent state consists of references to other entities and instances of *value* types. Values are primitives, collections, components and certain immutable objects. Unlike entities, values (in particular collections and components) *are* persisted and deleted by reachability. Since value objects (and primitives) are persisted and deleted along with their containing entity they may not be independently versioned. Values have no independent identity, so they cannot be shared by two entities or collections.

All NHibernate types except collections support null semantics if the .NET type is nullable (i.e. not derived from `System.ValueType`).

Up until now, we've been using the term "persistent class" to refer to entities. We will continue to do that. Strictly speaking, however, not all user-defined classes with persistent state are entities. A *component* is a user defined class with value semantics.

5.2.2. Basic value types

The *basic types* may be roughly categorized into three groups - `System.ValueType` types, `System.Object` types, and `System.Object` types for large objects. Just like the .NET Types, columns for System.ValueType types *can not* store `null` values and System.Object types *can* store `null` values.

Table 5.3. System.ValueType Mapping Types

NHibernate Type	.NET Type	Database Type	Remarks
`AnsiChar`	`System.Char`	`DbType.AnsiStringFixedLength - 1 char`	
`Boolean`	`System.Boolean`	`DbType.Boolean`	Default when no `type` attribute specified.
`Byte`	`System.Byte`	`DbType.Byte`	Default when no `type` attribute specified.
`Char`	`System.Char`	`DbType.StringFixedLength - 1 char`	Default when no `type` attribute specified.
`Date`	`System.DateTime`	`DbType.Date`	`type="Date"` must be specified.
`DateTime`	`System.DateTime`	`DbType.DateTime` - ignores the milliseconds	Default when no `type` attribute specified.
`DateTime2`	`System.DateTime`	`DbType.DateTime2`	`type="DateTime2"` must be specified.
`DbTimestamp`	`System.DateTime`	`DbType.DateTime` - as specific as database supports.	`type="DbTimestamp"` must be specified. When used as a `version` field, uses the database's current time rather than the client's current time.
`LocalDateTime`	`System.DateTime`	`DbType.DateTime` - ignores the milliseconds	Ensures the `DateTimeKind` is set to `DateTimeKind.Local`
`UtcDateTime`	`System.DateTime`	`DbType.DateTime` - ignores the milliseconds	Ensures the `DateTimeKind` is set to `DateTimeKind.Utc`
`Decimal`	`System.Decimal`	`DbType.Decimal`	Default when no `type` attribute specified.
`Double`	`System.Double`	`DbType.Double`	Default when no `type` attribute specified.
`Guid`	`System.Guid`	`DbType.Guid`	Default when no `type` attribute specified.
`Int16`	`System.Int16`	`DbType.Int16`	Default when no `type` attribute specified.
`Int32`	`System.Int32`	`DbType.Int32`	Default when no `type` attribute specified.
`Int64`	`System.Int64`	`DbType.Int64`	Default when no `type` attribute specified.
`PersistentEnum`	A `System.Enum`	The `DbType` for the underlying value.	Do not specify `type="PersistentEnum"` in the mapping. Instead specify the Assembly Qualified Name of the Enum or let NHibernate use Reflection to "guess" the Type. The UnderlyingType of the Enum is used to determine the correct `DbType`.
`Single`	`System.Single`	`DbType.Single`	Default when no `type` attribute specified.
`Ticks`	`System.DateTime`	`DbType.Int64`	`type="Ticks"` must be specified.
`Time`	`System.DateTime`	`DbType.Time`	`type="Time"` must be specified.
`TimeAsTimeSpan`	`System.TimeSpan`	`DbType.Time`	`type="TimeAsTimeSpan"` must be specified.
`TimeSpan`	`System.TimeSpan`	`DbType.Int64`	Default when no `type` attribute specified.
`Timestamp`	`System.DateTime`	`DbType.DateTime` - as specific as database supports.	`type="Timestamp"` must be specified.
`TrueFalse`	`System.Boolean`	`DbType.AnsiStringFixedLength` - 1 char either 'T' or 'F'	`type="TrueFalse"` must be specified.
`YesNo`	`System.Boolean`	`DbType.AnsiStringFixedLength` - 1 char either 'Y' or 'N'	`type="YesNo"` must be specified.

Table 5.4. System.Object Mapping Types

NHibernate Type	.NET Type	Database Type	Remarks
`AnsiString`	`System.String`	`DbType.AnsiString`	`type="AnsiString"` must be specified.
`CultureInfo`	`System.Globalization.CultureInfo`	`DbType.String` - 5 chars for culture	Default when no `type` attribute specified.
`Binary`	`System.Byte[]`	`DbType.Binary`	Default when no `type` attribute specified.
`Type`	`System.Type`	`DbType.String` holding Assembly Qualified Name.	Default when no `type` attribute specified.
`String`	`System.String`	`DbType.String`	Default when no `type` attribute specified.

Table 5.5. Large Object Mapping Types

NHibernate Type	.NET Type	Database Type	Remarks

`StringClob`	`System.String`	`DbType.String`	`type="StringClob"` must be specified. Entire field is read into memory.
`BinaryBlob`	`System.Byte[]`	`DbType.Binary`	`type="BinaryBlob"` must be specified. Entire field is read into memory.
`Serializable`	Any `System.Object` that is marked with SerializableAttribute.	`DbType.Binary`	`type="Serializable"` should be specified. This is the fallback type if no NHibernate Type can be found for the Property.

NHibernate supports some additional type names for compatibility with Java's Hibernate (useful for those coming over from Hibernate or using some of the tools to generate `hbm.xml` files). A `type="integer"` or `type="int"` will map to an `Int32` NHibernate type, `type="short"` to an `Int16` NHibernateType. To see all of the conversions you can view the source of static constructor of the class `NHibernate.Type.TypeFactory`.

5.2.3. Custom value types

It is relatively easy for developers to create their own value types. For example, you might want to persist properties of type `Int64` to `VARCHAR` columns. NHibernate does not provide a built-in type for this. But custom types are not limited to mapping a property (or collection element) to a single table column. So, for example, you might have a property `Name { get; set; }` of type `String` that is persisted to the columns `FIRST_NAME`, `INITIAL`, `SURNAME`.

To implement a custom type, implement either `NHibernate.UserTypes.IUserType` or `NHibernate.UserTypes.ICompositeUserType` and declare properties using the fully qualified name of the type. Check out `NHibernate.DomainModel.DoubleStringType` to see the kind of things that are possible.

```
<property name="TwoStrings" type="NHibernate.DomainModel.DoubleStringType, NHibernate.DomainModel">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

Notice the use of `<column>` tags to map a property to multiple columns.

The `ICompositeUserType`, `IEnhancedUserType`, `INullableUserType`, `IUserCollectionType`, and `IUserVersionType` interfaces provide support for more specialized uses.

You may even supply parameters to an `IUserType` in the mapping file. To do this, your `IUserType` must implement the `NHibernate.UserTypes.IParameterizedType` interface. To supply parameters to your custom type, you can use the `<type>` element in your mapping files.

```
<property name="priority">
    <type name="MyCompany.UserTypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

The `IUserType` can now retrieve the value for the parameter named `default` from the `IDictionary` object passed to it.

If you use a certain `UserType` very often, it may be useful to define a shorter name for it. You can do this using the `<typedef>` element. Typedefs assign a name to a custom type, and may also contain a list of default parameter values if the type is parameterized.

```
<typedef class="MyCompany.UserTypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

It is also possible to override the parameters supplied in a typedef on a case-by-case basis by using type parameters on the property mapping.

Even though NHibernate's rich range of built-in types and support for components means you will very rarely *need* to use a custom type, it is nevertheless considered good form to use custom types for (non-entity) classes that occur frequently in your application. For example, a `MonetaryAmount` class is a good candidate for an `ICompositeUserType`, even though it could easily be mapped as a component. One motivation for this is abstraction. With a custom type, your mapping documents would be future-proofed against possible changes in your way of representing monetary values.

5.2.4. Any type mappings

There is one further type of property mapping. The `<any>` mapping element defines a polymorphic association to classes from multiple tables. This type of mapping always requires more than one column. The first column holds the type of the associated entity. The remaining columns hold the identifier. It is impossible to specify a foreign key constraint for this kind of association, so this is most certainly not meant as the usual way of mapping (polymorphic) associations. You should use this only in very special cases (eg. audit logs, user session data, etc).

```
<any name="AnyEntity" id-type="Int64" meta-type="Eg.Custom.Class2TablenameType">
    <column name="table_name"/>
    <column name="id"/>
</any>
```

The `meta-type` attribute lets the application specify a custom type that maps database column values to persistent classes which have identifier properties of the type specified by `id-type`. If the meta-type returns instances of `System.Type`, nothing else is required. On the other hand, if it is a basic type like `String` or `Char`, you must specify the mapping from values to classes.

```
<any name="AnyEntity" id-type="Int64" meta-type="String">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="PropertyName"                               (1)
        id-type="idtypename"                              (2)
        meta-type="metatypename"                          (3)
        cascade="none|all|save-update"                    (4)
        access="field|property|nosetter|ClassName"        (5)
        optimistic-lock="true|false"                      (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name`: the property name.

(2) `id-type`: the identifier type.

(3) `meta-type` (optional - defaults to `Type`): a type that maps `System.Type` to a single database column or, alternatively, a type that is allowed for a discriminator mapping.

(4) `cascade` (optional - defaults to `none`): the cascade style.

(5) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

(6) `optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, define if a version increment should occur if this property is dirty.

5.3. SQL quoted identifiers

You may force NHibernate to quote an identifier in the generated SQL by enclosing the table or column name in backticks in the mapping document. NHibernate will use the correct quotation style for the SQL `Dialect` (usually double quotes, but brackets for SQL Server and backticks for MySQL).

```
<class name="LineItem" table="`Line Item`">
    <id name="Id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="ItemNumber" column="`Item #`"/>
    ...
</class>
```

5.4. Modular mapping files

It is possible to define `subclass` and `joined-subclass` mappings in saparate mapping documents, directly beneath `hibernate-mapping`. This allows you to extend a class hierachy just by adding a new mapping file. You must specify an `extends` attribute in the subclass mapping, naming a previously mapped superclass. Use of this feature makes the ordering of the mapping documents important!

```
<hibernate-mapping>
        <subclass name="Eg.Subclass.DomesticCat, Eg"
            extends="Eg.Cat, Eg" discriminator-value="D">
             <property name="name" type="string"/>
        </subclass>
</hibernate-mapping>
```

5.5. Generated Properties

Generated properties are properties which have their values generated by the database. Typically, NHibernate applications needed to `Refresh` objects which contain any properties for which the database was generating values. Marking properties as generated, however, lets the application delegate this responsibility to NHibernate. Essentially, whenever NHibernate issues an SQL INSERT or UPDATE for an entity which has defined generated properties, it immediately issues a select afterwards to retrieve the generated values.

Properties marked as generated must additionally be non-insertable and non-updateable. Only Section 5.1.8, “version (optional)”, Section 5.1.9, “timestamp (optional)”, and Section 5.1.10, “property” can be marked as generated.

`never` (the default) - means that the given property value is not generated within the database.

`insert` - states that the given property value is generated on insert, but is not regenerated on subsequent updates. Things like created-date would fall into this category. Note that even though Section 5.1.8, “version (optional)” and Section 5.1.9, “timestamp (optional)” properties can be marked as generated, this option is not available there...

`always` - states that the property value is generated both on insert and on update.

5.6. Auxiliary Database Objects

Allows CREATE and DROP of arbitrary database objects, in conjunction with NHibernate's schema evolution tools, to provide the ability to fully define a user schema within the NHibernate mapping files. Although designed specifically for creating and dropping things like triggers or stored procedures, really any SQL command that can be run via a `IDbCommand.ExecuteNonQuery()` method is valid here (ALTERs, INSERTS, etc). There are essentially two modes for defining auxiliary database objects.

The first mode is to explicitly list the CREATE and DROP commands out in the mapping file:

```
<nhibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</nhibernate-mapping>
```

The second mode is to supply a custom class which knows how to construct the CREATE and DROP commands. This custom class must implement the `NHibernate.Mapping.IAuxiliaryDatabaseObject` interface.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition, MyAssembly"/>
    </database-object>
</hibernate-mapping>
```

You may also specify parameters to be passed to the database object:

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition, MyAssembly">
            <param name="parameterName">parameterValue</param>
        </definition>
    </database-object>
</hibernate-mapping>
```

NHibernate will call `IAuxiliaryDatabaseObject.SetParameterValues` passing it a dictionary of parameter names and values.

Additionally, these database objects can be optionally scoped such that they only apply when certain dialects are used.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="NHibernate.Dialect.Oracle9iDialect"/>
        <dialect-scope name="NHibernate.Dialect.Oracle8iDialect"/>
    </database-object>
</hibernate-mapping>
```

Chapter 6. Collection Mapping

6.1. Persistent Collections

NHibernate requires that persistent collection-valued fields be declared as a generic interface type, for example:

```
public class Product
{
    private string serialNumber;
    private ISet<Part> parts = new HashSet<Part>();

    public ISet<Part> Parts
    {
        get { return parts; }
        set { parts = value; }
    }

    public string SerialNumber
    {
        get { return serialNumber; }
        set { serialNumber = value; }
    }
}
```

The actual interface might be `System.Collections.Generic.ICollection<T>`, `System.Collections.Generic.IList<T>`, `System.Collections.Generic.IDictionary<K, V>`, `System.Collections.Generic.ISet<T>` or ... anything you like! (Where "anything you like" means you will have to write an implementation of `NHibernate.UserType.IUserCollectionType`.)

Notice how we initialized the instance variable with an instance of `HashSet<Part>`. This is the best way to initialize collection valued properties of newly instantiated (non-persistent) instances. When you make the instance persistent - by calling `Save()`, for example - NHibernate will actually replace the `HashSet<Part>` with an instance of NHibernate's own implementation of `ISet<Part>`. Watch out for errors like this:

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
ISet<Cat> kittens = new HashSet<Cat>();
kittens.Add(kitten);
cat.Kittens = kittens;
session.Save(cat);
kittens = cat.Kittens; //Okay, kittens collection is an ISet<Cat>
HashSet<Cat> hs = (HashSet<Cat>) cat.Kittens; //Error!
```

Collection instances have the usual behavior of value types. They are automatically persisted when referenced by a persistent object and automatically deleted when unreferenced. If a collection is passed from one persistent object to another, its elements might be moved from one table to another. Two entities may not share a reference to the same collection instance. Due to the underlying relational model, collection-valued properties do not support null value semantics; NHibernate does not distinguish between a null collection reference and an empty collection.

You shouldn't have to worry much about any of this. Just use NHibernate's collections the same way you use ordinary .NET collections, but make sure you understand the semantics of bidirectional associations (discussed later) before using them.

Collection instances are distinguished in the database by a foreign key to the owning entity. This foreign key is referred to as the *collection key* . The collection key is mapped by the `<key>` element.

Collections may contain almost any other NHibernate type, including all basic types, custom types, entity types and components. This is an important definition: An object in a collection can either be handled with "pass by value" semantics (it therefore fully depends on the collection owner) or it can be a reference to another entity with an own lifecycle. Collections may not contain other collections. The contained type is referred to as the *collection element type*. Collection elements are mapped by `<element>`, `<composite-element>`, `<one-to-many>`, `<many-to-many>` or `<many-to-any>`. The first two map elements with value semantics, the other three are used to map entity associations.

All collection types except set and bag have an *index* column - a column that maps to an array or `IList<T>` index or `IDictionary<TKey, TValue>` key. The index of an `IDictionary<TKey, TValue>` may be of any basic type, an entity type or even a composite type (it may not be a collection). The index of an array or list is always of type `Int32`. Indexes are mapped using `<index>`, `<index-many-to-many>`, `<composite-index>` or `<index-many-to-any>`.

There are quite a range of mappings that can be generated for collections, covering many common relational models. We suggest you experiment with the schema generation tool to get a feeling for how various mapping declarations translate to database tables.

6.2. Mapping a Collection

Collections are declared by the `<set>`, `<list>`, `<map>`, `<bag>`, `<array>` and `<primitive-array>` elements. `<map>` is representative:

```
<map
    name="propertyName"                                         (1)
    table="table_name"                                          (2)
    schema="schema_name"                                        (3)
    lazy="true|false|extra"                                     (4)
    inverse="true|false"                                        (5)
    cascade="all|none|save-update|delete|all-delete-orphan"     (6)
    sort="unsorted|natural|comparatorClass"                     (7)
    order-by="column_name asc|desc"                             (8)
    where="arbitrary sql where condition"                       (9)
    fetch="select|join"                                         (10)
    batch-size="N"                                              (11)
    access="field|property|ClassName"                           (12)
    optimistic-lock="true|false"                                (13)
    generic="true|false"                                        (14)
>

    <key .... />
    <index .... />
    <element .... />
</map>
```

(1) `name` the collection property name

(2) `table` (optional - defaults to property name) the name of the collection table (not used for one-to-many associations)

(3) `schema` (optional) the name of a table schema to override the schema declared on the root element

(4) `lazy` (optional - defaults to `true`) may be used to disable lazy fetching and specify that the association is always eagerly fetched. Using `extra` fetches only the elements that are needed - see Section 19.1, “Fetching strategies” for more information.

(5) `inverse` (optional - defaults to `false`) mark this collection as the "inverse" end of a bidirectional association

(6) `cascade` (optional - defaults to `none`) enable operations to cascade to child entities

(7) `sort` (optional) specify a sorted collection with `natural` sort order, or a given comparator class

(8) `order-by` (optional) specify a table column (or columns) that define the iteration order of the `IDictionary<TKey, TValue>`, `ISet<T>` or bag, together with an optional `asc` or `desc`

(9) `where` (optional) specify an arbitrary SQL `WHERE` condition to be used when retrieving or removing the collection (useful if the collection should contain only a subset of the available data)

(10) `fetch` (optional) Choose between outer-join fetching and fetching by sequential select.

(11) `batch-size` (optional, defaults to `1`) specify a "batch size" for lazily fetching instances of this collection.

(12) `access` (optional - defaults to `property`): The strategy NHibernate should use for accessing the property value.

(13) `optimistic-lock` (optional - defaults to `true`): Species that changes to the state of the collection results in increment of the owning entity's version. (For one to many associations, it is often reasonable to disable this setting.)

(14) `generic` (optional, obsolete): Choose between generic and non-generic collection interfaces, currently NHibernate only supports generic collections.

The mapping of an `IList<T>` or array requires a saparate table column holding the array or list index (the `i` in `foo[i]`). If your relational model doesn't have an index column, e.g. if you're working with legacy data, use an unordered `ISet<T>` instead. This seems to put people off who assume that `IList<T>` should just be a more convenient way of accessing an unordered collection. NHibernate collections strictly obey the actual semantics attached to the `ISet<T>`, `IList<T>` and `IDictionary<TKey, TValue>` interfaces. `IList<T>` elements don't just spontaneously rearrange themselves!

On the other hand, people who planned to use the `IList<T>` to emulate *bag* semantics have a legitimate grievance here. A bag is an unordered, unindexed collection which may contain the same element multiple times. The .NET collections framework lacks an `IBag<T>` interface, hence you have to emulate it with an `IList<T>`. NHibernate lets you map properties of type `IList<T>` or `ICollection<T>` with the `<bag>` element. Note that bag semantics are not really part of the `ICollection<T>` contract and they actually conflict with the semantics of the `IList<T>` contract (however, you can sort the bag arbitrarily, discussed later in this chapter).

Note: Large NHibernate bags mapped with `inverse="false"` are inefficient and should be avoided; NHibernate can't create, delete or update rows individually, because there is no key that may be used to identify an individual row.

6.3. Collections of Values and Many-To-Many Associations

A collection table is required for any collection of values and any collection of references to other entities mapped as a many-to-many association (the natural semantics for a .NET collection). The table requires (foreign) key column(s), element column(s) and possibly index column(s).

The foreign key from the collection table to the table of the owning class is declared using a `<key>` element.

```
<key column="column_name"/>
```

(1) `column` (required): The name of the foreign key column.

For indexed collections like maps and lists, we require an `<index>` element. For lists, this column contains sequential integers numbered from zero. Make sure that your index really starts from zero if you have to deal with legacy data. For maps, the column may contain any values of any NHibernate type.

```
<index
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) `column` (required): The name of the column holding the collection index values.

(2) `type` (optional, defaults to `Int32`): The type of the collection index.

Alternatively, a map may be indexed by objects of entity type. We use the `<index-many-to-many>` element.

```
<index-many-to-many
        column="column_name"                (1)
        class="ClassName"                   (2)
/>
```

(1) `column` (required): The name of the foreign key column for the collection index values.

(2) `class` (required): The entity class used as the collection index.

For a collection of values, we use the `<element>` tag.

```
<element
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) `column` (required): The name of the column holding the collection element values.

(2) `type` (required): The type of the collection element.

A collection of entities with its own table corresponds to the relational notion of *many-to-many association*. A many to many association is the most natural mapping of a .NET collection but is not usually the best relational model.

```
<many-to-many
        column="column_name"                               (1)
        class="ClassName"                                  (2)
        fetch="join|select"                                (3)
        not-found="ignore|exception"                       (4)
    />
```

(1) `column` (required): The name of the element foreign key column.

(2) `class` (required): The name of the associated class.

(3) `fetch` (optional, defaults to `join`): enables outer-join or sequential select fetching for this association. This is a special case; for full eager fetching (in a single SELECT) of an entity and its many-to-many relationships to other entities, you would enable join fetching not only of the collection itself, but also with this attribute on the `<many-to-many>` nested element.

(4) `not-found` (optional - defaults to `exception`): Specifies how foreign keys that reference missing rows will be handled: `ignore` will treat a missing row as a null association.

Some examples, first, a set of strings:

```
<set name="Names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="String"/>
</set>
```

A bag containing integers (with an iteration order determined by the `order-by` attribute):

```
<bag name="Sizes" table="SIZES" order-by="SIZE ASC">
    <key column="OWNER"/>
    <element column="SIZE" type="Int32"/>
</bag>
```

An array of entities - in this case, a many to many association (note that the entities are lifecycle objects, `cascade="all"`):

```
<array name="Foos" table="BAR_FOOS" cascade="all">
    <key column="BAR_ID"/>
    <index column="I"/>
    <many-to-many column="FOO_ID" class="Eg.Foo, Eg"/>
</array>
```

A map from string indices to dates:

```
<map name="Holidays" table="holidays" schema="dbo" order-by="hol_name asc">
    <key column="id"/>
    <index column="hol_name" type="String"/>
    <element column="hol_date" type="Date"/>
</map>
```

A list of components (discussed in the next chapter):

```
<list name="CarComponents" table="car_components">
    <key column="car_id"/>
    <index column="posn"/>
    <composite-element class="Eg.Car.CarComponent">
            <property name="Price" type="float"/>
            <property name="Type" type="Eg.Car.ComponentType, Eg"/>
            <property name="SerialNumber" column="serial_no" type="String"/>
    </composite-element>
</list>
```

6.4. One-To-Many Associations

A *one to many association* links the tables of two classes *directly*, with no intervening collection table. (This implements a *one-to-many* relational model.) This relational model loses some of the semantics of .NET collections:

- No null values may be contained in a dictionary, set or list
- An instance of the contained entity class may not belong to more than one instance of the collection
- An instance of the contained entity class may not appear at more than one value of the collection index

An association from `Foo` to `Bar` requires the addition of a key column and possibly an index column to the table of the contained entity class, `Bar`. These columns are mapped using the `<key>` and `<index>` elements described above.

The `<one-to-many>` tag indicates a one to many association.

```
<one-to-many
        class="ClassName"                              (1)
        not-found="ignore|exception"                   (2)
    />
```

(1) `class` (required): The name of the associated class.

(2) `not-found` (optional - defaults to `exception`): Specifies how foreign keys that reference missing rows will be handled: `ignore` will treat a missing row as a null association.

Example:

```
<set name="Bars">
    <key column="foo_id"/>
    <one-to-many class="Eg.Bar, Eg"/>
</set>
```

Notice that the `<one-to-many>` element does not need to declare any columns. Nor is it necessary to specify the `table` name anywhere.

Very Important Note: If the `<key>` column of a `<one-to-many>` association is declared `NOT NULL`, NHibernate may cause constraint violations when it creates or updates the association. To prevent this problem, *you must use a bidirectional association* with the many valued end (the set or bag) marked as `inverse="true"`. See the discussion of bidirectional associations later in this chapter.

6.5. Lazy Initialization

Collections (other than arrays) may be lazily initialized, meaning they load their state from the database only when the application needs to access it. Initialization happens transparently to the user so the application would not normally need to worry about this (in fact, transparent lazy initialization is the main reason why NHibernate needs its own collection implementations). However, if the application tries something like this:

```
s = sessions.OpenSession();
ITransaction tx = sessions.BeginTransaction();
User u = s.CreateQuery("from User u where u.Name=:name").SetString("name", userName).UniqueResult<User>();
IDictionary permissions = u.Permissions;
tx.Commit();
s.Close();

int accessLevel = (int) permissions["accounts"];  // Error!
```

It could be in for a nasty surprise. Since the permissions collection was not initialized when the `ISession` was committed, the collection will never be able to load its state. The fix is to move the line that reads from the collection to just before the commit. (There are other more advanced ways to solve this problem, however.)

Alternatively, use a non-lazy collection. Since lazy initialization can lead to bugs like that above, non-laziness is the default. However, it is intended that lazy initialization be used for almost all collections, especially for collections of entities (for reasons of efficiency).

Exceptions that occur while lazily initializing a collection are wrapped in a `LazyInitializationException`.

Declare a lazy collection using the optional `lazy` attribute:

```
<set name="Names" table="NAMES" lazy="true">
    <key column="group_id"/>
    <element column="NAME" type="String"/>
</set>
```

In some application architectures, particularly where the code that accesses data using NHibernate, and the code that uses it are in different application layers, it can be a problem to ensure that the `ISession` is open when a collection is initialized. There are two basic ways to deal with this issue:

- In a web-based application, an event handler can be used to close the `ISession` only at the very end of a user request, once the rendering of the view is complete. Of course, this places heavy demands upon the correctness of the exception handling of your application infrastructure. It is vitally important that the `ISession` is closed and the transaction ended before returning to the user, even when an exception occurs during rendering of the view. The event handler has to be able to access the `ISession` for this approach. We recommend that the current `ISession` is stored in the `HttpContext.Items` collection (see chapter 1, Section 1.4, “Playing with cats”, for an example implementation).

- In an application with a saparate business tier, the business logic must "prepare" all collections that will be needed by the web tier before returning. This means that the business tier should load all the data and return all the data already initialized to the presentation/web tier that is required for a particular use case. Usually, the application calls `NHibernateUtil.Initialize()` for each collection that will be needed in the web tier (this call must occur before the session is closed) or retrieves the collection eagerly using a NHibernate query with a `FETCH` clause.

- You may also attach a previously loaded object to a new `ISession` with `Update()` or `Lock()` before accessing unitialized collections (or other proxies). NHibernate can not do this automatically, as it would introduce ad hoc transaction semantics!

You can use the `CreateFilter()` method of the NHibernate ISession API to get the size of a collection without initializing it:

```
ICollection countColl = s.CreateFilter( collection, "select count(*)" ).List();
IEnumerator countEn = countColl.GetEnumerator();
countEn.MoveNext();
int count = (int) countEn.Current;
```

`CreateFilter()` is also used to efficiently retrieve subsets of a collection without needing to initialize the whole collection.

6.6. Sorted Collections

NHibernate supports collections implemented by `System.Collections.SortedList` and `System.Collections.Generic.SortedSet<T>`. You must specify a comparer in the mapping file:

```
<set name="Aliases" table="person_aliases" sort="natural">
    <key column="person"/>
    <element column="name" type="String"/>
</set>

<map name="Holidays" sort="My.Custom.HolidayComparer, MyAssembly" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="String"/>
    <element column="hol_date" type="Date"/>
</map>
```

Allowed values of the `sort` attribute are `unsorted`, `natural` and the name of a class implementing `System.Collections.IComparer`.

If you want the database itself to order the collection elements use the `order-by` attribute of `set`, `bag` or `map` mappings. This performs the ordering in the SQL query, not in memory.

Setting the `order-by` attribute tells NHibernate to use `ListDictionary` or `ListSet` class internally for dictionaries and sets, maintaining the order of the elements. *Note that lookup operations on these collections are very slow if they contain more than a few elements.*

```
<set name="Aliases" table="person_aliases" order-by="name asc">
    <key column="person"/>
    <element column="name" type="String"/>
</set>

<map name="Holidays" order-by="hol_date, hol_name" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="String"/>
    <element column="hol_date type="Date"/>
</map>
```

Note that the value of the `order-by` attribute is an SQL ordering, not a HQL ordering!

Associations may even be sorted by some arbitrary criteria at runtime using a `Filter()`.

```
sortedUsers = s.Filter( group.Users, "order by this.Name" );
```

6.7. Using an `<idbag>`

If you've fully embraced our view that composite keys are a bad thing and that entities should have synthetic identifiers (surrogate keys), then you might find it a bit odd that the many to many associations and collections of values that we've shown so far all map to tables with composite keys! Now, this point is quite arguable; a pure association table doesn't seem to benefit much from a surrogate key (though a collection of composite values *might*). Nevertheless, NHibernate provides a feature that allows you to map many to many associations and collections of values to a table with a surrogate key.

The `<idbag>` element lets you map a `List` (or `Collection`) with bag semantics.

```
<idbag name="Lovers" table="LOVERS" lazy="true">
    <collection-id column="ID" type="Int64">
        <generator class="hilo"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Eg.Person" fetch="join"/>
</idbag>
```

As you can see, an `<idbag>` has a synthetic id generator, just like an entity class! A different surrogate key is assigned to each collection row. NHibernate does not provide any mechanism to discover the surrogate key value of a particular row, however.

Note that the update performance of an `<idbag>` is *much* better than a regular `<bag>`! NHibernate can locate individual rows efficiently and update or delete them individually, just like a list, map or set.

As of version 2.0, the `native` identifier generation strategy is supported for `<idbag>` collection identifiers.

6.8. Bidirectional Associations

A *bidirectional association* allows navigation from both "ends" of the association. Two kinds of bidirectional association are supported:

one-to-many

set or bag valued at one end, single-valued at the other

many-to-many

set or bag valued at both ends

Please note that NHibernate does not support bidirectional one-to-many associations with an indexed collection (list, map or array) as the "many" end, you have to use a set or bag mapping.

You may specify a bidirectional many-to-many association simply by mapping two many-to-many associations to the same database table and declaring one end as *inverse* (which one is your choice). Here's an example of a bidirectional many-to-many association from a class back to *itself* (each category can have many items and each item can be in many categories):

```
<class name="NHibernate.Auction.Category, NHibernate.Auction">
    <id name="Id" column="ID"/>
    ...
    <bag name="Items" table="CATEGORY_ITEM" lazy="true">
        <key column="CATEGORY_ID"/>
        <many-to-many class="NHibernate.Auction.Item, NHibernate.Auction" column="ITEM_ID"/>
    </bag>
</class>

<class name="NHibernate.Auction.Item, NHibernate.Auction">
    <id name="id" column="ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true" lazy="true">
        <key column="ITEM_ID"/>
        <many-to-many class="NHibernate.Auction.Category, NHibernate.Auction" column="CATEGORY_ID"/>
    </bag>
</class>
```

Changes made only to the inverse end of the association are *not* persisted. This means that NHibernate has two representations in memory for every bidirectional association, one link from A to B and another link from B to A. This is easier to understand if you think about the .NET object model and how we create a many-to-many relationship in C#:

```
category.Items.Add(item);          // The category now "knows" about the relationship
item.Categories.Add(category);     // The item now "knows" about the relationship

session.Update(item);                     // No effect, nothing will be saved!
session.Update(category);                 // The relationship will be saved
```

The non-inverse side is used to save the in-memory representation to the database. We would get an unneccessary INSERT/UPDATE and probably even a foreign key violation if both would trigger changes! The same is of course also true for bidirectional one-to-many associations.

You may map a bidirectional one-to-many association by mapping a one-to-many association to the same table column(s) as a many-to-one association and declaring the many-valued end `inverse="true"`.

```
<class name="Eg.Parent, Eg">
    <id name="Id" column="id"/>
    ....
    <set name="Children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="Eg.Child, Eg"/>
    </set>
</class>

<class name="Eg.Child, Eg">
    <id name="Id" column="id"/>
    ....
    <many-to-one name="Parent" class="Eg.Parent, Eg" column="parent_id"/>
</class>
```

Mapping one end of an association with `inverse="true"` doesn't affect the operation of cascades, both are different concepts!

6.9. Ternary Associations

There are two possible approaches to mapping a ternary association. One approach is to use composite elements (discussed below). Another is to use an `IDictionary` with an association as its index:

```
<map name="Contracts" lazy="true">
    <key column="employer_id"/>
    <index-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="Connections" lazy="true">
    <key column="node1_id"/>
    <index-many-to-many column="node2_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

6.10. Heterogeneous Associations

The `<many-to-any>` and `<index-many-to-any>` elements provide for true heterogeneous associations. These mapping elements work in the same way as the `<any>` element - and should also be used rarely, if ever.

6.11. Collection examples

The previous sections are pretty confusing. So lets look at an example. This class:

```
using System;
using System.Collections.Generic;

namespace Eg

    public class Parent
    {
        private long id;
        private ISet<Child> children;

        public long Id
        {
            get { return id; }
            set { id = value; }
        }

        private ISet<Child> Children
        {
            get { return children; }
            set { children = value; }
        }

        ....
        ....
    }
}
```

has a collection of `Eg.Child` instances. If each child has at most one parent, the most natural mapping is a one-to-many association:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="Eg" namespace="Eg">

    <class name="Parent">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <set name="Children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <property name="Name"/>
    </class>

</hibernate-mapping>
```

This maps to the following table definitions:

```
create table parent ( Id bigint not null primary key )
create table child ( Id bigint not null primary key, Name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

If the parent is *required*, use a bidirectional one-to-many association:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="Eg" namespace="Eg">

    <class name="Parent">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <set name="Children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <property name="Name"/>
        <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

Notice the `NOT NULL` constraint:

```
create table parent ( Id bigint not null primary key )
create table child ( Id bigint not null
                     primary key,
                     Name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

On the other hand, if a child might have multiple parents, a many-to-many association is appropriate:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="Eg" namespace="Eg">

    <class name="Parent">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <set name="Children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="Id">
            <generator class="sequence"/>
        </id>
        <property name="Name"/>
    </class>

</hibernate-mapping>
```

Table definitions:

```
create table parent ( Id bigint not null primary key )
create table child ( Id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

Chapter 7. Component Mapping

The notion of a *component* is re-used in several different contexts, for different purposes, throughout NHibernate.

7.1. Dependent objects

A component is a contained object that is persisted as a value type, not an entity. The term "component" refers to the object-oriented notion of composition (not to architecture-level components). For example, you might model a person like this:

```
public class Person
{
    private DateTime birthday;
    private Name name;
    private string key;

    public string Key
    {
        get { return key; }
        set { key = value; }
    }
    public DateTime Birthday
    {
        get { return birthday; }
        set { birthday = value; }
    }

    public Name Name
    {
        get { return name; }
        set { name = value; }
    }
    ......
    ......
}
```

```
public class Name
{
    char initial;
    string first;
    string last;

    public string First
    {
        get { return first; }
        set { first = value; }
    }

    public string Last
    {
        get { return last; }
        set { last = value; }
    }

    public char Initial
    {
        get { return initial; }
        set { initial = value; }
    }
}
```

Now `Name` may be persisted as a component of `Person`. Notice that `Name` defines getter and setter methods for its persistent properties, but doesn't need to declare any interfaces or identifier properties.

Our NHibernate mapping would look like:

```
<class name="Eg.Person, Eg" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="Birthday" type="date"/>
    <component name="Name" class="Eg.Name, Eg"> <!-- class attribute optional -->
        <property name="Initial"/>
        <property name="First"/>
        <property name="Last"/>
    </component>
</class>
```

The person table would have the columns `pid`, `Birthday`, `Initial`, `First` and `Last`.

Like all value types, components do not support shared references. The null value semantics of a component are *ad hoc*. When reloading the containing object, NHibernate will assume that if all component columns are null, then the entire component is null. This should be okay for most purposes.

The properties of a component may be of any NHibernate type (collections, many-to-one associations, other components, etc). Nested components should *not* be considered an exotic usage. NHibernate is intended to support a very fine-grained object model.

The `<component>` element allows a `<parent>` subelement that maps a property of the component class as a reference back to the containing entity.

```
<class name="Eg.Person, Eg" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="Birthday" type="date"/>
    <component name="Name" class="Eg.Name, Eg">
        <parent name="NamedPerson"/> <!-- reference back to the Person -->
        <property name="Initial"/>
        <property name="First"/>
        <property name="Last"/>
    </component>
```

```
</class>
```

7.2. Collections of dependent objects

Collections of components are supported (eg. an array of type `Name`). Declare your component collection by replacing the `<element>` tag with a `<composite-element>` tag.

```
<set name="SomeNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="Eg.Name, Eg"> <!-- class attribute required -->
        <property name="Initial"/>
        <property name="First"/>
        <property name="Last"/>
    </composite-element>
</set>
```

Note: if you define an `ISet<T>` of composite elements, it is very important to implement `Equals()` and `GetHashCode()` correctly.

Composite elements may contain components but not collections. If your composite element itself contains components, use the `<nested-composite-element>` tag. This is a pretty exotic case - a collection of components which themselves have components. By this stage you should be asking yourself if a one-to-many association is more appropriate. Try remodelling the composite element as an entity - but note that even though the object model is the same, the relational model and persistence semantics are still slightly different.

Please note that a composite element mapping doesn't support null-able properties if you're using a `<set>`. NHibernate has to use each columns value to identify a record when deleting objects (there is no separate primary key column in the composite element table), which is not possible with null values. You have to either use only not-null properties in a composite-element or choose a `<list>`, `<map>`, `<bag>` or `<idbag>`.

A special case of a composite element is a composite element with a nested `<many-to-one>` element. A mapping like this allows you to map extra columns of a many-to-many association table to the composite element class. The following is a many-to-many association from `Order` to `Item` where `PurchaseDate`, `Price` and `Quantity` are properties of the association:

```
<class name="Order" .... >
    ....
    <set name="PurchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="Purchase">
            <property name="PurchaseDate"/>
            <property name="Price"/>
            <property name="Quantity"/>
            <many-to-one name="Item" class="Item"/> <!-- class attribute is optional -->
        </composite-element>
    </set>
</class>
```

Even ternary (or quaternary, etc) associations are possible:

```
<class name="Order" .... >
    ....
    <set name="PurchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="OrderLine">
            <many-to-one name="PurchaseDetails class="Purchase"/>
            <many-to-one name="Item" class="Item"/>
        </composite-element>
    </set>
</class>
```

Composite elements may appear in queries using the same syntax as associations to other entities.

7.3. Components as IDictionary indices

The `<composite-index>` element lets you map a component class as the key of an `IDictionary`. Make sure you override `GetHashCode()` and `Equals()` correctly on the component class.

7.4. Components as composite identifiers

You may use a component as an identifier of an entity class. Your component class must satisfy certain requirements:

- It must be `Serializable`.
- It must re-implement `Equals()` and `GetHashCode()`, consistently with the database's notion of composite key equality.

You can't use an `IIdentifierGenerator` to generate composite keys. Instead the application must assign its own identifiers.

Since a composite identifier must be assigned to the object before saving it, we can't use `unsaved-value` of the identifier to distinguish between newly instantiated instances and instances saved in a previous session.

You may instead implement `IInterceptor.IsTransient()` if you wish to use `SaveOrUpdate()` or cascading save / update. As an alternative, you may also set the `unsaved-value` attribute on a `<version>` (or `<timestamp>`) element to specify a value that indicates a new transient instance. In this case, the version of the entity is used instead of the (assigned) identifier and you don't have to implement `IInterceptor.IsTransient()` yourself.

Use the `<composite-id>` tag (same attributes and elements as `<component>`) in place of `<id>` for the declaration of a composite identifier class:

```
<class name="Foo" table="FOOS">
    <composite-id name="CompId" class="FooCompositeID">
        <key-property name="String"/>
        <key-property name="Short"/>
        <key-property name="Date" column="date_" type="Date"/>
    </composite-id>
    <property name="Name"/>
    ....
</class>
```

Now, any foreign keys into the table `FOOS` are also composite. You must declare this in your mappings for other classes. An association to `Foo` would be declared like this:

```
<many-to-one name="Foo" class="Foo">
<!-- the "class" attribute is optional, as usual -->
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-one>
```

This new `<column>` tag is also used by multi-column custom types. Actually it is an alternative to the `column` attribute everywhere. A collection with elements of type `Foo` would use:

```
<set name="Foos">
    <key column="owner_id"/>
    <many-to-many class="Foo">
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </many-to-many>
</set>
```

On the other hand, `<one-to-many>`, as usual, declares no columns.

If `Foo` itself contains collections, they will also need a composite foreign key.

```
<class name="Foo">
    ....
    ....
    <set name="Dates" lazy="true">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="foo_string"/>
            <column name="foo_short"/>
            <column name="foo_date"/>
        </key>
        <element column="foo_date" type="Date"/>
    </set>
</class>
```

7.5. Dynamic components

You may even map a property of type `IDictionary`:

```
<dynamic-component name="UserAttributes">
    <property name="Foo" column="FOO"/>
    <property name="Bar" column="BAR"/>
    <many-to-one name="Baz" class="Baz" column="BAZ"/>
</dynamic-component>
```

The semantics of a `<dynamic-component>` mapping are identical to `<component>`. The advantage of this kind of mapping is the ability to determine the actual properties of the component at deployment time, just by editing the mapping document. (Runtime manipulation of the mapping document is also possible, using a DOM parser.)

Chapter 8. Inheritance Mapping

8.1. The Three Strategies

NHibernate supports the three basic inheritance mapping strategies.

- table per class hierarchy
- table per subclass
- table per concrete class

In addition, NHibernate supports a fourth, slightly different kind of polymorphism:

- implicit polymorphism

It is possible to use different mapping strategies for different branches of the same inheritance hierarchy, and then make use of implicit polymorphism to achieve polymorphism across the whole hierarchy. However, NHibernate does not support mixing `<subclass>`, and `<joined-subclass>` and `<union-subclass>` mappings under the same root `<class>` element. It is possible to mix together the table per hierarchy and table per subclass strategies, under the the same `<class>` element, by combining the `<subclass>` and `<join>` elements (see below).

It is possible to define `subclass`, `union-subclass`, and `joined-subclass` mappings in separate mapping documents, directly beneath `hibernate-mapping`. This allows you to extend a class hierachy just by adding a new mapping file. You must specify an `extends` attribute in the subclass mapping, naming a previously mapped superclass.

```
 <hibernate-mapping>
     <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
          <property name="name" type="string"/>
     </subclass>
 </hibernate-mapping>
```

8.1.1. Table per class hierarchy

Suppose we have an interface `IPayment`, with implementors `CreditCardPayment`, `CashPayment`, `ChequePayment`. The table-per-hierarchy mapping would look like:

```
<class name="IPayment" table="PAYMENT">
    <id name="Id" type="Int64" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="String"/>
    <property name="Amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Exactly one table is required. There is one big limitation of this mapping strategy: columns declared by the subclasses may not have `NOT NULL` constraints.

8.1.2. Table per subclass

A table-per-subclass mapping would look like:

```
<class name="IPayment" table="PAYMENT">
    <id name="Id" type="Int64" column="PAYMENT_ID">
```

```
        <generator class="native"/>
    </id>
    <property name="Amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

Four tables are required. The three subclass tables have primary key associations to the superclass table (so the relational model is actually a one-to-one association).

8.1.3. Table per subclass, using a discriminator

Note that NHibernate's implementation of table-per-subclass requires no discriminator column. Other object/relational mappers use a different implementation of table-per-subclass which requires a type discriminator column in the superclass table. The approach taken by NHibernate is much more difficult to implement but arguably more correct from a relational point of view. If you would like to use a discriminator column with the table per subclass strategy, you may combine the use of `<subclass>` and `<join>`, as follow:

```
<class name="Payment" table="PAYMENT">
    <id name="Id" type="Int64" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="Amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="CreditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

The optional `fetch="select"` declaration tells NHibernate not to fetch the `ChequePayment` subclass data using an outer join when querying the superclass.

8.1.4. Mixing table per class hierarchy with table per subclass

You may even mix the table per hierarchy and table per subclass strategies using this approach:

```
<class name="Payment" table="PAYMENT">
    <id name="Id" type="Int64" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="Amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="CreditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

For any of these mapping strategies, a polymorphic association to `IPayment` is mapped using `<many-to-one>`.

```
<many-to-one name="Payment" column="PAYMENT" class="IPayment"/>
```

8.1.5. Table per concrete class

There are two ways we could go about mapping the table per concrete class strategy. The first is to use `<union-subclass>`.

```
<class name="Payment">
    <id name="Id" type="Int64" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="Amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="CreditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

Three tables are involved for the subclasses. Each table defines columns for all properties of the class, including inherited properties.

The limitation of this approach is that if a property is mapped on the superclass, the column name must be the same on all subclass tables. (We might relax this in a future release of NHibernate.) The identity generator strategy is not allowed in union subclass inheritance, indeed the primary key seed has to be shared accross all unioned subclasses of a hierarchy.

If your superclass is abstract, map it with `abstract="true"`. Of course, if it is not abstract, an additional table (defaults to `PAYMENT` in the example above) is needed to hold instances of the superclass.

8.1.6. Table per concrete class, using implicit polymorphism

An alternative approach is to make use of implicit polymorphism:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="Id" type="Int64" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="Amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="Id" type="Int64" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="Amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="Id" type="Int64" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="Amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

Notice that nowhere do we mention the `IPayment` interface explicitly. Also notice that properties of `IPayment` are mapped in each of the subclasses. If you want to avoid duplication, consider using XML entities (e.g. `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` in the `DOCTYPE` declartion and `&allproperties;` in the mapping).

The disadvantage of this approach is that NHibernate does not generate SQL `UNION`s when performing polymorphic queries.

For this mapping strategy, a polymorphic association to `IPayment` is usually mapped using `<any>`.

```
<any name="Payment" meta-type="string" id-type="Int64">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

8.1.7. Mixing implicit polymorphism with other inheritance mappings

There is one further thing to notice about this mapping. Since the subclasses are each mapped in their own `<class>` element (and since `IPayment` is just an interface), each of the subclasses could easily be part of another table-per-class or table-per-subclass inheritance hierarchy! (And you can still use polymorphic queries against the `IPayment` interface.)

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="Id" type="Int64" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="String"/>
    <property name="Amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="Id" type="Int64" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="Amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="Amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

Once again, we don't mention `IPayment` explicitly. If we execute a query against the `IPayment` interface - for example, `from IPayment` - NHibernate automatically returns instances of `CreditCardPayment` (and its subclasses, since they also implement `IPayment`), `CashPayment` and `ChequePayment` but not instances of `NonelectronicTransaction`.

8.2. Limitations

There are certain limitations to the "implicit polymorphism" approach to the table per concrete-class mapping strategy. There are somewhat less restrictive limitations to `<union-subclass>` mappings.

The following table shows the limitations of table per concrete-class mappings, and of implicit polymorphism, in NHibernate.

Table 8.1. Features of inheritance mappings

Inheritance strategy	Polymorphic many-to-one	Polymorphic one-to-one	Polymorphic one-to-many	Polymorphic many-to-many	Polymorphic `load()/get()`	Polymorphic queries	Polymorphic joins	Outer join fetching
table per class-hierarchy	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.Get<IPayment>(id)`	`from IPayment p`	`from Order o join o.Payment p`	*supported*
table per subclass	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.Get<IPayment>(id)`	`from IPayment p`	`from Order o join o.Payment p`	*supported*
table per concrete-class (union-subclass)	`<many-to-one>`	`<one-to-one>`	`<one-to-many>` (for `inverse="true"` only)	`<many-to-many>`	`s.Get<IPayment>(id)`	`from IPayment p`	`from Order o join o.Payment p`	*supported*

table per concrete class (implicit polymorphism)	`<any>`	*not supported*	*not supported*	`<many-to-any>`	*use a query*	`from IPayment p`	*not supported*	*not supported*

Chapter 9. Manipulating Persistent Data

9.1. Creating a persistent object

An object (entity instance) is either *transient* or *persistent* with respect to a particular `ISession`. Newly instantiated objects are, of course, transient. The session offers services for saving (ie. persisting) transient instances:

```
DomesticCat fritz = new DomesticCat();
fritz.Color = Color.Ginger;
fritz.Sex = 'M';
fritz.Name = "Fritz";
long generatedId = (long) sess.Save(fritz);
```

```
DomesticCat pk = new DomesticCat();
pk.Color = Color.Tabby;
pk.Sex = 'F';
pk.Name = "PK";
pk.Kittens = new HashSet<Cat>();
pk.AddKitten(fritz);
sess.Save( pk, 1234L );
```

The single-argument `Save()` generates and assigns a unique identifier to `fritz`. The two-argument form attempts to persist `pk` using the given identifier. We generally discourage the use of the two-argument form since it may be used to create primary keys with business meaning.

Associated objects may be made persistent in any order you like unless you have a `NOT NULL` constraint upon a foreign key column. There is never a risk of violating foreign key constraints. However, you might violate a `NOT NULL` constraint if you `Save()` the objects in the wrong order.

9.2. Loading an object

The `Load()` methods of `ISession` give you a way to retrieve a persistent instance if you already know its identifier. One version takes a class object and will load the state into a newly instantiated object. The second version allows you to supply an instance into which the state will be loaded. The form which takes an instance is only useful in special circumstances (DIY instance pooling etc.)

```
Cat fritz = sess.Load<Cat>(generatedId);
```

```
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.Load<Cat>(pkId);
```

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.Load( cat, pkId );
ISet<Cat> kittens = cat.Kittens;
```

Note that `Load()` will throw an unrecoverable exception if there is no matching database row. If the class is mapped with a proxy, `Load()` returns an object that is an uninitialized proxy and does not actually hit the database until you invoke a method of the object. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database.

If you are not certain that a matching row exists, you should use the `Get()` method, which hits the database immediately and returns null if there is no matching row.

```
Cat cat = sess.Get<Cat>(id);
if (cat==null) {
    cat = new Cat();
    sess.Save(cat, id);
}
return cat;
```

You may also load an objects using an SQL `SELECT ... FOR UPDATE`. See the next section for a discussion of NHibernate `LockMode`s.

```
Cat cat = sess.Get<Cat>(id, LockMode.Upgrade);
```

Note that any associated instances or contained collections are *not* selected FOR UPDATE.

It is possible to re-load an object and all its collections at any time, using the Refresh() method. This is useful when database triggers are used to initialize some of the properties of the object.

```
sess.Save(cat);
sess.Flush(); //force the SQL INSERT
sess.Refresh(cat); //re-read the state (after the trigger executes)
```

An important question usually appears at this point: How much does NHibernate load from the database and how many SQL SELECTs will it use? This depends on the *fetching strategy* and is explained in Section 19.1, “Fetching strategies”.

9.3. Querying

If you don't know the identifier(s) of the object(s) you are looking for, use the CreateQuery() methods of ISession. NHibernate supports a simple but powerful object oriented query language.

```
IList<Cat> cats = sess.CreateQuery(
    "from Cat as cat where cat.Birthdate = :birthday"
).SetDate("birthday", date)
.List<Cat>();

IList<Cat> mates = sess.CreateQuery(
    "select mate from Cat as cat join cat.Mate as mate " +
    "where cat.name = :name"
).SetString("name", name)
.List<Cat>();

IList<Cat> cats = sess.CreateQuery( "from Cat as cat where cat.Mate.Birthdate is null" ).List<Cat>();

IList<Cat> moreCats = sess.CreateQuery(
    "from Cat as cat where " +
    "cat.Name = 'Fritz' or cat.id = :id1 or cat.id = :id2"
).SetInt64("id1", id1)
.SetInt67("id2", id2)
.List<Cat>();

IList<Cat> mates = sess.CreateQuery(
    "from Cat as cat where cat.Mate = :mate"
).SetParameter("mate", izi, NHibernateUtil.Entity(typeof(Cat))
).List<Cat>();

IList<Cat> problems = sess.CreateQuery(
    "from GoldFish as fish " +
    "where fish.Birthday > fish.Deceased or fish.Birthday is null"
).List<Cat>();
```

The NHibernateUtil class defines a number of static methods and constants, providing access to most of the built-in types, as instances of NHibernate.Type.IType.

If you expect your query to return a very large number of objects, but you don't expect to use them all, you might get better performance from the Enumerable() methods, which return a System.Collections.IEnumerable. The iterator will load objects on demand, using the identifiers returned by an initial SQL query (n+1 selects total).

```
// fetch ids
IEnumerable<Qux> en = sess.CreateQuery("from eg.Qux q order by q.Likeliness").Enumerable<Qux>();
foreach ( Qux qux in en )
{
    // something we couldnt express in the query
    if ( qux.CalculateComplicatedAlgorithm() ) {
        // dont need to process the rest
        break;
    }
}
```

The Enumerable() method also performs better if you expect that many of the objects are already loaded and cached by the session, or if the query results contain the same objects many times. (When no data is cached or repeated, Find() is almost always faster.) Heres an example of a query that should be called using Enumerable():

```
IEnumerable en = sess.CreateQuery(
    "select customer, product " +
    "from Customer customer, " +
    "Product product " +
    "join customer.Purchases purchase " +
    "where product = purchase.Product"
).Enumerable();
```

Calling the previous query using `Find()` would return a very large ADO.NET result set containing the same data many times.

NHibernate queries sometimes return tuples of objects, in which case each tuple is returned as an array:

```
IEnumerable foosAndBars = sess.CreateQuery(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.Date = foo.Date"
).Enumerable();
foreach (object[] tuple in foosAndBars)
{
    Foo foo = tuple[0]; Bar bar = tuple[1];
    ....
}
```

9.3.1. Scalar queries

Queries may specify a property of a class in the `select` clause. They may even call SQL aggregate functions. Properties or aggregates are considered "scalar" results.

```
IEnumerable results = sess.CreateQuery(
        "select cat.Color, min(cat.Birthdate), count(cat) from Cat cat " +
        "group by cat.Color"
).Enumerable();
foreach ( object[] row in results )
{
    Color type = (Color) row[0];
    DateTime oldest = (DateTime) row[1];
    int count = (int) row[2];
    .....
}
```

```
IEnumerable en = sess.CreateQuery(
    "select cat.Type, cat.Birthdate, cat.Name from DomesticCat cat"
).Enumerable();
```

```
IList<Cat> list = sess.CreateQuery(
    "select cat, cat.Mate.Name from DomesticCat cat"
).List<Cat>();
```

9.3.2. The IQuery interface

If you need to specify bounds upon your result set (the maximum number of rows you want to retrieve and / or the first row you want to retrieve) you should obtain an instance of `NHibernate.IQuery`:

```
IQuery q = sess.CreateQuery("from DomesticCat cat");
q.SetFirstResult(20);
q.SetMaxResults(10);
IList<Cat> cats = q.List<Cat>();
```

You may even define a named query in the mapping document. (Remember to use a `CDATA` section if your query contains characters that could be interpreted as markup.)

```
<query name="Eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from Eg.DomesticCat as cat
        where cat.Name = ?
        and cat.Weight > ?
] ]></query>
```

```
IQuery q = sess.GetNamedQuery("Eg.DomesticCat.by.name.and.minimum.weight");
q.SetString(0, name);
q.SetInt32(1, minWeight);
IList<Cat> cats = q.List<Cat>();
```

The query interface supports the use of named parameters. Named parameters are identifiers of the form `:name` in the query string. There are methods on `IQuery` for binding values to named or positional parameters. NHibernate numbers parameters from zero. The advantages of named parameters are:

- named parameters are insensitive to the order they occur in the query string
- they may occur multiple times in the same query
- they are self-documenting

```
//named parameter (preferred)
IQuery q = sess.CreateQuery("from DomesticCat cat where cat.Name = :name");
q.SetString("name", "Fritz");
IEnumerable<Cat> cats = q.Enumerable<Cat>();
```

```
//positional parameter
IQuery q = sess.CreateQuery("from DomesticCat cat where cat.Name = ?");
q.SetString(0, "Izi");
IEnumerable<Cat> cats = q.Enumerable<Cat>();
```

```
//named parameter list
IList<string> names = new List<string>();
names.Add("Izi");
names.Add("Fritz");
IQuery q = sess.CreateQuery("from DomesticCat cat where cat.Name in (:namesList)");
q.SetParameterList("namesList", names);
IList<Cat> cats = q.List<Cat>();
```

9.3.3. Filtering collections

A collection *filter* is a special type of query that may be applied to a persistent collection or array. The query string may refer to `this`, meaning the current collection element.

```
ICollection<Cat> blackKittens = session.CreateFilter(
    pk.Kittens, "where this.Color = ?", Color.Black, NHibernateUtil.Enum(typeof(Color))
).List<Cat>();
```

The returned collection is considered a bag.

Observe that filters do not require a `from` clause (though they may have one if required). Filters are not limited to returning the collection elements themselves.

```
ICollection<Cat> blackKittenMates = session.CreateFilter(
    pk.Kittens, "select this.Mate where this.Color = Eg.Color.Black"
).List<Cat>();
```

9.3.4. Criteria queries

HQL is extremely powerful but some people prefer to build queries dynamically, using an object oriented API, rather than embedding strings in their .NET code. For these people, NHibernate provides an intuitive `ICriteria` query API.

```
ICriteria crit = session.CreateCriteria<Cat>();
crit.Add( Expression.Eq("color", Eg.Color.Black) );
crit.SetMaxResults(10);
IList<Cat> cats = crit.List<Cat>();
```

If you are uncomfortable with SQL-like syntax, this is perhaps the easiest way to get started with NHibernate. This API is also more extensible than HQL. Applications might provide their own implementations of the `ICriterion` interface.

9.3.5. Queries in native SQL

You may express a query in SQL, using `CreateSQLQuery()`. You must enclose SQL aliases in braces.

```
IList<Cat> cats = session.CreateSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    typeof(Cat)
).List<Cat>();
```

```
IList<Cat> cats = session.CreateSQLQuery(
    "SELECT {cat}.ID AS {cat.Id}, {cat}.SEX AS {cat.Sex}, " +
           "{cat}.MATE AS {cat.Mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    typeof(Cat)
).List<Cat>()
```

SQL queries may contain named and positional parameters, just like NHibernate queries.

9.4. Updating objects

9.4.1. Updating in the same ISession

Transactional persistent instances (ie. objects loaded, saved, created or queried by the `ISession`) may be manipulated by the application and any changes to persistent state will be persisted when the `ISession` is *flushed* (discussed later in this chapter). So the most straightforward way to update the state of an object is to `Load()` it, and then manipulate it directly, while the `ISession` is open:

```
DomesticCat cat = (DomesticCat) sess.Load<Cat>( 69L );
cat.Name = "PK";
sess.Flush();  // changes to cat are automatically detected and persisted
```

Sometimes this programming model is inefficient since it would require both an SQL `SELECT` (to load an object) and an SQL `UPDATE` (to persist its updated state) in the same session. Therefore NHibernate offers an alternate approach.

9.4.2. Updating detached objects

Many applications need to retrieve an object in one transaction, send it to the UI layer for manipulation, then save the changes in a new transaction. (Applications that use this kind of approach in a high-concurrency environment usually use versioned data to ensure transaction isolation.) This approach requires a slightly different programming model to the one described in the last section. NHibernate supports this model by providing the method `ISession.Update()`.

```
// in the first session
Cat cat = firstSession.Load<Cat>(catId);
Cat potentialMate = new Cat();
firstSession.Save(potentialMate);

// in a higher tier of the application
cat.Mate = potentialMate;

// later, in a new session
secondSession.Update(cat);  // update cat
secondSession.Update(mate); // update mate
```

If the `Cat` with identifier `catId` had already been loaded by `secondSession` when the application tried to update it, an exception would have been thrown.

The application should individually `Update()` transient instances reachable from the given transient instance if and *only* if it wants their state also updated. (Except for lifecycle objects, discussed later.)

NHibernate users have requested a general purpose method that either saves a transient instance by generating a new identifier or update the persistent state associated with its current identifier. The `SaveOrUpdate()` method now implements this functionality.

NHibernate distinguishes "new" (unsaved) instances from "existing" (saved or loaded in a previous session) instances by the value of their identifier (or version, or timestamp) property. The `unsaved-value` attribute of the `<id>` (or `<version>`, or `<timestamp>`) mapping specifies which values should be interpreted as representing a "new" instance.

```
<id name="Id" type="Int64" column="uid" unsaved-value="0">
    <generator class="hilo"/>
</id>
```

The allowed values of `unsaved-value` are:

- `any` - always save
- `none` - always update
- `null` - save when identifier is null
- valid identifier value - save when identifier is null or the given value
- `undefined` - if set for `version` or `timestamp`, then identifier check is used

If `unsaved-value` is not specified for a class, NHibernate will attempt to guess it by creating an instance of the class using the no-argument constructor and reading the property value from the instance.

```
// in the first session
Cat cat = firstSession.Load<Cat>(catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.Mate = mate;

```

```
// later, in a new session
secondSession.SaveOrUpdate(cat);   // update existing state (cat has a non-null id)
secondSession.SaveOrUpdate(mate);  // save the new instance (mate has a null id)
```

The usage and semantics of `SaveOrUpdate()` seems to be confusing for new users. Firstly, so long as you are not trying to use instances from one session in another new session, you should not need to use `Update()` or `SaveOrUpdate()`. Some whole applications will never use either of these methods.

Usually `Update()` or `SaveOrUpdate()` are used in the following scenario:

- the application loads an object in the first session
- the object is passed up to the UI tier
- some modifications are made to the object
- the object is passed back down to the business logic tier
- the application persists these modifications by calling `Update()` in a second session

`SaveOrUpdate()` does the following:

- if the object is already persistent in this session, do nothing
- if the object has no identifier property, `Save()` it
- if the object's identifier matches the criteria specified by `unsaved-value`, `Save()` it
- if the object is versioned (`version` or `timestamp`), then the version will take precedence to identifier check, unless the versions `unsaved-value="undefined"` (default value)
- if another object associated with the session has the same identifier, throw an exception

The last case can be avoided by using `Merge(Object o)`. This method copies the state of the given object onto the persistent object with the same identifier. If there is no persistent instance currently associated with the session, it will be loaded. The method returns the persistent instance. If the given instance is unsaved or does not exist in the database, NHibernate will save it and return it as a newly persistent instance. Otherwise, the given instance does not become associated with the session. In most applications with detached objects, you need both methods, `SaveOrUpdate()` and `Merge()`.

9.4.3. Reattaching detached objects

The `Lock()` method allows the application to reassociate an unmodified object with a new session.

```
//just reassociate:
sess.Lock(fritz, LockMode.None);
//do a version check, then reassociate:
sess.Lock(izi, LockMode.Read);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.Lock(pk, LockMode.Upgrade);
```

9.5. Deleting persistent objects

`ISession.Delete()` will remove an object's state from the database. Of course, your application might still hold a reference to it. So it's best to think of `Delete()` as making a persistent instance transient.

```
sess.Delete(cat);
```

You may also delete many objects at once by passing a NHibernate query string to `Delete()`:.

```
sess.Delete("from Cat");
```

You may now delete objects in any order you like, without risk of foreign key constraint violations. Of course, it is still possible to violate a `NOT NULL` constraint on a foreign key column by deleting objects in the wrong order.

9.6. Flush

From time to time the `ISession` will execute the SQL statements needed to synchronize the ADO.NET connection's state with the state of objects held in memory. This process, *flush*, occurs by default at the following points

- from some invocations of `Find()` or `Enumerable()`
- from `NHibernate.ITransaction.Commit()`
- from `ISession.Flush()`

The SQL statements are issued in the following order

- all entity insertions, in the same order the corresponding objects were saved using `ISession.Save()`
- all entity updates
- all collection deletions
- all collection element deletions, updates and insertions
- all collection insertions
- all entity deletions, in the same order the corresponding objects were deleted using `ISession.Delete()`

(An exception is that objects using `native` ID generation are inserted when they are saved.)

Except when you explicity `Flush()`, there are absolutely no guarantees about *when* the `Session` executes the ADO.NET calls, only the *order* in which they are executed. However, NHibernate does guarantee that the `ISession.CreateQuery(..)` methods will never return stale data; nor will they return the wrong data.

It is possible to change the default behavior so that flush occurs less frequently, either by code or by configuration. The `FlushMode` class defines three different modes: only flush at commit time (and only when the NHibernate `ITransaction` API is used), flush automatically using the explained routine (will only work inside an explicit NHibernate `ITransaction`), or never flush unless `Flush()` is called explicitly. The last mode is useful for long running units of work, where an ISession is kept open and disconnected for a long time (see Section 11.4, "Optimistic concurrency control").

```
sess = sf.OpenSession();
ITransaction tx = sess.BeginTransaction();
sess.FlushMode = FlushMode.Commit; //allow queries to return stale state
Cat izi = sess.Load<Cat>(id);
izi.Name = "iznizi";
// execute some queries....
sess.CreateQuery("from Cat as cat left outer join cat.Kittens kitten");
//change to izi is not flushed!
...
tx.Commit(); //flush occurs
```

9.7. Ending a Session

Ending a session involves four distinct phases:

- flush the session
- commit the transaction
- close the session
- handle exceptions

9.7.1. Flushing the Session

If you happen to be using the `ITransaction` API, you don't need to worry about this step. It will be performed implicitly when the transaction is committed. Otherwise you should call `ISession.Flush()` to ensure that all changes are synchronized with the database.

9.7.2. Committing the database transaction

If you are using the NHibernate `ITransaction` API, this looks like:

```
tx.Commit(); // flush the session and commit the transaction
```

If you are managing ADO.NET transactions yourself you should manually `Commit()` the ADO.NET transaction.

```
sess.Flush();
currentTransaction.Commit();
```

If you decide *not* to commit your changes:

```
tx.Rollback();  // rollback the transaction
```

or:

```
currentTransaction.Rollback();
```

If you rollback the transaction you should immediately close and discard the current session to ensure that NHibernate's internal state is consistent.

9.7.3. Closing the ISession

A call to `ISession.Close()` marks the end of a session. The main implication of `Close()` is that the ADO.NET connection will be relinquished by the session.

```
tx.Commit();
sess.Close();
```

```
sess.Flush();
currentTransaction.Commit();
sess.Close();
```

If you provided your own connection, `Close()` returns a reference to it, so you can manually close it or return it to the pool. Otherwise `Close()` returns it to the pool.

9.8. Exception handling

NHibernate use might lead to exceptions, usually `HibernateException`. This exception can have a nested inner exception (the root cause), use the `InnerException` property to access it.

If the `ISession` throws an exception you should immediately rollback the transaction, call `ISession.Close()` and discard the `ISession` instance. Certain methods of `ISession` will *not* leave the session in a consistent state.

For exceptions thrown by the data provider while interacting with the database, NHibernate will wrap the error in an instance of `ADOException`. The underlying exception is accessible by calling `ADOException.InnerException`. NHibernate converts the `DbException` into an appropriate `ADOException` subclass using the `ISQLExceptionConverter` attached to the `SessionFactory`. By default, the `ISQLExceptionConverter` is defined by the configured dialect; however, it is also possible to plug in a custom implementation (see the api-docs for the ISQLExceptionConverter class for details).

The following exception handling idiom shows the typical case in NHibernate applications:

```
using (ISession sess = factory.OpenSession())
using (ITransaction tx = sess.BeginTransaction())
{
    // do some work
    ...
    tx.Commit();
}
```

Or, when manually managing ADO.NET transactions:

```
ISession sess = factory.openSession();
try
{
    // do some work
    ...
    sess.Flush();
    currentTransaction.Commit();
}
catch (Exception e)
{
    currentTransaction.Rollback();
    throw;
}
finally
{
    sess.Close();
}
```

9.9. Lifecyles and object graphs

To save or update all objects in a graph of associated objects, you must either

- `Save()`, `SaveOrUpdate()` or `Update()` each individual object OR
- map associated objects using `cascade="all"` or `cascade="save-update"`.

Likewise, to delete all objects in a graph, either

- `Delete()` each individual object OR
- map associated objects using `cascade="all"`, `cascade="all-delete-orphan"` or `cascade="delete"`.

Recommendation:

- If the child object's lifespan is bounded by the lifespan of the of the parent object make it a *lifecycle object* by specifying `cascade="all"`.
- Otherwise, `Save()` and `Delete()` it explicitly from application code. If you really want to save yourself some extra typing, use `cascade="save-update"` and explicit `Delete()`.

Mapping an association (many-to-one, one-to-one or collection) with `cascade="all"` marks the association as a *parent/child* style relationship where save/update/deletion of the parent results in save/update/deletion of the child(ren). Futhermore, a mere reference to a child from a persistent parent will result in save / update of the child. The metaphor is incomplete, however. A child which becomes unreferenced by its parent is *not* automatically deleted, except in the cases of `<one-to-many>` and `<one-to-one>` associations that have been mapped with `cascade="all-delete-orphan"` or `cascade="delete-orphan"`. The precise semantics of cascading operations are as follows:

- If a parent is saved, all children are passed to `SaveOrUpdate()`
- If a parent is passed to `Update()` or `SaveOrUpdate()`, all children are passed to `SaveOrUpdate()`
- If a transient child becomes referenced by a persistent parent, it is passed to `SaveOrUpdate()`
- If a parent is deleted, all children are passed to `Delete()`
- If a transient child is dereferenced by a persistent parent, *nothing special happens* (the application should explicitly delete the child if necessary) unless `cascade="all-delete-orphan"` or `cascade="delete-orphan"`, in which case the "orphaned" child is deleted.

NHibernate does not fully implement "persistence by reachability", which would imply (inefficient) persistent garbage collection. However, due to popular demand, NHibernate does support the notion of entities becoming persistent when referenced by another persistent object. Associations marked `cascade="save-update"` behave in this way. If you wish to use this approach throughout your application, it's easier to specify the `default-cascade` attribute of the `<hibernate-mapping>` element.

9.10. Interceptors

The `IInterceptor` interface provides callbacks from the session to the application allowing the application to inspect and / or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. One possible use for this is to track auditing information. For example, the following `IInterceptor` automatically sets the `CreateTimestamp` when an `IAuditable` is created and updates the `LastUpdateTimestamp` property when an `IAuditable` is updated.

```
using System;
using NHibernate.Type;

namespace NHibernate.Test
{
    [Serializable]
    public class AuditInterceptor : IInterceptor
    {

        private int updates;
        private int creates;

        public void OnDelete(object entity,
                             object id,
                             object[] state,
                             string[] propertyNames,
                             IType[] types)
        {
            // do nothing
        }

        public boolean OnFlushDirty(object entity,
                                    object id,
                                    object[] currentState,
                                    object[] previousState,
                                    string[] propertyNames,
                                    IType[] types) {

            if ( entity is IAuditable )
            {
                updates++;
                for ( int i=0; i < propertyNames.Length; i++ )
                {
                    if ( "LastUpdateTimestamp" == propertyNames[i] )
                    {
                        currentState[i] = DateTime.Now;
                        return true;
                    }
                }
            }
            return false;
        }

        public boolean OnLoad(object entity,
                              object id,
                              object[] state,
                              string[] propertyNames,
                              IType[] types)
        {
            return false;
        }

        public boolean OnSave(object entity,
                              object id,
                              object[] state,
                              string[] propertyNames,
                              IType[] types)
        {
            if ( entity is IAuditable )
            {
                creates++;
                for ( int i=0; i<propertyNames.Length; i++ )
                {
                    if ( "CreateTimestamp" == propertyNames[i] )
                    {
                        state[i] = DateTime.Now;
                        return true;
                    }
                }
            }
            return false;
        }

        public void PostFlush(ICollection entities)
        {
            Console.Out.WriteLine("Creations: {0}, Updates: {1}", creates, updates);
        }

        public void PreFlush(ICollection entities) {
            updates=0;
            creates=0;
        }

        ......
        ......
    }
}
```

The interceptor would be specified when a session is created.

```
ISession session = sf.OpenSession( new AuditInterceptor() );
```

You may also set an interceptor on a global level, using the `Configuration`:

```
new Configuration().SetInterceptor( new AuditInterceptor() );
```

9.11. Metadata API

NHibernate requires a very rich meta-level model of all entity and value types. From time to time, this model is very useful to the application itself. For example, the application might use NHibernate's metadata to implement a "smart" deep-copy algorithm that understands which objects should be copied (eg. mutable value types) and which should not (eg. immutable value types and, possibly, associated entities).

NHibernate exposes metadata via the `IClassMetadata` and `ICollectionMetadata` interfaces and the `IType` hierarchy. Instances of the metadata interfaces may be obtained from the `ISessionFactory`.

```
Cat fritz = ......;
IClassMetadata catMeta = sessionfactory.GetClassMetadata(typeof(Cat));
long id = (long) catMeta.GetIdentifier(fritz);
object[] propertyValues = catMeta.GetPropertyValues(fritz);
string[] propertyNames = catMeta.PropertyNames;
IType[] propertyTypes = catMeta.PropertyTypes;

// get an IDictionary of all properties which are not collections or associations
// TODO: what about components?

IDictionary namedValues = new Hashtable();
for ( int i=0; i<propertyNames.Length; i++ )
{
    if ( !propertyTypes[i].IsEntityType && !propertyTypes[i].IsCollectionType )
        {
        namedValues[ propertyNames[i] ] = propertyValues[i];
    }
}
```

Chapter 10. Read-only entities

Important

NHibernate's treatment of *read-only* entities may differ from what you may have encountered elsewhere. Incorrect usage may cause unexpected results.

When an entity is read-only:

- NHibernate does not dirty-check the entity's simple properties or single-ended associations;
- NHibernate will not update simple properties or updatable single-ended associations;
- NHibernate will not update the version of the read-only entity if only simple properties or single-ended updatable associations are changed;

In some ways, NHibernate treats read-only entities the same as entities that are not read-only:

- NHibernate cascades operations to associations as defined in the entity mapping.
- NHibernate updates the version if the entity has a collection with changes that dirties the entity;
- A read-only entity can be deleted.

Even if an entity is not read-only, its collection association can be affected if it contains a read-only entity.

For details about the affect of read-only entities on different property and association types, see Section 10.2, “Read-only affect on property type”.

For details about how to make entities read-only, see Section 10.1, “Making persistent entities read-only”

NHibernate does some optimizing for read-only entities:

- It saves execution time by not dirty-checking simple properties or single-ended associations.
- It saves memory by deleting database snapshots.

10.1. Making persistent entities read-only

Only persistent entities can be made read-only. Transient and detached entities must be put in persistent state before they can be made read-only.

NHibernate provides the following ways to make persistent entities read-only:

- you can map an entity class as *immutable*; when an entity of an immutable class is made persistent, NHibernate automatically makes it read-only. see Section 10.1.1, “Entities of immutable classes” for details
- you can change a default so that entities loaded into the session by NHibernate are automatically made read-only; see Section 10.1.2, “Loading persistent entities as read-only” for details
- you can make an HQL query or criteria read-only so that entities loaded when the query or criteria executes, or iterates,

are automatically made read-only; see Section 10.1.3, "Loading read-only entities from an HQL query/criteria" for details

- you can make a persistent entity that is already in the in the session read-only; see Section 10.1.4, "Making a persistent entity read-only" for details

10.1.1. Entities of immutable classes

When an entity instance of an immutable class is made persistent, NHibernate automatically makes it read-only.

An entity of an immutable class can created and deleted the same as an entity of a mutable class.

NHibernate treats a persistent entity of an immutable class the same way as a read-only persistent entity of a mutable class. The only exception is that NHibernate will not allow an entity of an immutable class to be changed so it is not read-only.

10.1.2. Loading persistent entities as read-only

Note

Entities of immutable classes are automatically loaded as read-only.

To change the default behavior so NHibernate loads entity instances of mutable classes into the session and automatically makes them read-only, call:

```
Session.DefaultReadOnly = true;
```

To change the default back so entities loaded by NHibernate are not made read-only, call:

```
Session.DefaultReadOnly = false;
```

You can determine the current setting by using the property:

```
Session.DefaultReadOnly;
```

If `Session.DefaultReadOnly` property returns true, entities loaded by the following are automatically made read-only:

- `Session.Load()` and `Session.Load<T>`
- `Session.Get()` and `Session.Get<T>`
- `Session.Merge()`
- executing, or iterating HQL queries and criteria; to override this setting for a particular HQL query or criteria see Section 10.1.3, "Loading read-only entities from an HQL query/criteria"

Changing this default has no effect on:

- persistent entities already in the session when the default was changed
- persistent entities that are refreshed via `Session.Refresh()`; a refreshed persistent entity will only be read-only if it was read-only before refreshing
- persistent entities added by the application via `Session.Persist()`, `Session.Save()`, and `Session.Update()` `Session.SaveOrUpdate()`

10.1.3. Loading read-only entities from an HQL query/criteria

Note

Entities of immutable classes are automatically loaded as read-only.

If Session.DefaultReadOnly returns false (the default) when an HQL query or criteria executes, then entities and proxies of mutable classes loaded by the query will not be read-only.

You can override this behavior so that entities and proxies loaded by an HQL query or criteria are automatically made read-only.

For an HQL query, call:

```
Query.SetReadOnly(true);
```

`Query.SetReadOnly(true)` must be called before `Query.List()`, `Query.UniqueResult()`, or `Query.Iterate()`

For an HQL criteria, call:

```
Criteria.SetReadOnly(true);
```

`Criteria.SetReadOnly(true)` must be called before `Criteria.List()`, or `Criteria.UniqueResult()`

Entities and proxies that exist in the session before being returned by an HQL query or criteria are not affected.

Uninitialized persistent collections returned by the query are not affected. Later, when the collection is initialized, entities loaded into the session will be read-only if Session.DefaultReadOnly returns true.

Using `Query.SetReadOnly(true)` or `Criteria.SetReadOnly(true)` works well when a single HQL query or criteria loads all the entities and intializes all the proxies and collections that the application needs to be read-only.

When it is not possible to load and initialize all necessary entities in a single query or criteria, you can temporarily change the session default to load entities as read-only before the query is executed. Then you can explicitly initialize proxies and collections before restoring the session default.

```
ISession session = factory.OpenSession();
ITransaction tx = session.BeginTransaction();

session.DefaultReadOnly = true;
Contract contract = session.CreateQuery("from Contract where CustomerName = 'Sherman'").UniqueResult<Contract>();
NHibernate.Initialize(contract.Plan);
NHibernate.Initialize(contract.Variations);
NHibernate.Initialize(contract.Notes);
session.DefaultReadOnly = false;
...
tx.Commit();
session.Close();
```

If Session.DefaultReadOnly returns true, then you can use Query.SetReadOnly(false) and Criteria.SetReadOnly(false) to override this session setting and load entities that are not read-only.

10.1.4. Making a persistent entity read-only

Note

Persistent entities of immutable classes are automatically made read-only.

To make a persistent entity or proxy read-only, call:

```
Session.SetReadOnly(entityOrProxy, true)
```

To change a read-only entity or proxy of a mutable class so it is no longer read-only, call:

```
Session.SetReadOnly(entityOrProxy, false)
```

Important

When a read-only entity or proxy is changed so it is no longer read-only, NHibernate assumes that the current state of the read-only entity is consistent with its database representation. If this is not true, then any non-flushed changes made before or while the entity was read-only, will be ignored.

To throw away non-flushed changes and make the persistent entity consistent with its database representation, call:

```
Session.Refresh(entity);
```

To flush changes made before or while the entity was read-only and make the database representation consistent with the current state of the persistent entity:

```
// evict the read-only entity so it is detached
session.Evict(entity);

// make the detached entity (with the non-flushed changes) persistent
session.Update(entity);
```

```
// now entity is no longer read-only and its changes can be flushed
s.Flush();
```

10.2. Read-only affect on property type

The following table summarizes how different property types are affected by making an entity read-only.

Table 10.1. Affect of read-only entity on property types

Property/Association Type	Changes flushed to DB?
Simple (Section 10.2.1, “Simple properties”)	no*
Unidirectional one-to-one Unidirectional many-to-one (Section 10.2.2.1, “Unidirectional one-to-one and many-to-one”)	no* no*
Unidirectional one-to-many Unidirectional many-to-many (Section 10.2.2.2, “Unidirectional one-to-many and many-to-many”)	yes yes
Bidirectional one-to-one (Section 10.2.3.1, “Bidirectional one-to-one”)	only if the owning entity is not read-only*
Bidirectional one-to-many/many-to-one inverse collection non-inverse collection (Section 10.2.3.2, “Bidirectional one-to-many/many-to-one”)	 only added/removed entities that are not read-only* yes
Bidirectional many-to-many (Section 10.2.3.3, “Bidirectional many-to-many”)	yes

* Behavior is different when the entity having the property/association is read-only, compared to when it is not read-only.

10.2.1. Simple properties

When a persistent object is read-only, NHibernate does not dirty-check simple properties.

NHibernate will not synchronize simple property state changes to the database. If you have automatic versioning, NHibernate will not increment the version if any simple properties change.

```
ISession session = factory.OpenSession();
ITransaction tx = session.BeginTransaction();

// get a contract and make it read-only
Contract contract = session.Get<Contract>(contractId);
session.SetReadOnly(contract, true);

// contract.CustomerName is "Sherman"
contract.CustomerName = "Yogi";
tx.Commit();

tx = session.BeginTransaction();

contract = session.Get<Contract>(contractId);
// contract.CustomerName is still "Sherman"
...
tx.Commit();
session.Close();
```

10.2.2. Unidirectional associations

10.2.2.1. Unidirectional one-to-one and many-to-one

NHibernate treats unidirectional one-to-one and many-to-one associations in the same way when the owning entity is read-only.

We use the term *unidirectional single-ended association* when referring to functionality that is common to unidirectional one-to-one and many-to-one associations.

NHibernate does not dirty-check unidirectional single-ended associations when the owning entity is read-only.

If you change a read-only entity's reference to a unidirectional single-ended association to null, or to refer to a different entity, that change will not be flushed to the database.

Note

If an entity is of an immutable class, then its references to unidirectional single-ended associations must be assigned when that entity is first created. Because the entity is automatically made read-only, these references can not be updated.

If automatic versioning is used, NHibernate will not increment the version due to local changes to unidirectional single-ended associations.

In the following examples, Contract has a unidirectional many-to-one association with Plan. Contract cascades save and update operations to the association.

The following shows that changing a read-only entity's many-to-one association reference to null has no effect on the entity's database representation.

```
// get a contract with an existing plan;
// make the contract read-only and set its plan to null
tx = session.BeginTransaction();
Contract contract = session.Get<Contract>(contractId);
session.SetReadOnly(contract, true);
contract.Plan = null;
tx.Commit();

// get the same contract
tx = session.BeginTransaction();
Contract contract = session.Get<Contract>(contractId);

// contract.Plan still refers to the original plan;

tx.Commit();
session.Close();
```

The following shows that, even though an update to a read-only entity's many-to-one association has no affect on the entity's database representation, flush still cascades the save-update operation to the locally changed association.

```
// get a contract with an existing plan;
// make the contract read-only and change to a new plan
tx = session.BeginTransaction();
Contract contract = session.Get<Contract>(contractId);
session.SetReadOnly(contract, true);
Plan newPlan = new Plan("new plan");
contract.Plan = newPlan;
tx.Commit();

// get the same contract
tx = session.BeginTransaction();
contract = session.Get<Contract>(contractId);
newPlan = session.Get<Plan>(newPlan.Id);

// contract.Plan still refers to the original plan;
// newPlan is non-null because it was persisted when
// the previous transaction was committed;

tx.Commit();
session.Close();
```

10.2.2.2. Unidirectional one-to-many and many-to-many

NHibernate treats unidirectional one-to-many and many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

NHibernate dirty-checks unidirectional one-to-many and many-to-many associations;

The collection can contain entities that are read-only, as well as entities that are not read-only.

Entities can be added and removed from the collection; changes are flushed to the database.

If automatic versioning is used, NHibernate will update the version due to changes in the collection if they dirty the owning entity.

10.2.3. Bidirectional associations

10.2.3.1. Bidirectional one-to-one

If a read-only entity owns a bidirectional one-to-one association:

- NHibernate does not dirty-check the association.
- updates that change the association reference to null or to refer to a different entity will not be flushed to the database.
- If automatic versioning is used, NHibernate will not increment the version due to local changes to the association.

Note

If an entity is of an immutable class, and it owns a bidirectional one-to-one association, then its reference must be assigned when that entity is first created. Because the entity is automatically made read-only, these references cannot be updated.

When the owner is not read-only, NHibernate treats an association with a read-only entity the same as when the association is with an entity that is not read-only.

10.2.3.2. Bidirectional one-to-many/many-to-one

A read-only entity has no impact on a bidirectional one-to-many/many-to-one association if:

- the read-only entity is on the one-to-many side using an inverse collection;
- the read-only entity is on the one-to-many side using a non-inverse collection;
- the one-to-many side uses a non-inverse collection that contains the read-only entity

When the one-to-many side uses an inverse collection:

- a read-only entity can only be added to the collection when it is created;
- a read-only entity can only be removed from the collection by an orphan delete or by explicitly deleting the entity.

10.2.3.3. Bidirectional many-to-many

NHibernate treats bidirectional many-to-many associations owned by a read-only entity the same as when owned by an entity that is not read-only.

NHibernate dirty-checks bidirectional many-to-many associations.

The collection on either side of the association can contain entities that are read-only, as well as entities that are not read-only.

Entities are added and removed from both sides of the collection; changes are flushed to the database.

If automatic versioning is used, NHibernate will update the version due to changes in both sides of the collection if they dirty the entity owning the respective collections.

Chapter 11. Transactions And Concurrency

NHibernate is not itself a database. It is a lightweight object-relational mapping tool. Transaction management is delegated to the underlying database connection. If the connection is enlisted with a distributed transaction, operations performed by the `ISession` are atomically part of the wider distributed transaction. NHibernate can be seen as a thin adapter to ADO.NET, adding object-oriented semantics.

11.1. Configurations, Sessions and Factories

An `ISessionFactory` is an expensive-to-create, threadsafe object intended to be shared by all application threads. An `ISession` is an inexpensive, non-threadsafe object that should be used once, for a single business process, and then discarded. For example, when using NHibernate in an ASP.NET application, pages could obtain an `ISessionFactory` using:

```
ISessionFactory sf = Global.SessionFactory;
```

Each call to a service method could create a new `ISession`, `Flush()` it, `Commit()` its transaction, `Close()` it and finally discard it. (The `ISessionFactory` may also be kept in a static *Singleton* helper variable.)

We use the NHibernate `ITransaction` API as discussed previously, a single `Commit()` of a NHibernate `ITransaction` flushes the state and commits any underlying database connection (with special handling of distributed transactions).

Ensure you understand the semantics of `Flush()`. Flushing synchronizes the persistent store with in-memory changes but *not* vice-versa. Note that for all NHibernate ADO.NET connections/transactions, the transaction isolation level for that connection applies to all operations executed by NHibernate!

The next few sections will discuss alternative approaches that utilize versioning to ensure transaction atomicity. These are considered "advanced" approaches to be used with care.

11.2. Threads and connections

You should observe the following practices when creating NHibernate Sessions:

- Never create more than one concurrent `ISession` or `ITransaction` instance per database connection.
- Be extremely careful when creating more than one `ISession` per database per transaction. The `ISession` itself keeps track of updates made to loaded objects, so a different `ISession` might see stale data.
- The `ISession` is *not* threadsafe! Never access the same `ISession` in two concurrent threads. An `ISession` is usually only a single unit-of-work!

11.3. Considering object identity

The application may concurrently access the same persistent state in two different units-of-work. However, an instance of a persistent class is never shared between two `ISession` instances. Hence there are two different notions of identity:

Database Identity

```
foo.Id.Equals( bar.Id )
```

CLR Identity

```
foo == bar
```

Then for objects attached to a *particular* `Session`, the two notions are equivalent. However, while the application might concurrently access the "same" (persistent identity) business object in two different sessions, the two instances will actually be "different" (CLR identity).

This approach leaves NHibernate and the database to worry about concurrency. The application never needs to synchronize on any business object, as long as it sticks to a single thread per `ISession` or object identity (within an `ISession` the application may safely use == to compare objects).

11.4. Optimistic concurrency control

Many business processes require a whole series of interactions with the user interleaved with database accesses. In web and enterprise applications it is not acceptable for a database transaction to span a user interaction.

Maintaining isolation of business processes becomes the partial responsibility of the application tier, hence we call this process a long running *application transaction*. A single application transaction usually spans several database transactions. It will be atomic if only one of these database transactions (the last one) stores the updated data, all others simply read data.

The only approach that is consistent with high concurrency and high scalability is optimistic concurrency control with versioning. NHibernate provides for three possible approaches to writing application code that uses optimistic concurrency.

11.4.1. Long session with automatic versioning

A single `ISession` instance and its persistent instances are used for the whole application transaction.

The `ISession` uses optimistic locking with versioning to ensure that many database transactions appear to the application as a single logical application transaction. The `ISession` is disconnected from any underlying ADO.NET connection when waiting for user interaction. This approach is the most efficient in terms of database access. The application need not concern itself with version checking or with reattaching detached instances.

```
// foo is an instance loaded earlier by the Session
session.Reconnect();
transaction = session.BeginTransaction();
foo.Property = "bar";
session.Flush();
transaction.Commit();
session.Disconnect();
```

The `foo` object still knows which `ISession` it was loaded it. As soon as the `ISession` has an ADO.NET connection, we commit the changes to the object.

This pattern is problematic if our `ISession` is too big to be stored during user think time, e.g. an `HttpSession` should be kept as small as possible. As the `ISession` is also the (mandatory) first-level cache and contains all loaded objects, we can propably use this strategy only for a few request/response cycles. This is indeed recommended, as the `ISession` will soon also have stale data.

11.4.2. Many sessions with automatic versioning

Each interaction with the persistent store occurs in a new `ISession`. However, the same persistent instances are reused for each interaction with the database. The application manipulates the state of detached instances originally loaded in another `ISession` and then "reassociates" them using `ISession.Update()` or `ISession.SaveOrUpdate()`.

```
// foo is an instance loaded by a previous Session
foo.Property = "bar";
session = factory.OpenSession();
transaction = session.BeginTransaction();
session.SaveOrUpdate(foo);
session.Flush();
transaction.Commit();
session.Close();
```

You may also call `Lock()` instead of `Update()` and use `LockMode.Read` (performing a version check, bypassing all caches) if you are sure that the object has not been modified.

11.4.3. Customizing automatic versioning

You may disable NHibernate's automatic version increment for particular properties and collections by setting the `optimistic-lock` mapping attribute to `false`. NHibernate will then no longer increment versions if the property is dirty.

Legacy database schemas are often static and can't be modified. Or, other applications might also access the same database and don't know how to handle version numbers or even timestamps. In both cases, versioning can't rely on a particular column in a table. To force a version check without a version or timestamp property mapping, with a comparison of the state of all fields in a row, turn on `optimistic-lock="all"` in the `<class>` mapping. Note that this concepetually only works if NHibernate can compare the old and new state, i.e. if you use a single long `ISession` and not session-per-request-with-detached-objects.

Sometimes concurrent modification can be permitted as long as the changes that have been made don't overlap. If you set `optimistic-lock="dirty"` when mapping the `<class>`, NHibernate will only compare dirty fields during flush.

In both cases, with dedicated version/timestamp columns or with full/dirty field comparison, NHibernate uses a single `UPDATE` statement (with an appropriate `WHERE` clause) per entity to execute the version check and update the information. If you use transitive persistence to cascade reattachment to associated entities, NHibernate might execute uneccessary updates. This is usually not a problem, but *on update* triggers in the database might be executed even when no changes have been made to detached instances. You can customize this behavior by setting `select-before-update="true"` in the `<class>` mapping, forcing NHibernate to `SELECT` the instance to ensure that changes did actually occur, before updating the row.

11.4.4. Application version checking

Each interaction with the database occurs in a new `ISession` that reloads all persistent instances from the database before manipulating them. This approach forces the application to carry out its own version checking to ensure application transaction isolation. (Of course, NHibernate will still *update* version numbers for you.) This approach is the least efficient in terms of database access.

```
// foo is an instance loaded by a previous Session
```

```
session = factory.OpenSession();
transaction = session.BeginTransaction();
int oldVersion = foo.Version;
session.Load( foo, foo.Key );
if ( oldVersion != foo.Version ) throw new StaleObjectStateException();
foo.Property = "bar";
session.Flush();
transaction.Commit();
session.close();
```

Of course, if you are operating in a low-data-concurrency environment and don't require version checking, you may use this approach and just skip the version check.

11.5. Session disconnection

The first approach described above is to maintain a single `ISession` for a whole business process thats spans user think time. (For example, a servlet might keep an `ISession` in the user's `HttpSession`.) For performance reasons you should

- commit the `ITransaction` and then
- disconnect the `ISession` from the ADO.NET connection

before waiting for user activity. The method `ISession.Disconnect()` will disconnect the session from the ADO.NET connection and return the connection to the pool (unless you provided the connection).

`ISession.Reconnect()` obtains a new connection (or you may supply one) and restarts the session. After reconnection, to force a version check on data you aren't updating, you may call `ISession.Lock()` on any objects that might have been updated by another transaction. You don't need to lock any data that you *are* updating.

Heres an example:

```
ISessionFactory sessions;
IList<Foo> fooList;
Bar bar;
....
ISession s = sessions.OpenSession();
ITransaction tx = null;

try
{
    tx = s.BeginTransaction())

    fooList = s.CreateQuery(
        "select foo from Eg.Foo foo where foo.Date = current date"
        // uses db2 date function
    ).List<Foo>();

    bar = new Bar();
    s.Save(bar);

    tx.Commit();
}
catch (Exception)
{
    if (tx != null) tx.Rollback();
    s.Close();
    throw;
}
s.Disconnect();
```

Later on:

```
s.Reconnect();

try
{
    tx = s.BeginTransaction();

    bar.FooTable = new HashMap();
    foreach (Foo foo in fooList)
    {
        s.Lock(foo, LockMode.Read);    //check that foo isn't stale
        bar.FooTable.Put( foo.Name, foo );
    }

    tx.Commit();
}
catch (Exception)
{
    if (tx != null) tx.Rollback();
    throw;
}
finally
{
    s.Close();
}
```

You can see from this how the relationship between `ITransaction`s and `ISession`s is many-to-one, An `ISession` represents a conversation between the application and the database. The `ITransaction` breaks that conversation up into atomic units of work at the database level.

11.6. Pessimistic Locking

It is not intended that users spend much time worring about locking strategies. It's usually enough to specify an isolation level for the ADO.NET connections and then simply let the database do all the work. However, advanced users may sometimes wish to obtain exclusive pessimistic locks, or re-obtain locks at the start of a new transaction.

NHibernate will always use the locking mechanism of the database, never lock objects in memory!

The `LockMode` class defines the different lock levels that may be acquired by NHibernate. A lock is obtained by the following mechanisms:

- `LockMode.Write` is acquired automatically when NHibernate updates or inserts a row.
- `LockMode.Upgrade` may be acquired upon explicit user request using `SELECT ... FOR UPDATE` on databases which support that syntax.
- `LockMode.UpgradeNoWait` may be acquired upon explicit user request using a `SELECT ... FOR UPDATE NOWAIT` under Oracle.
- `LockMode.Read` is acquired automatically when NHibernate reads data under Repeatable Read or Serializable isolation level. May be re-acquired by explicit user request.
- `LockMode.None` represents the absence of a lock. All objects switch to this lock mode at the end of an `ITransaction`. Objects associated with the session via a call to `Update()` or `SaveOrUpdate()` also start out in this lock mode.

The "explicit user request" is expressed in one of the following ways:

- A call to `ISession.Load()`, specifying a `LockMode`.
- A call to `ISession.Lock()`.
- A call to `IQuery.SetLockMode()`.

If `ISession.Load()` is called with `Upgrade` or `UpgradeNoWait`, and the requested object was not yet loaded by the session, the object is loaded using `SELECT ... FOR UPDATE`. If `Load()` is called for an object that is already loaded with a less restrictive lock than the one requested, NHibernate calls `Lock()` for that object.

`ISession.Lock()` performs a version number check if the specified lock mode is `Read`, `Upgrade` or `UpgradeNoWait`. (In the case of `Upgrade` or `UpgradeNoWait`, `SELECT ... FOR UPDATE` is used.)

If the database does not support the requested lock mode, NHibernate will use an appropriate alternate mode (instead of throwing an exception). This ensures that applications will be portable.

11.7. Connection Release Modes

The legacy (1.0.x) behavior of NHibernate in regards to ADO.NET connection management was that a `ISession` would obtain a connection when it was first needed and then hold unto that connection until the session was closed. NHibernate introduced the notion of connection release modes to tell a session how to handle its ADO.NET connections. Note that the following discussion is pertinent only to connections provided through a configured `IConnectionProvider`; user-supplied connections are outside the breadth of this discussion. The different release modes are identified by the enumerated values of `NHibernate.ConnectionReleaseMode`:

- `OnClose` - is essentially the legacy behavior described above. The NHibernate session obtains a connection when it first needs to perform some database access and holds unto that connection until the session is closed.
- `AfterTransaction` - says to release connections after a `NHibernate.ITransaction` has completed.

The configuration parameter `hibernate.connection.release_mode` is used to specify which release mode to use. The possible values:

- auto (the default) - equivalent to after_transaction in the current release. It is rarely a good idea to change this default behavior as failures due to the value of this setting tend to indicate bugs and/or invalid assumptions in user code.
- on_close - says to use ConnectionReleaseMode.OnClose. This setting is left for backwards compatibility, but its use is highly discouraged.
- after_transaction - says to use ConnectionReleaseMode.AfterTransaction. Note that with ConnectionReleaseMode.AfterTransaction, if a session is considered to be in auto-commit mode (i.e. no transaction was started) connections will be released after every operation.

As of NHibernate, if your application manages transactions through .NET APIs such as System.Transactions library, ConnectionReleaseMode.AfterTransaction may cause NHibernate to open and close several connections during one transaction, leading to unnecessary overhead and transaction promotion from local to distributed. Specifying ConnectionReleaseMode.OnClose will revert to the legacy behavior and prevent this problem from occuring.

Chapter 12. Interceptors and events

It is often useful for the application to react to certain events that occur inside NHibernate. This allows implementation of certain kinds of generic functionality, and extension of NHibernate functionality.

12.1. Interceptors

The IInterceptor interface provides callbacks from the session to the application allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. One possible use for this is to track auditing information. For example, the following IInterceptor automatically sets the createTimestamp when an IAuditable is created and updates the lastUpdateTimestamp property when an IAuditable is updated.

You may either implement IInterceptor directly or (better) extend EmptyInterceptor.

```
using System;

using NHibernate;
using NHibernate.Type;

public class AuditInterceptor : EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public override void OnDelete(object entity,
                                  object id,
                                  object[] state,
                                  string[] propertyNames,
                                  IType[] types)
    {
        // do nothing
    }

    public override bool OnFlushDirty(object entity,
                                      object id,
                                      object[] currentState,
                                      object[] previousState,
                                      string[] propertyNames,
                                      IType[] types)
    {
        if ( entity is IAuditable ) {
            updates++;
            for ( int i=0; i < propertyNames.Length; i++ ) {
                if ( "lastUpdateTimestamp".Equals( propertyNames[i] ) ) {
                    currentState[i] = new DateTime();
                    return true;
                }
            }
        }
        return false;
    }

    public override bool OnLoad(object entity,
                                object id,
                                object[] state,
                                string[] propertyNames,
                                IType[] types)
    {
        if ( entity is IAuditable ) {
            loads++;
        }
        return false;
    }

    public override bool OnSave(object entity,
                                object id,
                                object[] state,
                                string[] propertyNames,
                                IType[] types)
    {
        if ( entity is IAuditable ) {
            creates++;
            for ( int i=0; i<propertyNames.Length; i++ ) {
                if ( "createTimestamp".Equals( propertyNames[i] ) ) {
                    state[i] = new DateTime();
                    return true;
                }
            }
        }
        return false;
    }

    public override void AfterTransactionCompletion(ITransaction tx)
    {
        if ( tx.WasCommitted ) {
            System.Console.WriteLine("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

Interceptors come in two flavors: `ISession`-scoped and `ISessionFactory`-scoped.

An `ISession`-scoped interceptor is specified when a session is opened using one of the overloaded ISessionFactory.OpenSession() methods accepting an `IInterceptor`.

```
ISession session = sf.OpenSession( new AuditInterceptor() );
```

An `ISessionFactory`-scoped interceptor is registered with the `Configuration` object prior to building the `ISessionFactory`. In this case, the supplied interceptor will be applied to all sessions opened from that `ISessionFactory`; this is true unless a session is opened explicitly specifying the interceptor to use. `ISessionFactory`-scoped interceptors must be thread safe, taking care to not store session-specific state since multiple sessions will use this interceptor (potentially) concurrently.

```
new Configuration().SetInterceptor( new AuditInterceptor() );
```

12.2. Event system

If you have to react to particular events in your persistence layer, you may also use the NHibernate2 *event* architecture. The event system can be used in addition or as a replacement for interceptors.

Essentially all of the methods of the `ISession` interface correlate to an event. You have a `LoadEvent`, a `FlushEvent`, etc (consult the XML configuration-file XSD or the `NHibernate.Event` namespace for the full list of defined event types). When a request is made of one of these methods, the `ISession` generates an appropriate event and passes it to the configured event listeners for that type. Out-of-the-box, these listeners implement the same processing in which those methods always resulted. However, you are free to implement a customization of one of the listener interfaces (i.e., the `LoadEvent` is processed by the registered implemenation of the `ILoadEventListener` interface), in which case their implementation would be responsible for processing any `Load()` requests made of the `ISession`.

The listeners should be considered effectively singletons; meaning, they are shared between requests, and thus should not save any state as instance variables.

A custom listener should implement the appropriate interface for the event it wants to process and/or extend one of the convenience base classes (or even the default event listeners used by NHibernate out-of-the-box as their methods are declared virtual for this purpose). Custom listeners can either be registered programmatically through the `Configuration` object, or specified in the NHibernate configuration XML. Here's an example of a custom load event listener:

```
public class MyLoadListener : ILoadEventListener
{
    // this is the single method defined by the LoadEventListener interface
    public void OnLoad(LoadEvent theEvent, LoadType loadType)
    {
        if ( !MySecurity.IsAuthorized( theEvent.EntityClassName, theEvent.EntityId ) ) {
            throw new MySecurityException("Unauthorized access");
        }
    }
}
```

You also need a configuration entry telling NHibernate to use the listener in addition to the default listener:

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="MyLoadListener"/>
            <listener class="NHibernate.Event.Default.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

Instead, you may register it programmatically:

```
Configuration cfg = new Configuration();
ILoadEventListener[] stack = new ILoadEventListener[] { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners.LoadEventListeners = stack;
```

Listeners registered declaratively cannot share instances. If the same class name is used in multiple `<listener/>` elements, each reference will result in a separate instance of that class. If you need the capability to share listener instances between listener types you must use the programmatic registration approach.

Why implement an interface and define the specific type during configuration? Well, a listener implementation could implement multiple event listener interfaces. Having the type additionally defined during registration makes it easier to turn custom listeners on or off during configuration.

Chapter 13. Batch processing

A naive approach to inserting 100 000 rows in the database using NHibernate might look like this:

```
ISession session = sessionFactory.OpenSession();
ITransaction tx = session.BeginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.Save(customer);
}
tx.Commit();
session.Close();
```

This would fall over with an `OutOfMemoryException` somewhere around the 50 000th row. That's because NHibernate caches all the newly inserted `Customer` instances in the session-level cache.

In this chapter we'll show you how to avoid this problem. First, however, if you are doing batch processing, it is absolutely critical that you enable the use of ADO batching, if you intend to achieve reasonable performance. Set the ADO batch size to a reasonable number (say, 10-50):

```
adonet.batch_size 20
```

Note that NHibernate disables insert batching at the ADO level transparently if you use an `identiy` identifier generator.

You also might like to do this kind of work in a process where interaction with the second-level cache is completely disabled:

```
cache.use_second_level_cache false
```

However, this is not absolutely necessary, since we can explicitly set the `CacheMode` to disable interaction with the second-level cache.

13.1. Batch inserts

When making new objects persistent, you must `Flush()` and then `Clear()` the session regularly, to control the size of the first-level cache.

```
ISession session = sessionFactory.openSession();
ITransaction tx = session.BeginTransaction();
```

```
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.Save(customer);
    if ( i % 20 == 0 ) { //20, same as the ADO batch size
        //flush a batch of inserts and release memory:
        session.Flush();
        session.Clear();
    }
}

tx.Commit();
session.Close();
```

13.2. The StatelessSession interface

Alternatively, NHibernate provides a command-oriented API that may be used for streaming data to and from the database in the form of detached objects. A `IStatelessSession` has no persistence context associated with it and does not provide many of the higher-level life cycle semantics. In particular, a stateless session does not implement a first-level cache nor interact with any second-level or query cache. It does not implement transactional write-behind or automatic dirty checking. Operations performed using a stateless session do not ever cascade to associated instances. Collections are ignored by a stateless session. Operations performed via a stateless session bypass NHibernate's event model and interceptors. Stateless sessions are vulnerable to data aliasing effects, due to the lack of a first-level cache. A stateless session is a lower-level abstraction, much closer to the underlying ADO.

```
IStatelessSession session = sessionFactory.OpenStatelessSession();
ITransaction tx = session.BeginTransaction();

var customers = session.GetNamedQuery("GetCustomers")
    .Enumerable<Customer>();
while ( customers.MoveNext() ) {
    Customer customer = customers.Current;
    customer.updateStuff(...);
    session.Update(customer);
}

tx.Commit();
session.Close();
```

Note that in this code example, the `Customer` instances returned by the query are immediately detached. They are never associated with any persistence context.

The `insert(), update()` and `delete()` operations defined by the `StatelessSession` interface are considered to be direct database row-level operations, which result in immediate execution of a SQL `INSERT, UPDATE` or `DELETE` respectively. Thus, they have very different semantics to the `Save(), SaveOrUpdate()` and `Delete()` operations defined by the `ISession` interface.

13.3. DML-style operations

As already discussed, automatic and transparent object/relational mapping is concerned with the management of object state. This implies that the object state is available in memory, hence manipulating (using the SQL `Data Manipulation Language` (DML) statements: `INSERT`, `UPDATE`, `DELETE`) data directly in the database will not affect in-memory state. However, NHibernate provides methods for bulk SQL-style DML statement execution which are performed through the Hibernate Query Language (HQL).

The pseudo-syntax for `UPDATE` and `DELETE` statements is: `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`. Some points to note:

- In the from-clause, the FROM keyword is optional
- There can only be a single entity named in the from-clause; it can optionally be aliased. If the entity name is aliased, then any property references must be qualified using that alias; if the entity name is not aliased, then it is illegal for any property references to be qualified.
- No joins (either implicit or explicit) can be specified in a bulk HQL query. Sub-queries may be used in the where-clause; the subqueries, themselves, may contain joins.
- The where-clause is also optional.

As an example, to execute an HQL `UPDATE`, use the `IQuery.ExecuteUpdate()` method:

```
ISession session = sessionFactory.OpenSession();
ITransaction tx = session.BeginTransaction();
```

```
string hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or string hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.CreateQuery( hqlUpdate )
        .SetString( "newName", newName )
        .SetString( "oldName", oldName )
        .ExecuteUpdate();
tx.Commit();
session.Close();
```

HQL `UPDATE` statements, by default do not effect the [version](#) or the [timestamp](#) property values for the affected entities. However, you can force NHibernate to properly reset the `version` or `timestamp` property values through the use of a `versioned update`. This is achieved by adding the `VERSIONED` keyword after the `UPDATE` keyword.

```
ISession session = sessionFactory.OpenSession();
ITransaction tx = session.BeginTransaction();
string hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.CreateQuery( hqlUpdate )
        .SetString( "newName", newName )
        .SetString( "oldName", oldName )
        .ExecuteUpdate();
tx.Commit();
session.Close();
```

Note that custom version types (`NHibernate.Usertype.IUserVersionType`) are not allowed in conjunction with a `update versioned` statement.

To execute an HQL `DELETE`, use the same `IQuery.ExecuteUpdate()` method:

```
ISession session = sessionFactory.OpenSession();
ITransaction tx = session.BeginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.CreateQuery( hqlDelete )
        .SetString( "oldName", oldName )
        .ExecuteUpdate();
tx.Commit();
session.Close();
```

The `int` value returned by the `IQuery.ExecuteUpdate()` method indicate the number of entities effected by the operation. Consider this may or may not correlate to the number of rows effected in the database. An HQL bulk operation might result in multiple actual SQL statements being executed, for joined-subclass, for example. The returned number indicates the number of actual entities affected by the statement. Going back to the example of joined-subclass, a delete against one of the subclasses may actually result in deletes against not just the table to which that subclass is mapped, but also the "root" table and potentially joined-subclass tables further down the inheritence hierarchy.

The pseudo-syntax for `INSERT` statements is: `INSERT INTO EntityName properties_list select_statement`. Some points to note:

- Only the INSERT INTO ... SELECT ... form is supported; not the INSERT INTO ... VALUES ... form.

  The properties_list is analogous to the `column speficiation` in the SQL `INSERT` statement. For entities involved in mapped inheritence, only properties directly defined on that given class-level can be used in the properties_list. Superclass properties are not allowed; and subclass properties do not make sense. In other words, `INSERT` statements are inherently non-polymorphic.

- select_statement can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. Note however that this might cause problems between NHibernate `Type`s which are *equivalent* as opposed to *equal*. This might cause issues with mismatches between a property defined as a `NHibernate.Type.DateType` and a property defined as a `NHibernate.Type.TimestampType`, even though the database might not make a distinction or might be able to handle the conversion.

- For the id property, the insert statement gives you two options. You can either explicitly specify the id property in the properties_list (in which case its value is taken from the corresponding select expression) or omit it from the properties_list (in which case a generated value is used). This later option is only available when using id generators that operate in the database; attempting to use this option with any "in memory" type generators will cause an exception during parsing. Note that for the purposes of this discussion, in-database generators are considered to be `NHibernate.Id.SequenceGenerator` (and its subclasses) and any implementors of `NHibernate.Id.IPostInsertIdentifierGenerator`. The most notable exception here is `NHibernate.Id.TableHiLoGenerator`, which cannot be used because it does not expose a selectable way to get its values.

- For properties mapped as either `version` or `timestamp`, the insert statement gives you two options. You can either specify the property in the properties_list (in which case its value is taken from the corresponding select expressions) or omit it from the properties_list (in which case the `seed value` defined by the `NHibernate.Type.IVersionType` is used).

An example HQL `INSERT` statement execution:

```
ISession session = sessionFactory.OpenSession();
ITransaction tx = session.BeginTransaction();

var hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...";
int createdEntities = s.CreateQuery( hqlInsert )
        .ExecuteUpdate();
tx.Commit();
session.Close();
```

Chapter 14. HQL: The Hibernate Query Language

NHibernate is equiped with an extremely powerful query language that (quite intentionally) looks very much like SQL. But don't be fooled by the syntax; HQL is fully object-oriented, understanding notions like inheritence, polymorphism and association.

14.1. Case Sensitivity

Queries are case-insensitive, except for names of .NET classes and properties. So `SeLeCT` is the same as `sELEct` is the same as `SELECT` but `Eg.FOO` is not `Eg.Foo` and `foo.barSet` is not `foo.BARSET`.

This manual uses lowercase HQL keywords. Some users find queries with uppercase keywords more readable, but we find this convention ugly when embedded in C# code.

14.2. The from clause

The simplest possible NHibernate query is of the form:

```
from Eg.Cat
```

which simply returns all instances of the class `Eg.Cat`.

Most of the time, you will need to assign an *alias*, since you will want to refer to the `Cat` in other parts of the query.

```
from Eg.Cat as cat
```

This query assigns the alias `cat` to `Cat` instances, so we could use that alias later in the query. The `as` keyword is optional; we could also write:

```
from Eg.Cat cat
```

Multiple classes may appear, resulting in a cartesian product or "cross" join.

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

It is considered good practice to name query aliases using an initial lowercase, consistent with naming standards for local variables (eg. `domesticCat`).

14.3. Associations and joins

We may also assign aliases to associated entities, or even to elements of a collection of values, using a `join`.

```
from Eg.Cat as cat
    inner join cat.Mate as mate
    left outer join cat.Kittens as kitten

from Eg.Cat as cat left join cat.Mate.Kittens as kittens

from Formula form full join form.Parameter param
```

The supported join types are borrowed from ANSI SQL

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (not usually useful)

The `inner join`, `left outer join` and `right outer join` constructs may be abbreviated.

```
from Eg.Cat as cat
    join cat.Mate as mate
    left join cat.Kittens as kitten
```

In addition, a "fetch" join allows associations or collections of values to be initialized along with their parent objects, using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections. See Section 19.1, “Fetching strategies” for more information.

```
from Eg.Cat as cat
    inner join fetch cat.Mate
    left join fetch cat.Kittens
```

The associated objects are not returned directly in the query results. Instead, they may be accessed via the parent object.

It is possible to create a cartesian product by join fetching more than one collection in a query, so take care in this case. Join fetching multiple collection roles is also disabled for bag mappings. Note also that the `fetch` construct may not be used in queries called using `Enumerable()`. Finally, note that `full join fetch` and `right join fetch` are not meaningful.

14.4. The select clause

The `select` clause picks which objects and properties to return in the query result set. Consider:

```
select mate
from Eg.Cat as cat
    inner join cat.Mate as mate
```

The query will select `Mate`s of other `Cat`s. Actually, you may express this query more compactly as:

```
select cat.Mate from Eg.Cat cat
```

You may even select collection elements, using the special `elements` function. The following query returns all kittens of any cat.

```
select elements(cat.Kittens) from Eg.Cat cat
```

Queries may return properties of any value type including properties of component type:

```
select cat.Name from Eg.DomesticCat cat
where cat.Name like 'fri%'

select cust.Name.FirstName from Customer as cust
```

Queries may return multiple objects and/or properties as an array of type `object[]`

```
select mother, offspr, mate.Name
from Eg.DomesticCat as mother
    inner join mother.Mate as mate
    left outer join mother.Kittens as offspr
```

or as an actual typesafe object

```
select new Family(mother, mate, offspr)
from Eg.DomesticCat as mother
    join mother.Mate as mate
    left join mother.Kittens as offspr
```

assuming that the class `Family` has an appropriate constructor.

14.5. Aggregate functions

HQL queries may even return the results of aggregate functions on properties:

```
select avg(cat.Weight), sum(cat.Weight), max(cat.Weight), count(cat)
from Eg.Cat cat
```

Collections may also appear inside aggregate functions in the `select` clause.

```
select cat, count( elements(cat.Kittens) )
from Eg.Cat cat group by cat.Id, cat.Weight, ...
```

The supported aggregate functions are

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

The `distinct` and `all` keywords may be used and have the same semantics as in SQL.

```
select distinct cat.Name from Eg.Cat cat

select count(distinct cat.Name), count(cat) from Eg.Cat cat
```

14.6. Polymorphic queries

A query like:

```
from Eg.Cat as cat
```

returns instances not only of `Cat`, but also of subclasses like `DomesticCat`. NHibernate queries may name *any* .NET class or interface in the `from` clause. The query will return instances of all persistent classes that extend that class or implement the interface. The following query would return all persistent objects:

```
from System.Object o
```

The interface `INamed` might be implemented by various persistent classes:

```
from Eg.Named n, Eg.Named m where n.Name = m.Name
```

Note that these last two queries will require more than one SQL `SELECT`. This means that the `order by` clause does not correctly order the whole result set.

In order to use non-mapped base classes or interfaces in HQL queries, they have to be imported. See Section 5.1.21, “import” for more information.

14.7. The where clause

The `where` clause allows you to narrow the list of instances returned.

```
from Eg.Cat as cat where cat.Name='Fritz'
```

returns instances of `Cat` named 'Fritz'.

```
select foo
from Eg.Foo foo, Eg.Bar bar
where foo.StartDate = bar.Date
```

will return all instances of `Foo` for which there exists an instance of `Bar` with a `Date` property equal to the `StartDate` property of the `Foo`. Compound path expressions make the `where` clause extremely powerful. Consider:

```
from Eg.Cat cat where cat.Mate.Name is not null
```

This query translates to an SQL query with a table (inner) join. If you were to write something like

```
from Eg.Foo foo
where foo.Bar.Baz.Customer.Address.City is not null
```

you would end up with a query that would require four table joins in SQL.

The = operator may be used to compare not only properties, but also instances:

```
from Eg.Cat cat, Eg.Cat rival where cat.Mate = rival.Mate

select cat, mate
from Eg.Cat cat, Eg.Cat mate
where cat.Mate = mate
```

The special property (lowercase) `id` may be used to reference the unique identifier of an object. (You may also use its property name.)

```
from Eg.Cat as cat where cat.id = 123

from Eg.Cat as cat where cat.Mate.id = 69
```

The second query is efficient. No table join is required!

Properties of composite identifiers may also be used. Suppose `Person` has a composite identifier consisting of `Country` and `MedicareNumber`.

```
from Bank.Person person
where person.id.Country = 'AU'
    and person.id.MedicareNumber = 123456

from Bank.Account account
where account.Owner.id.Country = 'AU'
    and account.Owner.id.MedicareNumber = 123456
```

Once again, the second query requires no table join.

Likewise, the special property `class` accesses the discriminator value of an instance in the case of polymorphic persistence. A .Net class name embedded in the where clause will be translated to its discriminator value.

```
from Eg.Cat cat where cat.class = Eg.DomesticCat
```

You may also specify properties of components or composite user types (and of components of components, etc). Never try to use a path-expression that ends in a property of component type (as opposed to a property of a component). For example, if `store.Owner` is an entity with a component `Address`

```
store.Owner.Address.City    // okay
store.Owner.Address         // error!
```

An "any" type has the special properties `id` and `class`, allowing us to express a join in the following way (where `AuditLog.Item` is a property mapped with `<any>`).

```
from Eg.AuditLog log, Eg.Payment payment
where log.Item.class = 'Eg.Payment, Eg, Version=...' and log.Item.id = payment.id
```

Notice that `log.Item.class` and `payment.class` would refer to the values of completely different database columns in the above query.

14.8. Expressions

Expressions allowed in the `where` clause include most of the kind of things you could write in SQL:

- mathematical operators `+, -, *, /`
- binary comparison operators `=, >=, <=, <>, !=, like`
- logical operations `and, or, not`
- string concatenation ||
- SQL scalar functions like `upper()` and `lower()`
- Parentheses `( )` indicate grouping
- `in`, `between`, `is null`
- positional parameters `?`
- named parameters `:name`, `:start_date`, `:x1`

- SQL literals `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- Enumeration values and constants `Eg.Color.Tabby`

`in` and `between` may be used as follows:

```
from Eg.DomesticCat cat where cat.Name between 'A' and 'B'

from Eg.DomesticCat cat where cat.Name in ( 'Foo', 'Bar', 'Baz' )
```

and the negated forms may be written

```
from Eg.DomesticCat cat where cat.Name not between 'A' and 'B'

from Eg.DomesticCat cat where cat.Name not in ( 'Foo', 'Bar', 'Baz' )
```

Likewise, `is null` and `is not null` may be used to test for null values.

Booleans may be easily used in expressions by declaring HQL query substitutions in NHibernate configuration:

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

This will replace the keywords `true` and `false` with the literals `1` and `0` in the translated SQL from this HQL:

```
from Eg.Cat cat where cat.Alive = true
```

You may test the size of a collection with the special property `size`, or the special `size()` function.

```
from Eg.Cat cat where cat.Kittens.size > 0

from Eg.Cat cat where size(cat.Kittens) > 0
```

For indexed collections, you may refer to the minimum and maximum indices using `minIndex` and `maxIndex`. Similarly, you may refer to the minimum and maximum elements of a collection of basic type using `minElement` and `maxElement`.

```
from Calendar cal where cal.Holidays.maxElement > current date
```

There are also functional forms (which, unlike the constructs above, are not case sensitive):

```
from Order order where maxindex(order.Items) > 100

from Order order where minelement(order.Items) > 10000
```

The SQL functions `any, some, all, exists, in` are supported when passed the element or index set of a collection (`elements` and `indices` functions) or the result of a subquery (see below).

```
select mother from Eg.Cat as mother, Eg.Cat as kit
where kit in elements(mother.Kittens)

select p from Eg.NameList list, Eg.Person p
where p.Name = some elements(list.Names)

from Eg.Cat cat where exists elements(cat.Kittens)

from Eg.Player p where 3 > all elements(p.Scores)

from Eg.Show show where 'fizard' in indices(show.Acts)
```

Note that these constructs - `size`, `elements`, `indices`, `minIndex`, `maxIndex`, `minElement`, `maxElement` - have certain usage restrictions:

- in a `where` clause: only for databases with subselects
- in a `select` clause: only `elements` and `indices` make sense

Elements of indexed collections (arrays, lists, maps) may be referred to by index (in a where clause only):

```
from Order order where order.Items[0].id = 1234

select person from Person person, Calendar calendar
where calendar.Holidays['national day'] = person.BirthDay
    and person.Nationality.Calendar = calendar

select item from Item item, Order order
```

```
where order.Items[ order.DeliveredItemIndices[0] ] = item and order.id = 11

select item from Item item, Order order
where order.Items[ maxindex(order.items) ] = item and order.id = 11
```

The expression inside `[]` may even be an arithmetic expression.

```
select item from Item item, Order order
where order.Items[ size(order.Items) - 1 ] = item
```

HQL also provides the built-in `index()` function, for elements of a one-to-many association or collection of values.

```
select item, index(item) from Order order
    join order.Items item
where index(item) < 5
```

Scalar SQL functions supported by the underlying database may be used

```
from Eg.DomesticCat cat where upper(cat.Name) like 'FRI%'
```

If you are not yet convinced by all this, think how much longer and less readable the following query would be in SQL:

```
select cust
from Product prod,
    Store store
    inner join store.Customers cust
where prod.Name = 'widget'
    and store.Location.Name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.CurrentOrder.LineItems)
```

Hint: something like

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

14.9. The order by clause

The list returned by a query may be ordered by any property of a returned class or components:

```
from Eg.DomesticCat cat
order by cat.Name asc, cat.Weight desc, cat.Birthdate
```

The optional `asc` or `desc` indicate ascending or descending order respectively.

14.10. The group by clause

A query that returns aggregate values may be grouped by any property of a returned class or components:

```
select cat.Color, sum(cat.Weight), count(cat)
from Eg.Cat cat
group by cat.Color

select foo.id, avg( elements(foo.Names) ), max( indices(foo.Names) )
from Eg.Foo foo
group by foo.id
```

Note: You may use the `elements` and `indices` constructs inside a select clause, even on databases with no subselects.

A `having` clause is also allowed.

```
select cat.color, sum(cat.Weight), count(cat)
from Eg.Cat cat
group by cat.Color
having cat.Color in (Eg.Color.Tabby, Eg.Color.Black)
```

SQL functions and aggregate functions are allowed in the `having` and `order by` clauses, if supported by the underlying database (ie. not in MySQL).

```
select cat
from Eg.Cat cat
    join cat.Kittens kitten
group by cat.Id, cat.Name, cat.Other, cat.Properties
having avg(kitten.Weight) > 100
order by count(kitten) asc, sum(kitten.Weight) desc
```

Note that neither the `group by` clause nor the `order by` clause may contain arithmetic expressions. Also note that NHibernate currently does not expand a grouped entity, so you can't write `group by cat` if all properties of `cat` are non-aggregated. You have to list all non-aggregated properties explicitly.

14.11. Subqueries

For databases that support subselects, NHibernate supports subqueries within queries. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed.

```
from Eg.Cat as fatcat
where fatcat.Weight > (
    select avg(cat.Weight) from Eg.DomesticCat cat
)

from Eg.DomesticCat as cat
where cat.Name = some (
    select name.NickName from Eg.Name as name
)

from Eg.Cat as cat
where not exists (
    from eg.Cat as mate where mate.Mate = cat
)

from Eg.DomesticCat as cat
where cat.Name not in (
    select name.NickName from Eg.Name as name
)
```

14.12. HQL examples

NHibernate queries can be quite powerful and complex. In fact, the power of the query language is one of NHibernate's main selling points. Here are some example queries very similar to queries that I used on a recent project. Note that most queries you will write are much simpler than these!

The following query returns the order id, number of items and total value of the order for all unpaid orders for a particular customer and given minimum total value, ordering the results by total value. In determining the prices, it uses the current catalog. The resulting SQL query, against the `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` and `PRICE` tables has four inner joins and an (uncorrelated) subselect.

```
select order.id, sum(price.Amount), count(item)
from Order as order
    join order.LineItems as item
    join item.Product as product,
    Catalog as catalog
    join catalog.Prices as price
where order.Paid = false
    and order.Customer = :customer
    and price.Product = product
    and catalog.EffectiveDate < sysdate
    and catalog.EffectiveDate >= all (
        select cat.EffectiveDate
        from Catalog as cat
        where cat.EffectiveDate < sysdate
    )
group by order
having sum(price.Amount) > :minAmount
order by sum(price.Amount) desc
```

What a monster! Actually, in real life, I'm not very keen on subqueries, so my query was really more like this:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.LineItems as item
    join item.Product as product,
    Catalog as catalog
    join catalog.Prices as price
where order.Paid = false
    and order.Customer = :customer
    and price.Product = product
    and catalog = :currentCatalog
group by order
having sum(price.Amount) > :minAmount
order by sum(price.Amount) desc
```

The next query counts the number of payments in each status, excluding all payments in the `AwaitingApproval` status where the most recent status change was made by the current user. It translates to an SQL query with two inner joins and a correlated subselect against the `PAYMENT`, `PAYMENT_STATUS` and `PAYMENT_STATUS_CHANGE` tables.

```
select count(payment), status.Name
from Payment as payment
    join payment.CurrentStatus as status
    join payment.StatusChanges as statusChange
where payment.Status.Name <> PaymentStatus.AwaitingApproval
    or (
        statusChange.TimeStamp = (
            select max(change.TimeStamp)
            from PaymentStatusChange change
            where change.Payment = payment
        )
        and statusChange.User <> :currentUser
    )
group by status.Name, status.SortOrder
order by status.SortOrder
```

If I would have mapped the `StatusChanges` collection as a list, instead of a set, the query would have been much simpler to write.

```
select count(payment), status.Name
from Payment as payment
    join payment.CurrentStatus as status
where payment.Status.Name <> PaymentStatus.AwaitingApproval
    or payment.StatusChanges[ maxIndex(payment.StatusChanges) ].User <> :currentUser
group by status.Name, status.SortOrder
order by status.SortOrder
```

The next query uses the MS SQL Server `isNull()` function to return all the accounts and unpaid payments for the organization to which the current user belongs. It translates to an SQL query with three inner joins, an outer join and a subselect against the `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` and `ORG_USER` tables.

```
select account, payment
from Account as account
    left outer join account.Payments as payment
where :currentUser in elements(account.Holder.Users)
    and PaymentStatus.Unpaid = isNull(payment.CurrentStatus.Name, PaymentStatus.Unpaid)
order by account.Type.SortOrder, account.AccountNumber, payment.DueDate
```

For some databases, we would need to do away with the (correlated) subselect.

```
select account, payment
from Account as account
    join account.Holder.Users as user
    left outer join account.Payments as payment
where :currentUser = user
    and PaymentStatus.Unpaid = isNull(payment.CurrentStatus.Name, PaymentStatus.Unpaid)
order by account.Type.SortOrder, account.AccountNumber, payment.DueDate
```

14.13. Tips & Tricks

You can count the number of query results without actually returning them:

```
int count = (int) session.CreateQuery("select count(*) from ....").UniqueResult();
```

To order a result by the size of a collection, use the following query:

```
select usr.id, usr.Name
from User as usr
    left join usr.Messages as msg
group by usr.id, usr.Name
order by count(msg)
```

If your database supports subselects, you can place a condition upon selection size in the where clause of your query:

```
from User usr where size(usr.Messages) >= 1
```

If your database doesn't support subselects, use the following query:

```
select usr.id, usr.Name
from User usr
    join usr.Messages msg
group by usr.id, usr.Name
having count(msg) >= 1
```

As this solution can't return a `User` with zero messages because of the inner join, the following form is also useful:

```
select usr.id, usr.Name
from User as usr
    left join usr.Messages as msg
group by usr.id, usr.Name
having count(msg) = 0
```

Properties of an object can be bound to named query parameters:

```
IQuery q = s.CreateQuery("from foo in class Foo where foo.Name=:Name and foo.Size=:Size");
q.SetProperties(fooBean); // fooBean has properties Name and Size
IList foos = q.List();
```

Collections are pageable by using the `IQuery` interface with a filter:

```
IQuery q = s.CreateFilter( collection, "" ); // the trivial filter
q.setMaxResults(PageSize);
q.setFirstResult(PageSize * pageNumber);
IList page = q.List();
```

Collection elements may be ordered or grouped using a query filter:

```
ICollection orderedCollection = s.Filter( collection, "order by this.Amount" );
ICollection counts = s.Filter( collection, "select this.Type, count(this) group by this.Type" );
```

Chapter 15. Criteria Queries

NHibernate features an intuitive, extensible criteria query API.

15.1. Creating an `ICriteria` instance

The interface `NHibernate.ICriteria` represents a query against a particular persistent class. The `ISession` is a factory for `ICriteria` instances.

```
ICriteria crit = sess.CreateCriteria<Cat>();
crit.SetMaxResults(50);
List cats = crit.List();
```

15.2. Narrowing the result set

An individual query criterion is an instance of the interface `NHibernate.Expression.ICriterion`. The class `NHibernate.Expression.Expression` defines factory methods for obtaining certain built-in `ICriterion` types.

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
    .Add( Expression.Like("Name", "Fritz%") )
    .Add( Expression.Between("Weight", minWeight, maxWeight) )
    .List<Cat>();
```

Expressions may be grouped logically.

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
```

```
    .Add( Expression.Like("Name", "Fritz%") )
    .Add( Expression.Or(
        Expression.Eq( "Age", 0 ),
        Expression.IsNull("Age")
    ) )
    .List<Cat>();
```

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
    .Add( Expression.In( "Name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .Add( Expression.Disjunction()
        .Add( Expression.IsNull("Age") )
        .Add( Expression.Eq("Age", 0 ) )
        .Add( Expression.Eq("Age", 1 ) )
        .Add( Expression.Eq("Age", 2 ) )
    ) )
    .List<Cat>();
```

There are quite a range of built-in criterion types (`Expression` subclasses), but one that is especially useful lets you specify SQL directly.

```
        // Create a string parameter for the SqlString below
        IList<Cat> cats = sess.CreateCriteria<Cat>()
            .Add( Expression.Sql("lower({alias}.Name) like lower(?)", "Fritz%", NHibernateUtil.String )
            .List<Cat>();
```

The `{alias}` placeholder with be replaced by the row alias of the queried entity.

15.3. Ordering the results

You may order the results using `NHibernate.Expression.Order`.

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
    .Add( Expression.Like("Name", "F%")
    .AddOrder( Order.Asc("Name") )
    .AddOrder( Order.Desc("Age") )
    .SetMaxResults(50)
    .List<Cat>();
```

15.4. Associations

You may easily specify constraints upon related entities by navigating associations using `CreateCriteria()`.

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
    .Add( Expression.Like("Name", "F%")
    .CreateCriteria("Kittens")
        .Add( Expression.Like("Name", "F%") )
    .List<Cat>();
```

note that the second `CreateCriteria()` returns a new instance of `ICriteria`, which refers to the elements of the `Kittens` collection.

The following, alternate form is useful in certain circumstances.

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
    .CreateAlias("Kittens", "kt")
    .CreateAlias("Mate", "mt")
    .Add( Expression.EqProperty("kt.Name", "mt.Name") )
    .List<Cat>();
```

(`CreateAlias()` does not create a new instance of `ICriteria`.)

Note that the kittens collections held by the `Cat` instances returned by the previous two queries are *not* pre-filtered by the criteria! If you wish to retrieve just the kittens that match the criteria, you must use `SetResultTransformer(Transformers.AliasToEntityMap)`.

```
IList cats = sess.CreateCriteria<Cat>()
    .CreateCriteria("Kittens", "kt")
        .Add( Expression.Eq("Name", "F%") )
    .SetResultTransformer(Transformers.AliasToEntityMap)
    .List();
foreach ( IDictionary map in cats )
{
    Cat cat = (Cat) map[CriteriaUtil.RootAlias];
    Cat kitten = (Cat) map["kt"];
}
```

15.5. Dynamic association fetching

You may specify association fetching semantics at runtime using `SetFetchMode()`.

```
IList<Cat> cats = sess.CreateCriteria<Cat>()
    .Add( Expression.Like("Name", "Fritz%") )
    .SetFetchMode("Mate", FetchMode.Eager)
    .SetFetchMode("Kittens", FetchMode.Eager)
    .List<Cat>();
```

This query will fetch both `Mate` and `Kittens` by outer join. See Section 19.1, “Fetching strategies” for more information.

15.6. Example queries

The class `NHibernate.Expression.Example` allows you to construct a query criterion from a given instance.

```
Cat cat = new Cat();
cat.Sex = 'F';
cat.Color = Color.Black;
List<Cat> results = session.CreateCriteria<Cat>()
    .Add( Example.Create(cat) )
    .List<Cat>();
```

Version properties, identifiers and associations are ignored. By default, null-valued properties and properties which return an empty string from the call to `ToString()` are excluded.

You can adjust how the `Example` is applied.

```
Example example = Example.Create(cat)
    .ExcludeZeroes()           //exclude null- or zero-valued properties
    .ExcludeProperty("Color")  //exclude the property named "color"
    .IgnoreCase()              //perform case insensitive string comparisons
    .EnableLike();             //use like for string comparisons
IList<Cat> results = session.CreateCriteria<Cat>()
    .Add(example)
    .List<Cat>();
```

You can even use examples to place criteria upon associated objects.

```
IList<Cat> results = session.CreateCriteria<Cat>()
    .Add( Example.Create(cat) )
    .CreateCriteria("Mate")
        .Add( Example.Create( cat.Mate ) )
    .List<Cat>();
```

15.7. Projections, aggregation and grouping

The class `NHibernate.Expression.Projections` is a factory for `IProjection` instances. We apply a projection to a query by calling `SetProjection()`.

```
IList results = session.CreateCriteria<Cat>()
    .SetProjection( Projections.RowCount() )
    .Add( Expression.Eq("Color", Color.BLACK) )
    .List();
```

```
List results = session.CreateCriteria<Cat>()
    .SetProjection( Projections.ProjectionList()
        .Add( Projections.RowCount() )
        .Add( Projections.Avg("Weight") )
        .Add( Projections.Max("Weight") )
        .Add( Projections.GroupProperty("Color") )
    )
    .List();
```

There is no explicit "group by" necessary in a criteria query. Certain projection types are defined to be *grouping projections*, which also appear in the SQL `group by` clause.

An alias may optionally be assigned to a projection, so that the projected value may be referred to in restrictions or orderings. Here are two different ways to do this:

```
IList results = session.CreateCriteria<Cat>()
```

```
    .SetProjection( Projections.Alias( Projections.GroupProperty("Color"), "colr" ) )
    .AddOrder( Order.Asc("colr") )
    .List();
```

```
IList results = session.CreateCriteria<Cat>()
    .SetProjection( Projections.GroupProperty("Color").As("colr") )
    .AddOrder( Order.Asc("colr") )
    .List();
```

The `Alias()` and `As()` methods simply wrap a projection instance in another, aliased, instance of `IProjection`. As a shortcut, you can assign an alias when you add the projection to a projection list:

```
IList results = session.CreateCriteria<Cat>()
    .SetProjection( Projections.ProjectionList()
        .Add( Projections.RowCount(), "catCountByColor" )
        .Add( Projections.Avg("Weight"), "avgWeight" )
        .Add( Projections.Max("Weight"), "maxWeight" )
        .Add( Projections.GroupProperty("Color"), "color" )
    )
    .AddOrder( Order.Desc("catCountByColor") )
    .AddOrder( Order.Desc("avgWeight") )
    .List();
```

```
IList results = session.CreateCriteria<DomesticCat>("cat")
    .CreateAlias("kittens", "kit")
    .SetProjection( Projections.ProjectionList()
        .Add( Projections.Property("cat.Name"), "catName" )
        .Add( Projections.Property("kit.Name"), "kitName" )
    )
    .AddOrder( Order.Asc("catName") )
    .AddOrder( Order.Asc("kitName") )
    .List();
```

15.8. Detached queries and subqueries

The `DetachedCriteria` class lets you create a query outside the scope of a session, and then later execute it using some arbitrary `ISession`.

```
DetachedCriteria query = DetachedCriteria.For<Cat>()
    .Add( Expression.Eq("sex", 'F') );

ISession session = ....;
ITransaction txn = session.BeginTransaction();
IList results = query.GetExecutableCriteria(session).SetMaxResults(100).List();
txn.Commit();
session.Close();
```

A `DetachedCriteria` may also be used to express a subquery. ICriterion instances involving subqueries may be obtained via `Subqueries` .

```
DetachedCriteria avgWeight = DetachedCriteria.For<Cat>()
    .SetProjection( Projections.Avg("Weight") );
session.CreateCriteria<Cat>()
    .Add( Subqueries.Gt("Weight", avgWeight) )
    .List();
```

```
DetachedCriteria weights = DetachedCriteria.For<Cat>()
    .SetProjection( Projections.Property("Weight") );
session.CreateCriteria<Cat>()
    .add( Subqueries.GeAll("Weight", weights) )
    .list();
```

Even correlated subqueries are possible:

```
DetachedCriteria avgWeightForSex = DetachedCriteria.For<Cat>("cat2")
    .SetProjection( Projections.Avg("Weight") )
    .Add( Expression.EqProperty("cat2.Sex", "cat.Sex") );
session.CreateCriteria<Cat>("cat")
    .Add( Subqueries.Gt("weight", avgWeightForSex) )
    .List();
```

Chapter 16. QueryOver Queries

The ICriteria API is NHibernate's implementation of Query Object. NHibernate 3.0 introduces the QueryOver api, which combines the use of Extension Methods and Lambda Expressions (both new in .Net 3.5) to provide a statically typesafe wrapper round the ICriteria API.

QueryOver uses Lambda Expressions to provide some extra syntax to remove the 'magic strings' from your ICriteria queries.

So, for example:

```
.Add(Expression.Eq("Name", "Smith"))
```

becomes:

```
.Where<Person>(p => p.Name == "Smith")
```

With this kind of syntax there are no 'magic strings', and refactoring tools like 'Find All References', and 'Refactor->Rename' work perfectly.

Note: QueryOver is intended to remove the references to 'magic strings' from the ICriteria API while maintaining it's opaqueness. It is not a LINQ provider; NHibernate has a built-in Linq provider for this.

16.1. Structure of a Query

Queries are created from an ISession using the syntax:

```
IList<Cat> cats =
    session.QueryOver<Cat>()
        .Where(c => c.Name == "Max")
        .List();
```

Detached QueryOver (analagous to DetachedCriteria) can be created, and then used with an ISession using:

```
QueryOver<Cat> query =
    QueryOver.Of<Cat>()
        .Where(c => c.Name == "Paddy");

IList<Cat> cats =
    query.GetExecutableQueryOver(session)
        .List();
```

Queries can be built up to use restrictions, projections, and ordering using a fluent inline syntax:

```
var catNames =
    session.QueryOver<Cat>()
        .WhereRestrictionOn(c => c.Age).IsBetween(2).And(8)
        .Select(c => c.Name)
        .OrderBy(c => c.Name).Asc
        .List<string>();
```

16.2. Simple Expressions

The Restrictions class (used by ICriteria) has been extended to include overloads that allow Lambda Expression syntax. The Where() method works for simple expressions (<, <=, ==, !=, >, >=) so instead of:

```
ICriterion equalCriterion = Restrictions.Eq("Name", "Max")
```

You can write:

```
ICriterion equalCriterion = Restrictions.Where<Cat>(c => c.Name == "Max")
```

Since the QueryOver class (and IQueryOver interface) is generic and knows the type of the query, there is an inline syntax for restrictions that does not require the additional qualification of class name. So you can also write:

```
var cats =
    session.QueryOver<Cat>()
        .Where(c => c.Name == "Max")
        .And(c => c.Age > 4)
        .List();
```

Note, the methods Where() and And() are semantically identical; the And() method is purely to allow QueryOver to look similar to HQL/SQL.

Boolean comparisons can be made directly instead of comparing to true/false:

```
        .Where(p => p.IsParent)
        .And(p => !p.IsRetired)
```

Simple expressions can also be combined using the || and && operators. So ICriteria like:

```
        .Add(Restrictions.And(
                Restrictions.Eq("Name", "test name"),
                Restrictions.Or(
                    Restrictions.Gt("Age", 21),
                    Restrictions.Eq("HasCar", true))))
```

Can be written in QueryOver as:

```
        .Where(p => p.Name == "test name" && (p.Age > 21 || p.HasCar))
```

Each of the corresponding overloads in the QueryOver API allows the use of regular ICriterion to allow access to private properties.

```
        .Where(Restrictions.Eq("Name", "Max"))
```

It is worth noting that the QueryOver API is built on top of the ICriteria API. Internally the structures are the same, so at runtime the statement below, and the statement above, are stored as exactly the same ICriterion. The actual Lambda Expression is not stored in the query.

```
        .Where(c => c.Name == "Max")
```

16.3. Additional Restrictions

Some SQL operators/functions do not have a direct equivalent in C#. (e.g., the SQL `where name like '%anna%'`). These operators have overloads for QueryOver in the Restrictions class, so you can write:

```
        .Where(Restrictions.On<Cat>(c => c.Name).IsLike("%anna%"))
```

There is also an inline syntax to avoid the qualification of the type:

```
        .WhereRestrictionOn(c => c.Name).IsLike("%anna%")
```

While simple expressions (see above) can be combined using the || and && operators, this is not possible with the other restrictions. So this ICriteria:

```
        .Add(Restrictions.Or(
            Restrictions.Gt("Age", 5)
            Restrictions.In("Name", new string[] { "Max", "Paddy" })))
```

Would have to be written as:

```
        .Add(Restrictions.Or(
            Restrictions.Where<Cat>(c => c.Age > 5)
            Restrictions.On<Cat>(c => c.Name).IsIn(new string[] { "Max", "Paddy" })))
```

However, in addition to the additional restrictions factory methods, there are extension methods to allow a more concise inline syntax for some of the operators. So this:

```
        .WhereRestrictionOn(c => c.Name).IsLike("%anna%")
```

May also be written as:

```
        .Where(c => c..Name.IsLike("%anna%"))
```

16.4. Associations

QueryOver can navigate association paths using JoinQueryOver() (analagous to ICriteria.CreateCriteria() to create sub-criteria).

The factory method QuerOver<T>() on ISession returns an IQueryOver<T>. More accurately, it returns an IQueryOver<T,T> (which inherits from IQueryOver<T>).

An IQueryOver has two types of interest; the root type (the type of entity that the query returns), and the type of the 'current' entity being queried. For example, the following query uses a join to create a sub-QueryOver (analagous to creating sub-criteria in the ICriteria API):

```
IQueryOver<Cat,Kitten> catQuery =
    session.QueryOver<Cat>()
        .JoinQueryOver(c => c.Kittens)
            .Where(k => k.Name == "Tiddles");
```

The JoinQueryOver returns a new instance of the IQueryOver than has its root at the Kittens collection. The default type for restrictions is now Kitten (restricting on the name 'Tiddles' in the above example), while calling .List() will return an IList<Cat>. The type IQueryOver<Cat,Kitten> inherits from IQueryOver<Cat>.

Note, the overload for JoinQueryOver takes an IEnumerable<T>, and the C# compiler infers the type from that. If your collection type is not IEnumerable<T>, then you need to qualify the type of the sub-criteria:

```
IQueryOver<Cat,Kitten> catQuery =
    session.QueryOver<Cat>()
        .JoinQueryOver<Kitten>(c => c.Kittens)
            .Where(k => k.Name == "Tiddles");
```

The default join is an inner-join. Each of the additional join types can be specified using the methods `.Inner`, `.Left`, `.Right`, or `.Full`. For example, to left outer-join on Kittens use:

```
IQueryOver<Cat,Kitten> catQuery =
    session.QueryOver<Cat>()
        .Left.JoinQueryOver(c => c.Kittens)
            .Where(k => k.Name == "Tiddles");
```

16.5. Aliases

In the traditional ICriteria interface aliases are assigned using 'magic strings', however their value does not correspond to a name in the object domain. For example, when an alias is assigned using `.CreateAlias("Kitten", "kittenAlias")`, the string "kittenAlias" does not correspond to a property or class in the domain.

In QueryOver, aliases are assigned using an empty variable. The variable can be declared anywhere (but should be `null` at runtime). The compiler can then check the syntax against the variable is used correctly, but at runtime the variable is not evaluated (it's just used as a placeholder for the alias).

Each Lambda Expression function in QueryOver has a corresponding overload to allow use of aliases, and a .JoinAlias function to traverse associations using aliases without creating a sub-QueryOver.

```
Cat catAlias = null;
Kitten kittenAlias = null;

IQueryOver<Cat,Cat> catQuery =
    session.QueryOver<Cat>(() => catAlias)
        .JoinAlias(() => catAlias.Kittens, () => kittenAlias)
        .Where(() => catAlias.Age > 5)
        .And(() => kittenAlias.Name == "Tiddles");
```

16.6. Projections

Simple projections of the properties of the root type can be added using the `.Select` method which can take multiple Lambda Expression arguments:

```
IList selection =
    session.QueryOver<Cat>()
```

```
        .Select(
            c => c.Name,
            c => c.Age)
        .List<object[]>();
```

Because this query no longer returns a Cat, the return type must be explicitly specified. If a single property is projected, the return type can be specified using:

```
IList<int> ages =
    session.QueryOver<Cat>()
        .Select(c => c.Age)
        .List<int>();
```

However, if multiple properties are projected, then the returned list will contain object arrays, as per a projection in ICriteria. This could be fed into an anonymous type using:

```
var catDetails =
    session.QueryOver<Cat>()
        .Select(
            c => c.Name,
            c => c.Age)
        .List<object[]>()
        .Select(properties => new {
            CatName = (string)properties[0],
            CatAge = (int)properties[1],
            });

Console.WriteLine(catDetails[0].CatName);
Console.WriteLine(catDetails[0].CatAge);
```

Note that the second `.Select` call in this example is an extension method on IEnumerable<T> supplied in System.Linq; it is not part of NHibernate.

QueryOver allows arbitrary IProjection to be added (allowing private properties to be projected). The Projections factory class also has overloads to allow Lambda Expressions to be used:

```
IList selection =
    session.QueryOver<Cat>()
        .Select(Projections.ProjectionList()
            .Add(Projections.Property<Cat>(c => c.Name))
            .Add(Projections.Avg<Cat>(c => c.Age)))
        .List<object[]>();
```

In addition there is an inline syntax for creating projection lists that does not require the explicit class qualification:

```
IList selection =
    session.QueryOver<Cat>()
        .SelectList(list => list
            .Select(c => c.Name)
            .SelectAvg(c => c.Age))
        .List<object[]>();
```

Projections can also have arbitrary aliases assigned to them to allow result transformation. If there is a CatSummary DTO class defined as:

```
public class CatSummary
{
    public string Name { get; set; }
    public int AverageAge { get; set; }
}
```

... then aliased projections can be used with the AliasToBean<T> transformer:

```
CatSummary summaryDto = null;
IList<CatSummary> catReport =
    session.QueryOver<Cat>()
        .SelectList(list => list
            .SelectGroup(c => c.Name).WithAlias(() => summaryDto.Name)
            .SelectAvg(c => c.Age).WithAlias(() => summaryDto.AverageAge))
        .TransformUsing(Transformers.AliasToBean<CatSummary>())
        .List<CatSummary>();
```

16.7. Projection Functions

In addition to projecting properties, there are extension methods to allow certain common dialect-registered functions to be applied. For example you can write the following to extract just the year part of a date:

```
        .Where(p => p.BirthDate.YearPart() == 1971)
```

The functions can also be used inside projections:

```
        .Select(
            p => Projections.Concat(p.LastName, ", ", p.FirstName),
            p => p.Height.Abs())
```

16.8. Subqueries

The Subqueries factory class has overloads to allow Lambda Expressions to express sub-query restrictions. For example:

```
QueryOver<Cat> maximumAge =
    QueryOver.Of<Cat>()
        .SelectList(p => p.SelectMax(c => c.Age));

IList<Cat> oldestCats =
    session.QueryOver<Cat>()
        .Where(Subqueries.WhereProperty<Cat>(c => c.Age).Eq(maximumAge))
        .List();
```

The inline syntax allows you to use subqueries without requalifying the type:

```
IList<Cat> oldestCats =
    session.QueryOver<Cat>()
        .WithSubquery.WhereProperty(c => c.Age).Eq(maximumAge)
        .List();
```

There is an extension method `As()` on (a detached) QueryOver that allows you to cast it to any type. This is used in conjunction with the overloads `Where()`, `WhereAll()`, and `WhereSome()` to allow use of the built-in C# operators for comparison, so the above query can be written as:

```
IList<Cat> oldestCats =
    session.QueryOver<Cat>()
        .WithSubquery.Where(c => c.Age == maximumAge.As<int>())
        .List();
```

Chapter 17. Native SQL

You may also express queries in the native SQL dialect of your database. This is useful if you want to utilize database specific features such as query hints or the `CONNECT` keyword in Oracle. It also provides a clean migration path from a direct SQL/ADO.NET based application to NHibernate.

NHibernate allows you to specify handwritten SQL (including stored procedures) for all create, update, delete, and load operations.

17.1. Using an `ISQLQuery`

Execution of native SQL queries is controlled via the `ISQLQuery` interface, which is obtained by calling `ISession.CreateSQLQuery()`. The following describes how to use this API for querying.

17.1.1. Scalar queries

The most basic SQL query is to get a list of scalars (values).

```
sess.CreateSQLQuery("SELECT * FROM CATS")
 .AddScalar("ID", NHibernateUtil.Int64)
 .AddScalar("NAME", NHibernateUtil.String)
 .AddScalar("BIRTHDATE", NHibernateUtil.Date)
```

This query specified:

- the SQL query string
- the columns and types to return

This will return an IList of Object arrays (object[]) with scalar values for each column in the CATS table. Only these three columns will be returned, even though the query is using `*` and could return more than the three listed columns.

17.1.2. Entity queries

The above query was about returning scalar values, basically returning the "raw" values from the result set. The following shows how to get entity objects from a native SQL query via `AddEntity()`.

```
sess.CreateSQLQuery("SELECT * FROM CATS").AddEntity(typeof(Cat));
sess.CreateSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").AddEntity(typeof(Cat));
```

This query specified:

- the SQL query string
- the entity returned by the query

Assuming that Cat is mapped as a class with the columns ID, NAME and BIRTHDATE the above queries will both return an IList where each element is a Cat entity.

If the entity is mapped with a `many-to-one` to another entity it is required to also return its identifier when performing the native query, otherwise a database specific "column not found" error will occur. The additional columns will automatically be returned when using the * notation, but we prefer to be explicit as in the following example for a `many-to-one` to a `Dog`:

```
sess.CreateSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").AddEntity(typeof(Cat));
```

This will allow cat.Dog property access to function properly.

17.1.3. Handling associations and collections

It is possible to eagerly join in the `Dog` to avoid the possible extra roundtrip for initializing the proxy. This is done via the `AddJoin()` method, which allows you to join in an association or collection.

```
sess.CreateSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d WHERE c.DOG_ID = d.D_ID")
 .AddEntity("cat", typeof(Cat))
 .AddJoin("cat.Dog");
```

In this example the returned `Cat`'s will have their `Dog` property fully initialized without any extra roundtrip to the database. Notice that we added a alias name ("cat") to be able to specify the target property path of the join. It is possible to do the same eager joining for collections, e.g. if the `Cat` had a one-to-many to `Dog` instead.

```
sess.CreateSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE c.ID = d.CAT_ID")
 .AddEntity("cat", typeof(Cat))
 .AddJoin("cat.Dogs");
```

At this stage we are reaching the limits of what is possible with native queries without starting to enhance the SQL queries to make them usable in NHibernate; the problems start to arise when returning multiple entities of the same type or when the default alias/column names are not enough.

17.1.4. Returning multiple entities

Until now the result set column names are assumed to be the same as the column names specified in the mapping document. This can be problematic for SQL queries which join multiple tables, since the same column names may appear in more than one table.

Column alias injection is needed in the following query (which most likely will fail):

```
sess.CreateSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .AddEntity("cat", typeof(Cat))
 .AddEntity("mother", typeof(Cat))
```

The intention for this query is to return two Cat instances per row, a cat and its mother. This will fail since there is a conflict of names since they are mapped to the same column names and on some databases the returned column aliases will most likely be on the form "c.ID", "c.NAME", etc. which are not equal to the columns specificed in the mappings ("ID" and

"NAME").

The following form is not vulnerable to column name duplication:

```
sess.CreateSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .AddEntity("cat", typeof(Cat))
 .AddEntity("mother", typeof(Cat))
```

This query specified:

- the SQL query string, with placeholders for NHibernate to inject column aliases
- the entities returned by the query

The {cat.*} and {mother.*} notation used above is a shorthand for "all properties". Alternatively, you may list the columns explicity, but even in this case we let NHibernate inject the SQL column aliases for each property. The placeholder for a column alias is just the property name qualified by the table alias. In the following example, we retrieve Cats and their mothers from a different table (cat_log) to the one declared in the mapping metadata. Notice that we may even use the property aliases in the where clause if we like.

```
String sql = "SELECT ID as {c.Id}, NAME as {c.Name}, " +
         "BIRTHDATE as {c.BirthDate}, MOTHER_ID as {c.Mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.Mother} = c.ID";

IList loggedCats = sess.CreateSQLQuery(sql)
        .AddEntity("cat", typeof(Cat))
        .AddEntity("mother", typeof(Cat)).List();
```

17.1.4.1. Alias and property references

For most cases the above alias injection is needed, but for queries relating to more complex mappings like composite properties, inheritance discriminators, collections etc. there are some specific aliases to use to allow NHibernate to inject the proper aliases.

The following table shows the different possibilities of using the alias injection. Note: the alias names in the result are examples, each alias will have a unique and probably different name when used.

Table 17.1. Alias injection names

Description	Syntax	Example
A simple property	`{[aliasname].[propertyname]}`	`A_NAME as {item.Name}`
A composite property	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.Amount.Currency}, VALUE as {item.Amount.Value}`
Discriminator of an entity	`{[aliasname].class}`	`DISC as {item.class}`
All properties of an entity	`{[aliasname].*}`	`{item.*}`
A collection key	`{[aliasname].key}`	`ORGID as {coll.key}`
The id of an collection	`{[aliasname].id}`	`EMPID as {coll.id}`
The element of an collection	`{[aliasname].element}`	`XID as {coll.element}`
property of the element in the collection	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.Name}`
All properties of the element in the collection	`{[aliasname].element.*}`	`{coll.element.*}`
All properties of the the collection	`{[aliasname].*}`	`{coll.*}`

17.1.5. Returning non-managed entities

It is possible to apply an `IResultTransformer` to native sql queries. Allowing it to e.g. return non-managed entities.

```
sess.CreateSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .SetResultTransformer(Transformers.AliasToBean<CatDTO>())
```

This query specified:

- the SQL query string
- a result transformer

The above query will return a list of `CatDTO` which has been instantiated and injected the values of NAME and BIRTHNAME into its corresponding properties or fields.

IMPORTANT: The custom `IResultTransformer` should override `Equals` and `GetHashCode`, otherwise the query translation won't be cached. This also will result in memory leak.

17.1.6. Handling inheritance

Native SQL queries which query for entities that are mapped as part of an inheritance hierarchy must include all properties for the base class and all its subclasses.

17.1.7. Parameters

Native SQL queries support positional as well as named parameters:

```
Query query = sess.CreateSQLQuery("SELECT * FROM CATS WHERE NAME like ?").AddEntity(typeof(Cat));
IList pusList = query.SetString(0, "Pus%").List();

query = sess.CreateSQLQuery("SELECT * FROM CATS WHERE NAME like :name").AddEntity(typeof(Cat));
IList pusList = query.SetString("name", "Pus%").List();
```

17.2. Named SQL queries

Named SQL queries may be defined in the mapping document and called in exactly the same way as a named HQL query. In this case, we do *not* need to call `AddEntity()`.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.Name},
           person.AGE AS {person.Age},
           person.SEX AS {person.Sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
IList people = sess.GetNamedQuery("persons")
    .SetString("namePattern", namePattern)
    .SetMaxResults(50)
    .List();
```

The `<return-join>` and `<load-collection>` elements are used to join associations and define queries which initialize collections, respectively.

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.MailingAddress"/>
    SELECT person.NAME AS {person.Name},
           person.AGE AS {person.Age},
           person.SEX AS {person.Sex},
           adddress.STREET AS {address.Street},
           adddress.CITY AS {address.City},
           adddress.STATE AS {address.State},
           adddress.ZIP AS {address.Zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

A named SQL query may return a scalar value. You must declare the column alias and NHibernate type using the `<return-scalar>` element:

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="String"/>
    <return-scalar column="age" type="Int64"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

You can externalize the resultset mapping informations in a `<resultset>` element to either reuse them accross several named queries or through the `SetResultSetMapping()` API.

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.MailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.Name},
           person.AGE AS {person.Age},
           person.SEX AS {person.Sex},
           adddress.STREET AS {address.Street},
           adddress.CITY AS {address.City},
           adddress.STATE AS {address.State},
           adddress.ZIP AS {address.Zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

You can alternatively use the resultset mapping information in your .hbm.xml files directly in code.

```
IList cats = sess.CreateSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .SetResultSetMapping("catAndKitten")
    .List();
```

17.2.1. Using return-property to explicitly specify column/alias names

With `<return-property>` you can explicitly tell NHibernate what column aliases to use, instead of using the `{}`-syntax to let NHibernate inject its own aliases.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="Name" column="myName"/>
        <return-property name="Age" column="myAge"/>
        <return-property name="Sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` also works with multiple columns. This solves a limitation with the `{}`-syntax which can not allow fine grained control of multi-column properties.

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="Salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="EndDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.Employee}, EMPLOYER AS {emp.Employer},
        STARTDATE AS {emp.StartDate}, ENDDATE AS {emp.EndDate},
        REGIONCODE as {emp.RegionCode}, EID AS {emp.Id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

Notice that in this example we used `<return-property>` in combination with the `{}`-syntax for injection, allowing users to choose how they want to refer column and properties.

If your mapping has a discriminator you must use `<return-discriminator>` to specify the discriminator column.

17.2.2. Using stored procedures for querying

NHibernate introduces support for queries via stored procedures and functions. Most of the following documentation is equivalent for both. The stored procedure/function must return a resultset to be able to work with NHibernate. An example of such a stored function in MS SQL Server 2000 and higher is as follows:

```
CREATE PROCEDURE selectAllEmployments AS
    SELECT EMPLOYEE, EMPLOYER, STARTDATE, ENDDATE,
    REGIONCODE, EMPID, VALUE, CURRENCY
    FROM EMPLOYMENT
```

To use this query in NHibernate you need to map it via a named query.

```
<sql-query name="selectAllEmployments_SP">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    exec selectAllEmployments
</sql-query>
```

Notice that stored procedures currently only return scalars and entities. `<return-join>` and `<load-collection>` are not supported.

17.2.2.1. Rules/limitations for using stored procedures

To use stored procedures with NHibernate the procedures/functions have to follow some rules. If they do not follow those rules they are not usable with NHibernate. If you still want to use these procedures you have to execute them via `session.Connection`. The rules are different for each database, since database vendors have different stored procedure semantics/syntax.

Stored procedure queries can't be paged with `SetFirstResult()/SetMaxResults()`.

Recommended call form is dependent on your database. For MS SQL Server use `exec functionName <parameters>`.

For Oracle the following rules apply:

- A function must return a result set. The first parameter of a procedure must be an `OUT` that returns a result set. This is done by using a `SYS_REFCURSOR` type in Oracle 9i or later. In Oracle you need to define a `REF CURSOR` type, see Oracle literature.

For MS SQL server the following rules apply:

- The procedure must return a result set. NHibernate will use `IDbCommand.ExecuteReader()` to obtain the results.
- If you can enable `SET NOCOUNT ON` in your procedure it will probably be more efficient, but this is not a requirement.

17.3. Custom SQL for create, update and delete

NHibernate can use custom SQL statements for create, update, and delete operations. The class and collection persisters in NHibernate already contain a set of configuration time generated strings (insertsql, deletesql, updatesql etc.). The mapping tags `<sql-insert>`, `<sql-delete>`, and `<sql-update>` override these strings:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

Note that the custom `sql-insert` will not be used if you use `identity` to generate identifier values for the class.

The SQL is directly executed in your database, so you are free to use any dialect you like. This will of course reduce the portability of your mapping if you use database specific SQL.

Stored procedures are supported if the database-native syntax is used:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>exec createPerson ?, ?</sql-insert>
    <sql-delete>exec deletePerson ?</sql-delete>
    <sql-update>exec updatePerson ?, ?</sql-update>
</class>
```

The order of the positional parameters is currently vital, as they must be in the same sequence as NHibernate expects them.

You can see the expected order by enabling debug logging for the `NHibernate.Persister.Entity` level. With this level enabled NHibernate will print out the static SQL that is used to create, update, delete etc. entities. (To see the expected sequence, remember to not include your custom SQL in the mapping files as that will override the NHibernate generated static sql.)

The stored procedures are by default required to affect the same number of rows as NHibernate-generated SQL would. NHibernate uses `IDbCommand.ExecuteNonQuery` to retrieve the number of rows affected. This check can be disabled by using `check="none"` attribute in `sql-insert` element.

17.4. Custom SQL for loading

You may also declare your own SQL (or HQL) queries for entity loading:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.Name}, ID AS {pers.Id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

This is just a named query declaration, as discussed earlier. You may reference this named query in a class mapping:

```
<class name="Person">
    <id name="Id">
        <generator class="increment"/>
    </id>
    <property name="Name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

This even works with stored procedures.

You may even define a query for collection loading:

```
<set name="Employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.Employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

You could even define an entity loader that loads a collection by join fetching:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.Employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

Chapter 18. Filtering data

NHibernate provides an innovative new approach to handling data with "visibility" rules. A *NHibernate filter* is a global, named, parameterized filter that may be enabled or disabled for a particular NHibernate session.

18.1. NHibernate filters

NHibernate adds the ability to pre-define filter criteria and attach those filters at both a class and a collection level. A filter criteria is the ability to define a restriction clause very similiar to the existing "where" attribute available on the class and various collection elements. Except these filter conditions can be parameterized. The application can then make the decision at runtime whether given filters should be enabled and what their parameter values should be. Filters can be used like database views, but parameterized inside the application.

In order to use filters, they must first be defined and then attached to the appropriate mapping elements. To define a filter, use the `<filter-def/>` element within a `<hibernate-mapping/>` element:

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="String"/>
</filter-def>
```

Then, this filter can be attached to a class:

```
<class name="MyClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</class>
```

or, to a collection:

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</set>
```

or, even to both (or multiples of each) at the same time.

The methods on `ISession` are: `EnableFilter(string filterName)`, `GetEnabledFilter(string filterName)`, and `DisableFilter(string filterName)`. By default, filters are *not* enabled for a given session; they must be explcitly enabled through use of the `ISession.EnableFilter()` method, which returns an instance of the `IFilter` interface. Using the simple filter defined above, this would look like:

```
session.EnableFilter("myFilter").SetParameter("myFilterParam", "some-value");
```

Note that methods on the `NHibernate.IFilter` interface do allow the method-chaining common to much of NHibernate.

A full example, using temporal data with an effective record date pattern:

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="Department" column="dept_id" class="Department"/>
    <property name="EffectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="EffectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="Employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

Then, in order to ensure that you always get back currently effective records, simply enable the filter on the session prior to retrieving employee data:

```
ISession session = ...;
session.EnableFilter("effectiveDate").SetParameter("asOfDate", DateTime.Today);
IList<Employee> results = session.CreateQuery("from Employee as e where e.Salary > :targetSalary")
```

```
        .SetInt64("targetSalary", 1000000L)
        .List<Employee>();
```

In the HQL above, even though we only explicitly mentioned a salary constraint on the results, because of the enabled filter the query will return only currently active employees who have a salary greater than a million dollars.

Note: if you plan on using filters with outer joining (either through HQL or load fetching) be careful of the direction of the condition expression. It's safest to set this up for left outer joining; in general, place the parameter first followed by the column name(s) after the operator.

Default all filter definitions are applied to `<many-to-one/>` and `<one-to-one/>` elements. You can turn off this behaviour by using `use-many-to-one` attribute on `<filter-def/>` element.

```
<filter-def name="effectiveDate" use-many-to-one="false"/>
```

Chapter 19. Improving performance

19.1. Fetching strategies

A *fetching strategy* is the strategy NHibernate will use for retrieving associated objects if the application needs to navigate the association. Fetch strategies may be declared in the O/R mapping metadata, or overridden by a particular HQL or `Criteria` query.

NHibernate defines the following fetching strategies:

- *Join fetching* - NHibernate retrieves the associated instance or collection in the same `SELECT`, using an `OUTER JOIN`.
- *Select fetching* - a second `SELECT` is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you actually access the association.
- *Subselect fetching* - a second `SELECT` is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying `lazy="false"`, this second select will only be executed when you actually access the association.
- *"Extra-lazy" collection fetching* - individual elements of the collection are accessed from the database as needed. NHibernate tries not to fetch the whole collection into memory unless absolutely needed (suitable for very large collections)
- *Batch fetching* - an optimization strategy for select fetching - NHibernate retrieves a batch of entity instances or collections in a single `SELECT`, by specifying a list of primary keys or foreign keys.

NHibernate also distinguishes between:

- *Immediate fetching* - an association, collection or attribute is fetched immediately, when the owner is loaded.
- *Lazy collection fetching* - a collection is fetched when the application invokes an operation upon that collection. (This is the default for collections.)
- *Proxy fetching* - a single-valued association is fetched when a method other than the identifier getter is invoked upon the associated object.

We have two orthogonal notions here: *when* is the association fetched, and *how* is it fetched (what SQL is used). Don't confuse them! We use `fetch` to tune performance. We may use `lazy` to define a contract for what data is always available in any detached instance of a particular class.

19.1.1. Working with lazy associations

By default, NHibernate uses lazy select fetching for collections and lazy proxy fetching for single-valued associations. These defaults make sense for almost all associations in almost all applications.

However, lazy fetching poses one problem that you must be aware of. Access to a lazy association outside of the context of an open NHibernate session will result in an exception. For example:

```
s = sessions.OpenSession();
Transaction tx = s.BeginTransaction();

User u = s.CreateQuery("from User u where u.Name=:userName")
    .SetString("userName", userName).UniqueResult<User>();
IDictionary permissions = u.Permissions;
```

```
tx.Commit();
s.Close();

int accessLevel = (int) permissions["accounts"];  // Error!
```

Since the `permissions` collection was not initialized when the `ISession` was closed, the collection will not be able to load its state. *NHibernate does not support lazy initialization for detached objects*. The fix is to move the code that reads from the collection to just before the transaction is committed.

Alternatively, we could use a non-lazy collection or association, by specifying `lazy="false"` for the association mapping. However, it is intended that lazy initialization be used for almost all collections and associations. If you define too many non-lazy associations in your object model, NHibernate will end up needing to fetch the entire database into memory in every transaction!

On the other hand, we often want to choose join fetching (which is non-lazy by nature) instead of select fetching in a particular transaction. We'll now see how to customize the fetching strategy. In NHibernate, the mechanisms for choosing a fetch strategy are identical for single-valued associations and collections.

19.1.2. Tuning fetch strategies

Select fetching (the default) is extremely vulnerable to N+1 selects problems, so we might want to enable join fetching in the mapping document:

```
<set name="Permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="Mother" class="Cat" fetch="join"/>
```

The `fetch` strategy defined in the mapping document affects:

- retrieval via `Get()` or `Load()`
- retrieval that happens implicitly when an association is navigated
- `ICriteria` queries
- HQL queries if `subselect` fetching is used

No matter what fetching strategy you use, the defined non-lazy graph is guaranteed to be loaded into memory. Note that this might result in several immediate selects being used to execute a particular HQL query.

Usually, we don't use the mapping document to customize fetching. Instead, we keep the default behavior, and override it for a particular transaction, using `left join fetch` in HQL. This tells NHibernate to fetch the association eagerly in the first select, using an outer join. In the `ICriteria` query API, you would use `SetFetchMode(FetchMode.Join)`.

If you ever feel like you wish you could change the fetching strategy used by `Get()` or `Load()`, simply use a `ICriteria` query, for example:

```
User user = (User) session.CreateCriteria<User>()
                .SetFetchMode("Permissions", FetchMode.Join)
                .Add( Expression.Eq("Id", userId) )
                .UniqueResult();
```

(This is NHibernate's equivalent of what some ORM solutions call a "fetch plan".)

A completely different way to avoid problems with N+1 selects is to use the second-level cache.

19.1.3. Single-ended association proxies

Lazy fetching for collections is implemented using NHibernate's own implementation of persistent collections. However, a different mechanism is needed for lazy behavior in single-ended associations. The target entity of the association must be proxied. NHibernate implements lazy initializing proxies for persistent objects using runtime bytecode enhancement.

By default, NHibernate generates proxies (at startup) for all persistent classes and uses them to enable lazy fetching of `many-to-one` and `one-to-one` associations.

The mapping file may declare an interface to use as the proxy interface for that class, with the `proxy` attribute. By default, NHibernate uses a subclass of the class. *Note that the proxied class must implement a non-private default constructor. We* *recommend this constructor for all persistent classes!*

There are some gotchas to be aware of when extending this approach to polymorphic classes, eg.

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

Firstly, instances of `Cat` will never be castable to `DomesticCat`, even if the underlying instance is an instance of `DomesticCat`:

```
Cat cat = session.Load<Cat>(id);  // instantiate a proxy (does not hit the db)
if ( cat.IsDomesticCat ) {                  // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

Secondly, it is possible to break proxy ==.

```
Cat cat = session.Load<Cat>(id);            // instantiate a Cat proxy
DomesticCat dc =
        session.Load<DomesticCat>(id);  // acquire new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

However, the situation is not quite as bad as it looks. Even though we now have two references to different proxy objects, the underlying instance will still be the same object:

```
cat.Weight = 11.0;  // hit the db to initialize the proxy
Console.WriteLine( dc.Weight );  // 11.0
```

Third, you may not use a proxy for a `sealed` class or a class with any non-overridable public members.

Finally, if your persistent object acquires any resources upon instantiation (eg. in initializers or default constructor), then those resources will also be acquired by the proxy. The proxy class is an actual subclass of the persistent class.

These problems are all due to fundamental limitations in .NET's single inheritance model. If you wish to avoid these problems your persistent classes must each implement an interface that declares its business methods. You should specify these interfaces in the mapping file. eg.

```
<class name="CatImpl" proxy="ICat">
    ......
    <subclass name="DomesticCatImpl" proxy="IDomesticCat">
        .....
    </subclass>
</class>
```

where `CatImpl` implements the interface `ICat` and `DomesticCatImpl` implements the interface `IDomesticCat`. Then proxies for instances of `ICat` and `IDomesticCat` may be returned by `Load()` or `Enumerable()`. (Note that `List()` does not usually return proxies.)

```
ICat cat = session.Load<CatImpl>(catid);
IEnumerator iter = session.CreateQuery("from CatImpl as cat where cat.Name='fritz'").Enumerable().GetEnumerator();
iter.MoveNext();
ICat fritz = (ICat) iter.Current;
```

Relationships are also lazily initialized. This means you must declare any properties to be of type `ICat`, not `CatImpl`.

Certain operations do *not* require proxy initialization

- `Equals()`, if the persistent class does not override `Equals()`
- `GetHashCode()`, if the persistent class does not override `GetHashCode()`
- The identifier getter method

NHibernate will detect persistent classes that override `Equals()` or `GetHashCode()`.

19.1.4. Initializing collections and proxies

A `LazyInitializationException` will be thrown by NHibernate if an uninitialized collection or proxy is accessed outside of the scope of the `ISession`, ie. when the entity owning the collection or having the reference to the proxy is in the detached state.

Sometimes we need to ensure that a proxy or collection is initialized before closing the `ISession`. Of course, we can alway force initialization by calling `cat.Sex` or `cat.Kittens.Count`, for example. But that is confusing to readers of the code and is not convenient for generic code.

The static methods `NHibernateUtil.Initialize()` and `NHibernateUtil.IsInitialized()` provide the application with a convenient way of working with lazily initialized collections or proxies. `NHibernateUtil.Initialize(cat)` will force the initialization of a proxy, `cat`, as long as its `ISession` is still open. `NHibernateUtil.Initialize( cat.Kittens )` has a similar effect for the collection of kittens.

Another option is to keep the `ISession` open until all needed collections and proxies have been loaded. In some application architectures, particularly where the code that accesses data using NHibernate, and the code that uses it are in different application layers or different physical processes, it can be a problem to ensure that the `ISession` is open when a collection is initialized. There are two basic ways to deal with this issue:

- In a web-based application, a `HttpModule` can be used to close the `ISession` only at the very end of a user request, once the rendering of the view is complete (the *Open Session in View* pattern). Of course, this places heavy demands on the correctness of the exception handling of your application infrastructure. It is vitally important that the `ISession` is closed and the transaction ended before returning to the user, even when an exception occurs during rendering of the view. See the NHibernate Wiki for examples of this "Open Session in View" pattern.

- In an application with a separate business tier, the business logic must "prepare" all collections that will be needed by the web tier before returning. This means that the business tier should load all the data and return all the data already initialized to the presentation/web tier that is required for a particular use case. Usually, the application calls `NHibernateUtil.Initialize()` for each collection that will be needed in the web tier (this call must occur before the session is closed) or retrieves the collection eagerly using a NHibernate query with a `FETCH` clause or a `FetchMode.Join` in `ICriteria`. This is usually easier if you adopt the *Command* pattern instead of a *Session Facade*.

- You may also attach a previously loaded object to a new `ISession` with `Merge()` or `Lock()` before accessing uninitialized collections (or other proxies). No, NHibernate does not, and certainly *should* not do this automatically, since it would introduce ad hoc transaction semantics!

Sometimes you don't want to initialize a large collection, but still need some information about it (like its size) or a subset of the data.

You can use a collection filter to get the size of a collection without initializing it:

```
(int) s.CreateFilter( collection, "select count(*)" ).List()[0]
```

The `CreateFilter()` method is also used to efficiently retrieve subsets of a collection without needing to initialize the whole collection:

```
s.CreateFilter( lazyCollection, "").SetFirstResult(0).SetMaxResults(10).List();
```

19.1.5. Using batch fetching

NHibernate can make efficient use of batch fetching, that is, NHibernate can load several uninitialized proxies if one proxy is accessed (or collections. Batch fetching is an optimization of the lazy select fetching strategy. There are two ways you can tune batch fetching: on the class and the collection level.

Batch fetching for classes/entities is easier to understand. Imagine you have the following situation at runtime: You have 25 `Cat` instances loaded in an `ISession`, each `Cat` has a reference to its `Owner`, a `Person`. The `Person` class is mapped with a proxy, `lazy="true"`. If you now iterate through all cats and call `cat.Owner` on each, NHibernate will by default execute 25 `SELECT` statements, to retrieve the proxied owners. You can tune this behavior by specifying a `batch-size` in the mapping of `Person`:

```
<class name="Person" batch-size="10">...</class>
```

NHibernate will now execute only three queries, the pattern is 10, 10, 5.

You may also enable batch fetching of collections. For example, if each `Person` has a lazy collection of `Cats`, and 10 persons are currently loaded in the `ISesssion`, iterating through all persons will generate 10 `SELECTs`, one for every call to `person.Cats`. If you enable batch fetching for the `Cats` collection in the mapping of `Person`, NHibernate can pre-fetch collections:

```
<class name="Person">
    <set name="Cats" batch-size="3">
```

```
        ...
    </set>
</class>
```

With a `batch-size` of 3, NHibernate will load 3, 3, 3, 1 collections in four `SELECT`s. Again, the value of the attribute depends on the expected number of uninitialized collections in a particular `Session`.

Batch fetching of collections is particularly useful if you have a nested tree of items, ie. the typical bill-of-materials pattern. (Although a *nested set* or a *materialized path* might be a better option for read-mostly trees.)

19.1.6. Using subselect fetching

If one lazy collection or single-valued proxy has to be fetched, NHibernate loads all of them, re-running the original query in a subselect. This works in the same way as batch-fetching, without the piecemeal loading.

19.2. The Second Level Cache

A NHibernate `ISession` is a transaction-level cache of persistent data. It is possible to configure a cluster or process-level (`ISessionFactory`-level) cache on a class-by-class and collection-by-collection basis. You may even plug in a clustered cache. Be careful. Caches are never aware of changes made to the persistent store by another application (though they may be configured to regularly expire cached data). *In NHibernate 1.x the second level cache does not work correctly in combination with distributed transactions.*

By default, NHibernate uses HashtableCache for process-level caching. You may choose a different implementation by specifying the name of a class that implements `NHibernate.Cache.ICacheProvider` using the property `hibernate.cache.provider_class`.

Table 19.1. Cache Providers

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	`NHibernate.Cache.HashtableCacheProvider`	memory		yes
ASP.NET Cache (System.Web.Cache)	`NHibernate.Caches.SysCache.SysCacheProvider, NHibernate.Caches.SysCache`	memory		yes
Prevalence Cache	`NHibernate.Caches.Prevalence.PrevalenceCacheProvider, NHibernate.Caches.Prevalence`	memory, disk		yes

19.2.1. Cache mappings

The `<cache>` element of a class or collection mapping has the following form:

```
<cache
    usage="read-write|nonstrict-read-write|read-only"                (1)
    region="RegionName"                                              (2)
/>
```

(1) `usage` specifies the caching strategy: `read-write`, `nonstrict-read-write` or `read-only`

(2) `region` (optional, defaults to the class or collection role name) specifies the name of the second level cache region

Alternatively (preferrably?), you may specify `<class-cache>` and `<collection-cache>` elements in `hibernate.cfg.xml`.

The `usage` attribute specifies a *cache concurrency strategy*.

19.2.2. Strategy: read only

If your application needs to read but never modify instances of a persistent class, a `read-only` cache may be used. This is the simplest and best performing strategy. Its even perfectly safe for use in a cluster.

```
<class name="Eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

19.2.3. Strategy: read/write

If the application needs to update data, a `read-write` cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required. You should ensure that the transaction is completed when `ISession.Close()` or `ISession.Disconnect()` is called. If you wish to use this strategy in a cluster, you should ensure that the underlying cache implementation supports locking. The built-in cache providers do *not*.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="Kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

19.2.4. Strategy: nonstrict read/write

If the application only occasionally needs to update data (ie. if it is extremely unlikely that two transactions would try to update the same item simultaneously) and strict transaction isolation is not required, a `nonstrict-read-write` cache might be appropriate. When using this strategy you should ensure that the transaction is completed when `ISession.Close()` or `ISession.Disconnect()` is called.

The following table shows which providers are compatible with which concurrency strategies.

Table 19.2. Cache Concurrency Strategy Support

Cache	read-only	nonstrict-read-write	read-write
Hashtable (not intended for production use)	yes	yes	yes
SysCache	yes	yes	yes
PrevalenceCache	yes	yes	yes

Refer to Chapter 25, *NHibernate.Caches* for more details.

19.3. Managing the caches

Whenever you pass an object to `Save()`, `Update()` or `SaveOrUpdate()` and whenever you retrieve an object using `Load()`, `Get()`, `List()`, or `Enumerable()`, that object is added to the internal cache of the `ISession`.

When `Flush()` is subsequently called, the state of that object will be synchronized with the database. If you do not want this synchronization to occur or if you are processing a huge number of objects and need to manage memory efficiently, the `Evict()` method may be used to remove the object and its collections from the first-level cache.

```
IEnumerable<Cat> cats = sess.CreateQuery("from Eg.Cat as cat").Enumerable<Cat>(); //a huge result set
foreach( Cat cat in cats )
{
    DoSomethingWithACat(cat);
    sess.Evict(cat);
}
```

NHibernate will evict associated entities automatically if the association is mapped with `cascade="all"` or `cascade="all-delete-orphan"`.

The `ISession` also provides a `Contains()` method to determine if an instance belongs to the session cache.

To completely evict all objects from the session cache, call `ISession.Clear()`

For the second-level cache, there are methods defined on `ISessionFactory` for evicting the cached state of an instance, entire class, collection instance or entire collection role.

```
sessionFactory.Evict<Cat>(catId); //evict a particular Cat
sessionFactory.Evict<Cat>();  //evict all Cats
sessionFactory.EvictCollection("Eg.Cat.Kittens", catId); //evict a particular collection of kittens
sessionFactory.EvictCollection("Eg.Cat.Kittens"); //evict all kitten collections
```

19.4. The Query Cache

Query result sets may also be cached. This is only useful for queries that are run frequently with the same parameters. To use the query cache you must first enable it:

```
<add key="hibernate.cache.use_query_cache" value="true" />
```

This setting causes the creation of two new cache regions - one holding cached query result sets (`NHibernate.Cache.StandardQueryCache`), the other holding timestamps of the most recent updates to queryable tables (`NHibernate.Cache.UpdateTimestampsCache`). Note that the query cache does not cache the state of any entities in the result set; it caches only identifier values and results of value type. So the query cache should always be used in conjunction with the second-level cache.

Most queries do not benefit from caching, so by default queries are not cached. To enable caching, call `IQuery.SetCacheable(true)`. This call allows the query to look for existing cache results or add its results to the cache when it is executed.

If you require fine-grained control over query cache expiration policies, you may specify a named cache region for a particular query by calling `IQuery.SetCacheRegion()`.

```
IList<Blog> blogs = sess.CreateQuery("from Blog blog where blog.Blogger = :blogger")
    .SetEntity("blogger", blogger)
    .SetMaxResults(15)
    .SetCacheable(true)
    .SetCacheRegion("frontpages")
    .List<Blog>();
```

If the query should force a refresh of its query cache region, you may call `IQuery.SetForceCacheRefresh()` to `true`. This is particularly useful in cases where underlying data may have been updated via a saparate process (i.e., not modified through NHibernate) and allows the application to selectively refresh the query cache regions based on its knowledge of those events. This is a more efficient alternative to eviction of a query cache region via `ISessionFactory.EvictQueries()`.

19.5. Understanding Collection performance

We've already spent quite some time talking about collections. In this section we will highlight a couple more issues about how collections behave at runtime.

19.5.1. Taxonomy

NHibernate defines three basic kinds of collections:

- collections of values
- one to many associations
- many to many associations

This classification distinguishes the various table and foreign key relationships but does not tell us quite everything we need to know about the relational model. To fully understand the relational structure and performance characteristics, we must also consider the structure of the primary key that is used by NHibernate to update or delete collection rows. This suggests the following classification:

- indexed collections
- sets
- bags

All indexed collections (maps, lists, arrays) have a primary key consisting of the `<key>` and `<index>` columns. In this case collection updates are usually extremely efficient - the primary key may be efficiently indexed and a particular row may be efficiently located when NHibernate tries to update or delete it.

Sets have a primary key consisting of `<key>` and element columns. This may be less efficient for some types of collection element, particularly composite elements or large text or binary fields; the database may not be able to index a complex primary key as efficently. On the other hand, for one to many or many to many associations, particularly in the case of synthetic identifiers, it is likely to be just as efficient. (Side-note: if you want `SchemaExport` to actually create the primary key of a `<set>` for you, you must declare all columns as `not-null="true"`.)

`<idbag>` mappings define a surrogate key, so they are always very efficient to update. In fact, they are the best case.

Bags are the worst case. Since a bag permits duplicate element values and has no index column, no primary key may be defined. NHibernate has no way of distinguishing between duplicate rows. NHibernate resolves this problem by completely removing (in a single `DELETE`) and recreating the collection whenever it changes. This might be very inefficient.

Note that for a one-to-many association, the "primary key" may not be the physical primary key of the database table - but even in this case, the above classification is still useful. (It still reflects how NHibernate "locates" individual rows of the collection.)

19.5.2. Lists, maps, idbags and sets are the most efficient collections to update

From the discussion above, it should be clear that indexed collections and (usually) sets allow the most efficient operation in terms of adding, removing and updating elements.

There is, arguably, one more advantage that indexed collections have over sets for many to many associations or collections of values. Because of the structure of an `ISet<T>`, NHibernate doesn't ever `UPDATE` a row when an element is "changed". Changes to an `ISet<T>` always work via `INSERT` and `DELETE` (of individual rows). Once again, this consideration does not apply to one to many associations.

After observing that arrays cannot be lazy, we would conclude that lists, maps and idbags are the most performant (non-inverse) collection types, with sets not far behind. Sets are expected to be the most common kind of collection in NHibernate applications. This is because the "set" semantics are most natural in the relational model.

However, in well-designed NHibernate domain models, we usually see that most collections are in fact one-to-many associations with `inverse="true"`. For these associations, the update is handled by the many-to-one end of the association, and so considerations of collection update performance simply do not apply.

19.5.3. Bags and lists are the most efficient inverse collections

Just before you ditch bags forever, there is a particular case in which bags (and also lists) are much more performant than sets. For a collection with `inverse="true"` (the standard bidirectional one-to-many relationship idiom, for example) we can add elements to a bag or list without needing to initialize (fetch) the bag elements! This is because `IList.Add()` must always succeed for a bag or `IList` (unlike an `ISet<T>`). This can make the following common code much faster.

```
Parent p = sess.Load<Parent>(id);
    Child c = new Child();
    c.Parent = p;
    p.Children.Add(c);  //no need to fetch the collection!
    sess.Flush();
```

19.5.4. One shot delete

Occasionally, deleting collection elements one by one can be extremely inefficient. NHibernate isn't completely stupid, so it knows not to do that in the case of an newly-empty collection (if you called `list.Clear()`, for example). In this case, NHibernate will issue a single `DELETE` and we are done!

Suppose we add a single element to a collection of size twenty and then remove two elements. NHibernate will issue one `INSERT` statement and two `DELETE` statements (unless the collection is a bag). This is certainly desirable.

However, suppose that we remove eighteen elements, leaving two and then add thee new elements. There are two possible ways to proceed:

- Delete eighteen rows one by one and then insert three rows
- Remove the whole collection (in one SQL `DELETE`) and insert all five current elements (one by one)

NHibernate isn't smart enough to know that the second option is probably quicker in this case. (And it would probably be undesirable for NHibernate to be that smart; such behaviour might confuse database triggers, etc.)

Fortunately, you can force this behaviour (ie. the second strategy) at any time by discarding (ie. dereferencing) the original collection and returning a newly instantiated collection with all the current elements. This can be very useful and powerful from time to time.

Of course, one-shot-delete does not apply to collections mapped `inverse="true"`.

19.6. Batch updates

NHibernate supports batching SQL update commands (`INSERT`, `UPDATE`, `DELETE`) with the following limitations:

- the Nhibernate's drive used for your RDBMS may not supports batching,
- since the implementation uses reflection to access members and types in System.Data assembly which are not normally visible, it may not function in environments where necessary permissions are not granted,
- optimistic concurrency checking may be impaired since ADO.NET 2.0 does not return the number of rows affected by each statement in the batch, only the total number of rows affected by the batch.

Update batching is enabled by setting `adonet.batch_size` to a non-zero value.

19.7. Multi Query

This functionality allows you to execute several HQL queries in one round-trip against the database server. A simple use case is executing a paged query while also getting the total count of results, in a single round-trip. Here is a simple example:

```
IMultiQuery multiQuery = s.CreateMultiQuery()
    .Add(s.CreateQuery("from Item i where i.Id > ?")
            .SetInt32(0, 50).SetFirstResult(10))
    .Add(s.CreateQuery("select count(*) from Item i where i.Id > ?")
            .SetInt32(0, 50));
IList results = multiQuery.List();
IList items = (IList)results[0];
long count = (long)((IList)results[1])[0];
```

The result is a list of query results, ordered according to the order of queries added to the multi query. Named parameters can be set on the multi query, and are shared among all the queries contained in the multi query, like this:

```
IList results = s.CreateMultiQuery()
    .Add(s.CreateQuery("from Item i where i.Id > :id")
        .SetFirstResult(10) )
    .Add("select count(*) from Item i where i.Id > :id")
    .SetInt32("id", 50)
    .List();
IList items = (IList)results[0];
long count = (long)((IList)results[1])[0];
```

Positional parameters are not supported on the multi query, only on the individual queries.

As shown above, if you do not need to configure the query separately, you can simply pass the HQL directly to the `IMultiQuery.Add()` method.

Multi query is executed by concatenating the queries and sending the query to the database as a single string. This means that the database should support returning several result sets in a single query. At the moment this functionality is only enabled for Microsoft SQL Server and SQLite.

Note that the database server is likely to impose a limit on the maximum number of parameters in a query, in which case the limit applies to the multi query as a whole. Queries using `in` with a large number of arguments passed as parameters may easily exceed this limit. For example, SQL Server has a limit of 2,100 parameters per round-trip, and will throw an exception executing this query:

```
IList allEmployeesId  = ...; //1,500 items
IMultiQuery multiQuery = s.CreateMultiQuery()
        .Add(s.CreateQuery("from Employee e where e.Id in :empIds")
                .SetParameter("empIds", allEmployeesId).SetFirstResult(10))
        .Add(s.CreateQuery("select count(*) from Employee e where e.Id in :empIds")
                .SetParameter("empIds", allEmployeesId));
        IList results = multiQuery.List(); // will throw an exception from SQL Server
```

An interesting usage of this feature is to load several collections of an object in one round-trip, without an expensive cartesian product (blog * users * posts).

```
Blog blog = s.CreateMultiQuery()
    .Add("select b from Blog b left join fetch b.Users where b.Id = :id")
    .Add("select b from Blog b left join fetch b.Posts where b.Id = :id")
    .SetInt32("id", 123)
    .UniqueResult<Blog>();
```

19.8. Multi Criteria

This is the counter-part to Multi Query, and allows you to perform several criteria queries in a single round trip. A simple use case is executing a paged query while also getting the total count of results, in a single round-trip. Here is a simple example:

```
IMultiCriteria multiCrit = s.CreateMultiCriteria()
    .Add(s.CreateCriteria<Item>()
            .Add(Expression.Gt("Id", 50))
            .SetFirstResult(10))
    .Add(s.CreateCriteria<Item>()
            .Add(Expression.Gt("Id", 50))
            .SetProject(Projections.RowCount()));
IList results = multiCrit.List();
IList items = (IList)results[0];
long count = (long)((IList)results[1])[0];
```

The result is a list of query results, ordered according to the order of queries added to the multi criteria.

You can add `ICriteria` or `DetachedCriteria` to the Multi Criteria query. In fact, using DetachedCriteria in this fashion has some interesting implications.

```
DetachedCriteria customersCriteria = AuthorizationService.GetAssociatedCustomersQuery();
IList results = session.CreateMultiCriteria()
        .Add(customersCriteria)
        .Add(DetachedCriteria.For<Policy>()
                .Add( Subqueries.PropertyIn("id", CriteriaTransformer.Clone(customersCriteria)
                                                    .SetProjection(Projections.Id())
                     ) )
        ).List();

ICollection<Customer> customers = CollectionHelper.ToArray<Customer>(results[0]);
ICollection<Policy> policies = CollectionHelper.ToArray<Policy>(results[1]);
```

As you see, we get a query that represnt the customers we can access, and then we can utlize this query further in order to perform additional logic (getting the policies of the customers we are associated with), all in a single database roundtrip.

Chapter 20. Toolset Guide

Roundtrip engineering with NHibernate is possible using a set of commandline tools maintained as part of the NHibernate project, along with NHibernate support built into various code generation tools (MyGeneration, CodeSmith, ObjectMapper, AndroMDA).

The NHibernate main package comes bundled with the most important tool (it can even be used from "inside" NHibernate on-the-fly):

- DDL schema generation from a mapping file (aka `SchemaExport`, `hbm2ddl`)

Other tools directly provided by the NHibernate project are delivered with a separate package, *NHibernateContrib*. This package includes tools for the following tasks:

- C# source generation from a mapping file (aka `hbm2net`)
- mapping file generation from .NET classes marked with attributes (`NHibernate.Mapping.Attributes`, or NHMA for short)

Third party tools with NHibernate support are:

- CodeSmith, MyGeneration, and ObjectMapper (mapping file generation from an existing database schema)
- AndroMDA (MDA (Model-Driven Architecture) approach generating code for persistent classes from UML diagrams and their XML/XMI representation)

These 3rd party tools are not documented in this reference. Please refer to the NHibernate website for up-to-date information.

20.1. Schema Generation

The generated schema includes referential integrity constraints (primary and foreign keys) for entity and collection tables. Tables and sequences are also created for mapped identifier generators.

You *must* specify a SQL `Dialect` via the `hibernate.dialect` property when using this tool.

20.1.1. Customizing the schema

Many NHibernate mapping elements define an optional attribute named `length`. You may set the length of a column with this attribute. (Or, for numeric/decimal data types, the precision.)

Some tags also accept a `not-null` attribute (for generating a `NOT NULL` constraint on table columns) and a `unique` attribute (for generating `UNIQUE` constraint on table columns).

Some tags accept an `index` attribute for specifying the name of an index for that column. A `unique-key` attribute can be used to group columns in a single unit key constraint. Currently, the specified value of the `unique-key` attribute is *not* used to name the constraint, only to group the columns in the mapping file.

Examples:

```
<property name="Foo" type="String" length="64" not-null="true"/>
```

```
<many-to-one name="Bar" foreign-key="fk_foo_bar" not-null="true"/>

<element column="serial_number" type="Int64" not-null="true" unique="true"/>
```

Alternatively, these elements also accept a child `<column>` element. This is particularly useful for multi-column types:

```
<property name="Foo" type="String">
    <column name="foo" length="64" not-null="true" sql-type="text"/>
</property>

<property name="Bar" type="My.CustomTypes.MultiColumnType, My.CustomTypes"/>
    <column name="fee" not-null="true" index="bar_idx"/>
    <column name="fi" not-null="true" index="bar_idx"/>
    <column name="fo" not-null="true" index="bar_idx"/>
</property>
```

The `sql-type` attribute allows the user to override the default mapping of NHibernate type to SQL datatype.

The `check` attribute allows you to specify a check constraint.

```
<property name="Foo" type="Int32">
    <column name="foo" check="foo > 10"/>
</property>

<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="Bar" type="Single"/>
</class>
```

Table 20.1. Summary

Attribute	Values	Interpretation
`length`	number	column length/decimal precision
`not-null`	`true|false`	specfies that the column should be non-nullable
`unique`	`true|false`	specifies that the column should have a unique constraint
`index`	`index_name`	specifies the name of a (multi-column) index
`unique-key`	`unique_key_name`	specifies the name of a multi-column unique constraint
`foreign-key`	`foreign_key_name`	specifies the name of the foreign key constraint generated for an association, use it on <one-to-one>, <many-to-one>, <key>, and <many-to-many> mapping elements. Note that `inverse="true"` sides will not be considered by `SchemaExport`.
`sql-type`	`column_type`	overrides the default column type (attribute of `<column>` element only)
`check`	SQL expression	create an SQL check constraint on either column or table

20.1.2. Running the tool

The `SchemaExport` tool writes a DDL script to standard out and/or executes the DDL statements.

You may embed `SchemaExport` in your application:

```
Configuration cfg = ....;
new SchemaExport(cfg).Create(false, true);
```

20.1.3. Properties

Database properties may be specified

- as system properties with `-D`*<property>*
- in `hibernate.properties`
- in a named properties file with `--properties`

The needed properties are:

Table 20.2. SchemaExport Connection Properties

Property Name	Description
`hibernate.connection.driver_class`	jdbc driver class

hibernate.connection.url	jdbc url
hibernate.connection.username	database user
hibernate.connection.password	user password
hibernate.dialect	dialect

20.1.4. Using Ant

You can call `SchemaExport` from your Ant build script:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

If you don't specify `properties` or a `config` file, the `SchemaExportTask` will try to use normal Ant project properties instead. In other words, if you don't want or need an external configuration or properties file, you may put `hibernate.*` configuration properties in your build.xml or build.properties.

20.2. Code Generation

The NHibernate code generator may be used to generate skeletal C# implementation classes from a NHibernate mapping file. This tool is included in the NHibernate Contrib package (a saparate download in http://sourceforge.net/projects/nhcontrib/).

`hbm2net` parses the mapping files and generates fully working C# source files from these. Thus with `hbm2net` one could "just" provide the `.hbm` files, and then don't worry about hand-writing/coding the C# files.

`hbm2net` *options mapping_files*

Table 20.3. Code Generator Command Line Options

Option	Description
`-output:`*output_dir*	root directory for generated code
`-config:`*config_file*	optional file for configuring hbm2net

A more detailed guide of `hbm2net` is available in http://nhibernate.info/blog/2009/12/12/t4-hbm2net-alpha-2.html

Chapter 21. Example: Parent/Child

One of the very first things that new users try to do with NHibernate is to model a parent / child type relationship. There are two different approaches to this. For various reasons the most convenient approach, especially for new users, is to model both `Parent` and `Child` as entity classes with a `<one-to-many>` association from `Parent` to `Child`. (The alternative approach is to declare the `Child` as a `<composite-element>`.) Now, it turns out that default semantics of a one to many association (in NHibernate) are much less close to the usual semantics of a parent / child relationship than those of a composite element mapping. We will explain how to use a *bidirectional one to many association with cascades* to model a parent / child relationship efficiently and elegantly. It's not at all difficult!

21.1. A note about collections

NHibernate collections are considered to be a logical part of their owning entity; never of the contained entities. This is a crucial distinction! It has the following consequences:

- When we remove / add an object from / to a collection, the version number of the collection owner is incremented.
- If an object that was removed from a collection is an instance of a value type (eg, a composite element), that object

will cease to be persistent and its state will be completely removed from the database. Likewise, adding a value type instance to the collection will cause its state to be immediately persistent.

- On the other hand, if an entity is removed from a collection (a one-to-many or many-to-many association), it will not be deleted, by default. This behavior is completely consistent - a change to the internal state of another entity should not cause the associated entity to vanish! Likewise, adding an entity to a collection does not cause that entity to become persistent, by default.

Instead, the default behavior is that adding an entity to a collection merely creates a link between the two entities, while removing it removes the link. This is very appropriate for all sorts of cases. Where it is not appropriate at all is the case of a parent / child relationship, where the life of the child is bound to the lifecycle of the parent.

21.2. Bidirectional one-to-many

Suppose we start with a simple `<one-to-many>` association from `Parent` to `Child`.

```
<set name="Children">
    <key column="parent_id" />
    <one-to-many class="Child" />
</set>
```

If we were to execute the following code

```
Parent p = .....;
Child c = new Child();
p.Children.Add(c);
session.Save(c);
session.Flush();
```

NHibernate would issue two SQL statements:

- an `INSERT` to create the record for `c`
- an `UPDATE` to create the link from `p` to `c`

This is not only inefficient, but also violates any `NOT NULL` constraint on the `parent_id` column.

The underlying cause is that the link (the foreign key `parent_id`) from `p` to `c` is not considered part of the state of the `Child` object and is therefore not created in the `INSERT`. So the solution is to make the link part of the `Child` mapping.

```
<many-to-one name="Parent" column="parent_id" not-null="true"/>
```

(We also need to add the `Parent` property to the `Child` class.)

Now that the `Child` entity is managing the state of the link, we tell the collection not to update the link. We use the `inverse` attribute.

```
<set name="Children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

The following code would be used to add a new `Child`.

```
Parent p = session.Load<Parent>(pid);
Child c = new Child();
c.Parent = p;
p.Children.Add(c);
session.Save(c);
session.Flush();
```

And now, only one SQL `INSERT` would be issued!

To tighten things up a bit, we could create an `AddChild()` method of `Parent`.

```
public void AddChild(Child c)
{
    c.Parent = this;
    children.Add(c);
}
```

Now, the code to add a `Child` looks like

```
Parent p = session.Load<Parent>(pid);
Child c = new Child();
p.AddChild(c);
session.Save(c);
session.Flush();
```

21.3. Cascading lifecycle

The explicit call to `Save()` is still annoying. We will address this by using cascades.

```
<set name="Children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

This simplifies the code above to

```
Parent p = session.Load<Parent>(pid);
Child c = new Child();
p.AddChild(c);
session.Flush();
```

Similarly, we don't need to iterate over the children when saving or deleting a `Parent`. The following removes `p` and all its children from the database.

```
Parent p = session.Load<Parent>(pid);
session.Delete(p);
session.Flush();
```

However, this code

```
Parent p = session.Load<Parent>(pid);
// Get one child out of the set
IEnumerator childEnumerator = p.Children.GetEnumerator();
childEnumerator.MoveNext();
Child c = (Child) childEnumerator.Current;

p.Children.Remove(c);
c.Parent = null;
session.Flush();
```

will not remove `c` from the database; it will only remove the link to `p` (and cause a `NOT NULL` constraint violation, in this case). You need to explicitly `Delete()` the `Child`.

```
Parent p = session.Load<Parent>(pid);
// Get one child out of the set
IEnumerator childEnumerator = p.Children.GetEnumerator();
childEnumerator.MoveNext();
Child c = (Child) childEnumerator.Current;

p.Children.Remove(c);
session.Delete(c);
session.Flush();
```

Now, in our case, a `Child` can't really exist without its parent. So if we remove a `Child` from the collection, we really do want it to be deleted. For this, we must use `cascade="all-delete-orphan"`.

```
<set name="Children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Note: even though the collection mapping specifies `inverse="true"`, cascades are still processed by iterating the collection elements. So if you require that an object be saved, deleted or updated by cascade, you must add it to the collection. It is not enough to simply set its parent.

21.4. Using cascading `Update()`

Suppose we loaded up a `Parent` in one `ISession`, made some changes in a UI action and wish to persist these changes in a new ISession (by calling `Update()`). The `Parent` will contain a collection of children and, since cascading update is enabled, NHibernate needs to know which children are newly instantiated and which represent existing rows in the database. Let's assume that both `Parent` and `Child` have (synthetic) identifier properties of type `long`. NHibernate will use the identifier property value to determine which of the children are new. (You may also use the version or timestamp property, see Section 9.4.2, “Updating detached objects”.)

The `unsaved-value` attribute is used to specify the identifier value of a newly instantiated instance. *In NHibernate it is not necessary to specify `unsaved-value` explicitly.*

The following code will update `parent` and `child` and insert `newChild`.

```
//parent and child were both loaded in a previous session
parent.AddChild(child);
Child newChild = new Child();
parent.AddChild(newChild);
session.Update(parent);
session.Flush();
```

Well, thats all very well for the case of a generated identifier, but what about assigned identifiers and composite identifiers? This is more difficult, since `unsaved-value` can't distinguish between a newly instantiated object (with an identifier assigned by the user) and an object loaded in a previous session. In these cases, you will probably need to give NHibernate a hint; either

- define an `unsaved-value` on a `<version>` or `<timestamp>` property mapping for the class.
- set `unsaved-value="none"` and explicitly `Save()` newly instantiated children before calling `Update(parent)`
- set `unsaved-value="any"` and explicitly `Update()` previously persistent children before calling `Update(parent)`

`null` is the default `unsaved-value` for assigned identifiers, `none` is the default `unsaved-value` for composite identifiers.

There is one further possibility. There is a new `IInterceptor` method named `IsTransient()` which lets the application implement its own strategy for distinguishing newly instantiated objects. For example, you could define a base class for your persistent classes.

```
public class Persistent
{
    private bool _saved = false;

    public void OnSave()
    {
        _saved=true;
    }

    public void OnLoad()
    {
        _saved=true;
    }

    ......

    public bool IsSaved
    {
        get { return _saved; }
    }
}
```

(The `saved` property is non-persistent.) Now implement `IsTransient()`, along with `OnLoad()` and `OnSave()` as follows.

```
        public object IsTransient(object entity)
{
    if (entity is Persistent)
    {
        return !( (Persistent) entity ).IsSaved;
    }
    else
    {
        return null;
    }
}

public bool OnLoad(object entity,
    object id,
    object[] state,
    string[] propertyNames,
    IType[] types)
{
    if (entity is Persistent) ( (Persistent) entity ).OnLoad();
    return false;
}

public boolean OnSave(object entity,
    object id,
    object[] state,
    string[] propertyNames,
    IType[] types)
{
    if (entity is Persistent) ( (Persistent) entity ).OnSave();
    return false;
}
```

21.5. Conclusion

There is quite a bit to digest here and it might look confusing first time around. However, in practice, it all works out quite nicely. Most NHibernate applications use the parent / child pattern in many places.

We mentioned an alternative in the first paragraph. None of the above issues exist in the case of `<composite-element>` mappings, which have exactly the semantics of a parent / child relationship. Unfortunately, there are two big limitations to composite element classes: composite elements may not own collections, and they should not be the child of any entity other than the unique parent. (However, they *may* have a surrogate primary key, using an `<idbag>` mapping.)

Chapter 22. Example: Weblog Application

22.1. Persistent Classes

The persistent classes represent a weblog, and an item posted in a weblog. They are to be modelled as a standard parent/child relationship, but we will use an ordered bag, instead of a set.

```
using System;
using System.Collections.Generic;

namespace Eg
{
    public class Blog
    {
        private long _id;
        private string _name;
        private IList<BlogItem> _items;

        public virtual long Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public virtual IList<BlogItem> Items
        {
            get { return _items; }
            set { _items = value; }
        }

        public virtual string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }
}
```

```
using System;

namespace Eg
{
    public class BlogItem
    {
        private long _id;
        private DateTime _dateTime;
        private string _text;
        private string _title;
        private Blog _blog;

        public virtual Blog Blog
        {
            get { return _blog; }
            set { _blog = value; }
        }

        public virtual DateTime DateTime
        {
            get { return _dateTime; }
            set { _dateTime = value; }
        }

        public virtual long Id
        {
            get { return _id; }
            set { _id = value; }
        }

        public virtual string Text
        {
            get { return _text; }
            set { _text = value; }
        }

        public virtual string Title
        {
            get { return _title; }
            set { _title = value; }
        }
    }
}
```

22.2. NHibernate Mappings

The XML mappings should now be quite straightforward.

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="Eg" namespace="Eg">

    <class
        name="Blog"
        table="BLOGS"
        lazy="true">

        <id
            name="Id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="Name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="Items"
            inverse="true"
            lazy="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="Eg" namespace="Eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="Id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>
        <property
            name="Title"
            column="TITLE"
            not-null="true"/>

        <property
            name="Text"
            column="TEXT"
            not-null="true"/>

        <property
            name="DateTime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="Blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

22.3. NHibernate Code

The following class demonstrates some of the kinds of things we can do with these classes, using NHibernate.

```
using System;
using System.Collections.Generic;

using NHibernate.Tool.hbm2ddl;

namespace Eg
{
    public class BlogMain
    {
        private ISessionFactory _sessions;

        public void Configure()
        {
            _sessions = new Configuration()
                .AddClass(typeof(Blog))
                .AddClass(typeof(BlogItem))
                .BuildSessionFactory();
        }

        public void ExportTables()
        {
            Configuration cfg = new Configuration()
                .AddClass(typeof(Blog))
                .AddClass(typeof(BlogItem));
            new SchemaExport(cfg).create(true, true);
        }

        public Blog CreateBlog(string name)
        {
            Blog blog = new Blog();
            blog.Name = name;
            blog.Items = new List<BlogItem>();

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                session.Save(blog);
                tx.Commit();
            }

            return blog;
        }

        public BlogItem CreateBlogItem(Blog blog, string title, string text)
        {
            BlogItem item = new BlogItem();
            item.Title = title;
            item.Text = text;
            item.Blog = blog;
            item.DateTime = DateTime.Now;
            blog.Items.Add(item);

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                session.Update(blog);
                tx.Commit();
            }

            return item;
        }

        public BlogItem CreateBlogItem(long blogId, string title, string text)
        {
            BlogItem item = new BlogItem();
            item.Title = title;
            item.Text = text;
            item.DateTime = DateTime.Now;

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                Blog blog = session.Load<Blog>(blogId);
                item.Blog = blog;
                blog.Items.Add(item);
                tx.Commit();
            }

            return item;
        }

        public void UpdateBlogItem(BlogItem item, string text)
        {
            item.Text = text;

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                session.Update(item);
                tx.Commit();
            }
        }

        public void UpdateBlogItem(long itemId, string text)
        {
            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                BlogItem item = session.Load<BlogItem>(itemId);
                item.Text = text;
                tx.Commit();
            }
        }

        public IList<BlogItem> ListAllBlogNamesAndItemCounts(int max)
        {
            IList<BlogItem> result = null;

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                IQuery q = session.CreateQuery(
                    "select blog.id, blog.Name, count(blogItem) " +
                    "from Blog as blog " +
                    "left outer join blog.Items as blogItem " +
                    "group by blog.Name, blog.id " +
                    "order by max(blogItem.DateTime)"
                );
                q.SetMaxResults(max);
                result = q.List<BlogItem>();
                tx.Commit();
            }

            return result;
        }

        public Blog GetBlogAndAllItems(long blogId)
        {
            Blog blog = null;

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                IQuery q = session.createQuery(
                    "from Blog as blog " +
                    "left outer join fetch blog.Items " +
                    "where blog.id = :blogId"
                );
                q.SetParameter("blogId", blogId);
                blog  = (Blog) q.List()[0];
```

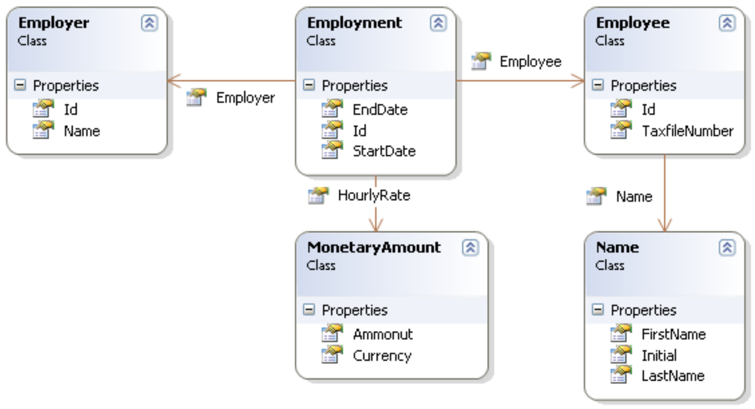
```
                tx.Commit();
            }

            return blog;
        }

        public IList<Blog> ListBlogsAndRecentItems()
        {
            IList<Blog> result = null;

            using (ISession session = _sessions.OpenSession())
            using (ITransaction tx = session.BeginTransaction())
            {
                IQuery q = session.CreateQuery(
                    "from Blog as blog " +
                    "inner join blog.Items as blogItem " +
                    "where blogItem.DateTime > :minDate"
                );

                DateTime date = DateTime.Now.AddMonths(-1);
                q.SetDateTime("minDate", date);

                result = q.List<Blog>();
                tx.Commit();
            }

            return result;
        }
    }
}
```

Chapter 23. Example: Various Mappings

This chapter shows off some more complex association mappings.

23.1. Employer/Employee

The following model of the relationship between `Employer` and `Employee` uses an actual entity class (`Employment`) to represent the association. This is done because there might be more than one period of employment for the same two parties. Components are used to model monetary values and employee names.



Here's a possible mapping document:

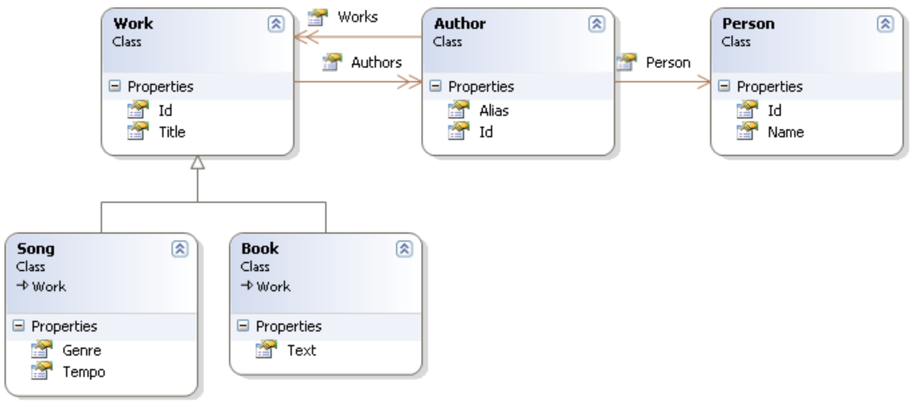
```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="..." namespace="...">

    <class name="Employer" table="employers">
        <id name="Id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="Name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="Id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="StartDate" column="start_date"/>
        <property name="EndDate" column="end_date"/>

        <component name="HourlyRate" class="MonetaryAmount">
            <property name="Amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="Currency" length="12"/>
        </component>

        <many-to-one name="Employer" column="employer_id" not-null="true"/>
        <many-to-one name="Employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="Id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="TaxfileNumber"/>
        <component name="Name" class="Name">
            <property name="FirstName"/>
            <property name="Initial"/>
            <property name="LastName"/>
        </component>
    </class>

</hibernate-mapping>
```

And here's the table schema generated by `SchemaExport`.

```
create table employers (
    Id BIGINT not null,
    Name VARCHAR(255),
    primary key (Id)
)

create table employment_periods (
    Id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    Currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (Id)
)

create table employees (
    Id BIGINT not null,
    FirstName VARCHAR(255),
    Initial CHAR(1),
    LastName VARCHAR(255),
    TaxfileNumber VARCHAR(255),
    primary key (Id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

23.2. Author/Work

Consider the following model of the relationships between `Work`, `Author` and `Person`. We represent the relationship between `Work` and `Author` as a many-to-many association. We choose to represent the relationship between `Author` and `Person` as one-to-one association. Another possibility would be to have `Author` extend `Person`.



The following mapping document correctly represents these relationships:

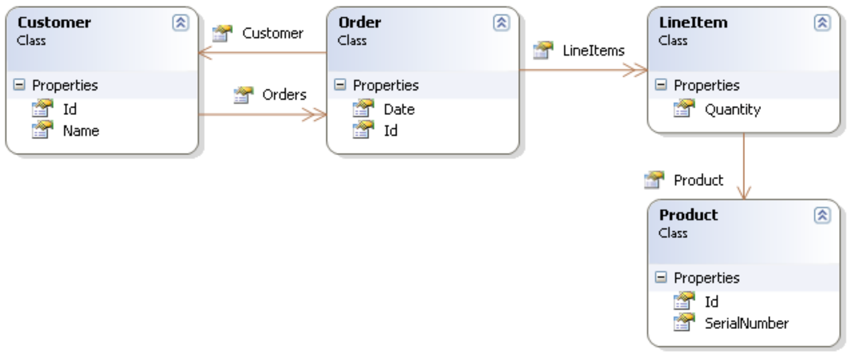
```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="..." namespace="...">

    <class name="Work" table="works" discriminator-value="W">

        <id name="Id" column="id" generator="native" />

        <discriminator column="type" type="character"/>

        <property name="Title"/>
        <set name="Authors" table="author_work" lazy="true">
            <key>
                <column name="work_id" not-null="true"/>
            </key>
            <many-to-many class="Author">
                <column name="author_id" not-null="true"/>
            </many-to-many>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="Text" column="text" />
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="Tempo" column="tempo" />
            <property name="Genre" column="genre" />
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="Id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="Alias" column="alias" />
        <one-to-one name="Person" constrained="true"/>

        <set name="Works" table="author_work" inverse="true" lazy="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="Id" column="id">
            <generator class="native"/>
        </id>
        <property name="Name" column="name" />
    </class>

</hibernate-mapping>
```

There are four tables in this mapping. `works`, `authors` and `persons` hold work, author and person data respectively. `author_work` is an association table linking authors to works. Heres the table schema, as generated by `SchemaExport`.

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

23.3. Customer/Order/Product

Now consider a model of the relationships between `Customer`, `Order` and `LineItem` and `Product`. There is a one-to-many association between `Customer` and `Order`, but how should we represent `Order` / `LineItem` / `Product`? I've chosen to map `LineItem` as an association class representing the many-to-many association between `Order` and `Product`. In NHibernate, this is called a composite element.



The mapping document:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
    assembly="..." namespace="...">

    <class name="Customer" table="customers">
        <id name="Id" column="id" generator="native" />
        <property name="Name" column="name"/>
        <set name="Orders" inverse="true" lazy="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="Id" column="id" generator="native" />
        <property name="Date" column="date"/>
        <many-to-one name="Customer" column="customer_id"/>
        <list name="LineItems" table="line_items" lazy="true">
            <key column="order_id"/>
            <index column="line_number"/>
            <composite-element class="LineItem">
                <property name="Quantity" column="quantity"/>
                <many-to-one name="Product" column="product_id"/>
            </composite-element>
        </list>
    </class>
    <class name="Product" table="products">
        <id name="Id" column="id">
            <generator class="native"/>
        </id>
        <property name="SerialNumber" column="serial_number" />
    </class>

</hibernate-mapping>
```

`customers`, `orders`, `line_items` and `products` hold customer, order, order line item and product data respectively. `line_items` also acts as an association table linking orders with products.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serial_number VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

Chapter 24. Best Practices

Write fine-grained classes and map them using `<component>`.

Use an `Address` class to encapsulate `street`, `suburb`, `state`, `postcode`. This encourages code reuse and simplifies refactoring.

Declare identifier properties on persistent classes.

NHibernate makes identifier properties optional. There are all sorts of reasons why you should use them. We recommend that identifiers be 'synthetic' (generated, with no business meaning) and of a non-primitive type. For maximum flexibility, use `Int64` or `String`.

Place each class mapping in its own file.

Don't use a single monolithic mapping document. Map `Eg.Foo` in the file `Eg/Foo.hbm.xml`. This makes particularly good sense in a team environment.

Embed mappings in assemblies.

Place mapping files along with the classes they map and declare them as `Embedded Resource`s in Visual Studio.

Consider externalising query strings.

This is a good practice if your queries call non-ANSI-standard SQL functions. Externalising the query strings to mapping files will make the application more portable.

Use parameters.

As in ADO.NET, always replace non-constant values by "?". Never use string manipulation to bind a non-constant value in a query! Even better, consider using named parameters in queries.

Don't manage your own ADO.NET connections.

NHibernate lets the application manage ADO.NET connections. This approach should be considered a last-resort. If you can't use the built-in connections providers, consider providing your own implementation of `NHibernate.Connection.IConnectionProvider`.

Consider using a custom type.

Suppose you have a type, say from some library, that needs to be persisted but doesn't provide the accessors needed to map it as a component. You should consider implementing `NHibernate.UserTypes.IUserType`. This approach frees the application code from implementing transformations to / from an NHibernate type.

Use hand-coded ADO.NET in bottlenecks.

In performance-critical areas of the system, some kinds of operations (eg. mass update / delete) might benefit from direct ADO.NET. But please, wait until you *know* something is a bottleneck. And don't assume that direct ADO.NET is necessarily faster. If need to use direct ADO.NET, it might be worth opening a NHibernate `ISession` and using that SQL connection. That way you can still use the same transaction strategy and underlying connection provider.

Understand `ISession` flushing.

From time to time the ISession synchronizes its persistent state with the database. Performance will be affected if this process occurs too often. You may sometimes minimize unnecessary flushing by disabling automatic flushing or even by changing the order of queries and other operations within a particular transaction.

In a three tiered architecture, consider using `SaveOrUpdate()`.

When using a distributed architecture, you could pass persistent objects loaded in the middle tier to and from the user interface tier. Use a new session to service each request. Use `ISession.Update()` or `ISession.SaveOrUpdate()` to update the persistent state of an object.

In a two tiered architecture, consider using session disconnection.

Database Transactions have to be as short as possible for best scalability. However, it is often neccessary to implement long running Application Transactions, a single unit-of-work from the point of view of a user. This Application Transaction might span several client requests and response cycles. Either use Detached Objects or, in two tiered architectures, simply disconnect the NHibernate Session from the ADO.NET connection and reconnect it for each subsequent request. Never use a single Session for more than one Application Transaction usecase, otherwise, you will run into stale data.

Don't treat exceptions as recoverable.

This is more of a necessary practice than a "best" practice. When an exception occurs, roll back the `ITransaction` and close the `ISession`. If you don't, NHibernate can't guarantee that in-memory state accurately represents persistent state. As a special case of this, do not use `ISession.Load()` to determine if an instance with the given identifier exists on the database; use `Get()` or a query instead.

Prefer lazy fetching for associations.

Use eager (outer-join) fetching sparingly. Use proxies and/or lazy collections for most associations to classes that are not cached in the second-level cache. For associations to cached classes, where there is a high probability of a cache hit, explicitly disable eager fetching using `fetch="select"`. When an outer-join fetch is appropriate to a particular use case, use a query with a `left join fetch`.

Consider abstracting your business logic from NHibernate.

Hide (NHibernate) data-access code behind an interface. Combine the *DAO* and *Thread Local Session* patterns. You can even have some classes persisted by handcoded ADO.NET, associated to NHibernate via an `IUserType`. (This advice is intended for "sufficiently large" applications; it is not appropriate for an application with five tables!)

Implement `Equals()` and `GetHashCode()` using a unique business key.

If you compare objects outside of the ISession scope, you have to implement `Equals()` and `GetHashCode()`. Inside the ISession scope, object identity is guaranteed. If you implement these methods, never ever use the database identifier! A transient object doesn't have an identifier value and NHibernate would assign a value when the object is saved. If the object is in an ISet<T> while being saved, the hash code changes, breaking the contract. To implement `Equals()` and `GetHashCode()`, use a unique business key, that is, compare a unique combination of class properties. Remember that this key has to be stable and unique only while the object is in an ISet<T>, not for the whole lifetime (not as stable as a database primary key). Never use collections in the `Equals()` comparison (lazy loading) and be careful with other associated classes that might be proxied.

Don't use exotic association mappings.

Good usecases for a real many-to-many associations are rare. Most of the time you need additional information stored in the "link table". In this case, it is much better to use two one-to-many associations to an intermediate link class. In fact, we think that most associations are one-to-many and many-to-one, you should be careful when using any other association style and ask yourself if it is really neccessary.

NHibernateContrib Documentation

Preface

The NHibernateContrib is various programs contributed to NHibernate by members of the NHibernate Team or by the end users. The projects in here are not considered core pieces of NHibernate but they extend it in a useful way.

Chapter 25. NHibernate.Caches

What is NHibernate.Caches?

NHibernate.Caches namespace contains several second-level cache providers for NHibernate. A cache is place where entities are kept after being loaded from the database; once cached, they can be retrieved without going to the database. This means that they are faster to (re)load.

An NHibernate session has an internal (first-level) cache where it keeps its entities. There is no sharing between these caches - a first-level cache belongs to a given session and is destroyed with it. NHibernate provides a *second-level cache* system; it works at the session factory level. A second-level cache is shared by all sessions created by the same session factory.

An important point is that the second-level cache *does not* cache instances of the object type being cached; instead it caches the individual values of the properties of that object. This provides two benefits. One, NHibernate doesn't have to worry that your client code will manipulate the objects in a way that will disrupt the cache. Two, the relationships and associations do not become stale, and are easy to keep up-to-date because they are simply identifiers. The cache is not a tree of objects but rather a map of arrays.

With the *session-per-request* model, a high number of sessions can concurrently access the same entity without hitting the database each time; hence the performance gain.

Several cache providers have been contributed by NHibernate users:

`NHibernate.Caches.Prevalence`

Uses `Bamboo.Prevalence` as the cache provider. Open the file `Bamboo.Prevalence.license.txt` for more information about its license; you can also visit its website.

`NHibernate.Caches.SysCache`

Uses `System.Web.Caching.Cache` as the cache provider. This means that you can rely on ASP.NET caching feature to understand how it works. For more information, read (on the MSDN): Caching Application Data.

`NHibernate.Caches.SysCache2`

Similar to `NHibernate.Caches.SysCache`, uses ASP.NET cache. This provider also supports SQL dependency-based expiration, meaning that it is possible to configure certain cache regions to automatically expire when the relevant data in the database changes.

SysCache2 requires Microsoft SQL Server 2000 or higher and .NET Framework version 2.0 or higher.

`NHibernate.Caches.MemCache`

Uses `memcached`. See memcached homepage for more information.

`NCache provider for NHibernate`

Uses `NCache`,NCache is a commercial distributed caching system with a provider for NHibernate. The NCache Express version is free for use, see NCache Express homepage for more information.

25.1. How to use a cache?

Here are the steps to follow to enable the second-level cache in NHibernate:

- Choose the cache provider you want to use and copy its assembly in your assemblies directory (`NHibernate.Caches.Prevalence.dll` or `NHibernate.Caches.SysCache.dll`).
- To tell NHibernate which cache provider to use, add in your NHibernate configuration file (can be `YourAssembly.exe.config` or `web.config` or a `.cfg.xml` file, in the latter case the syntax will be different from what is shown below):

```
<add key="hibernate.cache.provider_class" value="XXX" />(1)
<add key="expiration" value="120" />(2)
```

(1) "`XXX`" is the assembly-qualified class name of a class implementing `ICacheProvider`, eg. "`NHibernate.Caches.SysCache.SysCacheProvider, NHibernate.Caches.SysCache`".

(2) The `expiration` value is the number of seconds you wish to cache each entry (here two minutes). This example applies to SysCache only.

- Add `<cache usage="read-write|nonstrict-read-write|read-only"/>` (just after `<class>`) in the mapping of the entities you want to cache. It also works for collections (bag, list, map, set, ...).

Be careful. Caches are never aware of changes made to the persistent store by another process (though they may be configured to regularly expire cached data). As the caches are created at the session factory level, they are destroyed with the SessionFactory instance; so you must keep them alive as long as you need them.

25.2. Prevalence Cache Configuration

There is only one configurable parameter: `prevalenceBase`. This is the directory on the file system where the Prevalence engine will save data. It can be relative to the current directory or a full path. If the directory doesn't exist, it will be created.

25.3. SysCache Configuration

As SysCache relies on `System.Web.Caching.Cache` for the underlying implementation, the configuration is based on the available options that make sense for NHibernate to utilize.

`expiration`
: Number of seconds to wait before expiring each item.

`priority`
: A numeric cost of expiring each item, where 1 is a low cost, 5 is the highest, and 3 is normal. Only values 1 through 5 are valid.

SysCache has a config file section handler to allow configuring different expirations and priorities for different regions. Here's an example:

Example 25.1.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
        <configSections>
                <section name="syscache" type="NHibernate.Caches.SysCache.SysCacheSectionHandler,NHibernate.Caches.SysCache"
        </configSections>

        <syscache>
                <cache region="foo" expiration="500" priority="4" />
                <cache region="bar" expiration="300" priority="3" />
        </syscache>
</configuration>
```

25.4. SysCache2 Configuration

SysCache2 can use SqlCacheDependencies to invalidate cache regions when data in an underlying SQL Server table or query changes. Query dependencies are only available for SQL Server 2005. To use the cache provider, the application must be setup and configured to support SQL notifications as described in the MSDN documentation.

To configure cache regions with SqlCacheDependencies a `syscache2` config section must be defined in the application's configuration file. See the sample below.

Example 25.2.

```
<configSections>
        <section name="syscache2" type="NHibernate.Caches.SysCache2.SysCacheSection, NHibernate.Caches.SysCache2"/>
</configSections>
```

25.4.1. Table-based Dependency

A table-based dependency will monitor the data in a database table for changes. Table-based dependencies are generally used for a SQL Server 2000 database but will work with SQL Server 2005 or superior as well. Before you can use SQL Server cache invalidation with table based dependencies, you need to enable notifications for the database. This task is performed with the **aspnet_regsql** command. With table-based notifications, the application will poll the database for changes at a predefined interval. A cache region will not be invalidated immediately when data in the table changes. The cache will be invalidated the next time the application polls the database for changes.

To configure the data in a cache region to be invalidated when data in an underlying table is changed, a cache region must be configured in the application's configuration file. See the sample below.

Example 25.3.

```
<syscache2>
        <cacheRegion name="Product">
                <dependencies>
                        <tables>
                                <add name="price"
                                        databaseEntryName="Default"
                                        tableName="VideoTitle" />
                        </tables>
                </dependencies>
        </cacheRegion>
</syscache2>
```

Table-based Dependency Configuration Properties

`name`
: Unique name for the dependency

`tableName`
: The name of the database table that the dependency is associated with. The table must be enabled for notification support with the `AspNet_SqlCacheRegisterTableStoredProcedure`.

`databaseEntryName`
: The name of a database defined in the `databases` element for `sqlCacheDependency` for caching (ASP.NET Settings Schema) element of the application's `Web.config` file.

25.4.2. Command-Based Dependencies

A command-based dependency will use a SQL command to identify records to monitor for data changes. Command-based dependencies work only with SQL Server 2005.

Before you can use SQL Server cache invalidation with command-based dependencies, you need to enable the Service Broker for query notifications. The application must also start the listener for receiving change notifications from SQL Server. With command based notifications, SQL Server will notify the application when the data of a record returned in the results of a SQL query has changed. Note that a change will be indicated if the data in any of the columns of a record change, not just the columns returned by a query. The query is a way to limit the number of records monitored for changes, not the columns. As soon as data in one of the records is modified, the data in the cache region will be invalidated immediately.

To configure the data in a cache region to be invalidated based on a SQL command, a cache region must be configured in the application's configuration file. See the samples below.

Example 25.4. Stored Procedure

```
<cacheRegion name="Product" priority="High" >
        <dependencies>
                <commands>
                        <add name="price"
                                command="ActiveProductsStoredProcedure"
                                isStoredProcedure="true"/>
                </commands>
        </dependencies>
</cacheRegion>
```

Example 25.5. SELECT Statement

```
<cacheRegion name="Product" priority="High">
        <dependencies>
                <commands>
                        <add name="price"
                                command="Select VideoTitleId from dbo.VideoTitle where Active = 1"
                                connectionName="default"
                                connectionStringProviderType="NHibernate.Caches.SysCache2.ConfigConnectionStringProvider, NH
                </commands>
        </dependencies>
</cacheRegion>
```

Command Configuration Properties

`name`
: Unique name for the dependency

`command` (required)
: SQL command that returns results which should be monitored for data changes

`isStoredProcedure` (optional)
: Indicates if command is a stored procedure. The default is `false`.

`connectionName` (optional)
: The name of the connection in the applications configuration file to use for registering the cache dependency for change notifications. If no value is supplied for `connectionName` or `connectionStringProviderType`, the connection properties from the NHibernate configuration will be used.

`connectionStringProviderType` (optional)
: `IConnectionStringProvider` to use for retrieving the connection string to use for registering the cache dependency for change notifications. If no value is supplied for `connectionName`, the unnamed connection supplied by the provider will be used.

25.4.3. Aggregate Dependencies

Multiple cache dependencies can be specified. If any of the dependencies triggers a change notification, the data in the cache region will be invalidated. See the samples below.

Example 25.6. Multiple commands

```
<cacheRegion name="Product">
        <dependencies>
                <commands>
                        <add name="price"
                                command="ActiveProductsStoredProcedure"
                                isStoredProcedure="true"/>
                        <add name="quantity"
                                command="Select quantityAvailable from dbo.VideoAvailability"/>
                </commands>
        </dependencies>
</cacheRegion>
```

Example 25.7. Mixed

```
<cacheRegion name="Product">
        <dependencies>
                <commands>
                        <add name="price"
                                command="ActiveProductsStoredProcedure"
                                isStoredProcedure="true"/>
                </commands>
                <tables>
                        <add name="quantity"
                                databaseEntryName="Default"
                                tableName=" VideoAvailability" />
                </tables>
        </dependencies>
</cacheRegion>
```

25.4.4. Additional Settings

In addition to data dependencies for the cache regions, time based expiration policies can be specified for each item added to the cache. Time based expiration policies will not invalidate the data dependencies for the whole cache region, but serve as a way to remove items from the cache after they have been in the cache for a specified amount of time. See the samples below.

Example 25.8. Relative Expiration

```
<cacheRegion name="Product" relativeExpiration="300" priority="High" />
```

Example 25.9. Time of Day Expiration

```
<cacheRegion name="Product" timeOfDayExpiration="2:00:00" priority="High" />
```

Additional Configuration Properties

`relativeExpiration`

Number of seconds that an individual item will exist in the cache before being removed.

`timeOfDayExpiration`

24 hour based time of day that an item will exist in the cache until. 12am is specified as 00:00:00; 10pm is specified as 22:00:00. Only valid if relativeExpiration is not specified. Time of Day Expiration is useful for scenarios where items should be expired from the cache after a daily process completes.

`priority`

`System.Web.Caching.CacheItemPriority` that identifies the relative priority of items stored in the cache.

25.4.5. Patches

There is a known issue where some SQL Server 2005 notifications might not be received when an application subscribes to query notifications by using ADO.NET 2.0. To fix this problem install SQL hotfix for kb 913364.

Chapter 26. NHibernate.Mapping.Attributes

What is NHibernate.Mapping.Attributes?

NHibernate.Mapping.Attributes is an add-in for NHibernate contributed by Pierre Henri Kuaté (aka *KPixel*); the former implementation was made by John Morris. NHibernate require mapping streams to bind your domain model to your database. Usually, they are written (and maintained) in separated hbm.xml files.

With NHibernate.Mapping.Attributes, you can use .NET attributes to decorate your entities and these attributes will be used to generate these mapping .hbm.xml (as files or streams). So you will no longer have to bother with these *nasty* xml files ;).

Content of this library.

- NHibernate.Mapping.Attributes: That the only project you need (as end-user)
- Test: a working sample using attributes and HbmSerializer as NUnit TestFixture
- Generator: The program used to generate attributes and HbmWriter
- Refly : Thanks to Jonathan de Halleux for this library which make it so easy to generate code

Important

This library is generated using the file `/src/NHibernate.Mapping.Attributes/nhibernate-mapping.xsd` (which is embedded in the assembly to be able to validate generated XML streams). As this file can change at each new release of NHibernate, you should regenerate it before using it with a different version (open the Generator solution, compile and run the Generator project). But, no test has been done with versions prior to 0.8.

26.1. What's new?

NHibernate. introduces many new features, improvements and changes:

- It is possible to import classes by simply decorating them with `[Import] class ImportedClass1 {}`. Note that you must use `HbmSerializer.Serialize(assembly)`; The `<import/>` mapping will be added before the classes mapping. If you prefer to keep these imports in the class using them, you can specify them all on the class: `[Import(ClassType=typeof(ImportedClass1))] class Query {}`.

- `[RawXmlAttribute]` is a new attribute allowing to insert xml as-is in the mapping. This feature can be very useful to do complex mapping (eg: components). It may also be used to quickly move the mapping from xml files to attributes. Usage: `[RawXml(After=typeof(ComponentAttribute), Content="<component name="...">... </component>")]`. `After` tells after which kind of mapping the xml should be inserted (generally, it is the type of the mapping you are inserting); it is optional (in which case the xml is inserted on the top of the mapping). Note: At the moment, all raw xmls are prefixed by a `<!---->` (in the generated stream); this is a known side-effect.

- `[AttributeIdentifierAttribute]` is a new attribute allowing to provide the value of a defined "place holder". Eg:

  ```
  public class Base {
      [Id(..., Column="{{Id.Column}}")]
      [AttributeIdentifier(Name="Id.Column", Value="ID")] // Default value
      public int Id { ... }
  }
  [AttributeIdentifier(Name="Id.Column", Value="SUB_ID")]
  [Class] public class MappedSubClass : Base { }
  ```

  The idea is that, when you have a mapping which is shared by many subclasses but which has minor differences (like different column names), you can put the mapping in the base class with place holders on these fields and give their values in subclasses. Note that this is possible for any mapping field taking a string (column, name, type, access, etc.). And, instead of `Value`, you can use `ValueType` or `ValueObject` (if you use an enum, you can control its formatting with `ValueObject`).

  The "place holder" is defined like this: `{{XXX}}`. If you don't want to use these double curly brackets, you can change them using the properties `StartQuote` and `EndQuote` of the class `HbmWriter`.

- It is possible to register patterns (using Regular Expressions) to automatically transform fully qualified names of properties types into something else. Eg: `HbmSerializer.Default.HbmWriter.Patterns.Add(@"Namespace.(\S+), Assembly", "$1");` will map all properties with a not-qualified type name.

- Two methods have been added to allow writing: `cfg.AddInputStream( HbmSerializer.Default.Serialize(typeof(XXX)) )` and `cfg.AddInputStream( HbmSerializer.Default.Serialize(typeof(XXX).Assembly) )`. So it is no longer required to create a MemoryStream for these simple cases.

- Two `WriteUserDefinedContent()` methods have been added to `HbmWriter`. They improve the extensibility of this library; it is now very easy to create a .NET attribute and integrate it in the mapping.

- Attributes `[(Jcs)Cache]`, `[Discriminator]` and `[Key]` can be specified at class-level.

- Interfaces can be mapped (just like classes and structs).

- A notable "bug" fix is the re-ordering of (joined-)subclasses; This operation may be required when a subclass extends another subclass. In this case, the extended class mapping must come before the extending class mapping. Note that the re-ordering takes place only for "top-level" classes (that is not nested in other mapped classes). Anyway, it is quite unusual to put a interdependent mapped subclasses in a mapped class.

- There are also many other little changes; refer to the release notes for more details.

26.2. How to use it?

The *end-user class* is `NHibernate.Mapping.Attributes.HbmSerializer`. This class *serialize* your domain model to mapping streams. You can either serialize classes one by one or an assembly. Look at `NHibernate.Mapping.Attributes.Test` project for a working sample.

The first step is to decorate your entities with attributes; you can use: `[Class]`, `[Subclass]`, `[JoinedSubclass]` or `[Component]`. Then, you decorate your members (fields/properties); they can take as many attributes as required by your mapping. Eg:

```
    [NHibernate.Mapping.Attributes.Class]
    public class Example
    {
        [NHibernate.Mapping.Attributes.Property]
        public string Name;
    }
```

After this step, you use `NHibernate.Mapping.Attributes.HbmSerializer`: (here, we use `Default` which is an instance you can use if you don't need/want to create it yourself).

```
    NHibernate.Cfg.Configuration cfg = new NHibernate.Cfg.Configuration();
    cfg.Configure();
    NHibernate.Mapping.Attributes.HbmSerializer.Default.Validate = true; // Enable validation (optional)
    // Here, we serialize all decorated classes (but you can also do it class by class)
    cfg.AddInputStream( NHibernate.Mapping.Attributes.HbmSerializer.Default.Serialize(
        System.Reflection.Assembly.GetExecutingAssembly() ); );
    // Now you can use this configuration to build your SessionFactory...
```

Note

As you can see here: NHibernate.Mapping.Attributes is not (really) intrusive. Setting attributes on your objects doesn't force you to use them with NHibernate and doesn't break any constraint on your architecture. Attributes are purely informative (like documentation)!

26.3. Tips

- In production, it is recommended to generate a XML mapping file from NHibernate.Mapping.Attributes and use this file each time the SessionFactory need to be built. Use: `HbmSerializer.Default.Serialize(typeof(XXX).Assembly, "DomainModel.hbm.xml");` It is slightly faster.

- Use `HbmSerializer.Validate` to enable/disable the validation of generated xml streams (against NHibernate mapping schema); this is useful to quickly find errors (they are written in StringBuilder `HbmSerializer.Error`). If the error is due to this library then see if it is a know issue and report it; you can contribute a solution if you solve the problem :)

- Your classes, fields and properties (members) can be private; just make sure that you have the permission to access private members using reflection (`ReflectionPermissionFlag.MemberAccess`).

- Members of a mapped classes are also seek in its base classes (until we reach *mapped* base class). So you can decorate some members of a (not mapped) base class and use it in its (mapped) sub class(es).

- For a Name taking a `System.Type`, set the type with `Name="xxx"` (as `string`) or `NameType=typeof(xxx);` (add "`Type`" to "`Name`")

- By default, .NET attributes don't keep the order of attributes; so you need to set it yourself when the order matter (using the first parameter of each attribute); it is *highly* recommended to set it when you have more than one attribute on the same member.

- As long as there is no ambiguity, you can decorate a member with many unrelated attributes. A good example is to put class-related attributes (like `<discriminator>`) on the identifier member. But don't forget that the order matters (the `<discriminator>` must be after the `<id>`). The order used comes from the order of elements in the NHibernate mapping schema. Personally, I prefer using negative numbers for these attributes (if they come before!).

- You can add `[HibernateMapping]` on your classes to specify `<hibernate-mapping>` attributes (used when serializing the class in its stream). You can also use `HbmSerializer.Hbm*` properties (used when serializing an assembly or a type that is not decorated with `[HibernateMapping]`).

- Instead of using a string for `DiscriminatorValue` (in `[Class]` and `[Subclass]`), you can use any object you want. Example:

  ```
  [Subclass(DiscriminatorValueEnumFormat="d", DiscriminatorValueObject=DiscEnum.Val1)]
  ```

  Here, the object is an Enum, and you can set the format you want (the default value is "g"). Note that you must put it before! For others types, It simply use the `ToString()` method of the object.

- If you are using members of the type `Nullables.NullableXXX` (from the library Nullables), then they will be mapped to `Nullables.NHibernate.NullableXXXType` automatically; don't set `Type="..."` in `[Property]` (leave it null). This is also the case for `SqlTypes` (and you can add your own patterns). Thanks to *Michael Third* for the idea :)

- Each stream generated by NHibernate.Mapping.Attributes can contain a comment with the date of the generation; You may enable/disable this by using the property `HbmSerializer.WriteDateComment`.

- If you forget to provide a required xml attribute, it will obviously throw an exception while generating the mapping.

- The recommended and easiest way to map `[Component]` is to use `[ComponentProperty]`. The first step is to put `[Component]` on the component class and map its fields/properties. Note that you shouldn't set the `Name` in `[Component]`. Then, on each member in your classes, add `[ComponentProperty]`. But you can't override `Access`, Update or Insert for each member.

There is a working example in *NHibernate.Mapping.Attributes.Test* (look for the class CompAddress and its usage in others classes).

- Another way to map [Component] is to use the way this library works: If a mapped class contains a mapped component, then this component will be include in the class. *NHibernate.Mapping.Attributes.Test* contains the classes JoinedBaz and Stuff which both use the component Address.

  Basically, it is done by adding

  ```
  [Component(Name = "MyComp")] private class SubComp : Comp {}
  ```

  in each class. One of the advantages is that you can override Access, Update or Insert for each member. But you have to add the component subclass in each class (and it can not be inherited). Another advantage is that you can use [AttributeIdentifier].

- Finally, whenever you think that it is easier to write the mapping in XML (this is often the case for [Component]), you can use [RawXml].

- **About customization.** HbmSerializer uses HbmWriter to serialize each kind of attributes. Their methods are virtual; so you can create a subclass and override any method you want (to change its default behavior).

  Use the property HbmSerializer.HbmWriter to change the writer used (you may set a subclass of HbmWriter).

Example using some this tips: (0, 1 and 2 are position indexes)

```
    [NHibernate.Mapping.Attributes.Id(0, TypeType=typeof(int))] // Don't put it after [ManyToOne] !!!
        [NHibernate.Mapping.Attributes.Generator(1, Class="uuid.hex")]
    [NHibernate.Mapping.Attributes.ManyToOne(2, ClassType=typeof(Foo), OuterJoin=OuterJoinStrategy.True)]
    private Foo Entity;
```

Generates:

```
    <id type="Int32">
        <generator class="uuid.hex" />
    </id>
    <many-to-one name="Entity" class="Namespaces.Foo, SampleAssembly" outer-join="true" />
```

26.4. Known issues and TODOs

First, read TODOs in the source code ;)

A Position property has been added to all attributes to order them. But there is still a problem:

When a parent element "p" has a child element "x" that is also the child element of another child element "c" of "p" (preceding "x") :D Illustration:

```
    <p>
        <c>
            <x />
        </c>
        <x />
    </p>
```

In this case, when writing:

```
    [Attributes.P(0)]
        [Attributes.C(1)]
            [Attributes.X(2)]
        [Attributes.X(3)]
    public MyType MyProperty;
```

X(3) will always belong to C(1) ! (as X(2)).

It is the case for <dynamic-component> and <nested-composite-element>.

Another bad news is that, currently, XML elements coming after this elements can not be included in them. Eg: There is no way put a collection in <dynamic-component>. The reason is that the file nhibernate-mapping.xsd tells how elements are built and in which order, and NHibernate.Mapping.Attributes use this order.

Anyway, the solution would be to add a int ParentNode property to BaseAttribute so that you can create a real graph...

For now, you can fallback on `[RawXml]`.

Actually, there is no other know issue nor planned modification. This library should be stable and complete; but if you find a bug or think of an useful improvement, contact us!

On side note, it would be nice to write a better TestFixture than *NHibernate.Mapping.Attributes.Test* :D

26.5. Developer Notes

Any change to the schema (`nhibernate-mapping.xsd`) implies:

- Checking if there is any change to do in the Generator (like updating KnowEnums / AllowMultipleValue / IsRoot / IsSystemType / IsSystemEnum / CanContainItself)
- Updating `/src/NHibernate.Mapping.Attributes/nhibernate-mapping.xsd` (copy/paste) and running the Generator again (even if it wasn't modified)
- Running the Test project and make sure that no exception is thrown. A class/property should be modified/added in this project to be sure that any new breaking change will be caught (=> update the reference hbm.xml files and/or the project `NHibernate.Mapping.Attributes-2.0.csproj`)

This implementation is based on NHibernate mapping schema; so there is probably lot of "standard schema features" that are not supported...

The version of NHibernate.Mapping.Attributes should be the version of the NHibernate schema used to generate it (=> the version of NHibernate library).

In the design of this project, performance is a (*very*) minor goal :) Easier implementation and maintenance are far more important because you can (and should) avoid to use this library in production (Cf. the first tip in Section 26.3, “Tips”).