.NET

Reference Documentation

Version 1.0.2

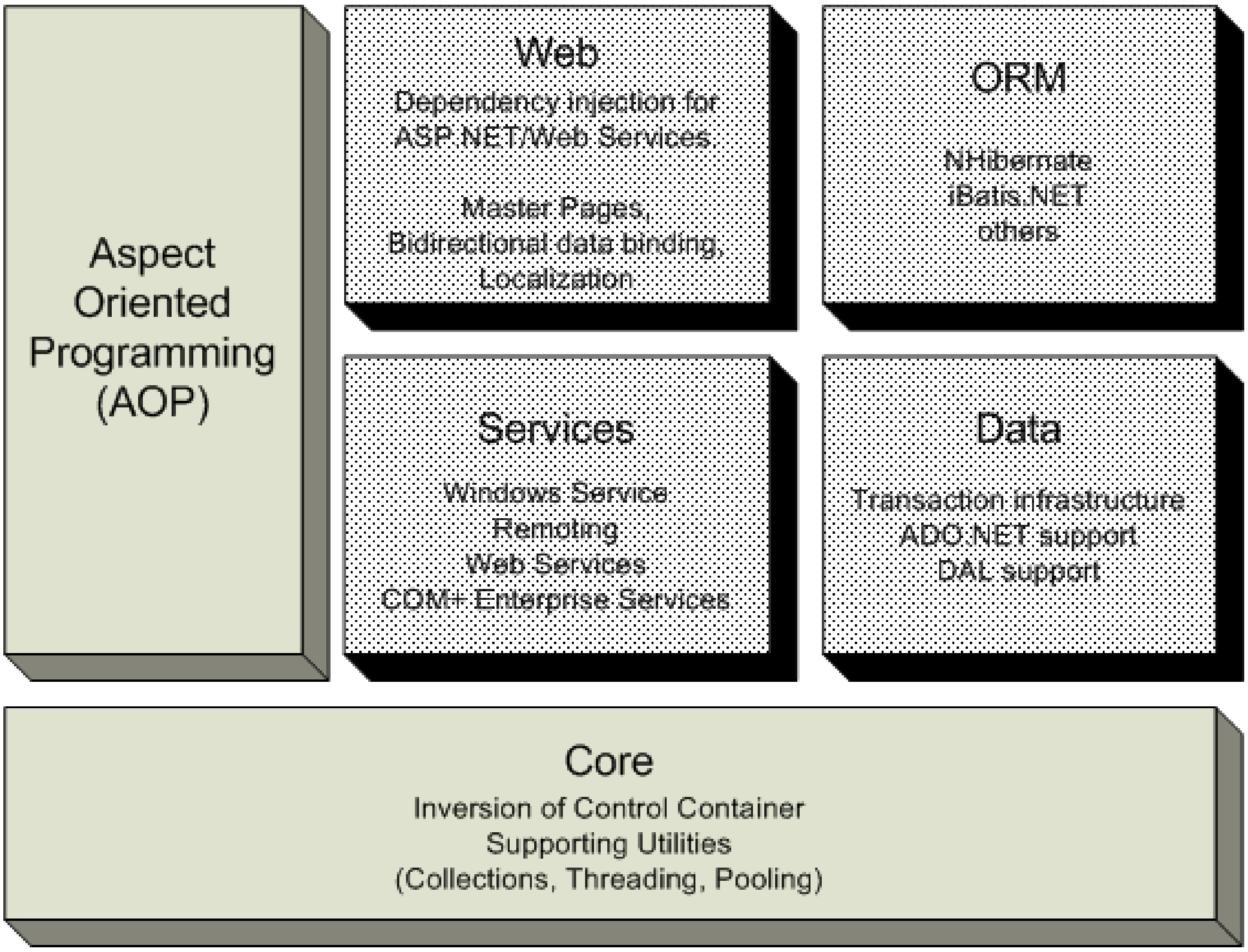
April 27, 2006 - (Work in progress)

Table of Contents

Chapter 1. Introduction

1.1. Overview

Spring.NET is an application framework focused on helping build enterprise .NET applications. It provides a wide range of functionality such as Dependency Injection, Aspect Oriented Programming (AOP), data access abstractions, and ASP.NET integration. Based on the Spring Framework for Java, the core concepts and values found in Spring.Java have been applied to .NET. The 1.0 release of Spring.NET contains a full featured Inversion of Control container and an AOP library. Subsequent releases will contain support for ASP.NET, Remoting, and data access. The diagram below shows the various modules of Spring .NET. The dark shaded modules are in the 1.0 release while the other modules are planned for future releases. In many cases you can already find working implementations for the planned modules available on our download site.


Web
Dependency injection for ASP.NET/Web Services
Master Pages, Bidirectional data binding, Localization
ORM
NHibernate
iBatis.NET
others
Aspect Oriented Programming (AOP)
Services
Windows Service
Remoting
Web Services
COM+ Enterprise Services
Data
Transaction infrastructure
ADO.NET support
DAL support
Core
Inversion of Control Container
Supporting Utilities
(Collections, Threading, Pooling)

The *Spring.Core* library is the most fundamental part of the framework, and provides Dependency Injection functionality. Most of the libraries in the Spring.NET distribution depend upon and extend the functionality provided by this core library. The basic concept here is provided by the `IObjectFactory` interface that provides a simple yet elegant factory pattern removing the need for programmatic singletons and numerous service locator stubs, allowing you to decouple the configuration and specification of dependencies from your actual program logic. An extension to the `IObjectFactory`, the `IApplicationContext` is also located here and adds more enterprise-centric features such as text localization using resource files, event-propagation, and resource-loading.

The *Spring.Aop* library provides Aspect Oriented Programming (AOP) support to your business objects. The

Spring.Aop library complements the IoC container in the Spring.Core library to provide a rock-solid foundation for building enterprise applications and applying services to business objects declaratively.

The *Spring.Web* library extends ASP.NET by adding a variety of features such as Dependency Injection for ASP.NET pages, Bidirectional data binding, Master pages for ASP.NET 1.1 and improved localization support.

The *Spring.Services* library let you expose any "normal" object (meaning an object that does not inherit from a special service base class) as an enterprise (COM+) service or remoting object. .NET Web services get additional configuration flexibility with support for depedency injection and overriding of attribute metadata. Windows Service intergration is also provided.

The *Spring.Data* library provides a Data Access Layer abstraction that can be used across a variety of data access providers, from ADO.NET to various ORM providers. It also contains an ADO.NET abstraction layer that removes the need for tedious coding and declarative transaction management for ADO.NET.

The *Spring.ORM* library provides integration layers for popular object relational mapping libraries. This provides functionality such as support for declarative transaction management

This document provides a reference guide to Spring.NET's features. Since this document is still very much a work-in-progress endeavour, if you have any requests or comments, please post them on the user forums at http://forum.springframework.net/ The latest version of this document can be found at http://www.springframework.net/doc/reference/index.html

Before we go on, a few words of gratitude are due to Chris Bauer (of the Hibernate project team), who prepared and adapted the DocBook-XSL software used to create Hibernate's reference guide, also allowing us to create this guide.

Chapter 2. Background information

2.1. Inversion of Control

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: *"the question, is what aspect of control are they inverting?"*. After talking about the term Inversion of Control Martin suggests renaming the pattern, or at least giving it a more self-explanatory name, and starts to use the term *Dependency Injection*. His article continues to explain some of the ideas behind this important software engineering principle.

Chapter 3. Objects, Object Factories, and Application Contexts

3.1. Introduction

(Available in 1.0)

The `Spring.Core` assembly provides the basis for the Spring.NET Inversion of Control (IoC - sometimes also referred to as Dependency Injection) features (see Section 2.1, “Inversion of Control” for some additional material describing this software engineering principle). The `IObjectFactory` interface from the `Spring.Core` assembly provides an advanced configuration mechanism capable of managing objects of any nature, using potentially any kind of storage facility. The `IApplicationContext` interface from the same assembly builds on top of the functionality provided by the `IObjectFactory` interface, complementing it with features such as integration with Spring.NET's Aspect Oriented Programming (AOP) features and message resource handling (for use in internationalization).

In short, the `IObjectFactory` provides the configuration framework and basic functionality, while the `IApplicationContext` adds more enterprise-centric functionality to it. In general, the `IApplicationContext` is a complete superset of the `IObjectFactory`, and any description of `IObjectFactory` capabilities and behavior should be considered to apply to `IApplicationContexts` as well.

This chapter is divided into two parts, with the first part covering the basic principles that apply to both the `IObjectFactory` and `IApplicationContext`, with the second part covering those features that apply only to the `IApplicationContext` interface.

If you are new to Spring.NET or IoC containers in general, you may want to consider starting with Chapter 16, *Quickstarts*, which contains a number of introductory level examples that actually demonstrate a lot of what is described in detail below. Don't worry if you don't absorb everything at once... those examples serve only to paint a picture of how Spring.NET hangs together in really broad brushstrokes. Once you have finished with those examples, you can come back to this section which will fill in all the fine detail.

3.2. Introduction to the IObjectFactory, IApplicationContext, and IObjectDefinition

3.2.1. The IObjectFactory and IApplicationContext

The `IObjectFactory` is the actual container that instantiates, configures, and manages a number of objects. These objects typically collaborate with one another, and thus can be said to have dependencies between themselves. These dependencies are reflected in the configuration data used by the `IObjectFactory` (although some dependencies may not be visible as configuration data, but rather be a function of programmatic interactions between objects at runtime).

An `IObjectFactory` is represented by the `Spring.Objects.Factory.IObjectFactory` interface, of which there are multiple implementations. The most commonly used simple `IObjectFactory` is the `Spring.Objects.Factory.Xml.XmlObjectFactory`. Interaction with the `IObjectFactory` interface is discussed in Section 3.7, “Interacting with the IObjectFactory”. Additional features offered by the `IApplicationContext` are discussed in section Section 3.12, “Introduction to the IApplicationContext”.

As mentioned previously in the introduction, one of the central tenets of the Spring.NET framework is non-invasiveness. Quite simply, your application code should not depend on any of the Spring.NET APIs. However, if one is going to take advantage of the features provided by Spring.NET's IoC container, through the use of either the `IObjectFactory` or the `Spring.Context.IApplicationContext` interfaces, at some point one **has** to instantiate an appropriate implementation of either of these two core interfaces. This can happen explicitly in user code via the use of the `new` operator (in C# - VB.NET developers have the equivalent `New` operator); or more easily by using a custom configuration section in the standard .NET application (or web) configuration file. Once the container has been created you may never need to explicitly interact with it again in your code.

What follows are some examples of how one can instantiate an actual implementation of the `IObjectFactory` interface. In the following example an object factory, complete with object definitions describing the services that we want to wire up and expose, is created from the contents of the `objects.xml` file; this file is passed in as an argument to one of the constructors of the `XmlObjectFactory` class.

```
[C#]
IResource input = new FileSystemResource ("objects.xml");
IObjectFactory factory = new XmlObjectFactory(input);
```

The above example uses Spring.NET's `IResource` abstraction. The `IResource` interface provides a simple and uniform interface to a wide array of IO resources that can represent themselves as `System.IO.Streams`. The `IResource` abstraction is explained further in Section 3.11, “The IResource abstraction”. These resources are most frequently files or URLs but can also be resources that have been embedded inside a .NET assembly. A simple URI syntax is used to describe the location of the resource, which follows the standard conventions for files, i.e. `file://object.xml` and other well known protocols such as http.

As previously mentioned, one is more likely to use the `IApplicationContext` interface. Any of the `IApplicationContext` implementations can be instantiated explicitly by specifying `IResource` URI locations to the constructor. Multiple configuration files can used to construct an `IApplicationContext` as shown below.

```
IApplicationContext context = new XmlApplicationContext(
        "file://objects.xml",
        "assembly://MyAssembly/MyProject/objects-dal-layer.xml");

// of course, an IApplicationContext is also an IObjectFactory...
IObjectFactory factory = (IObjectFactory) context;
```

The following snippet shows the use of the URI syntax for referring to a resource that has been embedded inside a .NET assembly, `assembly://<AssemblyName>/<NameSpace>/<ResourceName>`

Note

To create an embedded resource using Visual Studio you must set the Build Action of the .xml configuration file to Embedded Resource in the file property editor. Also, you will need to explicitly rebuild the project containing the configuration file if it is the only change you make between successive builds. If using NAnt to build, add a <resources> section to the csc task. For example usage, look at the Spring.Core.Tests.build file included the distribution.

The preferred way to create an `IApplicationContext` or `IObjectFactory` is to use a custom configuration section in the standard .NET application configuration file (one of `App.config` or `Web.config`). A custom configuration section that creates the same `IApplicationContext` as the previous example is

```
<spring>
  <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
    <resource uri="file://objects.xml"/>
    <resource uri="assembly://MyAssembly/MyProject/objects-dal-layer.xml"/>
  </context>
</spring>
```

The context type (specified as the value of the `type` attribute of the `context` element) is wholly optional, and defaults to the `Spring.Context.Support.XmlApplicationContext` class, so the following XML snippet is functionally equivalent to the first.

```
<spring>
  <context>
    <resource uri="file://objects.xml"/>
    <resource uri="assembly://MyAssembly/MyProject/objects-dal-layer.xml"/>
  </context>
</spring>
```

To acquire a reference to an `IApplicationContext` using a custom configuration section, one simply uses the following code; please note that the string literal '`spring/context`' is not arbitrary... you **must** use this string value (a constant is available in the `AbstractApplicationContext` class).

```
IApplicationContext ctx
        = ContextRegistry.GetContext();
```

The `ContextRegistry` is used to both instantiate the application context and to perform service locator style access to other objects. (See Section 3.17, “Service Locator access” for more information). The glue that makes this possible is an implementation of the Base Class Library (BCL) provided `IConfigurationSectionHandler` interface, namely the `Spring.Context.Support.ContextHandler` class. The handler class needs to be registered in the `configSections` section of the .NET configuration file as shown below.

```
<configSections>
  <sectionGroup name="spring">
    <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
  </sectionGroup>
</configSections>
```

This declaration now enables the use of a custom context section starting at the `spring` root element.

In some usage scenarios, user code will not have to explicitly instantiate an appropriate implementation of the `IObjectFactory` interface, since Spring.NET code will do it. For example, the ASP.NET web layer provides support code to load a Spring.NET `IApplicationContext` automatically as part of the normal startup process of an ASP.NET web application. Similar support for WinForms applications is being investigated.

While programmatic manipulation of `IObjectFactory` instances will be described later, the following sections will concentrate on describing the configuration of objects managed by `IObjectFactory` instances.

An `IObjectFactory` configuration consists of, at its most basic level, definitions of one or more objects that the `IObjectFactory` will manage. In an XML based factory these are defined as one or more `object` elements inside a top-level `objects` element.

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net
                http://www.springframework.net/xsd/spring-objects.xsd">
  <object id="..." type="...">
  ...
  </object>
  <object id="...." type="...">
  ...
  </object>
  ...
</objects>
```

Spring.NET comes with an XSD schema to make the validation of the XML object definitions a whole lot easier. The XSD document is thoroughly documented so feel free to take a peak inside (see Appendix A, *Spring.NET's spring-objects.xsd*). The XSD is currently used in the implementation code to validate the XML document. The XSD schema serves a dual purpose in that it also facilitates the editing of XML object definitions inside an XSD aware editor (typically VisualStudio.NET) by providing validation (and Intellisense support in the case of VisualStudio.NET). You may wish to refer to Chapter 15, *Visual Studio.NET Integration* for more information regarding such integration. You can also obtain the XSD that supports the latest release from the web at spring-objects.xsd.

Your XML object definitions can also be defined within the standard .NET application configuration file by registering the `Spring.Context.Support.DefaultSectionHandler` class as the configuration section handler for inline object definitions. This allows you to completely configure one or more `IApplicationContext` instances within a single standard .NET application configuration file as shown in the following example.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>

  <spring>

    <context>
      <resource uri="config://spring/objects"/>
    </context>

    <objects>
        ...
    </objects>

  </spring>

</configuration>
```

Other options available to structure the configuration files are described in Section 3.14.1, “Context Hierarchies” and Section 3.16, “Importing Object Definitions from One File Into Another”.

The `IApplicationContext` can be configured to register other resource handlers, custom parsers to integrate user-contributed XML schema into the object definitions section, type converters, and define type aliases. These features are discussed in section Section 3.13, “Configuration of IApplicationContext”

3.2.2. The IObjectDefinition

Object definitions describe objects managed by an `IObjectFactory` or `IApplicationContext`. Object definitions contain the following information:

- The object type, which is the actual implementation class (the .NET `System.Type`) of the object being described in the object definition.
- Object behavioral configuration elements, which state how the object should behave in the Spring.NET IoC container (i.e. prototype or singleton, autowiring mode, dependency checking mode, initialization and destruction methods).
- Property values to set in the newly created object. An example would be the number of threads to use in an object that manages a worker thread pool (either specified as a property or as a constructor argument), or the System.Type that should be used to create the thread pool.
- Other objects your object needs to do its work, i.e. *collaborators* (also specified as properties or as constructor arguments). These can also be called dependencies.

In the list above, we mentioned the use of property setters and constructor arguments. Spring.NET supports two types of IoC: Type 2 and Type 3 (Constructor Dependency Injection and Setter Dependency Injection respectively). What that basically means is that when new objects are constructed by the IoC container you can set properties of the object using both regular property setters, and also directly as arguments that you specify to a constructor.

The concepts listed above directly translate to a set of elements the object definition consists of. These elements are listed below, along with a link to further documentation about each of them.

Table 3.1. Object definition explanation

Feature	More info
type	Section 3.2.3, “Object Creation”
id and name	Section 3.2.5, “The object identifiers (id and name)”
singleton or prototype	Section 3.2.6, “Singleton & Prototype Scope”
object properties	Section 3.3.1, “Setting object properties and collaborators”
constructor arguments	Section 3.3.1, “Setting object properties and collaborators”
autowiring mode	Section 3.3.8, “Autowiring collaborators”
dependency checking mode	Section 3.3.9, “Checking for dependencies”
initialization method	Section 3.5.1, “Lifecycle interfaces”
destruction method	Section 3.5.1, “Lifecycle interfaces”

3.2.3. Object Creation

Every object definition needs to know the type (the .NET `System.Type`) of the object being defined (see Section 3.2.3.3, “Object creation via an instance factory method”, and Section 3.6, “Abstract and Child object definitions”for the exceptions). In the much more common case where the `IObjectFactory` itself directly creates the object instance by calling one of the objects constructors the type attribute specifies the type of the object that is to be instantiated. In the less common case where the `IObjectFactory` calls a so-called factory method on a type to create the object instance, the type attribute specifies the actual type containing the factory method. The type of the object returned from the invocation of this factory method may be the same type, or another type entirely, it doesn't matter.

3.2.3.1. Object creation via constructor invocation

When creating an object using the constructor approach, there are no special requirements as to what this class is or how it is implemented (i.e. your class does not have to implement a special interface to make it Spring.NET compatible), other than the fact that it must not be an interface. Just specifying the object type (and the assembly into which it has been compiled) should be enough. However, depending on what type of IoC you are going to use for that specific object, you may need to create a default constructor (i.e. a constructor that has no parameters) in the source code definition of your class.

The `XmlObjectFactory` implementation of the `IObjectFactory` interface can consume object definitions that have been defined in XML, for example...

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary"/>
```

This XML fragment describes an object definition that will be identified by the *exampleObject* name, instances of which will be of the `Examples.ExampleObject` type that has been compiled into the `ExamplesLibrary` assembly. Take special note of the structure of the `type` attribute's value... the namespace-qualified name of the class is specified, followed by a comma, followed by (at a bare minimum) the name of the assembly that contains the class. In the preceding example, the `ExampleObject` class is defined in the `Examples` namespace, and it has been compiled into the `ExamplesLibrary` assembly.

The name of the assembly that contains the type *must* be specified in the `type` attribute. Furthermore, it is recommended that you specify the fully qualified assembly name [1] in order to guarantee that the type that Spring.NET uses to instantiate your object (s) is indeed the one that you expect. Usually this is only an issue if you are using classes from (strongly named) assemblies that have been installed into the Global Assembly Cache (GAC).

If you are defining classes that have been compiled into assemblies that are available to your application (such as the `bin` directory in the case of ASP.NET applications) via the standard assembly probing mechanisms, then you can specify simply the name of the assembly (e.g. `ExamplesLibrary.Data`)... this way, when (or if) the assemblies used by your application are updated, you won't have to change the value of every `<object/>` definition's `type` attribute to reflect the new version number (if the version number has changed)... Spring.NET will automatically locate and use the newer versions of your assemblies (and their attendant classes) from that point forward.

3.2.3.2. Object creation via a static factory method

When defining an object which is to be created using a static factory method, along with the type attribute which specifies the type containing the static factory method, another attribute named factory-method is needed to specify the name of the factory method itself. Spring.NET expects to be able to call this method (with an optional list of arguments as described later) and get back a live object, which from that point on is treated as if it had been created normally via a constructor. One use for such an object definition is to call static factories in legacy code.

Following is an example of an object definition which specifies that the object is to be created by calling a factory-method. Note that the definition does not specify the type (class) of the returned object, only the type containing the factory method. In this example, `CreateInstance` must be a static method.

```
<object id="exampleObject"
      type="Examples.ExampleObjectFactory, ExamplesLibrary"
      factory-method="CreateInstance"/>
```

The mechanism for supplying (optional) arguments to the factory method, or setting properties of the object instance after it has been returned from the factory, will be described shortly.

3.2.3.3. Object creation via an instance factory method

Quite similar to using a static factory method to create an object, is the the use of an instance (non-static) factory method, where a factory method of an existing object from the factory is called to create the new object.

To use this mechanism, the type attribute must be left empty, and the factory-object attribute must specify the name of an object in the current or an ancestor object factory which contains the factory method. The factory method itself should still be set via the factory-method attribute.

Following is an example...

```
<!-- the factory object, which contains an instance method called 'CreateInstance' -->
```

[1] More information about assembly names can be found in the *Assembly Names* section of the .NET Framework Developer's Guide (installed as part of the .NET SDK), or online at Microsoft's MSDN website, by searching for *Assembly Names*.

```
<object id="exampleFactory" type="..."/>
<!-- the object that is to be created by the factory object -->
<object id="exampleObject"
      factory-method="CreateInstance"
      factory-object="exampleFactory"/>
```

Although the mechanisms for setting object properties are still to be discussed, one implication of this approach is that the factory object itself can be managed and configured via Dependency Injection, by the container.

3.2.4. Object creation of generic types

Generic types can also be created in much the same manner an non-generic types.

3.2.4.1. Object creation of generic types via constructor invocation

The following examples shows the definition of simple generic types and how they can be created in Spring's XML based configuration file.

```
namespace GenericsPlay
{
    public class FilterableList<T>
    {
        private List<T> list;

        private String name;

        public List<T> Contents
        {
            get { return list; }
            set { list = value; }
        }

        public String Name
        {
            get { return name; }
            set { name = value; }
        }

        public List<T> ApplyFilter(string filterExpression)
        {
            /// should really apply filter to list ;)
            return new List<T>();
        }

    }
}
```

The XML configuration to create and configure this object is shown below

```
<object id="myFilteredIntList" type="GenericsPlay.FilterableList&lt;int>, GenericsPlay">
  <property name="Name" value="My Integer List"/>
</object>
```

There are a few items to note in terms how to specify a generic type. First, the left bracket that specifies the generic type, i.e. <, is replaced with the string &lt; due to XML escape syntax for the less than symbol. Yes, we all realize this is less than ideal from the readability point of view. Second, the generic type arguments can not be fully assembly qualified as the comma is used to seperate generic type arguments. Alternative characters used to overcome the two quirks can be implemented in the future but so far, all proposals don't seem to help clarify the text. The suggested solution to improve readability is to use type aliases as shown below.candidate

```

<typeAliases>
 <alias name="GenericDictionary" type=" System.Collections.Generic.Dictionary&lt;,>" />
 <alias name="myDictionary" type="System.Collections.Generic.Dictionary&lt;int,string>" />
</typeAliases>
```

So that instead of something like this

```
<object id="myGenericObject"
        type="GenericsPlay.ExampleGenericObject&lt;System.Collections.Generic.Dictionary&lt;int , string>>, Gene
```

It can be shortened to

```
<object id="myOtherGenericObject"
        type="GenericsPlay.ExampleGenericObject&lt;GenericDictionary&lt;int , string>>, GenericsPlay" />
```

or even shorter

```
<object id="myOtherOtherGenericObject"
        type="GenericsPlay.ExampleGenericObject&lt;MyIntStringDictionary>, GenericsPlay" />
```

Refer to Section 3.13, “Configuration of IApplicationContext” for additional information on using type aliases.

3.2.4.2. Object creation of generic types via static factory method

The following classes are used to demonstrate the ability to create instances of generic types that themselves are created via a static generic factory method.

```
public class TestGenericObject<T, U>
{
    public TestGenericObject()
    {
    }

    private IList<T> someGenericList = new List<T>();

    private IDictionary<string, U> someStringKeyedDictionary =
        new Dictionary<string, U>();

    public IList<T> SomeGenericList
    {
        get { return someGenericList; }
        set { someGenericList = value; }
    }

    public IDictionary<string, U> SomeStringKeyedDictionary
    {
        get { return someStringKeyedDictionary; }
        set { someStringKeyedDictionary = value; }
    }

}
```

The accompanying factory class is

```

public class TestGenericObjectFactory
{
    public static TestGenericObject<V, W> StaticCreateInstance<V, W>()
    {
        return new TestGenericObject<V, W>();
    }

    public TestGenericObject<V, W> CreateInstance<V, W>()
    {
        return new TestGenericObject<V, W>();
    }
}
```

The XML snippit to create an instance of TestGenericObject where `V` is a List of integers and `W` is an integer is shown below

```
<object id="myTestGenericObject"
        type="GenericsPlay.TestGenericObjectFactory, GenericsPlay"
        factory-method="StaticCreateInstance&lt;System.Collections.Generic.List&lt;int>,int>"
/>
```

The StaticCreateInstance method is responsible for instantiating the object that will be associated with the id 'myTestGenericObject'.

3.2.4.3. Object creation of generic types via instance factory method

Using the class from the previous example the XML snippit to create an instance of a generic type via an instance factory method is shown below

```
<object id="exampleFactory" type="GenericsPlay.TestGenericObject&lt;int,string>, GenericsPlay"/>

<object id="anotherTestGenericObject"
        factory-object="exampleFactory"
        factory-method="CreateInstance&lt;System.Collections.Generic.List&lt;int>,int>"/>
```

This creates an instance of `TestGenericObject<List<int>,int>`

3.2.5. The object identifiers (`id` and `name`)

Every object has one or more ids (also called identifiers, or names; these terms refer to the same thing). These ids must be unique within the `IObjectFactory` or `IApplicationContext` that the object definition is hosted in. An object definition will almost always have only one id, but if an object has more than one id, the extra ones can essentially be considered aliases.

In an XML object definition, the `id` or `name` attributes of the `object` element are used to specify the object definition's id (s), and at least one id must be specified in one or both of these attributes. The `id` attribute allows you to specify one id, and since it is marked in the Spring.NET XSD as a bona-fide XML element ID attribute, the parser is able to do some extra validation when other elements point back to this one. As such, it is the preferred way to specify an object id. However, the XML specification does limit the characters that are legal in XML IDs. This is usually not a constraint, but if you have a need to use one of these characters, or want to introduce aliases to the object, you may also or instead specify one or more object ids (separated by a comma (,) or semicolon (;)) via the `name` attribute.

3.2.6. Singleton & Prototype Scope

Objects can be deployed in one of two modes: singleton or non-singleton (the latter is also called a prototype, although the term is used loosely as it doesn't quite fit). When an object definition is set to the singleton mode, only one *shared* instance of the object will be managed, and all requests for objects with an id or ids matching that object definition will result in that one specific object instance being returned (i.e. the object defined by the object definition will only ever be created *once*).

The non-singleton, prototype mode of an object deployment results in the creation of a *new* object instance every time a request for that specific object is received. For example, this is ideal for those situations where each user needs an independent user object or something similar.

Object definitions operate in singleton mode by default, unless you specify otherwise. Keep in mind that by changing the mode to non-singleton (prototype), each request for an object will result in a new instance and this might not be what you actually want. So only change the mode to prototype when absolutely necessary.

Note

When deploying an object in the prototype mode, the lifecycle of the object changes slightly. By definition, Spring.NET cannot manage the complete lifecycle of a non-singleton / prototype object, since after it is created, it is given to the client and the container does not keep track of it at all any longer. You can think of Spring.NET's role when talking about a non-singleton / prototype object as a replacement for the 'new' operator. Any lifecycle aspects past that point have to be handled by the client. The lifecycle of an object in the `IObjectFactory` is further described in Section 3.5.1, “Lifecycle interfaces”

In the XML fragment below, two objects are declared of which one is defined as a singleton, and the other one as a prototype. The `exampleObject` object is created each and every time a client asks the `IObjectFactory` for this object, whereas the `yetAnotherExample` object is only created once; a reference to the exact same instance is returned on each request for this object.

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary"
  singleton="false"/>
<object name="yetAnotherExample" type="Examples.ExampleObjectTwo, ExamplesLibrary"
  singleton="true"/>
```

The `lazy-init` attribute gives you control over when the singleton is instantiated. A common variation on the singleton design pattern is to lazily create the object, that is to create it only when it is first used. By default, the `lazy-init` attribute is set to false instructing the IoC container to pre-instantiate singletons object when the container is initialized. Setting the attribute to false will delay the creation until the object is first requested from the container, either directly from a user request or as a result of the resolving object dependencies.

3.3. Properties, collaborators, autowiring and dependency checking

3.3.1. Setting object properties and collaborators

Inversion of Control has already been referred to as *Dependency Injection*. The basic principle is that objects define their dependencies (i.e. the other objects they work with) only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually *inject* those dependencies when it creates the object. This is fundamentally the inverse (hence the name Inversion of Control) of the object instantiating or locating its dependencies on its own using direct construction of classes (via the `new` operator), or something like the *Service Locator* pattern. While we will not elaborate too much on the advantages of Dependency Injection, it becomes evident upon usage that code gets much cleaner and reaching a higher grade of decoupling is much easier when objects do not look up their dependencies, but are provided them, and additionally do not even know where the dependencies are located and of what actual type they are.

As touched on in the previous paragraph, Inversion of Control / Dependency Injection exists in two major variants:

- *setter-based* dependency injection is realized by calling property setters on your objects after invoking a no-argument constructor to instantiate your object. Spring.NET generally advocates the usage of setter-based dependency injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional.
- *constructor-based* dependency injection is realized by invoking a constructor with a number of arguments, each representing a collaborator or property. Although Spring.NET generally advocates the usage of setter-based dependency injection as much as possible, it does fully support the constructor-based approach as well, since you may wish to use it with pre-existing objects which provide only multi-argument constructors, and no setters. Additionally, for simpler objects, some people prefer the constructor approach as a means of ensuring objects can not be constructed in an invalid state.

The `IObjectFactory` supports both of these variants for injecting dependencies into objects it manages. The configuration for the dependencies comes in the form of the `IObjectDefinition` class, which is used together with `TypeConverters` to know how to convert properties from one format to another. However, most users of Spring.NET will not be dealing with these classes directly (i.e. programmatically), but rather with an XML definition file which will be converted internally into instances of these classes, and used to load an entire `IObjectFactory` or `IApplicationContext`.

Object dependency resolution generally happens as follows:

1. The `IObjectFactory` is created and initialized with a configuration which describes all the objects. Most Spring.NET users use an `IObjectFactory` or `IApplicationContext` variant that supports XML format configuration files.
2. Each object has dependencies expressed in the form of properties or constructor arguments. These will be provided to the object, *when the object is actually created*.
3. Each property or constructor-arg is either an actual definition of the value to set, or a reference to another object in the `IObjectFactory`. In the case of the `IApplicationContext`, the reference can be to an object in a parent `IApplicationContext`.
4. Each property or constructor argument which is a value must be able to be converted from whatever format it was specified in, to the actual `System.Type` of that property or constructor argument. By default Spring.NET can convert a value supplied in string format to all built-in types, such as `int`, `long`, `string`, `bool`, etc. Additionally, when talking about the XML based `IObjectFactory` variants (including the `IApplicationContext` variants), these have built-in support for defining `IList`, `IDictionary`, and `Set` collection types. Spring.NET uses `TypeConverter` definitions to be able to convert string values to other, arbitrary types. Refer to Section 4.3, “Type conversion” for more information regarding type conversion, and how you can design your classes to be convertible by Spring.NET.
5. It is important to realize that Spring.NET validates the configuration of each object in the `IObjectFactory` when the `IObjectFactory` is created, including the validation that properties that are object references are actually referring to valid objects (i.e. the objects being referred to are also defined in the `IObjectFactory`, or in a parent context in the case of `IApplicationContext`). However, the object properties themselves are not set until the object *is actually created*. For objects that have been defined as singletons and set to be pre-instantiated (such as singleton objects in an `IApplicationContext`), creation happens at the time that the `IObjectFactory` is created, but otherwise this is only when the object is requested. When an object actually has to be created, this will potentially cause a graph of other objects to be created, as its dependencies and its dependencies' dependencies (and so on) are created and assigned.
6. You can generally trust Spring.NET to do the right thing. It will pick up configuration issues, including references to non-existent object definitions and circular dependencies, at `IObjectFactory` load-time. It will actually set properties and resolve dependencies (i.e. create any dependent objects if needed) as late as possible, which is when the object is actually created. This does mean that an `IObjectFactory` which has loaded correctly can later generate an exception when you request an object, if there is a problem creating that object or one of its dependencies. This could happen if the object throws an exception as a result of a missing or invalid property, for example. This potentially delayed visibility of some configuration issues is why `IApplicationContext` by default pre-instantiates singleton objects. At the cost of some upfront time and memory to create these objects before they are actually needed, you find out about configuration issues when the `IApplicationContext` is created, not later. If you wish, you can still override this default behavior and set any of these singleton objects to lazy-load (not be preinstantiated).

Some examples (using XML based object definitions)...

First, an example of using the `IObjectFactory` for setter-based dependency injection. Below is a small part of an XML file specifying some object definitions. Following is the code for the actual main object itself, showing the appropriate setters declared.

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary">
    <property name="objectOne" ref="anotherExampleObject"/>
    <property name="objectTwo" ref="yetAnotherObject"/>
    <property name="IntegerProperty" value="1"/>
```

```
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>
```

```
[C#]
public class ExampleObject
{
  private AnotherObject objectOne;
  private YetAnotherObject objectTwo;
  private int i;

  public AnotherObject ObjectOne
  {
    set { this.objectOne = value; }
  }

  public YetAnotherObject ObjectTwo
  {
    set { this.objectTwo = value; }
  }

  public int IntegerProperty
  {
    set { this.i = value; }
  }
}
```

Now, an example of using the `IObjectFactory` for IoC Type 3 (Constructor Dependency Injection). Below is a snippet from an XML configuration that specifies constructor arguments and the actual object code, showing the constructor:

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <constructor-arg name="objectTwo" ref="yetAnotherObject"/>
    <constructor-arg name="IntegerProperty" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>
```

```
[Visual Basic.NET]
Public Class ExampleObject

    Private myObjectOne As AnotherObject
    Private myObjectTwo As YetAnotherObject
    Private i As Integer

    Public Sub New (
        anotherObject as AnotherObject,
        yetAnotherObject as YetAnotherObject,
        i as Integer)

        myObjectOne = anotherObject
        myObjectTwo = yetAnotherObject
        Me.i = i
    End Sub
End Class
```

As you can see, the constructor arguments specified in the object definition will be used to pass in as arguments to the constructor of the `ExampleObject`.

Note that Type 2 Setter Injection and Type 3 Constructor Injection IoC are not mutually exclusive... it is perfectly reasonable to use both for a single object definition, as can be seen in the following example:

```
<object id="exampleObject" type="Examples.MixedIocObject, ExamplesLibrary">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <property name="objectTwo" ref="yetAnotherObject"/>
    <property name="IntegerProperty" value="1"/>
```

```
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>
```

```
[C#]
public class MixedIocObject
{
  private AnotherObject objectOne;
  private YetAnotherObject objectTwo;
  private int i;

  public MixedIocObject (AnotherObject obj)
  {
    this.objectOne = obj;
  }

  public YetAnotherObject ObjectTwo
  {
    set { this.objectTwo = value; }
  }

  public int IntegerProperty
  {
    set { this.i = value; }
  }
}
```

Now consider a variant of this where instead of using a constructor, Spring is told to call a static factory method to return an instance of the object

```
<object id="exampleObject" type="Examples.ExampleFactoryMethodObject, ExamplesLibrary"
     factory-method="CreateInstance">
    <constructor-arg name="objectOne" ref="anotherExampleObject"/>
    <constructor-arg name="objectTwo" ref="yetAnotherObject"/>
    <constructor-arg name="intProp" value="1"/>
</object>

<object id="anotherExampleObject" type="Examples.AnotherObject, ExamplesLibrary"/>
<object id="yetAnotherObject" type="Examples.YetAnotherObject, ExamplesLibrary"/>
```

```
[C#]
public class ExampleFactoryMethodObject
{
  private AnotherObject objectOne;
  private YetAnotherObject objectTwo;
  private int i;

  // a private constructor
  private ExampleFactoryMethodObject()
  {
  }

  public static ExampleFactoryMethodObject CreateInstance(AnotherObject objectOne,
                              YetAnotherObject objectTwo,
                              int intProp)
  {
    ExampleFactoryMethodObject fmo = new ExampleFactoryMethodObject();
    fmo.AnotherObject = objectOne;
    fmo.YetAnotherObject = objectTwo;
    fmo.IntegerProperty = intProp;
    return fmo;
  }

  // Property definitions

}
```

Note that arguments to the static factory method are supplied via constructor-arg elements, exactly the same as if a constructor had actually been used. These arguments are optional. Also, it is important to realize that the type of the class being returned by the factory method does not have to be of the same type as the class which contains the static factory method, although in this example it is. An instance (non-static) factory method, mentioned previously, would be used in an essentially identical fashion (aside from the use of the factory-object attribute instead of the type attribute), so will not be detailed here.

3.3.2. Constructor Argument Resolution

Constructor argument resolution matching occurs using the argument's type. When another object is referenced, the type is known, and matching can occur. When a simple type is used, such as `<value>1</value>`, Spring.NET cannot determine the type of the value, and so cannot match by type without help. Consider the following class, which is used for the following two sections:

```
using System;

namespace SimpleApp
{
  public class ExampleObject
  {
    private int years;              //No. of years to the calculate the Ultimate Answer

    private string ultimateAnswer;  //The Answer to Life, the Universe, and Everything

    public ExampleObject(int years, string ultimateAnswer)
    {
       this.years = years;
       this.ultimateAnswer = ultimateAnswer;
    }

    public string UltimateAnswer
    {
      get { return this.ultimateAnswer; }
    }

    public int Years
    {
      get { return this.years; }
    }
  }
}
```

3.3.2.1. Constructor Argument Type Matching

The above scenario *can* use type matching with simple types by explicitly specifying the type of the constructor argument using the `type` attribute. For example:

```
<object name="exampleObject" type="SimpleApp.ExampleObject, SimpleApp">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="string" value="42"/>
</object>
```

The type attribute specifies the `System.Type` of the constructor argument, such as System.Int32. Alias' are available to for common simple types (and their array equivalents). These alias' are...

Table 3.2. Type aliases

Type	**Alias'**
System.Char	char, Char
System.Int16	short, Short
System.Int32	int, Integer

Type	**Alias'**
System.Int64	long, Long
System.UInt16	ushort
System.UInt32	uint
System.UInt64	ulong
System.Float	float, Single
System.Double	double, Double
System.Date	date, Date
System.Decimal	decimal, Decimal
System.Bool	bool, Boolean
System.String	string, String

3.3.2.2. Constructor Argument Index

Constructor arguments can have their index specified explicitly by use of the `index` attribute. For example:

```
<object name="exampleObject" type="SimpleApp.ExampleObject, SimpleApp">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</object>
```

As well as solving the ambiguity problem of multiple simple values, specifying an index also solves the problem of ambiguity where a constructor may have two arguments of the same type. Note that the *index is 0 based*.

3.3.2.3. Constructor Arguments by Name

Constructor arguments can also be specified by name by using the `name` attribute of the `<constructor-arg>` element.

```
<object name="exampleObject" type="SimpleApp.ExampleObject, SimpleApp">
  <constructor-arg name="years" value="7500000"/>
  <constructor-arg name="ultimateAnswer" value="42"/>
</object>
```

3.3.3. Object properties and constructor arguments in detail

As mentioned in the previous section, object properties and constructor arguments can be defined as either references to other managed objects (collaborators). The `XmlObjectFactory` (located in the `Spring.Objects.Factory.Xml` namespace) supports a number of sub-element types within its `property` and `constructor-arg` elements for this purpose.

The `value` element specifies a property or constructor argument as a human-readable string representation. As mentioned in detail previously, `TypeConverter` instances are used to convert these string values from a `System.String` to the actual property or argument type. Custom `TypeConverter` implementations in the `Spring.Objects.TypeConverters` namespace are used to augment the functionality offered by the .NET BCL's default `TypeConverter` implementations.

In the following example, we use a `SqlConnection` from the `System.Data.SqlClient` namespace of the BCL. This class (like many other existing classes) can easily be used in a Spring.NET object factory, as it offers a convenient public property for configuration of its `ConnectionString` property.

```
<objects>
  <object id="myConnection" type="System.Data.SqlClient.SqlConnection">
      <!-- results in a call to the setter of the ConnectionString property -->
      <property
          name="ConnectionString"
          value="Integrated Security=SSPI;database=northwind;server=mySQLServer"/>
  </object>
</objects>
```

3.3.3.1. Setting null values

The `<null>` element is used to handle `null` values. Spring.NET treats empty arguments for properties and constructor arguments as empty `string` instances. The following configuration demonstrates this behaviour...

```
<object type="Examples.ExampleObject, ExamplesLibrary">
    <property name="email"><value></value></property>
    <!-- equivalent, using value attribute as opposed to nested <value/> element...
    <property name="email" value=""/>
</object>
```

This results in the email property being set to the empty string value (`""`), in much the same way as can be seen in the following snippet of C# code...

```
exampleObject.Email = "";
```

The special `<null/>` element may be used to indicate a `null` value; to wit...

```
<object type="Examples.ExampleObject, ExamplesLibrary">
    <property name="email"><null/></property>
</object>
```

This results in the email property being set to `null`, again in much the same way as can be seen in the following snippet of C# code...

```
exampleObject.Email = null;
```

3.3.3.2. Setting collection values

The `list`, `set`, `name-values` and `dictionary` elements allow properties and arguments of the type `IList`, `ISet`, `NameValueCollection` and `IDictionary`, respectively, to be defined and set.

```
<objects>
  <object id="moreComplexObject" type="Example.ComplexObject">
      <!--
      results in a call to the setter of the SomeList (System.Collections.IList) property
      -->
      <property name="SomeList">
          <list>
              <value>a list element followed by a reference</value>
              <ref object="myConnection"/>
          </list>
      </property>
      <!--
      results in a call to the setter of the SomeDictionary (System.Collections.IDictionary) property
      -->
      <property name="SomeDictionary">
          <dictionary>
              <entry key="a string => string entry" value="just some string"/>
```

```
                <entry key-ref="myKeyObject" value-ref="myConnection"/>
            </dictionary>
        </property>
        <!--
        results in a call to the setter of the SomeNameValue (System.Collections.NameValueCollection) property
        -->
        <property name="SomeNameValue">
            <name-values>
                <add key="HarryPotter" value="The magic property"/>
                <add key="JerrySeinfeld" value="The funny (to Americans) property"/>
            </name-values>
        </property>
        <!--
        results in a call to the setter of the SomeSet (Spring.Collections.ISet) property
        -->
        <property name="someSet">
            <set>
                <value>just some string</value>
                <ref object="myConnection"/>
            </set>
        </property>
    </object>
</objects>
```

Many classes in the BCL expose only read-only properties for collection classes. When Spring.NET encounters a read-only collection, it will configure the collection by using the getter property to obtain a reference to the collection class and then proceed to add the additional elements to the existing collection. This results in an additive behavior for collection properties that are exposed in this manner.

Note that the value of a Dictionary entry, or a set value, can also again be any of the elements:

```
(object | ref | idref | list | set | dictionary | name-values | value | null)
```

The shortcut forms for value and references are useful to reduce XML verbosity when setting collection properties. See Section 3.3.3.8, "Value and ref shortcut forms" for more information.

Please be advised that the setting of multiple values for a `NameValueCollection` is planned for a future release.

3.3.3.3. Setting generic collection values

Spring supports setting values for classes that expose properties based on the generic collection interfaces `IList<T>` and `IDictionary<TKey, TValue>`. The type parameter for these collections is specified by using the XML attribute `element-type` for `IList<T>` and the XML attributes `key-type` and `value-type` for `IDictionary<TKey, TValue>`. The values of the collection are automaticaly converted from a string to the appropriate type. If you are using your own user-defined type as a generic type parameter you will likely need to register a custom type converter. Refer to Section 3.4, "Type conversion" for more information. The implementations of `IList<T>` and `IDictionary<TKey, TValue>` that is created are `System.Collections.Generic.List` and `System.Collections.Generic.Dictionary`.

The following class represents a lottery ticket and demonstrates how to set the values of a generic IList.

```
public class LotteryTicket
{
    List<int> list;
    DateTime date;
    public List<int> Numbers
    {
        set { list = value; }
        get { return list; }
    }

    public DateTime Date
    {
        get { return date; }
        set { date = value; }
    }
}
```

The XML fragment that can be used to configure this class is shown below

```
<object id="MyLotteryTicket" type="GenericsPlay.Lottery.LotteryTicket, GenericsPlay">
  <property name="Numbers">
    <list element-type="int">
      <value>11</value>
      <value>21</value>
      <value>23</value>
      <value>34</value>
      <value>36</value>
      <value>38</value>
    </list>
  </property>
  <property name="Date" value="4/16/2006"/>
</object>
```

The following shows the definition of a more complex class that demonstrates the use of generics using the `Spring.Expressions.IExpression` interface as the generic type parameter for the IList element-type and the value-type for IDictionary. `Spring.Expressions.IExpression` has an associated type converter, `Spring.Objects.TypeConverters.ExpressionConverter` that is already pre-registered with Spring.

```
public class GenericExpressionHolder
{
    private System.Collections.Generic.IList<IExpression> expressionsList;
    private System.Collections.Generic.IDictionary<string, IExpression> expressionsDictionary;

    public System.Collections.Generic.IList<IExpression> ExpressionsList
    {
        set { this.expressionsList = value; }
    }

    public System.Collections.Generic.IDictionary<string,IExpression> ExpressionsDictionary
    {
        set { this.expressionsDictionary = value; }
    }

    public IExpression this[int index]
    {
        get { return this.expressionsList[index]; }
    }

    public IExpression this[string key]
    {
        get { return this.expressionsDictionary[key]; }
    }
}
```

An example XML configuration of this class is shown below

```

<object id="genericExpressionHolder" type="Spring.Objects.Factory.Xml.GenericExpressionHolder, Spring.Core.Tests
  <property name="ExpressionsList">
    <list element-type="Spring.Expressions.IExpression, Spring.Core">
      <value>1 + 1</value>
      <value>date('1856-7-9').Month</value>
      <value>'Nikola Tesla'.ToUpper()</value>
      <value>DateTime.Today > date('1856-7-9')</value>
    </list>
  </property>
  <property name="ExpressionsDictionary">
    <dictionary key-type="string" value-type="Spring.Expressions.IExpression, Spring.Core">
      <entry key="zero">
        <value>1 + 1</value>
      </entry>
      <entry key="one">
        <value>date('1856-7-9').Month</value>
      </entry>
      <entry key="two">
        <value>'Nikola Tesla'.ToUpper()</value>
      </entry>
      <entry key="three">
        <value>DateTime.Today > date('1856-7-9')</value>
      </entry>
    </dictionary>
  </property>
</object>
```

3.3.3.4. Setting indexer properties

An indexer lets you set and get values from a collection using a familiar bracket `[]` notation. In the compiled class a property named `Item[type]` is created, where the type represent the indexer parameter type, typically a string or an int. If you use Reflector you can see this for yourself. Given this naming convention, Spring lets you configure indexer properties using the following syntax

```
<object id="objectWithIndexer" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="Item[0]" value="my string value"/>
</object>
```

would correspond to setting an indexer in a class with the followign declaration

```
public class TestObject
{
  public string this[int index]
  {
    ...
  }

  ...
}
```

You can also change the name used to identify the indexer by adorning your indexer method declaration with the attribute `[IndexerName("MyItemName")]`. You would then use the string `MyItemName[0]` to configure the first element of that indexer.

There are some limitations to be aware in the current indexer configuration. The indexer can only be of a single parameter that is convertable from a string to the indexer parameter type. Also, multiple indexers are not supported. You can get around that last limitation currently if you use the IndexerName attribute, but future versions will accomodate this case so you don't have to make any changes to your class (if even possible!). The Spring container is intended to be as non-invasive as possible.

3.3.3.5. Inline objects

An `object` element inside the `property` element is used to define an object value inline, instead of referring to an object defined elsewhere in the container. The inline object definition does not need to have any id or name defined (indeed, if any are defined, they will be ignored).

```
<object id="outer" type="...">
    <!-- Instead of using a reference to target, just use an inner object -->
    <property name="target">
        <object type="ExampleApp.Person, ExampleApp">
            <property name="name" value="Tony"/>
            <property name="age" value="51"/>
        </object>
   </property>
</object>
```

3.3.3.6. The idref element

An idref element is simply a shorthand and error-proof way to set a property to the String *id* or *name* of another object in the container.

```
<object id="theTargetObject" type="...">
</object>
<object id="theClientObject" type="...">
    <property name="targetName">
        <idref object="theTargetObject"/>
    </property>
</object>
```

This is exactly equivalent at runtime to the following fragment:

```
<object id="theTargetObject" type="...">
</object>
<object id="theClientObject" type="...">
    <property name="targetName" value="theTargetObject"/>
</object>
```

The main reason the first form is preferable to the second is that using the `idref` tag will allow Spring.NET to validate at deployment time that the other object actually exists. In the second variation, the class that is having its *targetName* property injected is forced to do its own validation, and that will only happen when that class is actually instantiated by the container, possibly long after the container is actually up and running.

Additionally, if the object being referred to is in the same actual XML file, and the object name is the object *id*, the `local` attribute may be used, which will allow the XML parser itself to validate the object name even earlier, at parse time.

```
<property name="targetName">
    <idref local="theTargetObject"/>
</property>
```

3.3.3.7. Referring to collaborating objects

The `ref` element is the final element allowed inside a `property` definition element. It is used to set the value of the specified property to be a reference to another object managed by the container, a collaborator, so to speak. As you saw in the previous example to set collection properties, we used the `SqlConnection` instance from the initial example as a collaborator and specified it using a <ref object/> element. As mentioned in a previous section, the referred-to object is considered to be a dependency of the object who's property is being set, and will be initialized on demand as needed (if it is a singleton object it may have already been initialized by the container) before the property is set. All references are ultimately just a reference to another object, but there are 3 variations on how the id/name of the other object may be specified, which determines how scoping and validation is handled.

Specifying the target object by using the `object` attribute of the `ref` tag is the most general form, and will allow creating a reference to any object in the same `IObjectFactory` / `IApplicationContext` (whether or not in the same XML file), or parent `IObjectFactory` / `IApplicationContext`. The value of the `object` attribute may be the same as either the `id` attribute of the target object, or one of the values in the `name` attribute of the target object.

```
<ref object="someObject"/>
```

Specifying the target object by using the `local` attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the `local` attribute must be the same as the `id` attribute of the target object. The XML parser will issue an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors are early as possible) if the target object is in the same XML file.

```
<ref local="someObject"/>
```

Specifying the target object by using the `parent` attribute allows a reference to be created to an object that is in a parent `IObjectFactory` (or`IApplicationContext`) of the current `IObjectFactory` (or `IApplicationContext`). The value of the `parent` attribute may be the same as either the `id` attribute of the target object, or one of the values in the `name` attribute of the target object, and the target object **must** be in a parent `IObjectFactory` or `IApplicationContext` of the current one. The main use of this object reference variant is when there is a need to wrap an existing object in a parent context with some sort of proxy (which may have the same name as the parent), and needs the original object so it may wrap it.

```
<ref parent="someObject"/>
```

3.3.3.8. Value and ref shortcut forms

There are also some shortcut forms that are less verbose than using the full `value` and `ref` elements. The `property`, `constructor-arg`, and `entry` elements all support a `value` attribute which may be used instead of embedding a full `value` element. Therefore, the following:

```
<property name="myProperty">
    <value>hello</value>
</property>

<constructor-arg>
    <value>hello</value>
</constructor-arg>

<entry key="myKey">
    <value>hello</value>
</entry>
```

are equivalent to:

```
<property name="myProperty" value="hello"/>

<constructor-arg value="hello"/>

<entry key="myKey" value="hello"/>
```

In general, when typing definitions by hand, you will probably prefer to use the less verbose shortcut form.

The `property` and `constructor-arg` elements support a similar shortcut `ref` attribute which may be used instead of a full nested `ref` element. Therefore, the following...

```
<property name="myProperty">
    <ref object="anotherObject"/>
</property>

<constructor-arg index="0">
    <ref object="anotherObject"/>
</constructor-arg>
```

is equivalent to...

```
<property name="myProperty" ref="anotherObject"/>

<constructor-arg index="0" ref="anotherObject"/>
```

Note

The shortcut form is equivalent to a `<ref object="xxx">` element; there is no shortcut for either the `<ref local="xxx">` or `<ref parent="xxx">` elements. For a local or parent `ref`, you must still use the long form.

Finally, the entry element allows a shortcut form the specify the key and/or value of a dictionary, in the form of key/key-ref and value/value-ref attributes. Therefore, the following

```
<entry>
  <key><ref object="MyKeyObject"/></key>
  <ref object="MyValueObject"/>
</entry>
```

Is equivalent to:

```
<entry key-ref="MyKeyObject" value-ref="MyValueObject"/>
```

As mentioned previously, the equivalence is to `<ref object="xxx">` and not the local or parent forms of object references.

3.3.3.9. Compound property names

Note that compound or nested property names are perfectly legal when setting object properties, as long as all components of the path except the final property name are non-null. For example, in this object definition:

```
<object id="foo" type="Spring.Foo, Spring.Foo">
    <property name="bar.baz.name" value="Bingo"/>
</object>
```

3.3.4. Method Injection

For most users, the majority of the objects in the container will be singletons. When a singleton object needs to collaborate with (use) another singleton object, or a non-singleton object needs to collaborate with another non-singleton object, the typical and common approach of handling this dependency (by defining one object to be a property of the other) is quite adequate. There is however a problem when the object lifecycles are different. Consider a singleton object A which needs to use a non-singleton (prototype) object B, perhaps on each method invocation on A. The container will only create the singleton object A once, and thus only get the opportunity to set its properties once. There is no opportunity for the container to provide object A with a new instance of object B every time one is needed.

One solution to this problem is to forego some inversion of control. Object A can be aware of the container by implementing the `IObjectFactoryAware` interface, and use programmatic means to ask the container via a `GetObject("B")` call for (a new) object B every time it needs it. This is generally not a desirable solution since the object code is then aware of and coupled to Spring.NET.

Method Injection, an advanced feature of supporting `IObjectFactoryAware` implementations, allows this use case to be handled in a clean fashion, along with some other scenarios.

3.3.4.1. Lookup Method Injection

Lookup method injection refers to the ability of the container to override `abstract` or concrete methods on managed objects in the container, and to return the result of looking up another named object in the container. The lookup will typically be of a non-singleton object as per the scenario described above (although it can also be a singleton). Spring.NET implements this through a dynamically generated subclass overriding the method using the classes in the `System.Reflection.Emit` namespace.

In the client class containing the method to be injected, the method definition must observe the following form:

```
protected abstract SingleShotHelper CreateSingleShotHelper();
```

If the method is not `abstract`, Spring.NET will simply override the existing implementation. In the `XmlObjectFactory` case, you instruct Spring.NET to inject / override this method to return a particular object from the container, by using the `lookup-method` element inside the object definition. For example:

```
<!-- a stateful object deployed as a prototype (non-singleton) -->
<object id="singleShotHelper" class="..." singleton="false"/>

<!-- myobject uses singleShotHelper -->
<object id="myObject" type="...">
  <lookup-method name="CreateSingleShotHelper" object="singleShotHelper"/>
  <property>
    ...
  </property>
</object>
```

The object identified as `myObject` will call its own method `CreateSingleShotHelper` whenever it needs a new instance of the `singleShotHelper` object. It is important to note that the person deploying the objects must be careful to deploy the `singleShotHelper` object as a non-singleton (if that is actually what is needed). If it is deployed as a singleton (either explicitly, or relying on the default `true` setting for this flag), the same instance of `singleShotHelper` will be returned each time!

Note that lookup method injection can be combined with Constructor Injection (supplying optional constructor arguments to the object being constructed), and also with Setter Injection (settings properties on the object being constructed).

3.3.4.2. Arbitrary method replacement

A less commonly useful form of method injection than Lookup Method Injection is the ability to replace arbitrary methods in a managed object with another method implementation. Users may safely skip the rest of this section (which describes this somewhat advanced feature), until this functionality is actually needed.

In an `XmlObjectFactory`, the `replaced-method` element may be used to replace an existing method implementation with another. Consider the following class, with a method `ComputeValue`, which we want to override:

```
public class MyValueCalculator {

  public virtual string ComputeValue(string input) {
    // ... some real code
  }

  // ... some other methods
}
```

A class implementing the `Spring.Objects.Factory.Support.IMethodReplacer` interface is needed to provide the new (injected) method definition.

```
/// <summary>
/// Meant to be used to override the existing ComputeValue(string)
/// implementation in MyValueCalculator.
/// </summary>
public class ReplacementComputeValue : IMethodReplacer
{
        public object Implement(object target, MethodInfo method, object[] arguments)
        {
                // get the input value, work with it, and return a computed result...
                string value = (string) arguments[0];
                // compute...
                return result;
        }
}
```

The attendant `IObjectFactory` definition to deploy the original class and specify the method override would look like:

```
<object id="myValueCalculator" type="Examples.MyValueCalculator, ExampleAssembly">
  <!-- arbitrary method replacement -->
  <replaced-method name="ComputeValue" replacer="replacementComputeValue">
    <arg-type match="String"/>
  </replaced-method>
</object>

<object id="replacementComputeValue" type="Examples.ReplaceMentComputeValue, ExampleAssembly"/>
```

One or more contained `arg-type` elements within the `replaced-method` element may be used to indicate the method signature of the method being overridden. Note that the signature for the arguments is actually only needed in the case that the method is actually overloaded and there are multiple variants within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, all the following would match `System.String`.

```
    System.String
    String
    Str
```

Since the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by just using the shortest string which will match an argument.

3.3.5. Setting a reference using the members of other objects and classes.

This section details those configuration scenarios that involve the setting of properties and constructor arguments using the members of other objects and classes. This kind of scenario is quite common, especially when dealing with legacy classes that you cannot (or won't) change to accommodate some of Spring.NET's conventions... consider the case of a class that has a contructor argument that can only be calculated by going to say, a database. The `MethodInvokingFactoryObject` handles exactly this scenario ... it will allow you to inject the result of an arbitrary method invocation into a constructor (as an argument) or as the value of a property setter. Similary, `PropertyRetrievingFactoryObject` and `FieldRetrievingFactoryObject` allow you to retrieve values from another objects property or field value. These classes implement the `IFactoryObject` interface which indicates to Spring.NET that this object is itself a factory and the factories product, not the factory itself, is what will be associated with the object id. Factory objects are discussed futher in Section 3.5.3, “IFactoryObject”

3.3.5.1. Setting a reference to the value of property.

The `PropertyRetrievingFactoryObject` is an `IFactoryObject` that addresses the scenario of setting one of the properties and / or constructor arguments of an object to the value of a property exposed on another object or class. One can use it to get the value of any **public** property exposed on either an instance or a class (in the case of a property exposed on a class, the property must obviously be static).

In the case of a property exposed on an instance, the target object that a `PropertyRetrievingFactoryObject` will evaluate can be either an object instance specified directly inline or a reference to another arbitrary object. In the case of a static property exposed on a class, the target object will be the class (the .NET `System.Type`) exposing the property.

The result of evaluating the property lookup may then be used in another object definition as a property value or constructor argument. Note that nested properties are supported for both instance and class property lookups. The `IFactoryObject` is discussed more generally in Section 3.5.3, “IFactoryObject”.

Here's an example where a property path is used against another object instance. In this case, an inner object definition is used and the property path is nested, i.e. spouse.age.

```
<object name="person" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="age" value="20"/>
  <property name="spouse">
    <object type="Spring.Objects.TestObject, Spring.Core.Tests">
      <property name="age" value="21"/>
    </object>
  </property>
</object>

// will result in 21, which is the value of property 'spouse.age' of object 'person'
<object name="theAge" type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="person"/>
  <property name="TargetProperty" value="spouse.age"/>
</object>
```

An example of using a `PropertyRetrievingFactoryObject` to evaluate a `static` property is shown below.

```
<object id="cultureAware"
        type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
  <property name="culture" ref="cultureFactory"/>
</object>

<object id="cultureFactory"
        type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject, Spring.Core">
  <property name="StaticProperty">
      <value>System.Globalization.CultureInfo.CurrentUICulture, Mscorlib</value>
  </property>
</object>
```

Similarly, an example showing the use of an instance property is shown below.

```
<object id="instancePropertyCultureAware"
        type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
  <property name="Culture" ref="instancePropertyCultureFactory"/>
</object>

<object id="instancePropertyCultureFactory"
        type="Spring.Objects.Factory.Config.PropertyRetrievingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="instancePropertyCultureAwareSource"/>
  <property name="TargetProperty" value="MyDefaultCulture"/>
</object>

<object id="instancePropertyCultureAwareSource"
        type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests"/>
```

3.3.5.2. Setting a reference to the value of field.

The `FieldRetrievingFactoryObject` class addresses much the same area of concern as the `PropertyRetrievingFactoryObject` described in the previous section. However, as its name might suggest, the `FieldRetrievingFactoryObject` class is concerned with looking up the value of a **public** field exposed on either an instance or a class (and similarly, in the case of a field exposed on a class, the field must obviously be static).

The following example demonstrates using a `FieldRetrievingFactoryObject` to look up the value of a (public, static) field exposed on a class

```
<object id="withTypesField"
        type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
  <property name="Types" ref="emptyTypesFactory"/>
```

```
</object>

<object id="emptyTypesFactory"
        type="Spring.Objects.Factory.Config.FieldRetrievingFactoryObject, Spring.Core">
  <property name="TargetType" value="System.Type, Mscorlib"/>
    <property name="TargetField" value="EmPTytypeS"/>
</object>
```

The example in the next section demonstrates the look up of a (public) field exposed on an object instance.

```
<object id="instanceCultureAware"
        type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests">
  <property name="Culture" ref="instanceCultureFactory"/>
</object>

<object id="instanceCultureFactory"
  type="Spring.Objects.Factory.Config.FieldRetrievingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="instanceCultureAwareSource"/>
  <property name="TargetField" value="Default"/>
</object>

<object id="instanceCultureAwareSource"
         type="Spring.Objects.Factory.Xml.XmlObjectFactoryTests+MyTestObject, Spring.Core.Tests"/>
```

3.3.5.3. Setting a property or constructor argument to the return value of a method invocation.

The `MethodInvokingFactoryObject` rounds out the trio of classes that permit the setting of properties and constructor arguments using the members of other objects and classes. Whereas the `PropertyRetrievingFactoryObject` and `FieldRetrievingFactoryObject` classes dealt with simply looking looking up and returning the value of property or field on an object or class, the `MethodInvokingFactoryObject` allows one to set a constructor or property to the return value of an arbitrary method invocation,

The `MethodInvokingFactoryObject` class handles both the case of invoking an (instance) method on another object in the container, and the case of a static method call on an arbitrary class. Additonally, it is sometimes neccessary to invoke a method just to perform some sort of initialization.... while the mechanisms for handling object initialization have yet to be introduced (see Section 3.5.1.1, “IInitializingObject / init-method ”), these mechanisms do not permit any arguments to be passed to any initialization method, and are confined to invoking an initialization method on the object that has just been instantiated by the container. The `MethodInvokingFactoryObject` allows one to invoke pretty much **any** method on any object (or class in the case of a static method).

The following example (in an XML based `IObjectFactory` definition) uses the `MethodInvokingFactoryObject` class to force a call to a static factory method prior to the instantiation of the object...

```
<object id="force-init"
  type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="StaticMethod">
    <value>ExampleNamespace.ExampleInitializerClass.Initialize</value>
  </property>
</object>
<object id="myService" depends-on="force-init"/>
```

Note that the definition for the `myService` object has used the `depends-on` attribute to refer to the `force-init` object, which will force the initialization of the `force-init` object first (and thus the calling of its configured `StaticMethod` static initializer method, when `myService` is first initialized. Please note that in order to effect this initialization, the `MethodInvokingFactoryObject` object **must** be operating in `singleton` mode (the default.. see the next paragraph).

Note that since this class is expected to be used primarily for accessing factory methods, this factory defaults to operating in `singleton` mode. As such, as soon as all of the properties for a `MethodInvokingFactoryObject` object have been set, and if the `MethodInvokingFactoryObject` object is still in `singleton` mode, the method will be invoked immediately and the return value cached for later access. The first request by the container for the factory to produce an object will cause the factory to return the cached return value for the current request (and all subsequent requests). The `IsSingleton` property may be set to false, to cause this factory to invoke the target method each time it is asked for an object (in this case there is obviously no caching of the return value).

A static target method may be specified by setting the `targetMethod` property to a string representing the static method name, with `TargetType` specifying the `Type` that the static method is defined on. Alternatively, a target instance method may be specified, by setting the `TargetObject` property to the name of another Spring.NET managed object definition (the target object), and the `TargetMethod` property to the name of the method to call on that target object.

Arguments for the method invocation may be specified in two ways (or even a mixture of both)... the first involves setting the `Arguments` property to the list of arguments for the method that is to be invoked. Note that the ordering of these arguments is significant... the order of the values passed to the `Arguments` property must be the same as the order of the arguments defined on the method signature, including the argument`Type`. This is shown in the example below

```
<object id="myObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetType" value="Whatever.MyClassFactory, MyAssembly"/>
  <property name="TargetMethod" value="GetInstance"/>

  <!-- the ordering of arguments is significant -->
  <property name="Arguments">
    <list>
      <value>1st</value>
      <value>2nd</value>
      <value>and 3rd arguments</value>
      <!-- automatic Type-conversion will be performed prior to invoking the method -->
    </list>
  </property>
</object>
```

The second way involves passing an arguments dictionary to the `NamedArguments` property... this dictionary maps argument names (`Strings`) to argument values (any object). The argument names are not case-sensitive, and order is (obviously) not significant (since dictionaries by definition do not have an order). This is shown in the example below

```
<object id="myObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetObject">
    <object type="Whatever.MyClassFactory, MyAssembly"/>
  </property>
  <property name="TargetMethod" value="Execute"/>

  <!-- the ordering of named arguments is not significant -->
  <property name="NamedArguments">
    <dictionary>
      <entry key="argumentName"><value>1st</value></entry>
      <entry key="finalArgumentName"><value>and 3rd arguments</value></entry>
      <entry key="anotherArgumentName"><value>2nd</value></entry>
    </dictionary>
  </property>
</object>
```

The following example shows how use `MethodInvokingFactoryObject` to call an instance method.

```
<object id="myMethodObject" type="Whatever.MyClassFactory, MyAssembly" />
```

```
<object id="myObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="myMethodObject"/>
  <property name="TargetMethod" value="Execute"/>
</object>
```

The above example could also have been written using an anonymous inner object definition... if the object on which the method is to be invoked is not going to be used outside of the factory object definition, then this is the preferred idiom because it limits the scope of the object on which the method is to be invoked to the surrounding factory object.

Finally, if you want to use `MethodInvokingFactoryObject` in conjunction with a method that has a variable length argument list, then please note that the variable arguments need to be passed (and configured) as a *list*. Let us consider the following method definition that uses the `params` keyword (in C#), and its attendant (XML) configuration...

```
[C#]
public class MyClassFactory
{
    public object CreateObject(Type objectType, params string[] arguments)
    {
        return ... // implementation elided for clarity...
    }
}
```

```
<object id="myMethodObject" type="Whatever.MyClassFactory, MyAssembly" />

<object id="paramsMethodObject" type="Spring.Objects.Factory.Config.MethodInvokingFactoryObject, Spring.Core">
  <property name="TargetObject" ref="myMethodObject"/>
  <property name="TargetMethod" value="CreateObject"/>
  <property name="Arguments">
    <list>
      <value>System.String</value>
      <!-- here is the 'params string[] arguments' -->
      <list>
        <value>1st</value>
        <value>2nd</value>
      </list>
    </list>
</object>
```

3.3.6. Additional IFactoryObject implementations

In addition to `PropertyRetrievingFactoryObject`, `MethodInvokingFactoryObject`, and `FieldRetrievingFactoryObject` Spring.NET comes with other useful implementations of the `IFactoryObject` interface. These are discussed below.

3.3.6.1. Log4Net

The `Log4NetFactoryObject` is useful when you would like to share a logging instance across a number of classes instead of creating a logging instance per class or class hierarchy. In the example shown below the same logging instance, with a log4Net name of "DAOLogger", is used in both the SimpleAccountDao and SimpleProducDao data access objects.

```
<objects xmlns="http://www.springframework.net"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.springframework.net
         http://www.springframework.net/xsd/spring-objects.xsd" >

    <object name="daoLogger" type="Spring.Objects.Factory.Config.Log4NetFactoryObject, Spring.Core">
      <property name="logName" value="DAOLogger"/>
    </object>

    <object name="productDao" type="PropPlayApp.SimpleProductDao, PropPlayApp ">
```

```
      <property name="maxResults" value="100"/>
      <property name="dbConnection" ref="myConnection"/>
      <property name="log" ref="daoLogger"/>
    </object>

    <object name="accountDao" type="PropPlayApp.SimpleAccountDao, PropPlayApp ">
      <property name="maxResults" value="100"/>
      <property name="dbConnection" ref="myConnection"/>
      <property name="log" ref="daoLogger"/>
    </object>

    <object name="myConnection" type="System.Data.Odbc.OdbcConnection, System.Data">
      <property name="connectionstring" value="dsn=MyDSN;uid=sa;pwd=myPassword;"/>
    </object>

</objects>
```

3.3.7. Using `depends-on`

Spring.NET uses the `<ref/>` element to express references to other dependant objects. Unless you have some special initialization requirements, you don't have to use the `<depends-on/>` element. However, when you are using statics that need initialization or some object needs to be initialized because of something else needing preparation, you can use the `<depends-on/>` element. This will ensure all objects you've listed as dependencies will get initialized before they're actually set on the object. For example...

```
<object id="objectOne" type="Examples.ExampleObject, ExamplesLibrary" depends-on="manager">
  <property name="manager" ref="manager"/>
</object>

<object id="manager" type="ManagerObject"/>
```

3.3.8. Autowiring collaborators

Spring.NET has autowire capabilities, meaning that it is possible to automatically let Spring.NET resolve collaborators (other objects) for your object by inspecting the object definitions in the container. Autowiring is specified *per* object and can thus be enabled for some objects, while other objects won't be autowired. Using autowiring, it is possible to reduce or even wholly eliminate the need to specify properties or constructor arguments. [2] The autowiring functionality has five modes...

Table 3.3. Autowiring modes

Mode	Explanation
no	No autowiring at all. This is the default value and you are encouraged not to change this for large applications, since specifying your collaborators explicitly gives you a feeling for what you're actually doing (always a bonus) and is a great way of somewhat documenting the structure of your system.
byName	This option will inspect the objects within the container, and look for an object named exactly the same as the property which needs to be autowired. For example, if you have an object definition that is set to autowire by name, and it contains a `Master` property, Spring.NET will look for an object definition named `Master`, and use it as the value of the `Master` property on your object definition.
byType	This option gives you the ability to resolve collaborators by type instead of by name. Supposing you have an `IObjectDefinition` with a collaborator typed `SqlConnection`, Spring.NET will search the entire object factory for an object definition of type

[2]See Section 3.3.1, “Setting object properties and collaborators”

Mode	Explanation
	`SqlConnection` and use it as the collaborator. *If 0 (zero) or more than 1 (one) object definitions of the desired type exist in the container, a failure will be reported and you won't be able to use autowiring for that specific object.*
constructor	This is analogous to *byType*, but applies to constructor arguments. If there isn't exactly one object of the constructor argument type in the object factory, a fatal error is raised.
autodetect	Chooses *constructor* or *byType* through introspection of the object class. If a default constructor is found, *byType* gets applied.

Note

Explicit dependencies always override autowiring. Autowire behavior can be combined with dependency checking, which will be performed after all autowiring has been completed.

3.3.9. Checking for dependencies

Spring.NET has the ability to try to check for the existence of unresolved dependencies of an object deployed into the container. These are properties of the object, which do not have actual values set for them in the object definition, or alternately provided automatically by the autowiring feature.

This feature is sometimes useful when you want to ensure that all properties (or all properties of a certain type) are set on an object. Of course, in many cases an object class will have default values for many properties, or some properties do not apply to all usage scenarios, so this feature is of limited use. Dependency checking can also be enabled and disabled per object, just as with the autowiring functionality. The default dependency checking mode is to *not* check dependencies. Dependency checking can be handled in several different modes. In XML-based configuration, this is specified via the `dependency-check` attribute in an object definition, which may have the following values.

Table 3.4. Dependency checking modes

Mode	Explanation
none	No dependency checking. Properties of the object which have no value specified for them are simply not set.
simple	Dependency checking is done for primitive types and collections (this means everything except collaborators).
object	Dependency checking is done for collaborators only.
all	Dependency checking is done for collaborators, primitive types and collections.

3.4. Type conversion

Type converters are responsible for converting objects from one type to another. When using the XML based file to configure the IoC container, string based property values are converted to the target property type. Spring will rely on the standard .NET support for type conversion unless an alternative `TypeConverter` is registered for a given type. How to register custom TypeConverters will be described shortly. As a reminder, the standard .NET type converter support works by associating a `TypeConverter` attribute with the class definition by passing the type of the converter as an attribute argument. [3] For example, an abbreviated class definition for the BCL type `Font` is shown below.

```
[Serializable, TypeConverter(typeof(FontConverter)), ...]
public sealed class Font : MarshalByRefObject, ICloneable, ISerializable, IDisposable
{
  // Methods

  ... etc ..
}
```

3.4.1. Type Conversion for Enumerations

The default type converter for enumerations is the `System.ComponentModel.EnumConverter` class. To specify the value for an enumerated property, simply use the name of the property. For example the `TestObject` class has a property of the enumerated type `FileMode`. One of the values for this enumeration is named `Create`. The following XML fragment shows how to configure this property

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="name" value="Rod"/>
  <property name="FileMode" value="Create"/>
</object>
```

3.4.2. Built-in TypeConverters

Spring.NET pre-registers a number of custom `TypeConverter` instances (for example, to convert a type expressed as a string into a real `System.Type` object). Each of those is listed below and they are all located in the `Spring.Objects.TypeConverters` namespace of the `Spring.Core` library.

Table 3.5. Built-in `TypeConverters`

Type	Explanation
`RuntimeTypeConverter`	Parses strings representing `System.Types` to actual `System.Types` and the other way around.
`FileInfoConverter`	Capable of resolving strings to a `System.IO.FileInfo` object.
`StringArrayConverter`	Capable of resolving a comma-delimited list of strings to a string-array and vice versa.
`UriConverter`	Capable of resolving a string representation of a URI to an actual `URI`-object.
`FileInfoConverter`	Capable of resolving a string representation of a FileInfo to an actual `FileInfo`-object.
`StreamConverter`	Capable of resolving Spring IResource URI (string) to its corresponding `InputStream`-object.
`ResourceConverter`	Capable of resolving Spring IResource URI (string) to an `IResource` object.
`ResourceManagerConverter`	Capable of resolving a two part string (resource name, assembly name) to a `System.Resources.ResourceManager` object.

[3] More information about creating custom `TypeConverter` implementations can be found online at Microsoft's MSDN website, by searching for *Implementing a Type Converter*.

Type	Explanation
`RGBColorConverter`	Capable of resolving a comma separated list of Red, Green, Blue integer values to a `System.Drawing.Color` structure.
`ExpressionConverter`	Capable of resolving a string into an instance of an object that implements the `IExpression` interface.

Spring.NET uses the standard .NET mechanisms for the resolution of `System.Types`, including, but not limited to checking any configuration files associated with your application, checking the Global Assembly Cache (GAC), and assembly probing.

3.4.3. Custom Type Conversion

There are a few ways to register custom type converters. The fundamental storage area in Spring for custom type converters is the `TypeConverterRegistry` class. The *most convenient way* if using an XML based implementation of `IObjectFactory` or `IApplicationContext` is to use the the custom configuration section handler `TypeConverterSectionHandler` This is demonstrated in section Section 3.13, “Configuration of IApplicationContext”

An alternate approach, present for legacy reasons in the port of Spring.NET from the Java code base, is to use the object factory post-processor `Spring.Objects.Factory.Config.CustomConverterConfigurer`. This is described in the next section.

If you are constructing your IoC container programmatically then you should use the `RegisterCustomConverter(Type requiredType, TypeConverter converter)` method of the `ConfigurableObjectFactory` interface.

3.4.3.1. Using CustomConverterConfigurer

This section shows in detail how to define a custom type converter that does not use the .NET `TypeConverter` attribute. The type converter class is standalone and inherits from the `TypeConverter` class. It uses the legacy factory post-processor approach.

Consider a user class *ExoticType*, and another class *DependsOnExoticType* which needs ExoticType set as a property:

```
public class ExoticType
{
    private string name;

    public ExoticType(string name)
    {
        this.name = name;
    }

    public string Name
    {
        get { return this.name; }
    }
}
```

and

```
public class DependsOnExoticType
{
    public DependsOnExoticType() {}

    private ExoticType exoticType;
    public ExoticType ExoticType
    {
        get { return this.exoticType; }
        set { this.exoticType = value; }
    }

    public override string ToString()
    {
        return exoticType.Name;
    }
}
```

When things are properly set up, we want to be able to assign the type property as a string, which a TypeConverter will convert into a real ExoticType object behind the scenes:

```
<object name="sample" type="SimpleApp.DependsOnExoticType, SimpleApp">
  <property name="exoticType" value="aNameForExoticType"/>
</object>
```

The TypeConverter looks like this:

```
public class ExoticTypeConverter : TypeConverter
{
        public ExoticTypeConverter()
        {
        }

        public override bool CanConvertFrom (
                ITypeDescriptorContext context,
                Type sourceType)
        {
                if (sourceType == typeof (string))
                {
                        return true;
                }
                return base.CanConvertFrom (context, sourceType);
        }

        public override object ConvertFrom (
                ITypeDescriptorContext context,
                CultureInfo culture, object value)
        {
                if (value is string)
                {
                        string s = value as string;
                        return new ExoticType(s.ToUpper());
                }
                return base.ConvertFrom (context, culture, value);
        }
}
```

Finally, we use the `CustomConverterConfigurer` to register the new `TypeConverter` with the `IApplicationContext`, which will then be able to use it as needed:

```
<object id="customConverterConfigurer"
        type="Spring.Objects.Factory.Config.CustomConverterConfigurer, Spring.Core">
  <property name="CustomConverters">
    <dictionary>
      <entry key="SimpleApp.ExoticType">
        <object type="SimpleApp.ExoticTypeConverter"/>
      </entry>
    </dictionary>
  </property>
</object>
```

3.5. Customizing the nature of an object

3.5.1. Lifecycle interfaces

Spring.NET uses several marker interfaces to change the behaviour of your object in the container, namely the Spring.NET specific `IInitializingObject` interface and the standard `System.IDisposable` interfaces. Implementing either of the aforementioned interfaces will result in the container calling the `AfterPropertiesSet()` method for the former and the `Dispose()` method for the latter, thus allowing you to do things upon the initialization and destruction of your objects.

Internally, Spring.NET uses implementations of the `IObjectPostProcessor` interface to process any marker interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring.NET doesn't offer out-of-the-box, you can implement an `IObjectPostProcessor` yourself. More information about this can be found in Section 3.8, “Customizing objects with IObjectPostProcessors”.

All the different lifecycle marker interfaces are described below.

3.5.1.1. IInitializingObject / `init-method`

The `Spring.Objects.Factory.IInitializingObject` interface gives you the ability to perform initialization work after all the necessary properties on an object are set by the container. The `IInitializingObject` interface specifies exactly one method:

- `void AfterPropertiesSet()`: called after all properties have been set by the container. This method enables you to do checking to see if all necessary properties have been set correctly, or to perform further initialization work. You can throw *any* `Exception` to indicate misconfiguration, initialization failures, etc.

Note

Generally, the use of the `IInitializingObject` can be avoided. The `Spring.Core` library provides support for a generic init-method, given to the object definition in the object configuration store (be it XML, or a database, etc).

```
<object id="exampleInitObject" type="Examples.ExampleObject" init-method="init"/>
[C#]
public class ExampleObject
{
    public void Init()
    {
        // do some initialization work
    }
}
```

Is exactly the same as...

```
<object id="exampleInitObject" type="Examples.AnotherExampleObject"/>
[C#]
public class AnotherExampleObject : IInitializingObject
{
    public void AfterPropertiesSet()
    {
        // do some initialization work
    }
}
```

but does not couple the code to Spring.NET.

Note

When deploying an object in `prototype` mode, the lifecycle of the object changes slightly. By definition, Spring.NET cannot manage the complete lifecycle of a non-singleton / `prototype` object, since after it is created, it is given to the client and the container no longer keeps a reference to the object. You can think of Spring.NET's role when talking about a non-singleton (`prototype`) object as a replacement for the new operator. Any lifecycle aspects past that point have to be handled by the client.

3.5.1.2. IDisposable / destroy-method

The System.IDisposable interface provides you with the ability to get a callback when an IObjectFactory is destroyed. The IDisposable interface specifies exactly one method:

- void Dispose(): and is called on destruction of the container. This allows you to release any resources you are keeping in this object (such as database connections). You can throw *any* Exception here... however, any such Exception will not stop the destruction of the container - it will only get logged.

Note

Note: If you choose you can avoid having your class implement IDisposable since the Spring.Core library provides support for a generic destroy-method, given to the object definition in the object configuration store (be it XML, or a database, etc).

```
<object id="exampleInitObject" type="Examples.ExampleObject" destroy-method="cleanup"/>
[C#]
public class ExampleObject
{
    public void cleanup()
    {
        // do some destruction work (such as closing any open connection (s))
    }
}
```

is exactly the same as:

```
<object id="exampleInitObject" type="Examples.AnotherExampleObject"/>
[C#]
public class AnotherExampleObject : IDisposable
{
    public void Dispose()
    {
        // do some destruction work
    }
}
```

3.5.2. Knowing who you are

3.5.2.1. IObjectFactoryAware

A class which implements the Spring.Objects.Factory.IObjectFactoryAware interface is provided with a reference to the IObjectFactory that created it. The interface specifies one (write-only) property:

- IObjectFactory ObjectFactory: the property that will be set *after the initialization methods* (AfterPropertiesSet and the init-method).

This allows objects to manipulate the IObjectFactory that created them programmatically, through the IObjectFactory interface, or by casting the reference to a known subclass of this which exposes additional functionality. Primarily this would consist of programmatic retrieval of other objects. While there are cases when this capability is useful, it should generally be avoided, since it couples the code to Spring.NET, and does not follow the Inversion of Control style, where collaborators are provided to objects as properties.

3.5.2.2. IObjectNameAware

The Spring.Objects.Factory.IObjectNameAware interface gives you the ability to let the container set the name of the object definition on the object instance itself. In those cases where your object needs to know what its name is, implement this interface.

- `string ObjectName`: the property that will be set to let the object know what its name is.

3.5.3. IFactoryObject

The `Spring.Objects.Factory.IFactoryObject` interface is to be implemented by objects that *are themselves factories*. The `IObjectFactory` interface provides one method and two (read-only) properties:

- `object GetObject()`: has to return an instance of the object this factory creates. The instance can possibly be shared (depending on whether this factory provides singletons or prototypes).
- `bool IsSingleton`: has to return `true` if this IFactoryObject returns singletons, `false` otherwise.
- `Type ObjectType`: has to return either the object type returned by the `GetObject()` method or `null` if the type isn't known in advance.

`IFactoryObject` implementions you already have examples of in Section 3.3.5, “Setting a reference using the members of other objects and classes.” are `ProprteryRetrievingFactoryObject` and `FieldRetrievingFactoryObject`.

3.6. Abstract and Child object definitions

An object definition potentially contains a large amount of configuration information, including container specific information (i.e. initialization method, static factory method name, etc.) and constructor arguments and property values. A child object definition is an object definition that inherits configuration data from a parent definition. It is then able to override some values, or add others, as needed. Using parent and child object definitions can potentially save a lot of typing. Effectively, this is a form of templating.

When working with an `IObjectFactory` programmatically, child object definitions are represented by the `ChildObjectDefinition` class. Most users will never work with them on this level, instead configuring object definitions declaratively in something like the `XmlObjectFactory`. In an `XmlObjectFactory` object definition, a child object definition is indicated simply by using the parent attribute, specifying the parent object definition as the value of this attribute.

```
<object id="inheritedTestObject" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsWithDifferentClass" type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObject" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</object>
```

A child object definition will use the object class from the parent definition if none is specified, but can also override it. In the latter case, the child object class must be compatible with the parent, i.e. it must accept the parent's property values.

A child object definition will inherit constructor argument values, property values and method overrides from the parent, with the option to add new values. If init method, destroy method and/or static factory method are specified, they will override the corresponding parent settings.

The remaining settings will always be taken from the child definition: `depends on`, `autowire mode`, `dependency check`, `singleton`, `lazy init`.

In the case where the parent definition does not specify a class...

```
<object id="inheritedTestObjectWithoutClass">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsWithClass" type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObjectWithoutClass" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</object>
```

... the parent object cannot be instantiated on its own since the definition is incomplete. The definition is also implicitly considered to be abstract. An object definition can also be explicitly declared as abstract using the abstract attribute. Valid values of the attribute are true and false. An abstract definition like this is usable just as a pure template or abstract object definition that will serve as a parent definition for child definitions. Trying to use such parent objects on its own (by referring to it as a ref property of another object, or doing an explicit `GetObject()` with the parent object id), will result in an error. Declaring the object as abstract will prevent it being instantiated and any attempt to instantiate the object will result in an `ObjectDefinitionIsAbstract` exception being thrown. The container's internal `PreInstantiateSingletons` method will completely ignore object definitions which are considered abstract.

Note

Application contexts (but not simple object factories) will by default pre-instantiate all singletons. Therefore it is important (at least for singleton objects) that if you have a (parent) object definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually attempt to pre-instantiate it.

```
<object id="abstractObject" abstract="true"
  type="Spring.Objects.DerivedTestObject, Spring.Core.Tests">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</object>
<object id="inheritsFromAbstract" type="Spring.Objects.DerivedTestObject, Spring.Core.Tests"
  parent="inheritedTestObjectWithoutClass" init-method="Initialize">
  <property name="name" value="override"/>
  <!-- age will inherit value of 1 from parent -->
</object>
```

3.7. Interacting with the IObjectFactory

The `IObjectFactory` is essentially nothing more than an advanced factory capable of maintaining a registry of different objects and their dependencies. The `IObjectFactory` enables you to read object definitions and access them using the object factory. When using just the `IObjectFactory` you would create an instance of one and then read in some object definitions in the XML format as follows:

```
[C#]
IResource input = new FileSystemResource ("objects.xml");
XmlObjectFactory factory = new XmlObjectFactory(input);
```

That is pretty much it. Using `GetObject(string)` (or the more concise indexer method `factory["string"]`) you can retrieve instances of your objects...

```
[C#]
object foo = factory.GetObject ("foo"); // gets the object defined as 'foo'
object bar = factory ["bar"];           // same thing, just using the indexer
```

You'll get a reference to the same object if you defined it as a singleton (the default) or you'll get a new instance each time if you set the `singleton` property of your object definition to *false*.

```
<object id="exampleObject" type="Examples.ExampleObject, ExamplesLibrary"/>
<object id="anotherObject" type="Examples.ExampleObject, ExamplesLibrary" singleton="false"/>
```

```
[C#]
object one = factory ["exampleObject"];   // gets the object defined as 'exampleObject'
object two = factory ["exampleObject"];
Console.WriteLine (one == two)            // prints 'true'
object three = factory ["anotherObject"]; // gets the object defined as 'anotherObject'
object four = factory ["anotherObject"];
Console.WriteLine (three == four);        // prints 'false'
```

The client-side view of the `IObjectFactory` is surprisingly simple. The `IObjectFactory` interface has only seven methods (and the aforementioned indexer) for clients to call:

- `bool ContainsObject(string)`: returns true if the `IObjectFactory` contains an object definition that matches the given name.
- `object GetObject(string)`: returns an instance of the object registered under the given name. Depending on how the object was configured by the `IObjectFactory` configuration, either a singleton (and thus shared) instance or a newly created object will be returned. An `ObjectsException` will be thrown when either the object could not be found (in which case it'll be a `NoSuchObjectDefinitionException`), or an exception occurred while instantiated and preparing the object.
- `Object this [string]`: this is the indexer for the `IObjectFactory` interface. It functions in all other respects in exactly the same way as the `GetObject(string)` method. The rest of this documentation will always refer to the `GetObject(string)` method, but be aware that you can use the indexer anywhere that you can use the `GetObject(string)` method.
- `Object GetObject(string, Type)`: returns an object, registered under the given name. The object returned will be cast to the given `Type`. If the object could not be cast, corresponding exceptions will be thrown (`ObjectNotOfRequiredTypeException`). Furthermore, all rules of the `GetObject(string)` method apply (see above).
- `bool IsSingleton(string)`: determines whether or not the object definition registered under the given name is a singleton or a prototype. If the object definition corresponding to the given name could not be found, an exception will be thrown (`NoSuchObjectDefinitionException`)
- `string[] GetAliases(string)`: returns the aliases for the given object name, if any were defined in the `IObjectDefinition`.
- `void ConfigureObject(object target):` Injects dependencies into the supplied target instance. The name of the abstract object definition is the `System.Type.FullName` of the target instance. This method is typically used when objects are instantiated outside the control of a developer, for example when ASP.NET instantiates web controls and when a WinForm application creates UserControls.
- `void ConfigureObject(object target, string name):` Offers the same functionality as the previously listed Configure method but uses a named object definition instead of using the type's full name.

3.7.1. Obtaining an `IFactoryObject`, not its product

Sometimes there is a need to ask an `IObjectFactory` for an actual `IFactoryObject` instance itself, not the object it produces. This may be done by prepending the object id with `&` when calling the `GetObject` method of the `IObjectFactory` and `IApplicationContext` interfaces. So for a given `IFactoryObject` with an id `myObject`, invoking `GetObject("myObject")` on the `IObjectFactory` will return the product of the

IFactoryObject, but invoking GetObject("&myObject") will return the IFactoryObject instance itself.

3.8. Customizing objects with IObjectPostProcessors

An object post-processor is a class that implements the Spring.Objects.Factory.Config.IObjectPostProcessor interface, which consists of two callback methods shown below.

```
object PostProcessBeforeInitialization(object instance, string name);

object PostProcessAfterInitialization(object instance, string name);
```

When such a class is registered as a post-processor with the container, for each object instance that is created by the container, the post-processor method PostProcessBeforeInitialization will be called by the container before any initialization methods such as the AfterPropertiesSet method of the IInitializingObject interface and any declared init method) are called. The post-processor method PostProcessAfterInitialization is called subsequent to any init methods.

The post-processor is free to do what it wishes with the object, including ignoring the callback completely. An object post-processor will typically check for marker interfaces, or do something such as wrap an object with a proxy. Some Spring.NET helper classes are implemented as object post-processors.

Other extensions to the the IObjectPostProcessors interface are IInstantiationAwareObjectPostProcessor and IDestructionAwareObjectPostProcessor defined below

```
public interface IInstantiationAwareObjectPostProcessor : IObjectPostProcessor
{
     object PostProcessBeforeInstantiation(Type objectType, string objectName);
}

public interface IDestructionAwareObjectPostProcessor : IObjectPostProcessor
{
     void PostProcessBeforeDestruction (object instance, string name);
}
```

The "BeforeInstantiation" callback method is called right before the container creates the object. If the object returned by this method is not null then the default instantiation behavor of the container is short circuited. The returned object is the one registered with the container and no other IObjectPostProcessor callbacks will be invoked on it. This mechanism is useful if you would like to expose a proxy to the object instead of the actual target object. The "BeforeDestruction" callack is called before a singletons destroy method is invoked.

It is important to know that the IObjectFactory treats object post-processors slightly differently than the IApplicationContext. An IApplicationContext will automatically detect any objects which are deployed into it that implement the IObjectPostProcessor interface, and register them as post-processors, to be then called appropriately by the factory on object creation. Nothing else needs to be done other than deploying the post-processor in a similar fashion to any other object. On the other hand, when using plain IObjectFactories, object post-processors have to manually be explicitly registered, with a code sequence such as...

```
ConfigurableObjectFactory factory = new .....; // create an IObjectFactory
... // now register some objects
// now register any needed IObjectPostProcessors
MyObjectPostProcessor pp = new MyObjectPostProcessor();
factory.AddObjectPostProcessor(pp);
// now start using the factory
...
```

Since this manual registration step is not convenient, and `IApplicationContexts` are functionally supersets of `IObjectFactories`, it is generally recommended that `IApplicationContext` variants are used when object post-processors are needed.

3.9. Customizing object factories with ObjectFactoryPostProcessors

An object factory post-processor is a class that implements the `Spring.Objects.Factory.Config.IObjectFactoryPostProcessor` interface. It is executed manually (in the case of the `IObjectFactory`) or automatically (in the case of the `IApplicationContext`) to apply changes of some sort to an entire ObjectFactory, after it has been constructed. When in need of special behavior in the `IObjectFactory` you need to consult the functionality provided by the `Spring.Objects.Factory.Config` namespace, which should provide you all the things you need. Using the types defined in the `Spring.Objects.Factory.Config` namespace you can for instance define a properties file of which the property-entries are used as replacements values in the `IObjectFactory`. You are free to define you own implementations of the `IObjectFactoryPostProcessor` interface. Spring.NET includes a number of pre-existing object factory post-processors, such as `PropertyResourceConfigurer` and `PropertyPlaceHolderConfigurer`, both described below.

In the case of an `IObjectFactory`, the process of applying an `IObjectFactoryPostProcessor` is manual, and will be similar to this:

```
XmlObjectFactory factory = new XmlObjectFactory(new FileSystemResource("objects.xml"));
// create placeholderconfigurer to bring in some property
// values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("ado.properties"));
// now actually do the replacement
cfg.PostProcessObjectFactory(factory);
```

An `IApplicationContext` will detect any objects which are deployed into it which implement the `ObjectFactoryPostProcessor` interface, and automatically use them as object factory post-processors, at the appropriate time. Nothing else needs to be done other than deploying these post-processor in a similar fashion to any other object.

Since this manual step is not convenient, and `IApplicationContexts` are functionally supersets of ObjectFactories, it is generally recommended that `IApplicationContext` variants are used when object factory post-processors are needed.

3.9.1. The `PropertyPlaceholderConfigurer`

The `PropertyPlaceholderConfigurer` is an excellent solution when you want to externalize a few properties from a file containing object definitions. Say you want to let the person responsible for deploying applications just fill in database URLs, usernames and passwords. You could of course let him or her change the XML file containing object definitions, but that would be risky. Instead, use the `PropertyPlaceHolderConfigurer` to - at runtime - replace those properties by values from a configuration file.

Note that `IApplicationContexts` are able to automatically recognize and apply objects deployed in them that implement the `IObjectFactoryPostProcessor` interface. This means that as described here, applying a `PropertyPlaceholderConfigurer` is much more convenient when using an `IApplicationContext`. For this reason, it is recommended that users wishing to use this or other object factory postprocessors use an `IApplicationContext` instead of an `IObjectFactory`.

In the example below a data access object needs to be configured with a database connection and also a value for the maximum number of results to return in a query. Instead of hard coding the values into the main Spring.NET configuration file we use place holders, in the NAnt style of ${variableName}, and obtain their values from NameValueSections in the standard .NET application configuration file. The Spring.NET configuration file looks like:

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    </sectionGroup>
    <section name="DaoConfiguration" type="System.Configuration.NameValueSectionHandler"/>
    <section name="DatabaseConfiguration" type="System.Configuration.NameValueSectionHandler"/>
  </configSections>

  <DaoConfiguration>
    <add key="maxResults" value="1000"/>
  </DaoConfiguration>

  <DatabaseConfiguration>
    <add key="connection.string" value="dsn=MyDSN;uid=sa;pwd=myPassword;"/>
  </DatabaseConfiguration>

  <spring>
    <context>
      <resource uri="assembly://DaoApp/DaoApp/objects.xml"/>
    </context>
  </spring>

</configuration>
```

Notice the presence of two NameValueSections in the configuration file. These name value pairs will be referred to in the Spring.NET configuration file. In this example we are using an embedded assembly resource for the location of the Spring.NET configuration file so as to reduce the chance of accidental tampering in deployment. This Spring.NET configuration file is shown below.

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net
                        http://www.springframework.net/xsd/spring-objects.xsd" >

    <object name="productDao" type="DaoApp.SimpleProductDao, DaoApp ">
      <property name="maxResults" value="${maxResults}"/>
      <property name="dbConnection" ref="myConnection"/>
    </object>

    <object name="myConnection" type="System.Data.Odbc.OdbcConnection, System.Data">
      <property name="connectionstring" value="${connection.string}"/>
    </object>

    <object name="appConfigPropertyHolder"
            type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">

        <property name="configSections">
          <value>DaoConfiguration,DatabaseConfiguration</value>
        </property>

    </object>
</objects>
```

The values of `${maxResults}` and `${connection.string}` match the key names used in the two NameValueSectionHandlers `DaoConfiguration` and `DatabaseConfiguration`. The `PropertyPlaceholderConfigurer` refers to these two sections via a comma delimited list of section names in the `configSections` property.

The `PropertyPlaceholderConfigurer` class also supports retrieving name value pairs from other `IResource` locations. These can be specified using the `Location` and `Locations` properties of the `PropertyPlaceHolderConfigurer` class.

If there are properties with the same name in different resource locations the default behavior is that the last property processed overrides the previous values. This is behavior is controlled by the `LastLocationOverrides` property. True enables overriding while false will append the values as one would normally expect using `NameValueCollection.Add`.

Note

In an ASP.NET environment you must specify the full, four-part name of the assembly when using a `NameValueFileSectionHandler`; to wit...

```
<section name="hibernateConfiguration"
          type="System.Configuration.NameValueFileSectionHandler, System,
                Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"/>
```

3.9.1.1. Replacement with Environment Variables

You may also use the value environment variables to replace property placeholders. The use of environment variables is controlled via the property `EnvironmentVariableMode`. This property is an enumeration of the type `EnvironmentVariablesMode` and has three values, Never, Fallback, and Override. `Fallback` is the default value and will resolve a property placeholder if it was not already done so via a value from a resource location. `Override` will apply environment variables before applying values defined from a resource location. `Never` will, quite appropriately, disable environment variable substitution. An example of how the `PropertyPlaceholderConfigurer`XML is modified to enable override usage is shown below

```
<object name="appConfigPropertyHolder"
        type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">
        <property name="configSections" value="DaoConfiguration,DatabaseConfiguration"/>
        <property name="EnvironmentVariableMode" value="Override"/>
    </object>
</objects>
```

3.9.2. The `PropertyOverrideConfigurer`

The `PropertyOverrideConfigurer`, another object factory post-processor, is similar to the `PropertyPlaceholderConfigurer`, but in contrast to the latter, the original definitions can have default values or no values at all for object properties. If an overriding configuration file file does not have an entry for a certain object property, the default context definition is used.

Note that the object factory definition is *not* aware of being overridden, so it is not immediately obvious when looking at the XML definition file that the override configurer is being used. In case that there are multiple `PropertyOverrideConfigurer` instances that define different values for the same object property, the last one will win (due to the overriding mechanism).

The example usage is similar to when using `PropertyPlaceHolderConfigurer` except that the key name refers to the name given to the object in the Spring.NET configuration file and is suffixed via 'dot' notation with the name of the property For example, if the application configuration file is

```
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    </sectionGroup>
    <section name="DaoConfigurationOverride" type="System.Configuration.NameValueSectionHandler"/>
  </configSections>

  <DaoConfigurationOverride>
    <add key="productDao.maxResults" value="1000"/>
  </DaoConfigurationOverride>

  <spring>
    <context>
      <resource uri="assembly://DaoApp/DaoApp/objects.xml"/>
    </context>
  </spring>

</configuration>
```

Then the value of 1000 will be used to overlay the value of 2000 set in the Spring.NET configuration file shown below

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects.xsd

    <object name="productDao" type="PropPlayApp.SimpleProductDao, PropPlayApp " >
      <property name="maxResults" value="2000"/>
      <property name="dbConnection" ref="myConnection"/>
      <property name="log" ref="daoLog"/>
    </object>

    <object name="daoLog" type="Spring.Objects.Factory.Config.Log4NetFactoryObject, Spring.Core">
      <property name="logName" value="DAOLogger"/>
    </object>

    <object name="myConnection" type="System.Data.Odbc.OdbcConnection, System.Data">
      <property name="connectionstring">
        <value>dsn=MyDSN;uid=sa;pwd=myPassword;</value>
      </property>
    </object>

    <object name="appConfigPropertyOverride" type="Spring.Objects.Factory.Config.PropertyOverrideConfigurer, Spr
      <property name="configSections">
        <value>DaoConfigurationOverride</value>
      </property>
    </object>

</objects>
```

3.10. Using the alias element to add aliases for existing objects

In an object definition itself, you may supply more than one name for the object, by using a combination of the id and name attributes as discussed in Section 3.2.5, “The object identifiers (id and name)”. This approach to aliasing objects has some limitations when you would like to assemble the main application configuration file from multiple files. See Section 3.16, “Importing Object Definitions from One File Into Another” for more information. This usage pattern is common when each configuration file represents a logical layer or component within the application. In this case you may want to refer to a common object dependency using a name that is specific to each file. If the common object dependency is defined in the main application configuration file itself, then one can use the name element as an alias mechansim. However, if the main application configuration file should not be responsible for defining the common object dependency, since it logically 'belongs' to one of the other layers or components, you can not use the name attribute to achive this goal.

In this case, you can define an alias using an explicit `alias` element contained in the main application configuration file.

```
<alias name="fromName" alias="toName"/>
```

This allows an object named `fromName` to be referred to as `toName` across all application configuration files.

As a concrete example, consider the case where the configuration file 'a.xml' (representing component A) defines a connection object called componentA-connection. In another file, 'b.xml' (representing component B) would like to refer to the connection as componentB-connection. And the main application, MyApp, defines its own XML fragment to assembles the final application configuration from all three fragments and would like to refer to the connection as myApp-connection. This scenario can be easily handled by adding to the MyApp XML fragement the following standalone aliases:

```
<alias    name="componentA-connection"    alias="componentB-connection"/>    <alias
name="componentA-connection" alias="myApp-connection"/>
```

Now each component and the main app can refer to the connection via a name that is unique and guaranteed not to clash with any other definition (effectively there is a namespace), yet they refer to the same object.

3.11. The IResource abstraction

The `IResource` interface contained in the `Spring.Core.IO` namespace provides a common interface to describe and access data from diverse resource locations. This abstraction lets you treat the `InputStream` from a file and from a URL in a polymorphic and protocol-independent manner... the .NET BCL does not provide such an abstraction. The `IResource` interface inherits from `IInputStream` that provides a single property `Stream InputStream`. The `IResource` interface adds descriptive information about the resource via a number of additional properties.

Table 3.6. IResource Properties

Property	Explanation
`InputStream`	Inherited from IInputStream. Opens and returns a `System.IO.Stream`.
`Exists`	Checks if the resource exists, returning false if it doesn't.
`IsOpen`	Will return true is multiple streams cannot be opened for this resource. This will be false for some resources, but file-based resources for instance, cannot be read multiple times concurrently.
`Description`	Returns a description of the resource, such as the fully qualified file name or the actual URL.
`Uri`	The Uri representation of the resource.
`File`	Returns a `System.IO.FileInfo` for this resource if it can be resolved to an absolute file path.

and the methods

Table 3.7. IResource Methods

Method	Explanation
`IResource`	Creates a resource relative to this resource using relative pathlike notation (./

Method	**Explanation**
`CreateRelative (string relativePath)`	and ../).

Note that the abstraction is not perfect since returning a FileInfo object can only be done for certain resource types. The actual behavior is implementation specific.

The resource implementations provided are

- `AssemblyResourceassembly://<AssemblyName>/<NameSpace>/<ResourceName>`
- `ConfigSectionResourceconfig://<path to section>`
- `FileSystemResourcefile://<filename>`
- `InputStreamResourceSystem.IO.Stream`
- `UriResource`

Refer to the MSDN documentation for more information on supported Uri scheme types.

To load resources given their Uri sytax, an implementation of the `IResourceLoader` is used. The default implementation is `ConfigurableResourceLoader`. Typically you will not need to access this class directly since the `IApplicationContext` implements the `IResourceLoader` interface that contains the single method `IResource GetResource(string location)`. The provided implementations of `IApplicationContext` delegate this method to an instance of `ConfigurableResourceLoader` which supports the Uri protocols/schemes listed previously. If you do not specify a protocol then the file protocol is used. The following shows some sample usage.

```
IResource resource = appContext.GetResource("http://www.springframework.net/license.html");
resource = appContext.GetResource("assembly://Spring.Core.Tests/Spring/TestResource.txt");
resource = appContext.GetResource("https://sourceforge.net/");
resource = appContext.GetResource("file:///C:/WINDOWS/ODBC.INI");

StreamReader reader = new StreamReader(resource.InputStream);
Console.WriteLine(reader.ReadToEnd());
```

Other protocols can be registered along with a new implementations of an IResource that must correctly parse a Uri string in its constructor. An example of this can be seen in the `Spring.Web` namespace that uses `Server.MapPath` to resolve the filename of a resource.

The `CreateRelative` method allows you to easily load resources based on a relative path name. In the case of relative assembly resources, the relative path navigates the namespace within an assembly. For example:

```
IResource res = new AssemblyResource("assembly://Spring.Core.Tests/Spring/TestResource.txt");
IResource res2 = res.CreateRelative("./Core.IO/TestIOResource.txt");
```

This loads the resource `TestResource.txt` and then navigates to the `Spring.Core.IO` namespace and loads the resource `TestIOResource.txt`

3.12. Introduction to the `IApplicationContext`

While the `Spring.Objects` namespace provides basic functionality for managing and manipulating objects, often in a programmatic way, the `Spring.Context` namespace introduces the `IApplicationContext` interface, which enhances the functionality provided by the `IObjectFactory` interface.

The basis for the context module is the `IApplicationContext` interface, located in the `Spring.Context` namespace. To be able to work in a more framework-oriented way, using layering and hierarchical contexts, the `Spring.Context` namespace also provides the following:

- *Loading of multiple (hierarchical) contexts*, allowing some of them to be focused and used on one particular layer, for example the web layer of an application.
- *Access to localized resources* at the application level by implementing `IMessageSource`.
- *Uniform access to resources* that can be read in as an InputStream, such as URLs and files by implementing `IResourceLoader`
- *Loosely Coupled Event Propagation*. Publishers and subscribers of events do not have to be directly aware of each other as they register their interest indirectly through the application context.

As the `IApplicationContext` includes all the functionality the object factory via its inheritance of the `IObjectFactory` interface, it is generally recommended to be used over the `IObjectFactory`. Classes that implement only the `IObjectFactory` interface, such as `XmlObjectFactory`, reflect the initial creation of the framework in Java to support resource constrained environments.

Since the `IApplicationContext` is a subclass of `IObjectFactory`, you can define objects using the XML format (as explained in Section 3.2.1, “The IObjectFactory and IApplicationContext” and Appendix A, *Spring.NET's spring-objects.xsd*) and retrieve them using their unique identifier. Also, you will be able to use all of the other features explained in the previous chapters, such as *autowiring*, *setter-based* or *constructor-based* dependency injection, *dependency checking*, *lifecycle interfaces*, etc.

The features unique to the `IApplicationContext` are described below. These are

- Singleton service locator style access. See Section 3.17, “Service Locator access”
- Automatic registration of objects that implement the `IObjectPostProcessors` interface. See Section 3.15, “Customized behavior in the ApplicationContext”
- Registration of classes used internally by `IApplicationContext` such as resource protocol handlers, XML parsers for object definitions, and type aliases. See Section 3.13, “Configuration of IApplicationContext”

3.13. Configuration of IApplicationContext

Well known locations in the .NET application configuration file are used to register resource handlers, custom parsers, type alias, and custom type converts in addition to the context and objects sections mentioned previously. A sample .NET application configuration file showing all these features is shown below. Each section require the use of a custom configuration section handler. Note that the types shown for resource handlers and parsers are fictional.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />

      <section name="configParsers" type="Spring.Context.Support.ConfigParsersSectionHandler, Spring.Core"/>
      <section name="resources" type="Spring.Context.Support.ResourcesSectionHandler, Spring.Core"/>
      <section name="typeAliases" type="Spring.Context.Support.TypeAliasesSectionHandler, Spring.Core"/>
      <section name="typeConverters" type="Spring.Context.Support.TypeConvertersSectionHandler, Spring.Core"/>

     </sectionGroup>
  </configSections>

  <spring>
    <configParsers>
      <parser namespace="http://schemas.springframework.net/aop" type="Spring.Aop.Support.AopConfigParser, Sprin
      <parser namespace="http://schemas.springframework.net/web" type="Spring.Web.Support.WebConfigParser, Sprin
    </configParsers>

    <resources>
      <resource protocol="db" type="MyCompany.MyApp.Resources.MyDbResource"/>
    </resources>
    <context caseSensitive="false">
      <resource uri="config://spring/objects"/>
      <resource uri="db://user:pass@dbName/MyDefinitionsTable"/>
    </context>

    <typeAliases>
        <alias name="WebServiceExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web"/>
        <alias name="DefaultPointcutAdvisor" type="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop"/>
        <alias name="AttributePointcut" type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop"/>
        <alias name="CacheAttribute" type="Spring.Attributes.CacheAttribute, Spring.Core"/>
        <alias name="MyType" type="MyCompany.MyProject.MyNamespace.MyType, MyAssembly"/>
     </typeAliases>

    <typeConverters>
      <converter for="Spring.Expressions.IExpression, Spring.Core" type="Spring.Objects.TypeConverters.Expressio
      <converter for="MyTypeAlias" type="MyCompany.MyProject.Converters.MyTypeConverter, MyAssembly"/>
    </typeConverters>

    <objects>
        ...
    </objects>

  </spring>

</configuration>
```

The new sections are described below. The attribute `caseSensitive` allows the for both `IObjectFactory` and `IApplicationContext` implementations to not pay attention to the case of the object names. This is important in web applications so that ASP.NET pages can be resolved in a case independent manner. The default value is true.

3.13.1. Registering custom parsers

Instead of using the default XML schema that is generic in nature to define an object's properties and dependencies, you can create your own XML schema specific to an application domain. This has the benefit of being easier to type and getting XML intellisence for the schema being used. The downside is that you need to write code that will transform this XML into Spring object definitions. One would typically implement a custom parser by deriving from the class `DefaultXmlObjectDefinitionParser` and overriding the methods `int ParseRootElement(XmlElement root, XmlResourceReader reader)` and `int ParseElement(XmlElement element, XmlResourceReader reader)`.

3.13.2. Creating custom resource handlers

Creating a custom resource handler means implementing the `IResource` interface. The base class `AbstractResource` is a useful starting point. Look at the Spring source for classes such as `FileSystemResource` or `AssemblyResource` for implementation tips.

3.13.3. Configuring Type Aliases

Type aliases allow you to simplify Spring configuration file by replacing fully qualified type name with an alias for frequently used types. Aliases can be registered both within config file and programatically and can be used anywhere in the context config file where fully qualified type name is expected. Type aliases can also be defined for generic types.

In order to configure type aliases you need to define custom config section in the Web/App.config file for your application, as well as the custom configuration section handler. See the previous XML configuration listing for an example that makes an alias for the WebServiceExporter type. Once you have aliases defined, you can simply use them anywhere where you would normally specify fully qualified type name:

```
<object id="MyWebService" type="WebServiceExporter">
    ...
</object>

<object id="cacheAspect" type="DefaultPointcutAdvisor">
    <property name="Pointcut">
        <object type="AttributePointcut">
            <property name="Attribute" value="CacheAttribute"/>
        </object>
    </property>
    <property name="Advice" ref="aspNetCacheAdvice"/>
</object>
```

For an example showing type aliases for generic types see Section 3.2.4, “Object creation of generic types”.

3.13.4. Registering Type Converters

The standard .NET mechanism for specifying a type converter is to add a `TypeConverter` attribute to type definition and specify the type of the Converter. This is the preferred way of defining type converters if you control the source code for the type that you want to define a converter for. However, this configuration section allows you to specify converters for the types that you don't control, and it also allows you to override some of the standard type converters, such as the ones that are defined for some of the types in the .NET Base Class Library.

3.14. Added functionality of the `IApplicationContext`

As already stated in the previous section, the `IApplicationContext` has a couple of features that distinguish it from the `IObjectFactory`. Let us review them one-by-one.

3.14.1. Context Hierarchies

You can structure the configuration information of application context into hierarchies that naturally reflect the internal layering of your application. As an example, abstract object definitions may appear in a parent application context configuration file, possibly as an embedded assembly resource so as not to invite accidental changes.

```
<spring>
  <context>
    <resource uri="assembly://MyAssembly/MyProject/root-objects.xml"/>
        <context name="mySubContext">
            <resource uri="file://objects.xml"/>
        </context>
  </context>
</spring>
```

The nesting of `context` elements reflects the parent-child hierarchy you are creating. The nesting can be to any level though it is unlikely one would need a deep application hierarchy. As a reminder, the `type` attribute is optional and defaults to `Spring.Context.Support.XmlApplicationContext` The name of the context can be used in conjunction with the service locator class, `ContextRegistry` discussed later.

3.14.2. Using the `IMessageSource`

The `IApplicationContext` interface extends an interface called `IMessageSource` and provides localization (i18n or internationalization) services for text messages and other resource data types such as images. This functionality makes it easier to use .NET's localization features at an application level and also offers some performance enhancements due to caching of retrieved resources. Together with the `NestingMessageSource`, capable of hierarchical message resolving, these are the basic interfaces Spring.NET provides to for localization. Let's quickly review the methods defined there:

- `string GetMessage(string name)`: retrieves a message from the `IMessageSource` and uses the CurrentUICulture.
- `string GetMessage(string name, CultureInfo cultureInfo)`: retrieves a message from the `IMessageSource` and using a specified culture.
- `string GetMessage(string name, params object[] args)`: retrieves a message from the `IMessageSource` using a variable list of arguments as replacement values in the message. The CurrentUICulture is used to resolve the message.
- `string GetMessage(string name, CultureInfo cultureInfo, params object[] args)`: retrieves a message from the `IMessageSource` using a variable list of arguments as replacement values in the message. The specified culture is used to resolve the message.
- `string GetMessage(IMessageSourceResolvable resolvable, CultureInfo culture)` : all properties used in the methods above are also wrapped in a class - the `MessageSourceResolvable`, which you can use in this method.
- `object GetResourceObject(string name)`:Get a localized resource object, i.e. Icon, Image, etc. given the resource name. The CurrentUICulture is used to resolve the resource object.
- `object GetResourceObject(string name, CultureInfo cultureInfo)`:Get a localized resource object, i.e. Icon, Image, etc. given the resource name. The specified culture is used to resolve the resource object.
- `void ApplyResources(object value, string objectName, CultureInfo cultureInfo)`: Uses a ComponentResourceManager to apply resources to all object properties that have a matching key name. Resource key names are of the form objectName.propertyName

When an `IApplicationContext` gets loaded, it automatically searches for an `IMessageSource` object defined in the context. The object has to have the name `messageSource`. If such an object is found, all calls to the methods described above will be delegated to the message source that was found. If no message source was found, the `IApplicationContext` checks to see if it has a parent containing a similar object, with a similar name. If so, it uses that object as the `IMessageSource`. If it can't find any source for messages, an empty `StaticMessageSource` will be instantiated in order to be able to accept calls to the methods defined above.

Fallback behavior

The fallback rules for localized resources seem to have a bug that is fixed by applying Service Pack 1 for .NET 1.1. This affects the use of IMessageSource.GetMessage methods that specify CultureInfo. The core of the issue in the .NET BCL is the method ResourceManager.GetObject that accepts CultureInfo. .

Spring.NET provides two `IMessageSource` implementations. These are `ResourceSetMessageSource` and `StaticMessageSource`. Both implement `IHierarchicalMessageSource` to resolve messages hierarchically. The `StaticMessageSource` is hardly ever used but provides programmatic ways to add messages to the source. The `ResourceSetMessageSource` is more interesting and an example is provided for in the distribution and discussed more extensively in the Chapter 16, *Quickstarts* section. The `ResourceSetMessageSource` is configured by providing a list of `ResourceManagers`. When a message code is to be resolved, the list of ResourceManagers is searched to resolve the code. For each `ResourceManager` a `ResourceSet` is retrieved and asked to resolve the code. Note that this search does not replace the standard hub-and-spoke search for localized resources. The ResourceManagers list specifies the multiple 'hubs' where the standard search starts.

```
<object name="messageSource" type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
  <property name="resourceManagers">
    <list>
       <value>Spring.Examples.AppContext.MyResource, Spring.Examples.AppContext</value>
    </list>
  </property>
```

```
</object>
```

You can specify the arguments to construct a ResourceManager as a two part string value containing the base name of the resource and the assembly name. This will be converted to a ResourceManager via the `ResourceManagerConverter` TypeConverter. This converter can be similarly used to set a property on any object that is of the type `ResourceManager.` You may also specify an instance of the `ResourceManager` to use with via an object reference. The convenience class `Spring.Objects.Factory.Config.ResourceManagerFactoryObject` can be used to conveniently create an instance of a ResourceManager.

```
<object name="myResourceManager" type="Spring.Objects.Factory.Config.ResourceManagerFactoryObject, Spring.Core">
  <property name="baseName">
    <value>Spring.Examples.AppContext.MyResource</value>
  </property>
  <property name="assemblyName">
    <value>Spring.Examples.AppContext</value>
  </property>
</object>
```

In application code, a call to `GetMessage` will retrieve a properly localized message string based on a code value. Any arguments present in the retrieved string are replaced using `String.Format` semantics. The ResourceManagers, ResourceSets and retrieved strings are cached to provide quicker lookup performance. The key 'HelloMessage' is contained in the resource file with a value of `Hello {0} {1}`. The following call on the application context will return the string `Hello Mr. Anderson`. *Note* that the caching of ResourceSets is via the concatination of ResourceManager base name and CultureInfo string. This combination must be unique.

```
string msg = ctx.GetMessage("HelloMessage",
                            new object[] {"Mr.", "Anderson"},
                            CultureInfo.CurrentCulture );
```

It is possible to chain the resolution of messages by passing arguments that are themselves messages to be resolved giving you greater flexibility in how you can structure you message resolution. This is achieved by passing as an argument a class that implements `IMessageResolvable` instead of a string literal. The convenience class `DefaultMessageResolvable` is available for this purpose. As an example if the resource file contains a key name `error.required` that has the value '`{0} is required {1}`' and another key name `field.firstname` with the value '`First name`'. The following code will create the string '`First name is required dude!`'

```
string[] codes = {"field.firstname"};
DefaultMessageResolvable dmr = new DefaultMessageResolvable(codes, null);
ctx.GetMessage("error.required",
                new object[] { dmr, "dude!" },
                CultureInfo.CurrentCulture ));
```

The examples directory in the distribution contains an example program, `Spring.Examples.AppContext`, that demonstrates the usage of these features.

3.14.3. Using resources within Spring.NET

A lot of applications need to access resources. Resources here, might mean files, but also news feeds from the Internet or normal web pages. Spring.NET provides a clean and transparent way of accessing resources in a protocol independent way. The `IApplicationContext` has a method (`GetResource(string)`) to take care of this. Refer to Section 3.11, “The IResource abstraction” for more information on the string format to use and the `IResource` abstraction in general.

3.14.4. Loosely coupled events

The Eventing Registry allows developers to utilize a loosely coupled event wiring mechanism. By decoupling the event publication and the event subscription, most of the mundane event wiring is handled by the IoC container. Event publishers can publish their event to a central registry, either all of their events or a subset based on criteria such as delegate type, name, return value, etc... Event subscribers can choose to subscribe to any number of published events. Subscribers can subscriber to events based on the type of object exposing them, allowing one subscriber to handle all events of a certain type without regards to how many different instances of that type are created. Other subscription criteria include name, delegate type, and regular expression matching.

The `Spring.Objects.Events.IEventRegistry` interface represents the central registry and defines publish and subscribe methods.

- `void PublishEvents( object sourceObject )`: publishes all events of the source object to subscribers that implement the correct handler methods.
- `void Subscribe(object subscriber )`: The subscriber receives all events from the source object for which it has matching handler methods.
- `void Subscribe(object subscriber, Type targetSourceType )`: The subscriber receives all events from a source object of a particular type for which it has matching handler methods.

`IApplicationContext` implements this interface and delegates the implementation to an instance of `Spring.Objects.Events.Support.EventRegistry`. You are free to create and use as many EventRegistries as you like but since it is common to use only one in an application, `IApplicationContext` provides convenient access to a single instance.

Within the `example/Spring/Spring.Examples.EventRegistry` directory you will find an example on how to use this functionality. When you open up the project, the most interesting file is the EventRegistryApp.cs file. This application loads a set of object definitions from the application configuration file into an `IApplicationContext` instance. From there, three objects are loaded up: one publisher and two subscribers. The publisher publishes its events to the `IApplicationContext` instance:

```
// Create the Application context using configuration file
IApplicationContext ctx = ContextRegistry.GetContext();

// Gets the publisher from the application context
MyEventPublisher publisher = (MyEventPublisher)ctx.GetObject("MyEventPublisher");

// Publishes events to the context.
ctx.PublishEvents( publisher );
```

One of the two subscribers subscribes to all events published to the `IApplicationContext` instance, using the publisher type as the filter criteria.

```
// Gets first instance of subscriber
MyEventSubscriber subscriber = (MyEventSubscriber)ctx.GetObject("MyEventSubscriber");

// Gets second instance of subscriber
MyEventSubscriber subscriber2 = (MyEventSubscriber)ctx.GetObject("MyEventSubscriber");

// Subscribes the first instance to the any events published by the type MyEventPublisher
ctx.Subscribe( subscriber, typeof(MyEventPublisher) );
```

This will wire the first subscriber to the original event publisher. Anytime the event publisher fires an event, (`publisher.ClientMethodThatTriggersEvent1();`) the first subscriber will handle the event, but the second subscriber will not. This allows for selective subscription, regardless of the original prototype definition.

3.14.5. Event notification from `IApplicationContext`

Event handling in the `IApplicationContext` is provided through the `IApplicationListener` interface that contains the single method `void OnApplicationEvent( object source, ApplicationEventArgs applicationEventArgs )`. Classes that implement the `IApplicationListener` interface are automatically registered as a listener with the `IApplicationContext`. Publishing an event is done via the context's `PublishEvent( ApplicationEventArgs eventArgs )` method. This implemenation is based on the traditional *Observer* design pattern.

The event argument type, `ApplicationEventArgs`, adds the time of the event firing as a property. The derived class `ContextEventArgs` is used to notify observers on the lifecycle events of the application context. It contains a property `ContextEvent Event` that returns the enumeration `Refreshed` or `Closed.`. The `Refreshed` enumeration value indicated that the `IApplicationContext` was either initialized or refreshed. Initialized here means that all objects are loaded, singletons are pre-instantiated and the `IApplicationContext` is ready for use. The `Closed` is published when the `IApplicationContext` is closed using the `Dispose()` method on the `IConfigurableApplicationContext` interface. Closed here means that singletons are destroyed.

Implementing custom events can be done as well. Simply call the `PublishEvent` method on the `IApplicationContext`, specifying a parameter which is an instance of your custom event argument subclass.

Let's have a look at an example. First, the `IApplicationContext`:

```
<object id="emailer" type="Example.EmailObject">
    <property name="blackList">
        <list>
            <value>black@list.org</value>
            <value>white@list.org</value>
            <value>john@doe.org</value>
        </list>
    </property>
</object>

<object id="blackListListener" type="Example.BlackListNotifier">
    <property name="notificationAddress">
        <value>spam@list.org</value>
    </property>
</object>
```

and then, the actual objects:

```
public class EmailObject : IApplicationContextAware {

    // the blacklist
    private IList blackList;

    public IList BlackList
    {
      set { this.blackList = value; }
    }

    public IApplicationContext ApplicationContext
    {
      set { this.ctx = value; }
    }

    public void SendEmail(string address, string text) {
        if (blackList.contains(address))
        {
            BlackListEvent evt = new BlackListEvent(address, text);
            ctx.publishEvent(evt);
            return;
        }
        // send email...
    }
}

public class BlackListNotifier : IApplicationListener
```

```
{

    // notification address
    private string notificationAddress;

    public string NotificationAddress
    {
      set { this.notificationAddress = value; }
    }

    public void OnApplicationEvent(ApplicationEvent evt)
    {
        if (evt instanceof BlackListEvent)
        {
            // notify appropriate person
        }
    }
}
```

3.15. Customized behavior in the ApplicationContext

The `IObjectFactory` already offers a number of mechanisms to control the lifecycle of objects deployed in it (such as marker interfaces like `IInitializingObject` and `System.IDisposable`, their configuration only equivalents such as `init-method` and `destroy-method`) attributes in an XmlObjectFactory configuration, and object post-processors. In an `IApplicationContext`, all of these still work, but additional mechanisms are added for customizing behavior of objects and the container.

3.15.1. `ApplicationContextAware` marker interface

All marker interfaces available with ObjectFactories still work. The `IApplicationContext` does add one extra marker interface which objects may implement, `IApplicationContextAware`. An object which implements this interface and is deployed into the context will be called back on creation of the object, using the interface's `ApplicationContext` property, and provided with a reference to the context, which may be stored for later interaction with the context.

3.15.2. The `IObjectPostProcessor`

Object post-processors are classes which implement the `Spring.Objects.Factory.Config.IObjectPostProcessor` interface, have already been mentioned. It is worth mentioning again here though, that post-processors are much more convenient to use in `IApplicationContexts` than in plain `IObjectFactory` instances. In an `IApplicationContext`, any deployed object which implements the above marker interface is automatically detected and registered as an object post-processor, to be called appropriately at creation time for each object in the factory.

3.15.3. The `IObjectFactoryPostProcessor`

Object factory post-processors are classes which implement the `Spring.Objects.Factory.Config.IObjectFactoryPostProcessor` interface, have already been mentioned... it is worth mentioning again here though, that object factory post-processors are much more convenient to use in `IApplicationContexts`. In an `IApplicationContext`, any deployed object which implements the above marker interface is automatically detected as an object factory post-processor, to be called at the appropriate time.

3.15.4. The `PropertyPlaceholderConfigurer`

The PropertyPlaceholderConfigurer has already been described in the context of its use within an IObjectFactory. It is worth mentioning here though, that it is generally more convenient to use it with an IApplicationContext, since the context will automatically recognize and apply any object factory post-processors, such as this one, when they are simply deployed into it like any other object. There is no need for a manual step to execute it.

3.16. Importing Object Definitions from One File Into Another

It is often useful to split up container definitions into multiple XML files. One way to then load an application context which is configured from all these XML fragments is to use the application context constructor which takes multiple resource locations. With an object factory, an object definition reader can be used multiple times to read definitions from each file in turn. This approach is shown in the example below that loads part of the configuration information from a file and another part from the assembly

```
IApplicationContext context = new XmlApplicationContext(
        "file://objects.xml", "assembly://MyAssembly/MyProject/objects-dal-layer.xml");
```

Generally, the Spring.NET team prefers the above approach, assembling individual files because it keeps container configurations files unaware of the fact that they are being combined with others. However, an alternate approach is to compose one XML object definition file using one or more occurences of the import element to load definitions from other files. Any import elements must be placed before object elements in the file doing the importing. Let's look at a sample:

```
<objects>
  <import resource="services.xml"/>
  <import resource="resources/messageSource.xml"/>
  <import resource="/resources/themeSource.xml"/>
  <object id="object1" type="..."/>
  <object id="object2" type="..."/>
```

In this example, external object definitions are being loaded from 3 files, services.xml, messageSource.xml, and themeSource.xml. All location paths are considered relative to the definition file doing the importing, so services.xml in this case must be in the same directory as the file doing the importing, while messageSource.xml and themeSource.xml must be in a resources location below the location of the importing file. As you can see, a leading slash is actually ignored, but given that these are considered relative paths, it is probably better form not to use the slash at all.

The contents of the files being imported must be fully valid XML object definition files according to the XSD, including the top level objects element.

3.17. Service Locator access

The majority of the code inside an application is best written in a Dependency Injection (Inversion of Control) style, where that code is served out of an IObjectFactory or IApplicationContext container, has its own dependencies supplied by the container when it is created, and is completely unaware of the container. However, there is sometimes a need for singleton (or quasi-singleton) style access to an IObjectFactory or IApplicationContext. For example, third party code may try to construct new object directly without the ability to force it to get these objects out of the IObjectFactory. Similarly, nested user control components in a WinForm application are created inside the generated code inside InitializeComponent. If this user control would like to obtain reference to objects contained in the container it can use the service locator style approach and 'reach out' from inside the code to obtain the object it requires.

The `Spring.Context.Support.ContextRegistry` class allows you to obtain a reference to an `IApplicationContext` via a static locator method. The `ContextRegistry` is initialized when creating an `IApplicationContext` through use of the `ContextHandler` discussed previously. The simple static method `GetContext()` can then be used to retrieve the context. Alternatively, if you create an `IApplicationContext` though other means you can register it with the `ContextRegistry` via the method `void RegisterContext(IApplicationContext context)` in the start-up code of your application. Hierarchical context retrieval is also supported though the use of the `GetContext(string name)` method, for example:

```
IApplicationContex ctx = ContextRegistry.GetContext("mySubContext");
```

This would retrieve the nested context for the context configuration shown previously.

```
<spring>
  <context>
    <resource uri="assembly://MyAssembly/MyProject/root-objects.xml"/>
        <context name="mySubContext">
            <resource uri="file://objects.xml"/>
        </context>
  </context>
</spring>
```

Chapter 4. The IObjectWrapper and Type conversion

4.1. Introduction

(Available in 1.0)

The concepts encapsulated by the `IObjectWrapper` interface are fundamental to the workings of the core Spring.NET libraries The typical application developer most probably will not ever have the need to use the `IObjectWrapper` directly... because this is reference documentation however, we felt that some explanation of this core interface might be right. The `IObjectWrapper` is explained in this chapter since if you were going to use it at all, you would probably do that when trying to bind data to objects, which, nicely enough, is precisely the area that the `IObjectWrapper` addresses.

4.2. Manipulating objects using the IObjectWrapper

One quite important concept of the `Spring.Objects` namespace is encapsulated in the definition `IObjectWrapper` interface and its corresponding implementation, the `ObjectWrapper` class. The functionality offered by the `IObjectWrapper` includes methods to set and get property values (either individually or in bulk), get property descriptors (instances of the `System.Reflection.PropertyInfo` class), and to query the readability and writability of properties. The `IObjectWrapper` also offers support for nested properties, enabling the setting of properties on subproperties to an unlimited depth. The `IObjectWrapper` usually isn't used by application code directly, but by framework classes such as the various `IObjectFactory` implementations.

The way the `IObjectWrapper` works is partly indicated by its name: *it wraps an object* to perform actions on a wrapped object instance... such actions would include the setting and getting of properties exposed on the wrapped object.

Note: the concepts explained in this section are not important to you if you're not planning to work with the `IObjectWrapper` directly.

4.2.1. Setting and getting basic and nested properties

Setting and getting properties is done using the `SetPropertyValue()` and `GetPropertyValue()` methods, for which there are a couple of overloaded variants. The details of the various overloads (including return values and method parameters) are all described in the extensive API documentation supplied as a part of the Spring.NET distribution.

The aforementioned `SetPropertyValue()` and `GetPropertyValue()` methods do have a number of conventions for indicating the path of a property. A property path is an expression that implementations of the `IObjectWrapper` interface can use to look up the properties of the wrapped object; some examples of property paths include...

Table 4.1. Examples of property paths

Path	Explanation
name	Indicates the `name` property of the wrapped object.
account.name	Indicates the nested property `name` of the `account` property of the wrapped object.
account[2]	Indicates the *third* element of the `account` property of the wrapped object. Indexed properties are typically collections such as `lists` and `dictionaries`, but can be any class that exposes an indexer.

Below you'll find some examples of working with the `IObjectWrapper` to get and set properties. Consider the following two classes:

```
[C#]
public class Company
  {
    private string name;
    private Employee managingDirector;

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

    public Employee ManagingDirector
    {
      get { return this.managingDirector; }
      set { this.managingDirector = value; }
    }
}
```

```
[C#]
public class Employee
{
    private string name;
    private float salary;

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

    public float Salary
    {
      get { return salary; }
      set { this.salary = value; }
    }
}
```

The following code snippets show some examples of how to retrieve and manipulate some of the properties of `IObjectWrapper`-wrapped `Company` and `Employee` instances.

```
[C#]
Company c = new Company();
IObjectWrapper owComp = new ObjectWrapper(c);
// setting the company name...
owComp.SetPropertyValue("name", "Salina Inc.");
// can also be done like this...
PropertyValue v = new PropertyValue("name", "Salina Inc.");
owComp.SetPropertyValue(v);

// ok, let's create the director and bind it to the company...
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("name", "Don Fabrizio");
```

```
owComp.SetPropertyValue("managingDirector", don);

// retrieving the salary of the ManagingDirector through the company
float salary = (float)owComp.GetPropertyValue("managingDirector.salary");
```

Note that since the various Spring.NET libraries are compliant with the Common Language Specification (CLS), the resolution of arbitrary strings to properties, events, classes and such is performed in a case-insensitive fashion. The previous examples were all written in the C# language, which is a case-sensitive language, and yet the `Name` property of the `Employee` class was set using the all-lowercase `'name'` string identifier. The following example (using the classes defined previously) should serve to illustrate this...

```
[C#]
// ok, let's create the director and bind it to the company...
Employee don = new Employee();
IObjectWrapper owDon = new ObjectWrapper(don);
owDon.SetPropertyValue("naMe", "Don Fabrizio");
owDon.GetPropertyValue("nAmE"); // gets "Don Fabrizio"

IObjectWrapper owComp = new ObjectWrapper(new Company());
owComp.SetPropertyValue("ManaGINGdirecToR", don);
owComp.SetPropertyValue("mANaGiNgdirector.salARY", 80000);
Console.WriteLine(don.Salary); // puts 80000
```

The case-insensitivity of the various Spring.NET libraries (dictated by the CLS) is not usually an issue... if you happen to have a class that has a number of properties, events, or methods that differ only by their case, then you might want to consider refactoring your code, since this is generally regarded as poor programming practice.

4.2.2. Other features worth mentioning

In addition to the features described in the preceding sections there a number of features that might be interesting to you, though not worth an entire section.

- *determining readability and writability*: using the `IsReadable()` and `IsWritable()` methods, you can determine whether or not a property is readable or writable.
- *retrieving PropertyInfo instances*: using `GetPropertyInfo(string)` and `GetPropertyInfos()` you can retrieve instances of the `System.Reflection.PropertyInfo` class, that might come in handy sometimes when you need access to the property metadata specific to the object being wrapped.

4.3. Type conversion

If you associate a `TypeConverter` with the definition of a custom `Type` using the standard .NET mechanism (see the example code below), Spring.NET will use the associated `TypeConverter` to do the conversion.

```
[C#]
[TypeConverter (typeof (FooTypeConverter))]
public class Foo
{
}
```

The `TypeConverter` class from the `System.ComponentModel` namespace of the .NET BCL is used extensively by the various classes in the `Spring.Core` library, as said class “... provides a unified way of converting types of values to other types, as well as for accessing standard values and subproperties.” [4]

[4] More information about creating custom `TypeConverter` implementations can be found online at Microsoft's MSDN website, by searching for *Implementing a Type Converter*.

For example, a date can be represented in a human readable format (such as `30th August 1984`), while we're still able to convert the human readable form to the original date format or (even better) to an instance of the `System.DateTime` class. This behavior can be achieved by using the standard .NET idiom of decorating a class with the `TypeConverterAttribute`. Spring.NET also offers another means of associating a `TypeConverters` with a class. You might want to do this to achieve a conversion that is not possible using standard idiom... for example, the `Spring.Core` library contains a custom `TypeConverter` that converts comma-delimited strings to String array instances. Registering custom converters on an `IObjectWrapper` instance gives the wrapper the knowledge of how to convert properties to the desired `Type`.

An example of where property conversion is used in Spring.NET is the setting of properties on objects, accomplished using the aforementioned `TypeConverters`. When mentioning `System.String` as the value of a property of some object (declared in an XML file for instance), Spring.NET will (if the type of the associated property is `System.Type`) use the `RuntimeTypeConverter` class to try to resolve the property value to a `Type` object. The example below demonstrates this automatic conversion of the `Example.Xml.SAXParser` (a string) into the corresponding `Type` instance for use in this factory-style class.

```
<objects>
<object id="parserFactory" type="Example.XmlParserFactory, ExamplesLibrary"
destroy-method="Close">
  <property name="ParserClass" value="Example.Xml.SAXParser, ExamplesLibrary"/>
</object>
</objects>
```

```
[C#]
public class XmlParserFactory
{
        private Type parserClass;

        public Type ParserClass
        {
          get { return this.parserClass; }
          set { this.parserClass = value; }
        }

        public XmlParser GetParser ()
        {
          return Activator.CreateInstance (ParserClass);
        }
}
```

4.3.1. Type Conversion for Enumerations

The default type converter for enumerations is the `System.ComponentModel.EnumConverter` class. To specify the value for an enumerated property, simply use the name of the property. For example the `TestObject` class has a property of the enumerated type `FileMode`. One of the values for this enumeration is named `Create`. The following XML fragment shows how to configure this property

```
<object id="rod" type="Spring.Objects.TestObject, Spring.Core.Tests">
  <property name="name" value="Rod"/>
  <property name="FileMode" value="Create"/>
</object>
```

4.4. Built-in TypeConverters

Spring.NET has a number of built-in `TypeConverters` to make life easy. Each of those is listed below and they are all located in the `Spring.Objects.TypeConverters` namespace of the `Spring.Core` library.

Table 4.2. Built-in `TypeConverters`

Type	**Explanation**
`RuntimeTypeConverter`	Parses strings representing `System.Types` to actual `System.Types` and the other way around.
`FileInfoConverter`	Capable of resolving strings to a `System.IO.FileInfo` object.
`StringArrayConverter`	Capable of resolving a comma-delimited list of strings to a string-array and vice versa.
`UriConverter`	Capable of resolving a string representation of a URI to an actual `URI`-object.
`FileInfoConverter`	Capable of resolving a string representation of a FileInfo to an actual `FileInfo`-object.
`StreamConverter`	Capable of resolving Spring IResource URI (string) to its corresponding `InputStream`-object.
`ResourceConverter`	Capable of resolving Spring IResource URI (string) to an `IResource` object.
`ResourceManagerConverter`	Capable of resolving a two part string (resource name, assembly name) to a `System.Resources.ResourceManager` object.
`RGBColorConverter`	Capable of resolving a comma separated list of Red, Green, Blue integer values to a `System.Drawing.Color` structure.

Spring.NET uses the standard .NET mechanisms for the resolution of `System.Types`, including, but not limited to checking any configuration files associated with your application, checking the Global Assembly Cache (GAC), and assembly probing.

Chapter 5. Threading and Concurrency Support

5.1. Introduction

(Available in 1.1)
The Spring.Threading library is a new addition so peer review and feedback are extremely appreciated. Please check the Spring.NET website for the latest updates to this document.

The purpose of the `Spring.Threading` namespace is to provide a place to keep useful concurrency abstractions that augment those in the BCL. Since Doug Lea has provided a wealth of mature public domain concurrency abstractions in his Java based 'EDU.oswego.cs.dl.util.concurrent' libraries we decided to port a few of his abstractions to .NET. So far, we've only ported three classes, the minimum necessary to provide basic object pooling functionality to support an AOP based pooling aspect and to provide a Semaphore class that was mistakenly not included in .NET 1.0/1.1.

As a side note, the classes present in the 'java.util.concurrent' package of Java 5 were based on the aforementioned libraries and underwent additional scrutiny and peer review. The Java 5 libraries and the new additions to the threading namespace in .NET 2.0 are potential sources for modifications to Spring's threading support.

5.2. Synchronization Primitives

When you take a look at these synchronization classes, you'll wonder why it's even necessary when `System.Threading` provides plenty of synchronization options. Although `System.Threading` provides great synchronization classes, it doesn't provide well-factored abstractions and interfaces for us. Without these abstractions, we will tend to code at a low-level. With enough experience, you'll eventually come up with some abstractions that work well. Doug Lea has already done a lot of that research and has a class library that we can take advantage of.

5.2.1. ISync

`ISync` is the central interface for all classes that control access to resources from multiple threads. It's a simple interface which has two basic use cases. The first case is to block indefinitely until a condition is met:

```
void ConcurrentRun(ISync lock) {
  lock.Acquire(); // block until condition met
  try {
    // ... access shared resources
  }
  finally {
    lock.Release();
  }
}
```

The other case is to specify a maximum amount of time to block before the condition is met:

```
void ImpatientConcurrentRun(ISync lock) {
  // block for at most 10 milliseconds for condition
  if ( lock.Attempt(10) ) {
    try {
      // ... access shared resources
    }
    finally {
      lock.Release();
    }
  } else {
    // complain of time out
  }
}
```

5.2.2. SyncHolder

The `SyncHolder` class implements the `System.IDisposable` interface and so provides a way to use an `ISync` with the `using` C# keyword: the `ISync` will be automatically `Acquired` and then `Released` on exiting from the block.

This should simplify the programming model for code using (!) an `ISync`:

```

ISync sync = ...
...
using (new SyncHolder(sync))
  {
    // ... code to be executed
    // holding the ISync lock
  }
```

There is surely the timed version, a little more cumbersome as you must deal with timeouts:

```

ISync sync = ...
long msecs = 100;
...
// try to acquire the ISync for msecs milliseconds
try
{
  using (new SyncHolder(sync, msecs))
  {
    // ... code to be executed
    // holding the ISync lock
  }
}
catch (TimeoutException)
{
  // deal with failed lock acquisition
}
```

5.2.3. Latch

The `Latch` class implements the `ISync` interface and provides an implementation of a *latch*. A latch is a boolean condition that is set at most once, ever. Once a single release is issued, all acquires will pass. It is similar to a `ManualResetEvent` initialized unsignalled (Reset) and can only be `Set()`. A typical use is to act as a start signal for a group of worker threads.

```
class Boss {
  Latch _startPermit;

  void Worker() {
    // very slow worker initialization ...
    // ... attach to messaging system
    // ... connect to database
    _startPermit.Acquire();
    // ... use resources initialized in Mush
    // ... do real work
  }

  void Mush() {
    _startPermit = new Latch();
    for (int i=0; i<10; ++i) {
      new Thread(new ThreadStart(Worker)).Start();
    }
    // very slow main initialization ...
    // ... parse configuration
    // ... initialize other resources used by workers
    _startPermit.Release();
  }

}
```

5.2.4. Semaphore

The `Semaphore` class implements the `ISync` interface and provides an implementation of a semaphore. Conceptually, a semaphore maintains a set of permits. Each `Acquire()` blocks if necessary until a permit is available, and then takes it. Each `Release()` adds a permit. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. A typical use is to control access to a pool of shared objects.

```
class LimitedConcurrentUploader {
  // ensure we don't exceed maxUpload simultaneous uploads
  Semaphore _available;
  public LimitedConcurrentUploader(maxUploads) {
    _available = new Semaphore(maxUploads);
  }
  // no matter how many threads call this method no more
  // than maxUploads concurrent uploads will occur.
  public Upload(IDataTransfer upload) {
    _available.Acquire();
    try {
      upload.TransferData();
    }
    finally {
      _available.Release();
    }
  }
}
```

Chapter 6. Object Pooling

6.1. Introduction

(Available in 1.0)

The Spring.Pool namespace contains a generic API for implementing pools of objects. Object pooling is a well known technique to minimize the creation of objects that can take a significant amount of time. Common examples are to create a pool of database connections such that each request to the database can reuse an existing connection instead of creating one per client request. Threads are also another common candidate for pooling in order to increase responsiveness of an application to multiple concurrent client requests.

.NET contains support for object pooling in these common scenarios. Support for database connection pools is direclty supported by ADO.NET data providers as a configuration option. Similarly, thread pooling is supported via the System.ThreadPool class. Support for pooling of other objects can be done using the CLR managed API to COM+ found in the System.EnterpriseServices namespace.

Despite this built-in support there are scenarios where you would like to use alternative pool implementations. This maybe because the default implementations, such as System.ThreadPool, do not meet your requirements. (For a discussion on advanced ThreadPool usage see Smart Thread Pool by Ami Bar.) Alternatively, you may want to pool classes that do not inherit from `System.EnterpriseServices.ServicedComponent`. Instead of making changes to the object model to meet this inheritance requirement, Spring .NET provides similar support for pooling, but for any object, by using using AOP proxies and a generic pool API for managing object instances.

Note, that if you are only concerned only with applying pooling to an existing object, the pooling APIs discussed here are not very important. Instead the use and configuration of `Spring.Aop.Target.SimplePoolTargetSource` is more relevant. Pooling of objects can either be done programmatically or through the XML configuration of the Spring .NET container. Attribute support for pooling, similar to the ServicedComponent approach, will be available in a future relase of .NET.

Chapter 16, *Quickstarts* contains an example that shows the use of the pooling API independent of AOP functionality.

6.2. Interfaces and Implementations

The `Spring.Pool` namespace provides two simple interfaces to manage pools of objects. The first interface, `IObjectPool` describes how to take and put back an object from the pool. The second interface `IPoolableObjectFactory` is meant to be used in conjunction with implementations of the `IObjectPool` to provide guidance in calling call various lifecycle events on the objects managed by the pool. These interfaces are based off the Jakarta Commons Pool API. `Spring.Pool.Support.SimplePool` is a default implementation of `IObjectPool` and `Spring.Aop.Target.SimplePoolTargetSource` is the implementation of `IPoolableObjectFactory` for use with AOP. The current goal of the Spring.Pool namespace is not to provide a one-for-one replacement of the Jakarta Commons Pool API, but rather to support basic object pooling needs for in common AOP scenarios. Consequently, other interfaces and base classes available in the Jakarta package are not available.

Chapter 7. Expression Evaluation

7.1. Introduction

The Spring.Expressions namespace provides a powerful expression language for manipulating an object at runtime. The language supports setting and getting of property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithemtic operators, named variables, and retrieval of objects by name from Spring's IoC container.

The functionality provided in this namespace serves as the foundation for a variety of other features in Spring.NET such as enhanced property evaluation in the XML based configuration of the IoC container, a Data Validation framework, and a Data Binding framework for ASP.NET. You will likely find other cool uses for this library in your own work where run-time evaluation of criteria based on an object's state is requried. For those with a Java background, the Spring.Expressions namespace provides functionality similar to the Java based Object Graph Navigation Language, OGNL

This chapter covers the features of the expression language using an Inventor and Inventor's Society class as the target objects for expression evaluation. The class declarations and the data used to populate them are listed at the end of the chapter in section Section 7.4, “Classes used in the examples”. These classes are blatently taken from the NUnit tests for the Expressions namespace which you can refer to for additional example usage.

7.2. Evaluating Expressions

The central class in Spring.Expressions is `ExpressionEvaluator` whose methods are shown below.

```
public static object GetValue(object root, string expression);

public static object GetValue(object root, string expression, IDictionary variables)

public static void SetValue(object root, string expression, object newValue)

public static void SetValue(object root, string expression, IDictionary variables, object newValue)
```

The first argument is the 'root' object that the expression string (2nd argument) will be evaluated against. The third argument is used to to support variables in the expression and will be discussed later. Simple usage to get the value of an object property is shown below using the `Inventor` class. You can find the class listing in section Section 7.4, “Classes used in the examples”.

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1856, 7, 9), "Serbian");

tesla.PlaceOfBirth.City = "Smiljan";

string evaluatedName = (string) ExpressionEvaluator.GetValue(tesla, "Name");

string evaluatedCity = (string) ExpressionEvaluator.GetValue(tesla, "PlaceOfBirth.City"));
```

The value of 'evaluatedName' is 'Nikola Tesla' and that of 'evaluatedCity' is 'Smiljan'. A period is used to navigate the nested properties of the object. Similarly to set the property of an object, say we want to rewrite history and change Tesla's city of birth, we would simply add the following line

```
ExpressionEvaluator.SetValue(tesla, "PlaceOfBirth.City", "Novi Sad");
```

The other class in the namespace you may have occasion to use is `Expression`. It is useful if you intend to perform many evaluations of an expression against the same object and want to increase the performance of the evaluation. The motivation for this class is that the `ExpressionEvaluator` classes parses the expression on each invocation of `GetValue` or `SetValue`. The `Expression` class on the other hand will cache the parsed expression as well as reflection look-ups for increased performance. The methods of this class are listed below

```
public static IExpression Parse(string expression)

public override object Get(object context, IDictionary variables)

public override void Set(object context, IDictionary variables, object newValue)
```

The retrieval of the Name property in the previous example using the Expression class is shown below

```
IExpression exp = Expression.Parse("Name");

string evaluatedName = (string) exp.GetValue(tesla, null);
```

There are a few exception classes to be aware of when using the `ExpressionEvaluator`. These are `InvalidPropertyException`, when you refer to a property that doesn't exist, `NullValueInNestedPathException`, when a null value is encountered when traversing through the nested property list, and `ArgumentException` and `NotSupportedException` when you pass in values that are in error in some other manner.

The expression language is based on a grammar and uses ANTLR to construct the lexer and parser. Errors relating to bad syntax of the langage will be caught at this level of the language implementation. For those interested in the digging deeper into the implementation, the grammar file is named Expression.g and is located in the src directory of the namespace. As a side note, the release version of the ANTLR DLL included with Spring.NET was signed with the Spring.NET key. Upcoming releases of ANTLR will provide strongly signed assemblies.

7.3. Language Reference

7.3.1. Literal expressions

The types of literal expressions supported are strings, dates, numeric values (int, real, and hex), boolean and null. String are delimited by single quotes. To put a single quote itself in a string use the backslash character. The following listing shows simple usage of literals. Typically they would not be used in isolation like this, but as part of a more complex expression, for example using a literal on one side of a logical comparison operator.

```
string helloWorld = (string) ExpressionEvaluator.GetValue(null, "'Hello World'"); // evals to "Hello World"

string tonyPizza  = (string) ExpressionEvaluator.GetValue(null, "'Tony\\'s Pizza'"); // evals to "Tony's Pizza"

double avogadrosNumber = (double) ExpressionEvaluator.GetValue(null, "6.0221415E+23");

int maxValue = (int)  ExpressionEvaluator.GetValue(null, "0x7FFFFFFF");  // evals to 2147483647

DateTime birthday = (DateTime) ExpressionEvaluator.GetValue(null, "date('1974/08/24')");

DateTime exactBirthday =
  (DateTime) ExpressionEvaluator.GetValue(null, " date('19740824T131030', 'yyyyMMddTHHmmss')");

bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "true");

object nullValue = ExpressionEvaluator.GetValue(null, "null");
```

Note that the extra backslash character in Tony's Pizza is to satisfy C# escape syntax. Numbers support the use of the negative sign, exponential notation, and decimal points. By default real numbers are parsed using `Double.Parse` unless the format character "M" or "F" is supplied, in which case `Decimal.Parse` and `Single.Parse` would be used respectfully. As shown above, if two arguments are given to the date literal then

`DateTime.ParseExact` will be used. Note that all parse methods of classes that are used internally reference the `CultureInfo.InvariantCulture`.

7.3.2. Properties, Arrays, Lists, Dictionaries, Indexers

As shown in the previous example in Section 7.2, “Evaluating Expressions”, navigating through properties is easy, just use a period to indicate a nested property value. The instances of `Inventor` class, pupin and tesla, were populated with data listed in section Section 7.4, “Classes used in the examples”. To navigate "down" and get Tesla's year of birth and Pupin's city of birth the following expressions are used

```
int year = (int) ExpressionEvaluator.GetValue(tesla, "DOB.Year"));  // 1856

string city = (string) ExpressionEvaluator.GetValue(pupin, "PlaCeOfBirTh.CiTy");  // "Idvor"
```

For the sharp-eyed, that isn't a typo in the property name for place of birth. The expression uses mixed cases to demonstrate that the evaluation is case insensitive.

The contents of arrays and lists are obtained using square bracket notation.

```
// Inventions Array
string invention = (string) ExpressionEvaluator.GetValue(tesla, "Inventions[3]"); // "Induction motor"

// Members List
string name = (string) ExpressionEvaluator.GetValue(ieee, "Members[0].Name"); // "Nikola Tesla"

// List and Array navigation
string invention = (string) ExpressionEvaluator.GetValue(ieee, "Members[0].Inventions[6]") // "Wireless communic
```

The contents of dictionaries are obtained by specifying the literal key value between single quotes.

```
// Officer's Dictionary
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers['president']";

string city = (string) ExpressionEvaluator.GetValue(ieee, "Officers['president'].PlaceOfBirth.City"); // "Idvor"

ExpressionEvaluator.SetValue(ieee, "Officers['advisors'][0].PlaceOfBirth.Country", "Croatia");
```

You may also specify non literal values in place of the quoted literal values by using another expression inside the square brackets such as variable names or static properties/methods on other types. These features are discussed on other sections.

Indexers are similarly referenced using square brackets. The following is a small example that shows the use of indexers. Multidimensional indexers are also supported.

```
public class Bar
{
    private int[] numbers = new int[] {1, 2, 3};

    public int this[int index]
    {
        get { return numbers[index];}
        set { numbers[index] = value; }
    }
}

Bar b = new Bar();

int val = (int) ExpressionEvaluator.GetValue(bar, "[1]") // evaluated to 2

ExpressionEvaluator.SetValue(bar, "[1]", 3);  // set value to 3
```

7.3.3. Methods

Methods are invoked in typical C# programming syntax. You may also invoke methods on literals.

```
//string literal
char[] chars = (char[]) ExpressionEvaluator.GetValue(null, "'test'.ToCharArray(1, 2)"))  // 't','e'

//date literal
int year = (int) ExpressionEvaluator.GetValue(null, "date('1974/08/24').AddYears(31).Year") // 2005

// object usage, calculate age of tesla navigating from the IEEE society.

ExpressionEvaluator.GetValue(ieee, "Members[0].GetAge(date('2005-01-01')") // 149 (eww..a big anniversary is com
```

7.3.4. Operators

7.3.4.1. Relational operator

The relational operators; equal, not equal, less than, less than or equal, greater than, and greater than or equal are supported using standard operator notation. These operators take into account if the object implements the `IComparable` interface. Enumerations are also supported but you will need to register the enumeration type, as described in Section Section 7.3.8, “Type Registration”, in order to use an enumeration value in an expression if it is not contained in the mscorlib.

```
ExpressionEvaluator.GetValue(null, "2 == 2")  // true

ExpressionEvaluator.GetValue(null, "date('1974-08-24') != DateTime.Today"  // true

ExpressionEvaluator.GetValue(null, "2 < -5.0") // false

ExpressionEvaluator.GetValue(null, "DateTime.Today <= date('1974-08-24')") // false

ExpressionEvaluator.GetValue(null, "'Test' >= 'test'") // true
```

Enumerations can be evaluated as shown below

```
FooColor fColor = new FooColor();

ExpressionEvaluator.SetValue(fColor, "Color", KnownColor.Blue);

bool trueValue = (bool) ExpressionEvaluator.GetValue(fColor, "Color == KnownColor.Blue"); //true
```

Where FooColor is the following class.

```
public class FooColor
{
    private KnownColor knownColor;

    public KnownColor Color
    {
        get { return knownColor;}
        set { knownColor = value; }
    }
}
```

7.3.4.2. Logical operators

The logical operators that are supported are *and*, *or*, and *not*. Their use is demonstrated below

```
// AND
bool falseValue = (bool) ExpressionEvaluator.GetValue(null, "true and false"); //false

string expression = @"IsMember('Nikola Tesla') and IsMember('Mihajlo Pupin')";
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, expression);  //true
```

```
// OR
bool trueValue = (bool) ExpressionEvaluator.GetValue(null, "true or false");  //true

string expression = @"IsMember('Nikola Tesla') or IsMember('Albert Einstien')";
bool trueValue = (bool) ExpressionEvaluator.GetValue(ieee, expression); // true

// NOT
bool falseValue = (bool) ExpressionEvaluator.GetValue(null, "!true");

// AND and NOT
string expression = @"IsMember('Nikola Tesla') and !IsMember('Mihajlo Pupin')";
bool falseValue = (bool) ExpressionEvaluator.GetValue(ieee, expression);
```

7.3.4.3. Mathmatical operators

The addition operator can be used on numbers, string, dates. Subtraction can be used on numbers and dates. Multiplication and division can be used only on numbers. Other mathmatical operators supported are modulus (%) and exponential power (^). Standard operator precedence is enforced. These operators are demonstrated below

```
// Addition
int two = (int)ExpressionEvaluator.GetValue(null, "1 + 1"); // 2

String testString = (String)ExpressionEvaluator.GetValue(null, "'test' + ' ' + 'string'"); //'test string'

DateTime dt = (DateTime)ExpressionEvaluator.GetValue(null, "date('1974-08-24') + 5"); // 8/29/1974

// Subtraction

int four = (int) ExpressionEvaluator.GetValue(null, "1 - -3"); //4

Decimal dec = (Decimal) ExpressionEvaluator.GetValue(null, "1000.00m - 1e4"); // 9000.00

TimeSpan ts = (TimeSpan) ExpressionEvaluator.GetValue(null, "date('2004-08-14') - date('1974-08-24')"); //10948.

// Multiplication

int six = (int) ExpressionEvaluator.GetValue(null, "-2 * -3"); // 6

int twentyFour = (int) ExpressionEvaluator.GetValue(null, "2.0 * 3e0 * 4"); // 24

// Division

int minusTwo = (int) ExpressionEvaluator.GetValue(null, "6 / -3"); // -2

int one = (int) ExpressionEvaluator.GetValue(null, "8.0 / 4e0 / 2"); // 1

// Modulus

int three = (int) ExpressionEvaluator.GetValue(null, "7 % 4"); // 3

int one = (int) ExpressionEvaluator.GetValue(null, "8.0 % 5e0 % 2"); // 1

// Exponent

int sixteen = (int) ExpressionEvaluator.GetValue(null, "-2 ^ 4"); // 16

// Operator precedence

int minusFortyFive = (int) ExpressionEvaluator.GetValue(null, "1+2-3*8^2/2/2"); // -45
```

7.3.5. Assignment

Setting of a property is done by using the equals operator. This would typically be done within a call to `GetValue` since in the simple case `SetValue` offers the same functionality. Assignment in this manner is useful when combining multiple operators in an expression list, discussed in the next section. Some examples of assignment are shown below

```
Inventor inventor = new Inventor();
String aleks = (String) ExpressionEvaluator.GetValue(inventor, "Name = 'Aleksandar Seovic'");
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(inventor, "DOB = date('1974-08-24')");

//Set the vice president of the society
Inventor tesla = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers['vp'] = Members[0]");
```

7.3.6. Expression lists

Multiple expressions can be executed at the same time by separating them with a semicolon and enclosing the entire expression within curly braces. The value returned is the value of the last expression in the list. Examples of this are shown below

```
//Perform property assignments and then return name property.

String pupin = (String) ExpressionEvaluator.GetValue(ieee.Members,
  "{ [1].PlaceOfBirth.City = 'Beograd'; [1].PlaceOfBirth.Country = 'Serbia'; [1].Name}"));

// pupin = "Mihajlo Pupin"
```

7.3.7. Types

To represent a `System.Type` in an expression use the following syntax.

```
Type dateType = (Type) ExpressionEvaluator.GetValue(null, "type('System.DateTime')"

Type evalType = (Type) ExpressionEvaluator.GetValue(null, "type('Spring.Expressions.ExpressionEvaluator, Spring.

bool trueValue = (bool) ExpressionEvaluator.GetValue(tesla, "type('System.DateTime') == DOB.GetType()")
```

The implemention delegates to Spring's `ObjectUtils.ResolveType` for the actual type resolution.

7.3.8. Type Registration

To refer to a type within an expression that is not in the mscorlib you need to register it with the `TypeRegistry`. This will allow you to refer to a shorthand name of the type within your expressions. This is commonly used in expression that use the new operator or refer to a static properties of an object. Example usage is shown below.

```
TypeRegistry.RegisterType("Society", typeof(Society));

Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers[Society.President]");
```

7.3.9. Constructors

Constructors can be invoked using the new operator. For classes outside mscorlib you will need to register your types so they can be resolved. Examples of using constructors are shown below

```
// simple ctor
DateTime dt = (DateTime) ExpressionEvaluator.GetValue(null, "new DateTime(1974, 8, 24)");

// Register Inventor type then create new inventor instance within Add method inside an expression list.
// Then return the new count of the Members collection.

TypeRegistry.RegisterType("Inventor", typeof(Inventor));
int three = (int) ExpressionEvaluator.GetValue(ieee.Members, "{ Add(new Inventor('Aleksandar Seovic', date('1974
```

7.3.10. Variables

Variables can referenced in the expression using the syntax *#variableName*. The variables are passed in and out of the expression using the dictionary parameter in `ExpressionEvaluator`'s `GetValue` or `SetValue` methods.

```
public static object GetValue(object root, string expression, IDictionary variables)

public static void SetValue(object root, string expression, IDictionary variables, object newValue)
```

The variable name is the key value of the dictionary. Example usage is shown below;

```
IDictionary vars = new Hashtable();
vars["newName"] = "Mike Tesla";
ExpressionEvaluator.GetValue(tesla, "Name = #newName", vars));
```

You can also use the dictionary as a place to store values of the object as they are evaluated inside the expression. For example to change Tesla's first name back again and keep the old value;

```
ExpressionEvaluator.GetValue(tesla, "{ #oldName = Name; Name = 'Nikola Tesla' }", vars);
String oldName = (String)vars["oldName"]; // Mike Tesla
```

Variable names can also be used inside indexers or maps instead of literal values. For example;

```
vars["prez"] = "president";
Inventor pupin = (Inventor) ExpressionEvaluator.GetValue(ieee, "Officers[#prez]", vars);
```

7.3.10.1. The 'this' variable

7.3.10.2. The 'root' variable

7.3.11. If then else...

You can use the ternary operator for performing if-then-else conditional logic inside the expression. A minimal example is;

```
String aTrueString  = (String) ExpressionEvaluator.GetValue(null, "false ? 'trueExp' : 'falseExp'") // trueExp
```

In this case, the boolean false results in returning the string value 'trueExp'. A less artificial example is shown below

```
ExpressionEvaluator.SetValue(ieee, "Name", "IEEE");
IDictionary vars = new Hashtable();
vars["queryName"] = "Nikola Tesla";

string expression = @"IsMember(#queryName)
                      ? #queryName + ' is a member of the ' + Name + ' Society'
                      : #queryName + ' is not a member of the ' + Name + ' Society'";

String queryResultString = (String) ExpressionEvaluator.GetValue(ieee, expression, vars));

// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

7.3.12. Spring Object References

Expressions can refer to objects that are declared in Spring's application context using the syntax *@contextName*:*objectName*. If no contextName is specified the default root context name (Spring.RootContext) is used. Using the application context defined in the MovieFinder example from Chapter 16, *Quickstarts*, the following expression returns the number of movies directed by Roberto Benigni.

```
public static void Main()
{
  . . .

// Retrieve context defined in the spring/context section of
// the standard .NET configuration file.
IApplicationContext ctx = ContextRegistry.GetContext();

int numMovies = (int) ExpressionEvaluator.GetValue(null,
                      "@MyMovieLister.MoviesDirectedBy('Roberto Benigni').Length");

  . . .
}
```

The variable numMovies is evaluated to 2 in this example.

7.3.13. Null Context

If you do not specify a root object, i.e. pass in null, then the expressions evaluated either have to be literal values, i.e. ExpressionEvaluator.GetValue(null, "2 + 3.14"), refer to classes that have static methods or properties, i.e. ExpressionEvaluator.GetValue(null, "DateTime.Today"), create new instances of objects, i.e. ExpressionEvaluator.GetValue(null, "new DateTime(2004, 8, 14)") or refer to other objects such as those in the variable dictionary or in the IoC container. The latter two usages will be discussed later.

7.4. Classes used in the examples

The following simple classes are used to demonstrate the functionality of the expression language.

```
public class Inventor
{
    public string Name;
    public string Nationality;
    public string[] Inventions;
    private DateTime dob;
    private Place pob;

    public Inventor() : this(null, DateTime.MinValue, null)
    {}

    public Inventor(string name, DateTime dateOfBirth, string nationality)
    {
        this.Name = name;
        this.dob = dateOfBirth;
        this.Nationality = nationality;
        this.pob = new Place();
    }

    public DateTime DOB
    {
        get { return dob; }
        set { dob = value; }
    }

    public Place PlaceOfBirth
    {
        get { return pob; }
    }

    public int GetAge(DateTime on)
    {
        // not very accurate, but it will do the job ;-)
        return on.Year - dob.Year;
    }
}

public class Place
{
    public string City;
    public string Country;
}

public class Society
{
    public string Name;
    public static string Advisors = "advisors";
    public static string President = "president";

    private IList members = new ArrayList();
    private IDictionary officers = new Hashtable();

    public IList Members
    {
        get { return members; }
    }

    public IDictionary Officers
    {
        get { return officers; }
    }

    public bool IsMember(string name)
    {
        bool found = false;
        foreach (Inventor inventor in members)
        {
            if (inventor.Name == name)
            {
                found = true;
                break;
            }
        }
        return found;
    }
}
```

The code listings in this chapter use instances of the data populated with the following information.

```
Inventor tesla = new Inventor("Nikola Tesla", new DateTime(1856, 7, 9), "Serbian");
tesla.Inventions = new string[]
    {
        "Telephone repeater", "Rotating magnetic field principle",
        "Polyphase alternating-current system", "Induction motor",
        "Alternating-current power transmission", "Tesla coil transformer",
        "Wireless communication", "Radio", "Fluorescent lights"
    };
tesla.PlaceOfBirth.City = "Smiljan";

Inventor pupin = new Inventor("Mihajlo Pupin", new DateTime(1854, 10, 9), "Serbian");
pupin.Inventions = new string[] {"Long distance telephony & telegraphy", "Secondary X-Ray radiation", "Sonar"};
pupin.PlaceOfBirth.City = "Idvor";
pupin.PlaceOfBirth.Country = "Serbia";

Society ieee = new Society();
ieee.Members.Add(tesla);
ieee.Members.Add(pupin);
ieee.Officers["president"] = pupin;
ieee.Officers["advisors"] = new Inventor[] {tesla, pupin};
```

Chapter 8. Validation Framework

8.1. Introduction

Spring provides an object validation framework that leverages the functionality provided in the Expressions namespace. (See Chapter 7, *Expression Evaluation* for more information). The validation framework is in Spring.Core assembly under the namespace Spring.Validation. We apologize that reference documentation for this functionality is not yet available. You can currently get some information on this functionality by looking at the SDK documentation and also by running the SpringAir sample application provided in the nightly builds. Please post to the Spring.NET forums with any questions.

Chapter 9. Aspect Oriented Programming with Spring.NET

9.1. Introduction

(Available in 1.0)

Aspect-Oriented Programming (*AOP*) complements OOP by providing another way of thinking about program structure. Whereas OO decomposes applications into a hierarchy of objects, AOP decomposes programs into *aspects* or *concerns*. This enables the modularization of concerns such as transaction management that would otherwise cut across multiple objects (such concerns are often termed *crosscutting* concerns).

One of the key components of Spring.NET is the *AOP framework*. While the Spring.NET IoC container does not depend on AOP, meaning you don't need to use AOP if you don't want to, AOP complements Spring.NET IoC to provide a very capable middleware solution.

AOP is used in Spring.NET:

- To provide declarative enterprise services, especially as a replacement for COM+ declarative services. The most important such service is *declarative transaction management*, which builds on Spring.NET's transaction abstraction. This functionality is planed for an upcoming release of Spring.NET
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Thus you can view Spring.NET AOP as either an enabling technology that allows Spring.NET to provide declarative transaction management without COM+; or use the full power of the Spring.NET AOP framework to implement custom aspects.

For those who would like to hit the ground running and start exploring how to use Spring's AOP functionality, head on over to Chapter 17, *AOP Guide*.

9.1.1. AOP concepts

Let us begin by defining some central AOP concepts. These terms are not Spring.NET-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring.NET used its own terminology.

- *Aspect*: A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in enterprise applications. Aspects are implemented using Spring.NET as Advisors or interceptors.
- *Joinpoint*: Point during the execution of a program, such as a method invocation or a particular exception being thrown.
- *Advice*: Action taken by the AOP framework at a particular joinpoint. Different types of advice include "around," "before" and "throws" advice. Advice types are discussed below. Many AOP frameworks, including Spring.NET, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.
- *Pointcut*: A set of joinpoints specifying when an advice should fire. An AOP framework must allow

developers to specify pointcuts: for example, using regular expressions.

- *Introduction*: Adding methods or fields to an advised class. Spring.NET allows you to introduce new interfaces to any advised object. For example, you could use an introduction to make any object implement an `IAuditable` interface, to simplify the tracking of changes to an object's state.
- *Target object*: Object containing the joinpoint. Also referred to as *advised* or *proxied* object.
- *AOP proxy*: Object created by the AOP framework, including advice. In Spring.NET, an AOP proxy is a dynamic proxy that uses IL code generated at runtime.
- *Weaving*: Assembling aspects to create an advised object. This can be done at compile time (using the Gripper-Loom.NET compiler, for example), or at runtime. Spring.NET performs weaving at runtime.

Different advice types include:

- *Around advice*: Advice that surrounds a joinpoint such as a method invocation. This is the most powerful kind of advice. Around advice will perform custom behaviour before and after the method invocation. They are responsible for choosing whether to proceed to the joinpoint or to shortcut executing by returning their own return value or throwing an exception.
- *Before advice*: Advice that executes before a joinpoint, but which does not have the ability to prevent execution flow proceeding to the joinpoint (unless it throws an exception).
- *Throws advice*: Advice to be executed if a method throws an exception. Spring.NET provides strongly typed throws advice, so you can write code that catches the exception (and subclasses) you're interested in, without needing to cast from Exception.
- *After returning advice*: Advice to be executed after a joinpoint completes normally: for example, if a method returns without throwing an exception.

Spring.NET provides a full range of advice types. We recommend that you use the least powerful advice type that can implement the required behaviour. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you don't need to invoke the `proceed()` method on the `IMethodInvocation` used for around advice, and hence can't fail to invoke it.

The pointcut concept is the key to AOP, distinguishing AOP from older technologies offering interception. Pointcuts enable advice to be targeted independently of the OO hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects. Thus pointcuts provide the structural element of AOP.

9.1.2. Spring.NET AOP capabilities

Spring.NET AOP is implemented in pure C#. There is no need for a special compilation process - all weaving is done at runtime. Spring.NET AOP does not need to control or modify the way in which assemblies are loaded, nor does it rely on unmanaged APIs, and is thus suitable for use in any CLR environment.

Spring.NET currently supports interception of method invocations. Field interception is not implemented, although support for field interception could be added without breaking the core Spring.NET AOP APIs.
Field interception arguably violates OO encapsulation. We don't believe it is wise in application development.

Spring.NET provides classes to represent pointcuts and different advice types. Spring.NET uses the term *advisor* for an object representing an aspect, including both an advice and a pointcut targeting it to specific joinpoints.

Different advice types are `IMethodInterceptor` (from the AOP Alliance interception API); and the advice interfaces defined in the `Spring.Aop` namespace. All advices must implement the `AopAlliance.Aop.IAdvice` tag interface. Advices supported out the box are `IMethodInterceptor` ; `IThrowsAdvice`; `IBeforeAdvice`; and `IAfterReturningAdvice`. We'll discuss advice types in detail below.

Spring.NET provides a .NET translation of the Java interfaces defined by the *AOP Alliance*. Around advice must implement the AOP Alliance `AopAlliance.Interceptr.IMethodInterceptor` interface. Whilst there is wide support for the AOP Alliance in Java, Spring.NET is currently the only .NET AOP framework that makes use of these interfaces. In the short term, this will provide a consistent programming model for those doing development in both .NET and Java, and in the longer term, we hope to see more .NET projects adopt the AOP Alliance interfaces.
The aim of Spring.NET AOP support is not to provide a comprehensive AOP implementation on par with the functionality available in AspectJ. However, Spring.NET AOP provides an excellent solution to most problems in .NET applications that are amenable to AOP.
Thus, it is common to see Spring.NET's AOP functionality used in conjunction with a Spring.NET IoC container. AOP advice is specified using normal object definition syntax (although this allows powerful "autoproxying" capabilities); advice and pointcuts are themselves managed by Spring.NET IoC.

9.1.3. AOP Proxies in Spring.NET

Spring.NET generates AOP proxies at runtime using classes from the System.Reflection.Emit namespace to create necessary IL code for the proxy class.This results in the proxies that are very efficient and do not impose any restrictions on the inheritance hierarchy.

Another common approach to AOP proxy implementation in .NET is to use ContextBoundObject and the .NET remoting infrastructure as an interception mechanism. We are not very fond of ContextBoundObject approach because it requires classes that need to be proxied to inherit from the ContextBoundObject either directly or indirectly. In our opinion this an unnecessary restriction that influences how you should design your object model and also excludes applying AOP to "3rd party" classes that are not under your direct control. Context-bound proxies are also an order of magnitude slower than IL-generated proxies, due to the overhead of the context switching and .NET remoting infrastructure.

Spring.NET AOP proxies are also "smart" - in that because proxy configuration is known during proxy generation, the generated proxy can be optimized to invoke target method via reflection only when necessary (i.e. when there are advices applied to the target method). In all other cases target method will be called directly, thus avoiding performance hit caused by the reflective invocation.

Finally, Spring.NET AOP proxies will never return a raw reference to a target object. Whenever target method returns raw reference to a target object (i.e. "return this;"), AOP proxy will recognize what happened and will replace the return value with a reference to itself instead.

The current implementation of the AOP proxy generator uses object composition to delegate calls from the proxy to a target object, similar to how you would implement a classic Decorator pattern. This means that classes that need to be proxied have to implement one or more interfaces, which is in our opinion not only a less-intruding requirement than ContextBoundObject inheritance requirements, but also a good practice that should be followed anyway for the service classes that are most common targets for AOP proxies.

In the future release we will implement proxies using inheritance, which will allow you to proxy classes without interfaces as well and will remove some of the remaining raw reference issues that cannot be solved using composition-based proxies.

9.2. Pointcuts in Spring.NET

Let's look at how Spring.NET handles the crucial pointcut concept.

9.2.1. Concepts

Spring.NET's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The `Spring.Aop.IPointcut` interface is the central interface, used to target advices to particular types and methods. The complete interface is shown below:

```
public interface IPointcut
{
    ITypeFilter TypeFilter { get; }

    IMethodMatcher MethodMatcher { get; }
}
```

Splitting the `IPointcut` interface into two parts allows reuse of type and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ITypeFilter` interface is used to restrict the pointcut to a given set of target classes. If the `Matches()` method always returns true, all target types will be matched:

```
public interface ITypeFilter
{
    bool Matches(Type type);
}
```

The `IMethodMatcher` interface is normally more important. The complete interface is shown below:

```
public interface IMethodMatcher
{
    bool IsRuntime { get; }

    bool Matches(MethodInfo method, Type targetType);

    bool Matches(MethodInfo method, Type targetType, object[] args);
}
```

The `Matches(MethodInfo, Type)` method is used to test whether this pointcut will ever match a given method on a target type. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the `IsRuntime` property for the `IMethodMatcher` returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most `IMethodMatchers` are static, meaning that their `IsRuntime` property returns false. In this case, the 3-argument `Matches` method will never be invoked.
Whenever possible, try to make pointcuts static... this allows the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

9.2.2. Operations on pointcuts

Spring.NET supports operations on pointcuts: notably, *union* and *intersection*.

Union means the methods that either pointcut matches.

Intersection means the methods that both pointcuts match.

Union is usually more useful.

Pointcuts can be composed using the static methods in the *Spring.Aop.Support.Pointcuts* class, or using the *ComposablePointcut* class in the same namespace.

9.2.3. Convenience pointcut implementations

Spring.NET provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

9.2.3.1. Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient--and best--for most usages. It's possible for Spring.NET to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.NET.

9.2.3.1.1. Regular expression pointcuts

One obvious way to specify static pointcuts is using regular expressions. Several AOP frameworks besides Spring.NET make this possible. The `Spring.Aop.Support.SdkRegularExpressionMethodPointcut` class is a generic regular expression pointcut, that uses the regular expression classes from the .NET BCL.

Using this class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true (so the result is effectively the union of these pointcuts.)

The usage is shown below:

```
<object id="settersAndAbsquatulatePointcut"
    type="Spring.Aop.Support.SdkRegularExpressionMethodPointcut, Spring.Aop">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</object>
```

As a convenience, Spring provides the `RegularExpressionMethodPointcutAdvisor` class that allows us to reference an `IAdvice` instance as well as defining the pointcut rules (remember that an `IAdvice` instance can be an interceptor, before advice, throws advice etc.) This simplifies wiring, as the one object serves as both pointcut and advisor, as shown below:

```
<object id="settersAndAbsquatulateAdvisor"
    type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor, Spring.Aop">
    <property name="advice">
        <ref local="objectNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</object>
```

The `RegularExpressionMethodPointcutAdvisor` class can be used with any `Advice` type.

9.2.3.1.2. Attribute pointcuts

Pointcuts can be specified by matching an attribute type that is associated with a method. Advice associated with this pointcut can then read the metadata assocatied with the attribute to configure itself. The class `AttributeMatchMethodPointcut` provides this functionality. Sample usage that will match all methods that have the attribute `Spring.Attributes.CacheAttribute` is shown below.

```
<object id="cachePointcut" type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop">
    <property name="Attribute" value="Spring.Attributes.CacheAttribute, Spring.Core"/>
</object>
```

This can be used with a `DefaultPointcutAdvisor` as shown below

```
<object id="cacheAspect" type="Spring.Aop.Support.DefaultPointcutAdvisor, Spring.Aop">
  <property name="Pointcut">
      <object type="Spring.Aop.Support.AttributeMatchMethodPointcut, Spring.Aop">
          <property name="Attribute" value="Spring.Attributes.CacheAttribute, Spring.Core"/>
      </object>
  </property>
  <property name="Advice" ref="aspNetCacheAdvice"/>
</object>
```

where aspNetCacheAdvice is an implementation of an `IMethodInterceptor` that caches method return values. See the SDK docs for `Spring.Aop.Advice.CacheAdvice` for more information on this particular advice.

As a convenience the class `AttributeMatchMethodPointcutAdvisor` is provided to defining an attribute based Advisor as a somewhat shorter alternative to using the generic DefaultPointcutAdvisor. An example is shown below.

```
<object id="AspNetCacheAdvice" type="Spring.Aop.Support.AttributeMatchMethodPointcutAdvisor, Spring.Aop">
   <property name="advice">
      <object type="Aspect.AspNetCacheAdvice, Aspect"/>
   </property>
   <property name="attribute" value="Framework.AspNetCacheAttribute, Framework" />
</object>
```

9.2.3.2. Dynamic Pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

9.2.3.2.1. Control Flow Pointcuts

Spring.NET control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below another pointcut.). A control flow pointcut is dynamic because it is evaluated against the current call stack for each method invocation. For example, if method ClassA.A() calls ClassB.B() then the execution of ClassB.B() has occured in ClassA.A()'s control flow. A control flow pointcut allows advice to be applied to the method ClassA.A() but only when called from ClassB.B() and not when ClassA.A() is executed from another call stack. Control flow pointcuts are specified using the `Spring.Aop.Support.ControlFlowPointcut` class.

Note

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts.

When using control flow point cuts some attention should be paid to the fact that at runtime the JIT compiler can inline the methods, typically for increased performance, but with the consequence that the method no longer appears in the current call stack. This is because inlining takes the callee's IL code and inserts it into the caller's IL code effectively removing the method call. The information returned from `System.Diagnostics.StackTrace`, used in the implementation of `ControlFlowPointcut` is subject to these optimizations and therefore a control flow pointcut will not match if the the method has been inlined.

Generally speaking, a method will be a candidate for inlining when its code is 'small', just a few lines of code (less than 32 bytes of IL). For some interesting reading on this process read David Notario's blog entries (JIT Optimizations I and JIT Optimizations II). Additionally, when an assembly is compiled with a Release configuration the assembly metadata instructs the CLR to enable JIT optimizations. When compiled with a Debug configuration the CLR will disable (some?) these optimizations. Emperically, method inlining is turned off in a Debug configuration.

The way to ensure that your control flow pointcut will not be overlooked because of method inlining is to apply the `System.Runtime.CompilerServices.MethodImplAttribute` attribute with the value `MethodImplOptions.NoInlining`. In this (somewhat artifical) simple example, if the code is compiled in release mode it will not match a control flow pointcut for the method "GetAge".

```
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

However, applying the attributes as shown below will prevent the method from being inlined even in a release build.

```
[MethodImpl(MethodImplOptions.NoInlining)]
public int GetAge(IPerson person)
{
    return person.GetAge();
}
```

9.2.4. Custom pointcuts

Because pointcuts in Spring.NET are .NET types, rather than language features (as in AspectJ) it is possible to declare custom pointcuts, whether static or dynamic. However, there is no support out of the box for the sophisticated pointcut expressions that can be coded in the AspectJ syntax. However, custom pointcuts in Spring.NET can be as arbitrarily complex as any object model.

Spring.NET provides useful pointcut superclasses to help you implement your own pointcuts.

Because static pointcuts are the most common and generally useful pointcut type, you'll probably subclass `StaticMethodMatcherPointcut`, as shown below. This requires you to implement just one abstract method (although it is possible to override other methods to customize behaviour):

```
public class TestStaticPointcut : StaticMethodMatcherPointcut {

    public override bool Matches(MethodInfo method, Type targetType) {
        // return true if custom criteria match
    }
}
```

9.3. Advice types in Spring.NET

Let's now look at how Spring.NET AOP handles advice.

9.3.1. Advice Lifecycle

Spring.NET advices can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

9.3.2. Advice types

Spring.NET provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

9.3.2.1. Interception Around Advice

The most fundamental advice type in Spring.NET is *interception around advice*.

Spring.NET is compliant with the AOP Alliance interface for around advice using method interception. Around advice is implemented using the following interface:

```
public interface IMethodInterceptor : IInterceptor
{
    object Invoke(IMethodInvocation invocation);
}
```

The `IMethodInvocation` argument to the `Invoke()` method exposes the method being invoked; the target joinpoint; the AOP proxy; and the arguments to the method. The `Invoke()` method should return the invocation's result: the return value of the joinpoint.

A simple `IMethodInterceptor` implementation looks as follows:

```
public class DebugInterceptor : IMethodInterceptor {

    public object Invoke(IMethodInvocation invocation) {
        Console.WriteLine("Before: invocation=[{0}]", invocation);
        object rval = invocation.Proceed();
        Console.WriteLine("Invocation returned");
        return rval;
    }
}
```

Note the call to the IMethodInvocation's `Proceed()` method. This proceeds down the interceptor chain towards the joinpoint. Most interceptors will invoke this method, and return its return value. However, an IMethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the `Proceed()` method. However, you don't want to do this without good reason!

9.3.2.2. Before advice

A simpler advice type is a **before advice**. This does not need an `IMethodInvocation` object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the `Proceed()`method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The `IMethodBeforeAdvice` interface is shown below.

```
public interface IMethodBeforeAdvice : IBeforeAdvice
{
    void Before(MethodInfo method, object[] args, object target);
}
```

Note the the return type is `void`. Before advice can insert custom behaviour before the joinpoint executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring.NET, which counts all methods that return normally:

```
public class CountingBeforeAdvice : IMethodBeforeAdvice {

    private int count;

    public void Before(MethodInfo method, object[] args, object target) {
        ++count;
    }

    public int Count  {
        get { return count; }
    }
}
```

Before advice can be used with any pointcut.

9.3.2.3. Throws advice

Throws advice is invoked after the return of the joinpoint if the joinpoint threw an exception. The `Spring.Aop.IThrowsAdvice` interface does not contain any methods: it is a tag interface identifying that the implementing advice object implements one or more typed throws advice methods. These throws advice methods must be of the form:

```
AfterThrowing([MethodInfo method, Object[] args, Object target], Exception subclass)
```

Throws-advice methods must be named `'AfterThrowing'`. The return value will be ignored by the Spring.NET AOP framework, so it is typically `void`. With regard to the method arguments, only the last argument is required. Thus there are *exactly* one *or* four arguments, depending on whether the advice method is interested in the method, method arguments and the target object.

The following method snippets show examples of throws advice.

This advice will be invoked if a `RemotingException` is thrown (including subclasses):

```
public class RemoteThrowsAdvice : IThrowsAdvice {

    public void AfterThrowing(RemotingException ex) {
        // Do something with remoting exception
    }
}
```

The following advice is invoked if a `SqlException` is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

```
public class SqlExceptionThrowsAdviceWithArguments : IThrowsAdvice {

    public void AfterThrowing(MethodInfo method, object[] args, object target, SqlException ex) {
        // Do something will all arguments
    }
}
```

The final example illustrates how these two methods could be used in a single class, which handles both `RemotingException` and `SqlException`. Any number of throws advice methods can be combined in a single class, as can be seen in the following example.

```
public class CombinedThrowsAdvice : IThrowsAdvice {

    public void AfterThrowing(RemotingException ex)  {
        // Do something with remoting exception
    }

    public void AfterThrowing(MethodInfo method, object[] args, object target, SqlException ex) {
        // Do something will all arguments
    }
}
```

Finally, it is worth stating that throws advice is only applied to the actual exception being thrown. What does this mean? Well, it means that if you have defined some throws advice that handles `RemotingExceptions`, the applicable `AfterThrowing` method will **only** be invoked if the type of the thrown exception is `RemotingException`... if a `RemotingException` has been thrown and subsequently wrapped inside another exception before the exception bubbles up to the throws advice interceptor, then the throws advice that handles `RemotingExceptions` will **never** be called. Consider a business method that is advised by throws advice that handles `RemotingExceptions`; if during the course of a method invocation said business method throws a RemoteException... and subsequently wraps said `RemotingException` inside a business-specific `BadConnectionException` (see the code snippet below) before throwing the exception, then the throws advice will never be able to respond to the `RemotingException`... because all the throws advice sees is a `BadConnectionException`. The fact that the `RemotingException` is wrapped up inside the `BadConnectionException` is immaterial.

```
public void BusinessMethod()
    {
        try
        {
            // do some business operation...
        }
        catch (RemotingException ex)
        {
            throw new BadConnectionException("Couldn't connect.", ex);
        }
    }
```

Note

Please note that throws advice can be used with any pointcut.

9.3.2.4. After Returning advice

An after returning advice in Spring.NET must implement the `Spring.Aop.IAfterReturningAdvice` interface, shown below:

```
public interface IAfterReturningAdvice : IAdvice
{
  void AfterReturning(object returnValue, MethodBase method, object[] args, object target);
}
```

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

```
public class CountingAfterReturningAdvice : IAfterReturningAdvice {
    private int count;

    public void AfterReturning(object returnValue, MethodBase m, object[] args, object target) {
        ++count;
    }

    public int Count  {
        get { return count; }
    }
}
```

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.

Note

Please note that after-returning advice can be used with any pointcut.

9.3.2.5. Introduction advice

Spring.NET allows you to add new methods and properties to an advised class. This would typically be done when the functionality you wish to add is a crosscutting concern and want to introduce this functionality as a change to the static structure of the class hierarchy. For example, you may want to cast objects to the introduction interface in your code. Introductions are also a means to emulate multiple inheritance.

Introduction advice is defined by using a normal interface declaration that implements the tag interface `IAdvice`.

Note

The need for implementing this marker interface will likely be removed in future versions.

As an example, consider the interface `IAuditable` that describes the last modified time of an object.

```
public interface IAuditable : IAdvice
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

where

```
public interface IAdvice
{
}
```

Access to the advised object can be obtained by implementing the interface `ITargetAware`

```
public interface ITargetAware
{
  IAopProxy TargetProxy
  {
    set;
  }
}
```

with the `IAopProxy` reference providing a layer of indirection through which the advised object can be accessed.

```
public interface IAopProxy
{
  object GetProxy();
}
```

A simple class that demonstrates this functionality is shown below.

```
public interface IAuditable : IAdvice, ITargetAware
{
    DateTime LastModifiedDate
    {
        get;
        set;
    }
}
```

A class that implements this interface is shown below.

```
public class AuditableMixin : IAuditable
{
    private DateTime date;
    private IAopProxy targetProxy;

    public AuditableMixin()
    {
      date = new DateTime();
    }

    public DateTime LastModifiedDate
    {
       get { return date; }
       set { date = value; }
    }

    public IAopProxy TargetProxy
    {
       set { targetProxy = value; }
    }
}
```

Introduction advice is not associated with a pointcut, since it applies at the class and not the method level. As such, introductions use their own subclass of the interface `IAdvisor`, namely `IIntroductionAdvisor`, to specify the types that the introduction can be applied to.

```
public interface IIntroductionAdvisor : IAdvisor
{
    ITypeFilter TypeFilter { get; }

    Type[] Interfaces { get; }

    void ValidateInterfaces();
}
```

The `TypeFilter` property returns the filter that determines which target classes this introduction should apply to.

The `Interfaces` property returns the interfaces introduced by this advisor.

The `ValidateInterfaces()` method is used internally to see if the introduced interfaces can be implemented by the introduction advice.

Spring.NET provides a default implementation of this interface (the `DefaultIntroductionAdvisor` class) that should be sufficient for the majority of situations when you need to use introductions. The most simple implementation of an introduction advisor is a subclass that simply passes a a new instance the the base constructor. Passing a new instance is important since we want a new instance of the mixin classed used for each advised object.

```
public class AuditableAdvisor : DefaultIntroductionAdvisor
{
  public AuditableAdvisor() : base(new AuditableMixin())
  {
  }
}
```

Other contructors let you explicity specify the interfaces of the class that will be introduced. See the SDK documentation for more details.

We can apply this advisor programmatically, using the `IAdvised.AddIntroduction(),` method, or (the recommended way) in XML configuration using the `IntroductionNames` property on `ProxyFactoryObject`, which will be discussed later.
Unlike the AOP implementation in the Spring Framework for Java, introduction advice in Spring.NET is not implemented as a specialized type of interception advice. The advantage of this approach is that introductions are not kept in the interceptor chain, which allows some significant performace optimizations. When a method is called that has no interceptors, a direct call is used instead of reflection regardless of whether target method is on the target object itself or one of the introductions. This means that introduced methods perform the same as target object methods, which could be useful for adding introductions to fine grained objects. The disadvantage is that if the mixin functionality would benefit from having access to the calling stack, it is not available. Introductions with this functionality will be addressed in a future version of Spring.NET AOP.

9.4. Advisors in Spring.NET

In Spring.NET, an advisor is a modularization of an aspect. Advisors typically incorporate both an advice and a pointcut.

Apart from the special case of introductions, any advisor can be used with any advice. The `Spring.Aop.Support.DefaultPointcutAdvisor` class is the most commonly used advisor implementation. For example, it can be used with a `IMethodInterceptor`, `IBeforeAdvice` or `IThrowsAdvice` and any pointcut definition.

Other convenience implementations provided are: `AttributeMatchMethodPointcutAdvisor` shown in usage previously in Section 9.2.3.1.2, “Attribute pointcuts” for use with attribute based pointcuts. `RegularExpressionMethodPointcutAdvisor` that will apply pointcuts based on the matching a regular expression to method names.

It is possible to mix advisor and advice types in Spring.NET in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring.NET will automatically create the necessary interceptor chain.

9.5. Using the ProxyFactoryObject to create AOP proxies

If you're using the Spring.NET IoC container for your business objects - generally a good idea - you will want to use one of Spring.NET's AOP-specific `IFactoryObject` implementations (remember that a factory object introduces a layer of indirection, enabling it to create objects of a different type - Section 3.3.5, “Setting a reference using the members of other objects and classes.”).

The basic way to create an AOP proxy in Spring.NET is to use the `Spring.Aop.Framework.ProxyFactoryObject` class. This gives complete control over ordering and application of the pointcuts and advice that will apply to your business objects. However, there are simpler options that are preferable if you don't need such control.

9.5.1. Basics

The `ProxyFactoryObject`, like other Spring.NET `IFactoryObject` implementations, introduces a level of indirection. If you define a `ProxyFactoryObject` with name `foo`, what objects referencing `foo` see is not the `ProxyFactoryObject` instance itself, but an object created by the `ProxyFactoryObject's` implementation of the `GetObject()` method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using an `ProxyFactoryObject` or other IoC-aware classes that create AOP proxies, is that it means that advice and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

9.5.2. ProxyFactoryObject Properties

Like most `IFactoryObject` implementations provided with Spring.NET, the `ProxyFactoryObject` is itself a Spring.NET configurable object. Its properties are used to:

- Specify the target object that is to be proxied.
- Specify the advice that is to be applied to the proxy.

Some key properties are inherited from the `Spring.Aop.Framework.ProxyConfig` class: this class is the superclass for all AOP proxy factories in Spring.NET. Some of the key properties include:

- `ProxyTargetType`: a boolean value that should be set to true if the target class is to be proxied directly, as opposed to just proxying the interfaces exposed on the target class.
- `Optimize`: whether to apply aggressive optimization to created proxies. Don't use this setting unless you understand how the relevant AOP proxy handles optimization. The exact meaning of this flag will differ between proxy implementations and will generally result in a tradeoff between proxy creation time and runtime performance. Optimizations may be ignored by certain proxy implementations and may be disabled silently based on the value of other properties such as `ExposeProxy`.

- `IsFrozen`: whether advice changes should be disallowed once the proxy factory has been configured. The default is false.
- `ExposeProxy`: whether the current proxy should be exposed in via the `AopContext` so that it can be accessed by the target. (It's available via the `IMethodInvocation` without the need for the `AopContext`.) If a target needs to obtain the proxy and `ExposeProxy` is `true`, the target can use the `AopContext.CurrentProxy` property.
- `AopProxyFactory`: the implementation of `IAopProxyFactory` to use when generating a proxy. Offers a way of customizing whether to use remoting proxies, IL generation or any other proxy strategy. The default implementation will use IL generation to create composition-based proxies.

Other properties specific to the `ProxyFactoryObject` class include:

- `ProxyInterfaces`: the array of `string` interface names we're proxying.
- `InterceptorNames`: `string` array of `IAdvisor`, interceptor or other advice names to apply. Ordering is significant... first come, first serve that is. The first interceptor in the list will be the first to be able to interceptor the invocation (of course if it concerns a regular MethodInterceptor or BeforeAdvice).

  The names are object names in the current container, including objectnames from container hierarchies. You can't mention object references here since doing so would result in the `ProxyFactoryObject` ignoring the singleton setting of the advise.
- `IntroductionNames`: The names of objects in the container that will be used as introductions to the target object. If the object refered to by name does not implement the `IIntroductionAdvisor` it will be passed to the default constructor of `DefaultIntroductionAdvisor` and all of the objects interfaces will be added to the target object. Objects that implement the `IIntroductionAdvisor` interface will be used as is, giving you a finer level of control over what interfaces you may want to expose and the types for which they will be matched against.
- `IsSingleton` whether or not the factory should return a single proxy object, no matter how often the `GetObject()` method is called. Several `IFactoryObject` implementations offer such a method. Default value is `true`. If you would like to be able to apply advice on a per-proxy object basis, use a `IsSingleton` value of `false` and a `IsFrozen` value of `false`. If you want to use stateful advice--for example, for stateful mixins--use prototype advices along with a `IsSingleton` value of `false`.

9.5.3. Proxying Interfaces

Let's look at a simple example of `ProxyFactoryObject` in action. This example involves:

- A target object that will be proxied. This is the "personTarget" object definition in the example below.
- An `IAdvisor` and an `IInterceptor` used to provide advice.
- An AOP proxy object definition specifying the target object (the personTarget object) and the interfaces to proxy, along with the advices to apply.

```
<object id="personTarget" type="MyCompany.MyApp.Person, MyCompany">
    <property name="name" value="Tony"/>
    <property name="age" value="51"/>
</object>

<object id="myCustomInterceptor" type="MyCompany.MyApp.MyCustomInterceptor, MyCompany">
    <property name="customProperty" value="configuration string"/>
```

```
</object>

<object id="debugInterceptor" type="Spring.Aop.Advice.DebugAdvice, Spring.Aop">
</object>

<object id="person" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

    <property name="proxyInterfaces" value="MyCompany.MyApp.IPerson"/>

    <property name="target" ref="personTarget"/>

    <property name="interceptorNames">
        <list>
            <value>debugInterceptor</value>
            <value>myCustomInterceptor</value>
        </list>
    </property>

</object>
```

Note that the `InterceptorNames` property takes a list of `strings`: the object names of the interceptor or advisors in the current context. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.
You might be wondering why the list doesn't hold object references. The reason for this is that if the `ProxyFactoryObject's` singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it's necessary to be able to obtain an instance of the prototype from the context; holding a reference isn't sufficient.

The "person" object definition above can be used in place of an `IPerson` implementation, as follows:

```
IPerson person = (IPerson) factory.GetObject("person");
```

Other objects in the same IoC context can express a strongly typed dependency on it, as with an ordinary .NET object:

```
<object id="personUser" type="MyCompany.MyApp.PersonUser, MyCompany">
    <property name="person" ref="person"/>
</object>
```

The `PersonUser` class in this example would expose a property of type `IPerson`. As far as it's concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its type would be a proxy type. It would be possible to cast it to the `IAdvised` interface (discussed below).

It's possible to conceal the distinction between target and proxy using an anonymous *inline object*, as follows. (for more information on inline objects see Section 3.3.3.5, “Inline objects”.) Only the `ProxyFactoryObject` definition is different; the advice is included only for completeness:

```
<object id="myCustomInterceptor" type="MyCompany.MyApp.MyCustomInterceptor, MyCompany">
    <property name="customProperty" value="configuration string"/>
</object>

<object id="debugInterceptor" type="Spring.Aop.Advice.DebugAdvice, Spring.Aop">
</object>

<object id="person" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">

    <property name="proxyInterfaces" value="MyCompany.MyApp.IPerson"/>

    <property name="target">
      <!-- Instead of using a reference to target, just use an inline object -->
      <object type="MyCompany.MyApp.Person, MyCompany">
        <property name="name" value="Tony"/>
        <property name="age" value="51"/>
      </object>
    </property>

    <property name="interceptorNames">
        <list>
            <value>debugInterceptor</value>
            <value>myCustomInterceptor</value>
        </list>
    </property>

</object>
```

This has the advantage that there's only one object of type `Person`: useful if we want to prevent users of the application context obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There's also arguably an advantage in that the ProxyFactoryObject definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

9.5.1. Applying advice on a per-proxy basis.

Let's look at an example of configuring the proxy objects retrieved from `ProxyFactoryObject`.

```

<!-- create the object to reference -->
<object id="RealObjectTarget" type="MyRealObject" singleton="false"/>
<!-- create the proxied object for everyone to use-->
<object id="MyObject" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
        <property name="proxyInterfaces" value="MyInterface" />
        <property name="isSingleton" value="false"/>
        <property name="targetName" value="RealObjectTarget" />
</object>

```

If you are using a prototype as the target you must set the `TargetName` property with the name/object id of your object and not use the property `Target` with a reference to that object. This will then allow a new proxy to be created around a new prototype target instance./para>

Consider the above Spring.Net object configuration. Notice that the `IsSingleton` property of the `ProxyFactoryObject` instance is set to false. This means that each proxy object will be unique. Thus, you can configure each proxy object with its' own individual advice(s) using the following syntax

```
MyInterface myProxyObject1 = (MyInterface)ctx.GetObject("MyObject"); // Will return un-advised instance of proxy

IAdvised advised = (IAdvised)myProxyObject1;
advised.AddAdvice( new DebugAdvice() ); // myProxyObject1 instance now has an advice attached to it.

MyInterface myProxyObject2 = (MyInterface)ctx.GetObject("MyObject"); // Will return a new, un-advised instance o
```

9.5.4. Proxying Classes

This functionalty is not yet available. Stay tuned!

9.6. Creating AOP Proxies Programmatically with the ProxyFactory

It's easy to create AOP proxies programmatically using Spring.NET. This enables you to use Spring.NET AOP without dependency on Spring.NET IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.AddAdvice(myMethodInterceptor);
factory.AddAdvisor(myAdvisor);
IBusinessInterface tb = (IBusinessInterface) factory.GetProxy();
```

The first step is to contruct a object of type `Spring.Aop.Framework.ProxyFactory`. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add interceptors or advisors, and manipulate them for the life of the `ProxyFactory`. *TODO: note on introductions.*

There are also convenience methods on `ProxyFactory` (inherited from `AdvisedSupport`) allowing you to add other advice types such as before and throws advice. `AdvisedSupport` is the superclass of both `ProxyFactory` and `ProxyFactoryObject`.

Note

Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from .NET code with AOP, as in general.

9.7. Manipulating Advised Objects

However you create AOP proxies, you can manipulate them using the `Spring.Aop.Framework.IAdvised` interface. Any AOP proxy can be cast to this interface, whatever other interfaces it implements. This interface includes the following methods and properties:

```
public interface IAdvised
{
    IAdvisor[] Advisors { get; }

    IIntroductionAdvisor[] Introductions { get; }

    void  AddInterceptor(IInterceptor interceptor);

    void  AddInterceptor(int pos, IInterceptor interceptor);

    void  AddAdvisor(IAdvisor advisor);

    void  AddAdvisor(int pos, IAdvisor advisor);

    void  AddIntroduction(IIntroductionAdvisor advisor);

    void  AddIntroduction(int pos, IIntroductionAdvisor advisor);

    int IndexOf(IAdvisor advisor);

    int IndexOf(IIntroductionAdvisor advisor);

    bool RemoveAdvisor(IAdvisor advisor);

    void RemoveAdvisor(int index);

    bool RemoveInterceptor(IInterceptor interceptor);

    bool RemoveIntroduction(IIntroductionAdvisor advisor);

    void RemoveIntroduction(int index);

    void ReplaceIntroduction(int index, IIntroductionAdvisor advisor);

    bool ReplaceAdvisor(IAdvisor a, IAdvisor b);
}
```

The `Advisors` property will return an `IAdvisor` for every advisor, interceptor or other advice type that has been added to the factory. If you added an `IAdvisor`, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring.NET will have wrapped this in an advisor with a `IPointcut` that always returns `true`. Thus if you added an `IMethodInterceptor`, the advisor returned for this index will be a `DefaultPointcutAdvisor` returning your `IMethodInterceptor` and an `IPointcut` that matches all types and methods.

The `AddAdvisor()` methods can be used to add any `IAdvisor`. Usually this will be the generic `DefaultPointcutAdvisor`, which can be used with any advice or pointcut (but not for introduction).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. *TODO: Need to query this for introductions* The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)
It's questionable whether it's advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a `Frozen` flag, in which case the `IAdvised` `IsFrozen` property will return `true`, and any attempts to modify advice through addition or removal will result in an `AopConfigException`. The ability to freeze the state of an advised object is useful in some cases: For example, to prevent calling code removing a security interceptor.

9.8. Using the "autoproxy" facility

So far we've considered explicit creation of AOP proxies using a `ProxyFactoryObject` or similar factory objects. For applications that would like create many AOP proxies, say across all the classes in a service layer, this approach can lead to a lengthy configuration file. To simplify the creation of many AOP proxies Spring provides "autoproxy" capabilities that will automatically proxy object definitions based on higher level criteria that will group together multiple objects as candidates to be proxied.

This functionality is built on Spring "object post-processor" infrastructure, which enables modification of any object definition as the container loads. Refer to Section 3.8, “Customizing objects with IObjectPostProcessors” for general information on object post-processors.

In this model, you set up some special object definitions in your XML object definition file configuring the auto proxy infrastructure. This allows you just to declare the targets eligible for autoproxying: you don't need to use `ProxyFactoryObject`.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling GetObject("MyBusinessObject1") on an ApplicationContext will return an AOP proxy, not the target business object. The "inline object" idiom shown earlier in Section 9.5.3, “Proxying Interfaces” also offers this benefit.)

The namespace `Spring.Aop.Framework.AutoProxy` provides generic autoproxy infrastructure, should you choose to write your own autoproxy implementations, as well as several out-of-the-box implementations. The basis for autoproxy object post-processors is the abstract base class `AbstractAutoProxyCreator`. Two implementations are provided, `ObjectNameAutoProxyCreator` and `DefaultAdvisorAutoProxyCreator`. These are discussed in the following sections.

9.8.1. ObjectNameAutoProxyCreator

The `ObjectNameAutoProxyCreator` automatically creates AOP proxies for object with names matching literal values or wildcards. The following simple classes are used to demonstrate this autoproxy functionality.

```
public enum Language
{
    English = 1,
    Portuguese = 2,
    Italian = 3
}


public interface IHelloWorldSpeaker
{
   void SayHello();
}


public class HelloWorldSpeaker : IHelloWorldSpeaker
{
    private Language language;

    public Language Language
    {
        set { language = value; }
        get { return language; }
    }

    public void SayHello()
    {
        switch (language)
        {
            case Language.English:
                Console.WriteLine("Hello World!");
                break;
            case Language.Portuguese:
                Console.WriteLine("Oi Mundo!");
                break;
            case Language.Italian:
                Console.WriteLine("Ciao Mondo!");
                break;
        }
    }
}


public class DebugInterceptor : IMethodInterceptor
{
    public object Invoke(IMethodInvocation invocation)
    {
        Console.WriteLine("Before: " + invocation.Method.ToString());
        object rval = invocation.Proceed();
        Console.WriteLine("After:  " + invocation.Method.ToString());
        return rval;
    }

}
```

The following XML is used to automatically create an AOP proxy and apply a Debug interceptor to object definitions whose names match "English*" and "PortugueseSpeaker".

```
<object id="ProxyCreator" type="Spring.Aop.Framework.AutoProxy.ObjectNameAutoProxyCreator, Spring.Aop">
  <property name="ObjectNames">
      <list>
          <value>English*</value>
          <value>PortugeseSpeaker</value>
      </list>
  </property>
  <property name="InterceptorNames">
      <list>
          <value>debugInterceptor</value>
      </list>
  </property>
</object>

<object id="debugInterceptor" type="AopPlay.DebugInterceptor, AopPlay"/>

<object id="EnglishSpeakerOne" type="AopPlay.HelloWorldSpeaker, AopPlay">
  <property name="Language" value="English"/>
</object>

<object id="EnglishSpeakerTwo" type="AopPlay.HelloWorldSpeaker, AopPlay">
  <property name="Language" value="English"/>
</object>

<object id="PortugeseSpeaker" type="AopPlay.HelloWorldSpeaker, AopPlay">
  <property name="Language" value="Portuguese"/>
</object>

<object id="ItalianSpeakerOne" type="AopPlay.HelloWorldSpeaker, AopPlay">
  <property name="Language" value="Italian"/>
</object>
```

As with `ProxyFactoryObject`, there is an InterceptorNames property rather than a list of interceptors, to allow correct behavior for prototype advisors. Named "interceptors" can be advisors or any advice type.

The same advice will be applied to all matching objects. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different objects.

Running the following simple program demonstrates the application of the AOP interceptor.

```
IApplicationContext ctx = ContextRegistry.GetContext();
IDictionary speakerDictionary = ctx.GetObjectsOfType(typeof(IHelloWorldSpeaker));
foreach (DictionaryEntry entry in speakerDictionary)
{
    string name = (string)entry.Key;
    IHelloWorldSpeaker worldSpeaker = (IHelloWorldSpeaker)entry.Value;
    Console.Write(name + " says; ");
    worldSpeaker.SayHello();
}
```

The output is shown below

```
ItalianSpeakerOne says; Ciao Mondo!
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After:  Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
```

9.8.2. DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is `DefaultAdvisorAutoProxyCreator`. This will automatically apply eligible advisors in the current appliation context, without the need to include specific object names in the autoproxy advisor's object definition. It offers the same merit of consistent configuration and avoidance of duplication as `ObjectNameAutoProxyCreator`.

Using this mechanism involves:

- Specifying a `DefaultAdvisorAutoProxyCreator` object definition

- Specifying any number of Advisors in the same or related contexts. Note that these *must* be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate object definitions.

The `DefaultAdvisorAutoProxyCreator` will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each object defined in the application context.

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied.

The `DefaultAdvisorAutoProxyCreator` is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily--for example, tracing or performance monitoring aspects--with minimal change to configuration.

The following example demonstrates the use of `DefaultAdvisorAutoProxyCreator`. Expanding on the previous example code used to demonstrate `ObjectNameAutoProxyCreator` we will add a new class, `SpeakerDao`, that acts as a Data Access Object to find and store `IHelloWorldSpeaker` objects.

```
public interface ISpeakerDao
{
    IList FindAll();

    IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker);
}

public class SpeakerDao : ISpeakerDao
{
    public System.Collections.IList FindAll()
    {
        Console.WriteLine("Finding speakers...");
        // just a demo...fake the retrieval.
        Thread.Sleep(10000);
        HelloWorldSpeaker speaker = new HelloWorldSpeaker();
        speaker.Language = Language.Portuguese;

        IList list = new ArrayList();
        list.Add(speaker);
        return list;
    }

    public IHelloWorldSpeaker Save(IHelloWorldSpeaker speaker)
    {
        Console.WriteLine("Saving speaker...");
        // just a demo...not really saving...
        return speaker;
    }

}
```

The XML configuration specifies two Advisors, that is, the combination of advice (the behavior to add) and a pointcut (where the behavior should be applied). A `RegularExpressionMethodPointcutAdvisor` is used as a convenience to specify the pointcut as a regular expression that matches methods names. Other pointcuts of your own creation could be used, in which case a `DefaultPointcutAdvisor` would be used to define the Advisor. The object definitions for these advisors, advice, and SpeakerDao object are shown below

```
<object id="SpeachAdvisor" type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor, Spring.Aop">

    <property name="advice" ref="debugInterceptor"/>
    <property name="patterns">
        <list>
            <value>.*Say.*</value>
        </list>
    </property>

</object>
```

```
<object id="AdoAdvisor" type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor, Spring.Aop">

    <property name="advice" ref="timingInterceptor"/>
    <property name="patterns">
        <list>
            <value>.*Find.*</value>
        </list>
    </property>

</object>

// Advice
<object id="debugInterceptor" type="AopPlay.DebugInterceptor, AopPlay"/>

<object id="timingInterceptor" type="AopPlay.TimingInterceptor, AopPlay"/>

// Speaker DAO Object - has 'FindAll' Method.
<object id="speakerDao" type="AopPlay.SpeakerDao, AopPlay"/>

// HelloWorldSpeaker objects as previously listed.
```

Adding an instance of `DefaultAdvisorAutoProxyCreator` to the configuration file

```
<object id="ProxyCreator" type="Spring.Aop.Framework.AutoProxy.DefaultAdvisorAutoProxyCreator, Spring.Aop"/>
```

will apply the debug interceptor on all objects in the context that have a method that contains the text "Say" and apply the timing interceptor on objects in the context that have a method that contains the text "Find". Running the following code demonstrates this behavior. Note that the "Save" method of SpeakerDao does not have any advice applied to it.

```
IApplicationContext ctx = ContextRegistry.GetContext();
IDictionary speakerDictionary = ctx.GetObjectsOfType(typeof(IHelloWorldSpeaker));
foreach (DictionaryEntry entry in speakerDictionary)
{
    string name = (string)entry.Key;
    IHelloWorldSpeaker worldSpeaker = (IHelloWorldSpeaker)entry.Value;
    Console.Write(name + " says; ");
    worldSpeaker.SayHello();
}
ISpeakerDao dao = (ISpeakerDao)ctx.GetObject("speakerDao");
IList speakerList = dao.FindAll();
IHelloWorldSpeaker speaker = dao.Save(new HelloWorldSpeaker());
```

This produces the following output

```
ItalianSpeakerOne says; Before: Void SayHello()
Ciao Mondo!
After:  Void SayHello()
EnglishSpeakerTwo says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
PortugeseSpeaker says; Before: Void SayHello()
Oi Mundo!
After:  Void SayHello()
EnglishSpeakerOne says; Before: Void SayHello()
Hello World!
After:  Void SayHello()
Finding speakers...
Elapsed time = 00:00:10.0154745
Saving speaker...
```

The DefaultAdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the `Spring.Core.IOrdered` interface to ensure correct ordering if this is an issue. The default is unordered.

9.9. Using TargetSources

Spring.NET offers the concept of a *TargetSource*, expressed in the `Spring.Aop.ITargetSource` interface. This interface is responsible for returning the "target object" implementing the joinpoint. The `TargetSource` implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring.NET AOP don't normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let's look at the standard target sources provided with Spring.NET, and how you can use them.
When using a custom target source, your target will usually need to be a prototype rather than a singleton object definition. This allows Spring.NET to create a new target instance when required.

9.9.1. Hot swappable target sources

The `org.Spring.NETframework.aop.target.HotSwappableTargetSource` exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source's target takes effect immediately. The `HotSwappableTargetSource` is threadsafe.

You can change the target via the `swap()` method on HotSwappableTargetSource as follows:

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) objectFactory.GetObject("swapper");
object oldTarget = swapper.swap(newTarget);
```

The XML definitions required look as follows:

```
<object id="initialTarget" type="MyCompany.OldTarget, MyCompany">
</object>

<object id="swapper"
    type="Spring.Aop.Target.HotSwappableTargetSource, Spring.Aop">
    <constructor-arg><ref local="initialTarget"/></constructor-arg>
</object>

<object id="swappable"
    type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop"
>
    <property name="targetSource">
        <ref local="swapper"/>
    </property>
</object>
```

The above `swap()` call changes the target of the swappable object. Clients who hold a reference to that object will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn't add any advice--and it's not necessary to add advice to use a `TargetSource`--of course any `TargetSource` can be used in conjunction with arbitrary advice.

9.9.2. Pooling target sources

Using a pooling target source provides a programming model in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring.NET pooling and pooling in .NET Enterprise Services pooling is that Spring.NET pooling can be applied to any PONO. (Plain old .NET object). As with Spring.NET in general, this service can be applied in a non-invasive way.

Spring.NET provides out-of-the-box support using a pooling implementation based on Jakarta Commons Pool 1.1, which provides a fairly efficient pooling implementation. You'll need to reference the Spring.Pool assembly to use this feature. It's also possible to subclass `Spring.Aop.Target.AbstractPoolingTargetSource` to support any other pooling API.

Sample configuration is shown below:

```
<object id="businessObjectTarget" type="MyCompany.MyBusinessObject, MyCompany"
    singleton="false">
    ... properties omitted
</object>

<object id="poolTargetSource"
    type="Spring.Aop.Target.SimplePoolTargetSource, Spring.Aop">
    <property name="targetObjectName"><value>businessObjectTarget</value></property>
    <property name="maxSize"><value>25</value></property>
</object>

<object id="businessObject"
    type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop"
>
    <property name="targetSource"><ref local="poolTargetSource"/></property>
    <property name="interceptorNames"><value>myInterceptor</value></property>
</object>
```

Note that the target object--"businessObjectTarget" in the example--*must* be a prototype. This allows the `PoolingTargetSource` implementation to create new instances of the target to grow the pool as necessary. See the SDK documentation for for `AbstractPoolingTargetSource` and the concrete subclass you wish to use for information about it's properties: maxSize is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn't necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don't set the interceptorNames property at all.

It's possible to configure Spring.NET so as to be able to cast any pooled object to the `Spring.Aop.Target.PoolingConfig` interface, which exposes information about the configuration and current size of the pool through an introduction. You'll need to define an advisor like this:

```
<object id="poolConfigAdvisor"
    type="Spring.Object.Factory.Config.MethodInvokingFactoryObject, Spring.Aop">
    <property name="target" ref="poolTargetSource" />
    <property name="targetMethod" value="getPoolingConfigMixin" />
</object>
```

This advisor is obtained by calling a convenience method on the `AbstractPoolingTargetSource` class, hence the use of `MethodInvokingFactoryObject`. This advisor's name (`'poolConfigAdvisor'` here) must be in the list of interceptor names in the `ProxyFactoryObject` exposing the pooled object.

The cast will look as follows:

```
PoolingConfig conf = (PoolingConfig) objectFactory.GetObject("businessObject");
Console.WriteLine("Max pool size is " + conf.getMaxSize());
```

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally threadsafe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using autoproxying. It's possible to set the TargetSources used by any autoproxy creator.

9.9.3. Prototype target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object may not be hihgh, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn't use this approach without very good reason.

To do this, you could modify the `poolTargetSource` definition shown above as follows. (the name of the definition has also been changed, for clarity.)

```
<object id="prototypeTargetSource"
        type="Spring.Aop.Target.PrototypeTargetSource, Spring.Aop">
    <property name="targetObjectName" value="businessObject" />
</object>
```

There is only one property: the name of the target object. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target object must be a prototype object definition.

9.10. Defining new Advice types

Spring.NET AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to interception around, before, throws, and after returning advice, which are supported out of the box.

The `Spring.Aop.Framework.Adapter` package is an SPI (Service Provider Interface) package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement the `AopAlliance.Aop.IAdvice` tag interface.

Please refer to the `Spring.Aop.Framework.Adapter` namespace documentation for further information.

9.11. Further reading and resources

The Spring.NET team recommends the excellent *AspectJ in Action* by Ramnivas Laddad (Manning, 2003) for an introduction to AOP.

If you are interested in more advanced capabilities of Spring.NET AOP, take a look at the test suite as it illustrates advanced features not discussed in this document.

Chapter 10. .NET Remoting

10.1. Introduction

(Available in 1.1)

The main goal of Spring.Services is to provide location transparency for business services. We believe that users should be able to implement services the simplest way possible, using service interfaces and implementations in the form of plain .NET classes. We also think that decision on how a particular service is exposed to clients should be a configuration concern and not an implementation concern.

In preview status is support for exposing a plain .NET object as a web service, System.EnterpriseServices serviced component, and remoted objects. By "plain .NET object" we mean classes that do not inherit from a specific infrastructure base class (such as MarshalByRefObject) or infrastructure attributes (such as WebMethod). The current implementation relies on your plain .NET object to implement a business interface. Spring's .NET Remoting exporters will automatically create a proxy that implements MarshalByRefObject. On the server side you can register SAO types as either SingleCall or Singleton and also configure on a per-object basis lifetime and leasing parameters. Additionally for SAO objects you can export an object that has had AOP advice applied to it. On the client side you can obtain CAO references to server proxy objects in a manner that promotes interface based design best practices when developing .NET remoting applications. An remoting specific xml-schema is also provided to simplify the remoting configuration, although you can still use the standard reflection-like property based configuration schema.

If you want to use these features please get the code from CVS (instructions) or from the nightly release vailable from the following download page. A sample application, often referred to in this documentation, is under the directory "examples\Spring\Spring.Examples.Calculator"/>

10.2. Publishing a Singleton SAO on the Server

Exposing a Singleton SAO service can be done in two ways. The first is through programatic or administrative type registration that makes calls to `RemotingConfiguration.RegisterWellKnownServiceType`. This method has the limitation that you must use a default constructor and you can not easily configure the singleton state at runtime since it is created on demand. The second way is to publish an object instance using `RemotingServices.Marshal`. This method overcomes the limitations of the first method. Example server side code for publishing an SAO singleton object with a predefined state is shown below

```
AdvancedMBRCalculator calc = new AdvancedMBRCalculator(217);
RemotingServices.Marshal(calc, "MyRemotedCalculator");
```

The class AdvancedMBRCalculator used above inherits from MarshalByRefObject.

If your design calls for configuring a singleton SAO, or using a non-default constructor, you can use Spring IoC container to create the SAO instance, configure it, and register it with the .NET remoting infrastructure. The `SaoExporter` class performs this task and most importantly, will automatically create a proxy class that inherits from MarshalbyRefObject if your business object does not already do so. The following XML taken from the Remoting QuickStart demonstrates its usage.

```
<object id="singletonCalculator" type="Services.AdvancedCalculator, Services">
  <constructor-arg type="int" value="217"/>
</object>

<!-- Registers the calculator service as a SAO in 'Singleton' mode. -->
```

```
<object name="saoSingletonCalculator" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculator" />
  <property name="ServiceName" value="RemotedSaoSingletonCalculator" />
  <property name="Infinite" value="true" />
</object>
```

This XML fragment shows how an existing object "singletonCalculator" defined in the Spring context is exposed under the url-path name "RemotedSaoSingletonCalculator". (The fully qualified url is tcp://localhost:8005/RemotedSaoSingleCallCalculator using the standard .NET channel configuration shown futher below.) The `AdvancedCalculator` class implements the business interface `IAdvancedCalculator`. The current proxy implementation requires that your business objects implement an interface. The interfaces' methods will be the ones exposed in the generated .NET remoting proxy. The inital memory of the calculator is set to 217 via the constructor. The class `Services.AdvancedCalcualtor` *does not* inherit from `MarshalByRefObject`. The `Infinite` property is used to set the lifetime of the object to be infinite so that the the singleton will not be garbage collected after 5 minutes (the default lease time). Other lifecycle properties you can set are InitialLeaseTime, RenewOnCallTime, and SponsorshipTimeout.

The following XML fragment shows to to expose the calculator service in SAO 'SingleCall' mode.

```
<object id="prototypeCalculator" type="Services.AdvancedCalculator, Services" singleton="false">
  <constructor-arg type="int" value="217"/>
</object>

<object name="saoSingleCallCalculator" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="prototypeCalculator" />
  <property name="ServiceName" value="RemotedSaoSingleCallCalculator" />
</object>
```

The object refered to in the `TargetName` parameter can be an AOP proxy to a business object. For example, if we were to apply some simple logging advice to the singleton calculator, the following standard AOP configuration is used to create the target for the SaoExporter

```
<object id="singletonCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
        <property name="target" ref="singletonCalculator"/>
        <property name="interceptorNames">
                <list>
                        <value>Log4NetLoggingAroundAdvice</value>
                </list>
        </property>
</object>
<object name="saoSingletonCalculatorWeaved" type="Spring.Remoting.SaoExporter, Spring.Services">
        <property name="TargetName" value="singletonCalculatorWeaved" />
        <property name="ServiceName" value="RemotedSaoSingletonCalculatorWeaved" />
        <property name="Infinite" value="true" />
</object>
```

When using `SaoExporter` you can still use the standard remoting administration section in the application configuration file to register the channel. `ChannelServices` as shown below

```
<system.runtime.remoting>
        <application>
                <channels>
                        <channel ref="tcp" port="8005" />
                </channels>
        </application>
</system.runtime.remoting>
```

A console application that will host this remoted object needs to initialize the .NET Remoting infrastructure with a call to RemotingConfiguration (since we are using the .config file for channel registration) and then start the Spring application context. This is shown below

```
RemotingConfiguration.Configure("RemoteServer.exe.config");
```

```
IApplicationContext ctx = ContextRegistry.GetContext();

Console.Out.WriteLine("Server listening...");

Console.ReadLine();
```

The readline prevents the console application from exiting. You can refer to the the code in RemoteServer in the Remoting QuickStart to see ths code in action. Deployment as a windows service should follow in a similar manner but without the ReadLine as the process does not exit. Deployment inside IIS is not yet tested.

Note

As generally required with a .NET Remoting application, the arguments to your service methods should be Serializable.

10.3. Accessing a SAO on the Client

Administrative type registration on the client side lets you easily obtain a reference to a SAO object. When a type is registered on the client, using the new operator or using the reflection API will return a proxy to the remote object instead of a local reference. Administrative type registration on the client for a SAO object is performed using the `wellknown` element in the client configuration section. However, this approach requires that you expose the implemenation of the class on the client side. Practially speaking this would mean linking in the server assembly to the client application, a generally recognized bad practice. This is dependency can be removed by developing remote services based on a business interface. Aside from remoting considerations, the separation of interface and implementation is considered a good practice when designing OO systems. In the context of remoting, this means that the client can obtain a proxy to a specific implemenation with *only* a reference to the interface assembly. To achieve the decoupling of client and server, a separate assembly containing the interface definitions is created and shared between the client and server applications.

There is a simple means for following this design when the remote object is a SAO object. A call to `Activator.GetObject` will instantiate a SAO proxy on the client. For CAO objects another mechanism is used and is discussed later. The code to obtain the SAO proxy is shown below

```
ICalculator calc = (ICalculator)Activator.GetObject (
  typeof (ICalculator),
  "tcp://localhost:8005/MyRemotedCalculator");
```

To obtain a reference to a SAO proxy within the IoC container, you can use the object factory `SaoFactoryObject` in the Spring configuration file. The following XML taken from the Remoting QuickStart demonstrates its usage.

```
<object id="calculatorService" type="Spring.Remoting.SaoFactoryObject, Spring.Services">
        <property name="ServiceInterface" value="Interfaces.IAdvancedCalculator, Interfaces" />
        <property name="ServiceUrl" value="tcp://localhost:8005/RemotedSaoSingletonCalculator" />
</object>
```

The ServiceInterface property specifies the type of proxy to create while the ServiceUrl property creates a proxy bound to the specified server and published object name.

Other objects in the IoC container that depend on an implementation of the interface `ICalculator` can now refer to the object "calculatorService", thereby using using a remote implementation of this interface. The exposure of dependencies among objects within the IoC container lets you easily switch the implementation of `ICalculator`. By using the IoC container changing the application to use a local instead of remote implementation is a configuration file change, not a code change. By promoting interface based programing, the ability to switch implementation makes it easier to unit test the client application, since unit testing can be done with a mock implementation of the interface. Similarly, development of the client can proceed independent of the server implementation. This increases productivity when there are separate client and server development teams. The two teams agree on interfaces before starting development. The client team can quickly create a simple, but functional implementation and then integrate with the server implementation when it is ready.

10.4. CAO best practices

Creating a client activiated object (CAO) is typically done by administrative type registration, either programmatically or via the standard .NET remoting configuration section. The registration process allows you to use the 'new' operator to create the remote object and requires that the implementation of the object be distributed to the client. As mentioned before, this is not a desirable approach to developing distributed systems. The best practice approach that avoids this problem is to create an SAO based factory class on the server that will return CAO references to the client. In a maner similar to how Spring's generic object factory can be used as a replacement to creating a factory per class, we can create a generic SAO object factory to return CAO references to objects defined in Spring's application context. This functionality is encapsulated in Spring's `CaoExporter` class. On the client side a reference is obtained using `CaoFactoryObject`. The client side factory object supports creation of the CAO object using constructor arguments. In addition to reducing the clutter and tedium around creating factory classes specific to each object type you which to expose in this manner, this approach has the additional benefit of not requiring any type registration on the client or server side. This is because the act of returning an instance of a class that inherits from MarshalByRefObject across a remoting boundary automatically returns a CAO object reference. For more information on this best-practice, refer to the last section, Section 10.8, “Additional Resources”, for some links to additional resources.

10.5. Registering a CAO object on the Server

10.6. Accessing a CAO on the Client

Creating a client activiated object (CAO) is typically done by administrative type registration, either programmatically or via the standard .NET remoting configuration section. The registration process allows you to use the 'new' operator to create the remote object and requires that the implementation of the object be distributed to the client. As mentioned before, this is not a desirable approach to developing distributed systems. The best practice approach that avoids this problem is to create an SAO based factory class on the server that will return CAO references to the client. In a maner similar to how Spring's generic object factory can be used as a replacement to creating a factory per class, we can create a generic SAO object factory for use with remoting. An additional benefit of this approach is that you don't have to perform any type registration on the client or server side. This is because the act of returning an instance of a class that inherits from MarshalByRefObject across a remoting boundary automatically returns a CAO object reference. For more information on this best-practice, refer to the last section, Section 10.8, “Additional Resources”, for some links to additional resources.

The interface `Spring.Remoting.IRemoteFactory` provides this funcationality and defines the following three methods

```
public interface IRemoteFactory
{
  MarshalByRefObject GetObject(string name);

  MarshalByRefObject GetObject(string name, object[] constructorArguments);

  MarshalByRefObject GetSaoObject(string name);
}
```

The first two methods are used to return references to CAO objects by specifying the name of the object, as registered with Spring on the server side, and an optional array of constructor arguments. The matching of constructor arguments is done by type which should cover most cases. The class that you use on the server side that implements this interface is `Spring.Remoting.RemoteFactory.` and it contains a reference to the standard Spring application context (i.e. Object Factory) which is used to return instances of an object by name. You can use the `SaoServiceExporter` to publish this object to a well known location as shown below from the quick start example.

```
<object name="prototypeCalculator" type="Server.RemotedCalculator, Server"
                                singleton="false"/>

<object name="remoteFactory" type="Spring.Remoting.RemoteFactory, Spring.Services"/>

<object name="publishedRemoteFactory" type="Spring.Remoting.SaoServiceExporter, Spring.Services">
  <property name="Service" ref="remoteFactory"/>
  <property name="ServiceName" value="RemoteFactory"/>
</object>
```

The object that we will want to get by name on the client side is named, `prototypeCalculator` Be sure to remember to set the singleton attribute to false, otherwise multiple calls to `GetObject` will return a CAO reference to the same object instance. On the client side we can obtain a reference to the `IRemoteFactory` using the `SaoFactoryObject` helper class as described previously. If you request an object by name that does not inherit from MarshalByRefObject will throw

Although it is not commonly mentioned, one can also return a SAO reference to an object using the same mechanism. In this case, the server returns the result of calling `Activator.GetObject` to the client instead of returning an instance of the object itself. The method `GetSaoObject` of `Spring.Remoting.RemoteFactory` performs this task. To use this functionality you need to tell the RemoteFactory the System.Type of the object, the URI of the object as well as the name that will be used by the client to obtain a reference to the SAO object. This information is captured using the class `Spring.Remoting.NamedServiceEntry`. The relevant propreties of this class are shown below

```
public class NamedServiceEntry : IInitializingObject
{
        . . .
        /// <summary>
        /// The name used to identify the SAO entry.
        /// </summary>
        public string Name
        { . . . }

        /// <summary>
        /// The service to export, populated via an object reference.
        /// </summary>
        public Type ServiceType
        { . . . }

        /// <summary>
        /// The uri of the well known object
        /// </summary>
        public string ServiceUrl
        { . . . }

        . . .

}
```

An collection of these parameters is used to configure the RemoteFactory using the property `ServiceEntries`. The configuration code from the quick start example is shown below.

```

<object name="namedSaoCalculatorService" type="Spring.Remoting.NamedServiceEntry">
  <property name="name" value="MyRemotedCalculatorFromFactory" />
  <property name="ServiceType" value="Server.RemotedCalculator, Server" />
  <property name="ServiceUrl" value="tcp://localhost:8005/MyRemotedCalculator" />
</object>

<object name="remoteFactory" type="Spring.Remoting.RemoteFactory, Spring.Services">

    <property name="serviceEntries">
        <list>
            <ref object="namedSaoCalculatorService"/>
        </list>
    </property>

</object>

<object name="publishedRemoteFactory" type="Spring.Remoting.SaoServiceExporter, Spring.Services">
        <property name="Service" ref="remoteFactory"/>
        <property name="ServiceName" value="RemoteFactory"/>
</object>
```

On the client side one calls the method GetSaoObject using the name specified with for the `NamedServiceEntry`, i.e. MyRemotedCalculatorFromFactory.

```
IRemoteFactory remoteFactory = (IRemoteFactory) ctx["remoteFactory"];

ICalculator calc = (ICalculator) remoteFactory.GetSaoObject("MyRemotedCalculatorFromFactory");
```

This approach results in less configuration on the client side as both CAO and SAO objects can be retrieved by name using only one registered remoting class, the RemoteFactory.

10.7. XML Schema for configuration

To be done.

10.8. Additional Resources

Two articles that describe the process of creating a standard SAO factory for returning CAO objects are Implementing Broker with .NET Remoting Using Client-Activated Objects on MSDN and Step by Step guide to CAO creation through SAO class factories on Glacial Components website.

Chapter 11. Spring Services

11.1. Introduction

The main goal of Spring.Services is to provide location transparency for business services. We believe that users should be able to implement services the simplest way possible, using service interfaces and implementations in the form of plain .Net classes. We also think that decision on how a particular service is exposed to clients should be a configuration concern and not an implementation concern.

Spring.Services provides infrastructure that allows you to expose *any* normal object as a web service, System.EnterpriseServices serviced component or remoting object using service exporter definitions in the configuration file.

We believe that this approach will also provide the easiest migration path to Indigo. As long as your services are coded against interfaces and implemented as regular classes, we should be able to implement Indigo exporter for them once the Indigo ships.

In the meantime, you get the benefit of being able to expose business services any way you find appropriate using existing exporters and Spring configuration.

11.2. Serviced Components

Services components in .NET are able to use COM+ services such as declarative and distributed transactions, role based security, object pooling messaging. To access these services your class needs to derive from the class `System.EnterpriseServices.ServicedComponent`, adorn your class and assemblies with relevant attributes, and configure your application by registering your serviced components with the COM+ catalog. The overall landscape of accessing and using COM+ services within .NET goes by the name .NET Enterprise Services.

Many of these services can be provided without the need to derive from a ServicedComponent though the use of Spring's Aspect-Oriented Programming functionality. Nevertheless, you may be interested in exporting your class as a serviced component and having client access that component in a location transparent manner. By using Spring's `ServicedComponentExporter`, `EnterpriseServicesExporter`and `ServicedComponentFactory` you can easily create and consume serviced components without having your class inherit from `ServicedComponent` and automate the manual deployment process that involves strongly signing your assembly and using the `regsvcs` utility.

Note that the following sections do not delve into the details of programming .NET Enterprise Services. An excellent reference for such information is Christian Nagel's "Enterprise Services with the .NET Framework"

11.3. Server Side

One of the main challenges for the exported to host a serviced component is the need for them to be contained within a physical assembly on the file system in order to be registered with the COM+ Services. To make things more complicated, this assembly has to be strongly named before it can be successfully registered.

Spring provides two classes that allow all of this to happen.

- `Spring.Enterprise.ServicedComponentExporter`
- `Spring.Enterprise.EnterpriseServicesExporter`

Let's say that we have a simple service interface and implementation class, such as these:

```
namespace MyApp.Services
{
    public interface IUserManager
    {
        User GetUser(int userId);
        void SaveUser(User user);
    }

    public class SimpleUserManager : IUserManager
    {
        private IUserDao userDao;
        public IUserDao UserDao
        {
            get { return userDao; }
            set { userDao = value; }
        }

        public User GetUser(int userId)
        {
            return UserDao.FindUser(userId);
        }

        public void SaveUser(User user)
        {
            if (user.IsValid)
            {
                UserDao.SaveUser(user);
            }
        }
    }
}
```

And the corresponding object definition for it in the application context config file:

```
<object id="userManager" type="MyApp.Services.SimpleUserManager">
    <property name="UserDao"><ref object="userDao"/></property>
            </object>
```

Let's say that we want to expose user manager as an serviced component so we can leverage its support for transactions. First we need to export our service using the exporter `ServicedComponentExporter` as shown below

```
<object id="MyApp.EnterpriseServices.UserManager" type="Spring.Enterprise.ServicedComponentExporter, Spring.Serv
    <property name="TargetName"><value>userManager</value></property>
    <property name="ClassAttributes">
        <list>
            <object type="System.EnterpriseServices.TransactionAttribute, System.EnterpriseServices"/>
        </list>
    </property>
    <property name="MethodAttributes">
        <dictionary>
            <entry key="*">
                <list>
                    <object type="System.EnterpriseServices.AutoCompleteAttribute, System.EnterpriseServices"/>
                </list>
            </entry>
        </dictionary>
    </property>
</object>
```

The exporter defined above will create a composition proxy for our SimpleUserManager class that extends `ServicedComponent` and delegates method calls to SimpleUserManager instance. It will also adorn the proxy class with a `TransactionAtribute` and all methods with an `AutoCompleteAttribute`.

The next thing we need to do is configure an exporter for the COM+ application that will host our new component:

```
<object id="MyComponentExporter" type="Spring.Enterprise.EnterpriseServicesExporter, Spring.Services">
    <property name="ApplicationName"><value>My COM+ Application</value></property>
    <property name="Description"><value>My enterprise services application.</value></property>
    <property name="AccessControl">
        <object type="System.EnterpriseServices.ApplicationAccessControlAttribute, System.EnterpriseServices">
            <property name="AccessChecksLevel"><value>ApplicationComponent</value></property>
        </object>
    </property>
    <property name="Roles">
        <list>
            <value>Admin : Administrator role</value>
            <value>User : User role</value>
            <value>Manager : Administrator role</value>
        </list>
    </property>
    <property name="Components">
        <list>
            <ref object="MyApp.EnterpriseServices.UserManager"/>
        </list>
    </property>
    <property name="Assembly"><value>MyComPlusApp</value></property>
            </object>
```

This exporter will put all proxy classes for the specified list of components into the specified assembly, sign the assembly, and register it with the specified COM+ application name. If application does not exist it will create it and configure it using values specified for Description, AccessControl and Roles properties.

11.4. Client Side

Because serviced component classes are dynamically generated and registered, you cannot instantiate them in your code using new operator. Instead, you need to use `Spring.Enterprise.ServicedComponentFactory` definition, which also allows you to specify configuration template for the component as well as the name of the remote server component is running on, if necessary. An example is shown below

```
<object id="enterpriseUserManager" type="Spring.Enterprise.ServicedComponentFactory, Spring.Services">
    <property name="Name"><value>MyApp.EnterpriseServices.UserManager</value></property>
    <property name="Template"><value>userManager</value></property>
            </object>
```

You can then inject this instance of the IUserManager into a client class and use it just like you would use original SimpleUserManager implementation. As you can see, by coding your services as plain .Net objects, against well defined service interfaces, you can achieve true location transparency for your services through configuration.

Chapter 12. Windows Services

12.1. Remarks

This is functionality that will be included after the 1.0 release. If you want to use these features please get the code from CVS http://opensource.atlassian.com/confluence/spring/display/NET/Project+Structure (instructions) or from the download section of the Spring.NET website that contains an .zip with the full CVS tree. In addition to this documentation you can refer to the example program located at `examples\Spring\Spring.Examples.WindowsService` to better understand the package. Please check the Spring.NET website for the latest updates to this document.

12.2. Introduction

Developers usually create Windows Services using the Visual Studio .NET wizard. While not difficult to do, this procedure is repetative and does not encourage separation between infrastructure code (windows service) and application code. This is generally considered a "bad thing" but you can certainly disagree.

As Spring.NET can provide an explicitly managed initialize/destroy lifecycle for singleton objects, there is a natural synergy with the lifecycle of a Windows service. As such, it could be very convenient to expose a Spring application context as a Windows service. Starting and stopping the service corresponds to creating and destroying an application context and its contained objects. This approach provides a high level means to declare what objects are created and destroyed when developing a Windows service.

To do that, Spring.NET requires the installation of one physical service able to run as services as many applications as you want - each a logical independent service in their own application domain. By default, the deployment and updating of the service can also be done by copying the relevant executables to a special directory.

The executable that at present provides these features is the `Spring.Services.WindowsService.Process.exe` assembly. It makes heavy use of classes and interfaces definde in the `Spring.Services.WindowsService.Common.dll` assembly. You should reference the common assembly it if you want to follow the advice on customization contained in the following sections

The benefits of this approach, a part from those given by separating infrastructure code and application code (a field where Spring.NET tries hard to succeed) is that you can think about installing a new service at client site by simply dropping a new application assembly in a remote directory[5].

12.3. The `Spring.Services.WindowsService.Process.exe` application

12.3.1. Installing

The installation can be done in two ways, using the .NET SDK `installutil.exe` tool or using the more mundane `Spring.Services.WindowsService.Installer.exe`; while the former is the standard, the latter is probably more flexible. It allows you to customize the name/display name of the service and has the ability to install multiple times the same assembly with different names. This can be useful in a number of scenarios, especially where you don't like, for some reasons, to run several different logical services under the same physical windows service.

Be aware of the fact that the service will be installed as running with the system account (installing with a specific user account seems a bit buggy on Windows XP)

That said, while `installutil` is documented on its own , the command line for `Spring.Services.WindowsService.Installer.exe` is as follow:

```
Spring.Services.WindowsService.Installer.exe

usage:
 install service-exe-path service-display-name service-name
 uninstall service-name      [i|u] service-exe-path service-display-name service-name
```

for example, to install, you can invoke it with the following:

```
... install  Spring.Services.WindowsService.Process.exe "Spring.Service Support" spring-service
```

and to uninstall it:

```
... uninstall  spring-service
```

12.3.2. Configuration

The standard .NET `.config` file can be used to tune some parameters of `Spring.Services.WindowsService.Process.exe`, (including log4net settings, for which it is recomended to consult the log4net documentation).

This file also define the context run by this process; here the file in its current beauty:

```

<configuration>

    <configSections>
        <section name="log4net" type="System.Configuration.IgnoreSectionHandler" />
        <sectionGroup name="spring">
            <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
            <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>

    <spring>
        <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
            <resource uri="file://~/service-process-definition.xml" />
        </context>
    </spring>

    <system.runtime.remoting>
    <application>
        <channels>
        <channel ref="http" port="1234" />
        </channels>
    </application>
    </system.runtime.remoting>

  <log4net>
    <!-- see http://logging.apache.org/log4net/release/manual/introduction.html -->
    <appender name="RollingFile" type="log4net.Appender.RollingFileAppender">
      <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%d [%t] %-5p %c{1} - %m%n" />
      </layout>
      <file value="logs/Spring.Service.Process.log" />
      <appendToFile value="true" />
      <maximumFileSize value="500KB" />
      <maxSizeRollBackups value="5" />
    </appender>
    <appender name="OutputDebugString" type="log4net.Appender.OutputDebugStringAppender">
      <layout type="log4net.Layout.PatternLayout">
```

```
        <conversionPattern value="%d{HH:mm:ss,fff} %-5p %c{2}(line:%L) - %m%n" />
      </layout>
    </appender>
    <root>
      <level value="OFF" />
    </root>
      <logger name="Spring.Services">
          <level value="ALL" />
      <appender-ref ref="RollingFile" />
      <appender-ref ref="OutputDebugString" />
      </logger>
  </log4net>

</configuration>
```

As you see, the context is defined in another file: let's review the objects it defines.

Firstly, it is worth notice that in order to 'localize' the service (i.e. to know where it is installed to use that directory as base for the deploy dir as in the above file) you should define an object like this: the name is not very important, it is important that it is an `IObjectFactoryPostProcessor` and so will be automatically applied to this application context:

```

<!-- provides access to the ${spring.services.process.base.dir} property -->
<object
    name="localizer"
    type="Spring.Services.WindowsService.Common.Localizer+ForProcess, Spring.Services.WindowsService.Common">
    <!-- change this to access the property with another prefix, for example ${foo.process.base.dir}
    <property name="prefix" value="foo"/>
    -->
</object>
```

In that object definition you can customize the prefix for the following string

```

public static readonly string SpringServicesProcessBaseDirFormat = "{0}.process.base.dir";
```

but you usually won't need it; the default value is

```

public static readonly string DefaultPrefix = "spring.services";
```

The sole important object defined by this context, i.e. the main object run by the service. The thing you can (and should) configure is the path to the folder you will use as the deploy location; the current definition, to avoid the need for a fully qualified path (e.g.: `c:\spring\services`) uses the properties made available by the `localizer` above:

```

<object
    name="service"
    type="Spring.Services.WindowsService.Common.DefaultService, Spring.Services.WindowsService.Common"
    init-method="Start"
    destroy-method="Stop">
    <property name="DeployPath" value="${spring.services.process.base.dir}/deploy"/>
</object>
```

The above object is then easily remoted using spring remoting utilities (please notice you should tune the remoting configuration listed in the standard .NET `.config` file, listed above):

```

<object name="remoted.service" type="Spring.Remoting.SaoServiceExporter, Spring.Services">
    <property name="Service" ref="service"/>
    <property name="ServiceName" value="SpringWindowsService.rem"/>
</object>
```

12.4. Running an application context as a windows service

If you package an application using the layout and conventions described here, you'll be able to run an application context as a Windows Service. The conventions used are modeled after those used by ASP.NET and are very easy to follow.

As already said, you'll have a Spring.NET application context running in a dedicated `AppDomain` hosted in a process running as a windows service: that process is able to run many application contexts simultaneously.

A complete application runable as service consists of a directory containing:

- The .NET configuration file `service.config`: this file should define your application context. Moreover this files will be used by the CLR to configure the application domain your application will run in, exactly as you expect. This file has the same role of ASP.NET `Web.config` file.
- Optional: an xml context file (`watcher.xml`) defining the watcher for your application.

  The watcher controls the automatic redeployment of the service and is discussed more in the following section.
- Recomended: along the lines of ASP.NET convention, a `bin` subdirectory containing all the assemblies your application needs; you can of course put your assemblies in the same directory where you put `service.config` but this is not encouraged ...

12.4.1. `service.config`

This is the standard .NET configuration file for the `AppDomain` that will host your application. It is semantically equivalent to the ASP.NET `Web.config` file. [6]

This file should also define your application context. When the service is started and stopped, the corresponding lifecycle methods are called on all the singletons defined. Of course, singletons are automatically instantiated by the application context when the service starts. For more information on lifecycles in Spring.NET see Section 3.5.1, “Lifecycle interfaces” Here an example taken from the tests:

```
<configuration>

    <configSections>
        <sectionGroup name="spring">
            <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
            <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>

    <appSettings>
        <add key="port" value="10"/>
    </appSettings>

    <spring>
        <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
            <resource uri="file://~/service.xml" />
        </context>
    </spring>

</configuration>
```

In this case the context is (again!) defined in another file (author's personal taste...) and the only 'service' is the

[6] `log4net` users please notice that (as of 1.2 beta 9) file appenders, when dealing with a relative file name, assume it is relative to the application domain code base. If you use log4net, it is very handy with the mechanics used by Spring Windows Service as every log file you will specify will be relative the directory containing the service application.

echo object (there is also a PropertyPlaceholderConfigurer just to make the example more realistic):

```
<objects xmlns="http://www.springframework.net"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects.xsd">

    <object name="echo"
      type="Spring.Services.WindowsService.Samples.Echo, Spring.Services.WindowsService.Tests"
      init-method="Start" destroy-method="Stop">
      <property name="port"><value>${port}</value></property>
    </object>

    <object id="configurer" type="Spring.Objects.Factory.Config.PropertyPlaceholderConfigurer, Spring.Core">
      <property name="locations">
         <list>
            <value>file://~/service.config</value>
         </list>
      </property>
      <property name="configSections">
         <list>
            <value>appSettings</value>
         </list>
      </property>
    </object>

</objects>
```

12.4.1.1. Let the application know where it is

There are some properties you may need at runtime, when your services will run, and you cannot know in advance. Hopefully, your xml definition file will allow to find the information it needs using some predefined variables you can use inside the service definition file with the standard NAnt style `${property name}` syntax.

These properies are:

- `spring.services.application.fullpath` that will be replaced with the full path of the application's `AppDomain.BaseDirectory`, i.e., where your application has been deployed;
- `spring.services.application.name` that will be replaced with the name of the subdirectory where the application has been deployed. Each application is deployed in its own directory, of course;

These properties are accessible only if one defines a localizer in the context like this (the localizer is a special `IObjectFactoryPostProcessor`:

```
<!-- provides access to the ${spring.services.application.*} properties -->
<object
    name="localizer"
    type="Spring.Services.WindowsService.Common.Localizer+ForApplication, Spring.Services.WindowsService.Common"
    <!-- change this to access the property with another prefix, for example ${foo.application.base.dir}
    -->
    <property name="prefix" value="myPrefix"/>
</object>
```

As you can see above, one can easily change the prefix used by that localizer and then write someting like:

```
<object name="simple"
  type="Spring.Services.WindowsService.Samples.Simple, Spring.Services.WindowsService.Tests"
  init-method="Start" destroy-method="Stop">
  <constructor-arg index="0" value="${myPrefix.application.name},${myPrefix.application.fullPath}"/>
  <property name="AppName">
    <value>${myPrefix.application.name}</value>
  </property>
  <property name="AppFullPath">
    <value>${myPrefix.application.fullpath}</value>
  </property>
```

```
</object>
```

12.4.2. `watcher.xml` - optional

This file allows you to optionally define a watcher for your application that can automatically redeploy it when needed.

The important thing to notice is that you can define your own application watcher, named `watcher`. Here it is used a watcher that listen for changes on the filesystem, configured to listen for some changes and to ignore others.

You can provide your own implementation defining an object named `watcher` that implements `Spring.Services.WindowsService.Common.Deploy.IApplicationWatcher`:

```
/// <summary>
Interface defining the contract for an application watcher.
<p>An application watcher is responsible to dispatch an
<see cref="IApplicationWatcherFactory">event</see> whenever it thinks the
application has been updated.</p>
<p>Usually it should not raise other kind of events
as they are usually raised by the <see cref="FileSystemApplicationWatcher"/>
that creates the watcher itself</p>
</summary>
<remarks>Usually instances of this interface need to be disposed</remarks>
<seealso cref="DeployEventArgs"/>
<seealso cref="IDeployLocation"/>
<seealso cref="DeployEventType.ApplicationUpdated"/>
<seealso cref="DeployEventAggregator"/>
<seealso cref="IDeployLocation"/>
<seealso cref="DeployEventType"/>
public interface IApplicationWatcher : IDisposable
{
    /// <summary>
    /// The watched application
    /// </summary>
    IApplication Application {get; }

    /// <summary>
    /// Start to watch the application, using the given dispatcher to
    /// dispatch deply events
    /// </summary>
    /// <param name="dispatcher">the dispatcher used to raise deploy events</param>
    void StartWatching (IDeployEventDispatcher dispatcher);

    /// <summary>
    /// Stop to watch the application.
    /// </summary>
    void StopWatching ();

    /// <summary>
    /// If physical events watched by this watcher should be filtered, this methods
    /// will allow to set filters that allows and disallows the event to be raised
    /// by the watcher.
    /// </summary>
    /// <param name="allows">the list of allowing filters</param>
    /// <param name="disallows">the list of disallowing filters</param>
    /// <seealso cref="FilteringSupport"/>
    /// <seealso cref="RegularExpressionFilter"/>
    void SetFilters (IList allows, IList disallows);
}
```

Please notice that this interface is currently a movable target and will probably change before the first official release (this will probably affect also the way a watcher will know about the application it should monitor, as shown in a few lines).

A tipical example of this file is give here:

```
<objects>

    <object name='watcher'
      type='Spring.Services.WindowsService.Common.Deploy.FileSystem.FileSystemApplicationWatcher'>
      <!--
      we can get access to the IApplication we are asked to monitor
      using a reference like the following
      -->
      <constructor-arg ref='.injected.application'/>

      <!-- sometimes the windows OS will decide to not give you the same case you see in explorer:
           in fact one should consider this OS case-insensitive with regard to file names ...
           The following property, true by default can however be tuned
      <property name="ignoreCase" value="false"/>
      -->

      <property name="includes">
        <list>
          <value>wwwroot/bin/*.*</value>
          <value>service.config</value>
          <value>service.xml</value>
        </list>
      </property>

      <!--
      <property name="excludes">
        <list>
          <value>Db/**/*.*</value>
          <value>Jobs</value>
          <value>Jobs</value>
          <value>**/*.log</value>
        </list>
      </property>
      -->

    </object>

</objects>
```

As you can see, if you need it, you can reference the `Spring.Services.WindowsService.Common.IApplication` object that your watcher should watch using the name `.injected.application`.

12.4.3. `bin` directory - optional

This is, by default, the folder where your assemblies are placed in the same way they are in an ASP.NET application.

Putting assemblies there is more a convention and maybe a good practice (they are isolated from other artifacts, but maybe you will prefer to use another directory (modify the `service.config` file accordingly) or the application directory directly (= `bin` parent).

Be aware of the fact that the process in which your application will run will have its own PATH environmental variable. As such don't expect to be successfull using dlls imported with [DllImport] if they are not in the system PATH of the hosting machine: while it is well known that the CLR fusion algorithm will not consider the PATH variable, you may be biten by assemblies using non-system dlls (SQLite and Firebird ADO.NET providers are good examples).

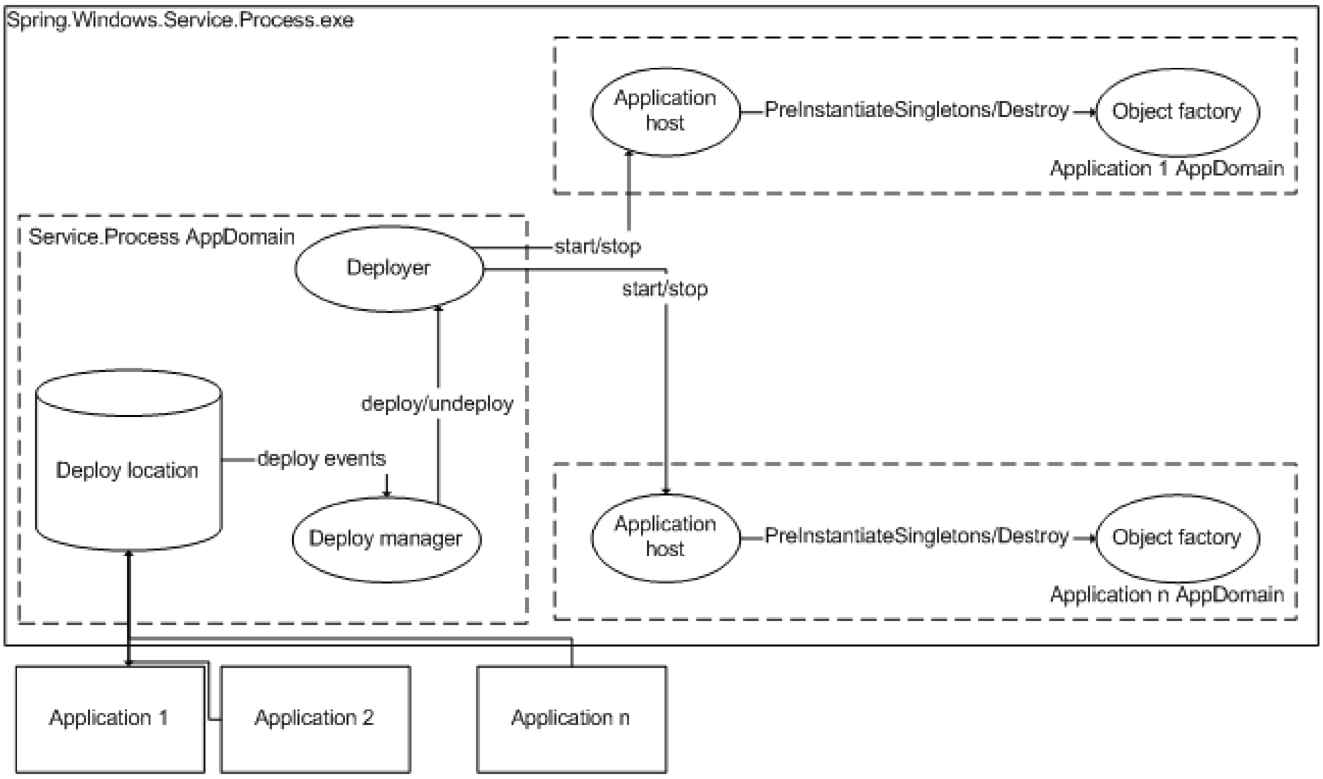
Reiterating, one can put assemblies in another directory under the application directory tree, and write the .NET configuration file (`service.config`) accordingly: .NET probing algorithm is always in place.

Please notice that it is not required that your application uses or include any of the Spring.NET assemblies: any object in any assembly, given it has lifecycle methods, can be run as a service: non invasive infrastructure support courtesy of Spring.NET!

12.5. Customizing or extending

It should be said that support for windows service has been initially developed with a clear but limited set of 'extension points' in mind, mainly related to the way you can deploy your services: deploy location (filesystem, zip archives, mailbox, urls, ...), (auto-)updating features, and so on.

To better understand the following discussion, the following figure depicts some of the inner details of `Spring.Services.WindowsService.Process.exe` at run-time:



12.5.1. The `.config` file

The executable `Spring.Services.WindowsService.Process.exe` is somewhat configured by the corresponding `.config` file. Please notice that this file is the most important extension point for windows service support, and it will probably be made more powerful and flexible in the future.

For applications deployed in the standard way (i.e. on the filesystem as explained above) the updating features are configured by the `watcher.xml` file, *if present*, as already seen.

There should be however, other ways to deploy your applications, maybe just as zip files dropped somewhere on the web or sent via e-mail.

For these scenarios, your deploy location will be something that implements `Spring.Services.WindowsService.Common.Deploy.IDeployLocation`.

Please notice that, while questionable, it actually entends `IDisposable` [7]:

```
/// <summary>
```

[7]this has been done as it is possible that a deploy location holds resources that should be released, for example network connections, lock files or the like

```
/// Interface defining how a deploy location should look like
/// </summary>
public interface IDeployLocation : IDeployEventSource, IDisposable
{
    /// <summary>
    /// The list of applications deployed at this location
    /// Usually non-valid applications are not listed
    /// </summary>
    /// <seealso cref="Application"/>
    IList Applications { get; }
}
```

```

/// <summary>
/// Interface defining the contract for an object acting as the source of
/// deploy events (application added, removed, updated)
/// </summary>
/// <seealso cref="DeployEventArgs"/>
/// <seealso cref="DeployEventHandler"/>
public interface IDeployEventSource
{
    /// <summary>
    /// The multicaster for deploy events
    /// </summary>
    event DeployEventHandler DeployEvent;
}
```

Chapter 13. Web framework

13.1. Introduction

(Available in 1.1)

It is the considered opinion of the Spring.NET team that ASP.NET already provides a decent implementation of the basic MVC pattern (with very few shortcomings) and that Spring.NET's web library must only provide those extensions that are necessary to overcome these shortcomings. The separation between HTML view code in an `.aspx` file and the code-behind class is a good one and is more than sufficient for the vast majority of enterprise applications; the code-behind class can be used as a controller (in the classic MVC sense), with event handlers effectively representing actions that the controller executes.

Having said that, event handlers in controllers (code-behind classes) should not have to deal with ASP.NET UI controls directly. Such event handlers should rather work with the data model of the page, represented either as a hierarchy of domain objects or an ADO.NET `DataSet`. It is for that reason that the Spring.NET team implemented bidirectional data binding to handle the mapping of values to and from the controls on a page to the underlying data model. Data binding transparently also takes care of type conversion and formatting, enabling application developers to work with fully typed data (domain) objects in the event handler's of code-behind files.

The flow of control through an application is another area of concern that is addressed by Spring.NET's web support. Typical ASP.NET applications will use `Response.Redirect` or `Server.Transfer` calls within `Page` logic to navigate to an appropriate page after an action is executed. This typically leads to hard-coded target URLs in the `Page`, which is never a good thing. Result mapping solves this problem by allowing application developers to specify aliases for action results that map to target URLs based on information in an external configuration file that easily can be edited.

Standard localization support is also limited in versions of ASP.NET prior to ASP.NET 2.0. Even though Visual Studio.NET 2003 generates a local resource file for each ASP.NET `Page` and user control, those resources are never used by the ASP.NET infrastructure. This means that application developers have to deal directly with resource managers whenever they need access to localized resources, which in the opinion of the Spring.NET team should not be the case. Spring.NET's web library (hereafter referred to as Spring.Web) adds comprehensive support for localization using both local resource files and global resources that are configured within and for a Spring.NET container.

Spring.Web also adds support for applying the dependency injection principle to one's ASP.NET `Pages` and `Controls`. This means that application developers can easily inject service dependencies into web controllers by leveraging the power of the Spring.NET IoC container.

In addition to the aforementioned features that can be considered to be the *'core'* features of the Spring.Web library, Spring.Web also ships with a number of other lesser features that might be useful to a large number of application developers. Some of these additional features include back-ports of ASP.NET 2.0 features that can be used with ASP.NET 1.1 (such as Master Page support); others are simply features that are generally useful, such as support for wizards (pages where the business process spans multiple distinct requests and / or form submissions).

In order to implement some of the above mentioned features the Spring.NET team had to extend (as in the object-oriented sense) the standard ASP.NET `Page` and `UserControl`. This means that in order to take advantage of the *full* feature stack of Spring.Web (most notably bidirectional data binding, localization and result mapping), your code-behind classes will have to extend Spring.Web specific base classes such as

`Spring.Web.UI.Page`; however, some very powerful features such as dependency injection for ASP.NET `Pages` can be leveraged without having to extend Spring.Web-specific base classes. It is worth stating that by taking advantage of *some* of the more useful features offered by Spring.Web you will be coupling the view tier of your application (s) to Spring.Web... the choice of whether or not this is appropriate is of course left to the end user.

Finally, please be aware that the standard Spring.NET distribution (as of v1.1) ships with a complete reference application, SpringAir. This reference application has a Spring.Web-enabled ASP.NET web frontend, so please do refer to it as you are reading this (reference) material (see Chapter 19, *SpringAir - Reference Application*).

13.2. Automatic context loading and hierarchical contexts

13.2.1. Configuration

Unsurprisingly, the Spring.Web library builds on top of the Spring.NET IoC container, and makes heavy use (internally) of the easy pluggability and standardized configuration afforded by the IoC container. This also means that all of the controllers (ASP.NET `Pages`) that make up a typical Spring.Web enabled application will be configured using the same standard Spring.NET XML configuration syntax. Spring.Web uses a custom `PageHandlerFactory` implementation to load and configure a Spring.NET IoC container, which is in turn used to locate an appropriate `Page` to handle a HTTP request.

The instantiation and configuration of the Spring.NET IoC container by the Spring.Web infrastructure is wholly transparent to application developers, who will typically never have to explicitly instatiate and configure an IoC container manually (by for example using the `new` operator in C#). In order to effect the transparent bootstrapping of the IoC container, the Spring.Web infrastructure requires the insertion of the following configuration snippet into each and every Spring.Web-enabled web application's root `web.config` file (the `verb` and `path` properties can of course be changed from the values that are shown below):

```
<system.web>
    <httpHandlers>
        <add verb="*" path="*.aspx" type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
    </httpHandlers>
    ...
</system.web>
```

Please note that this snippet of standard ASP.NET configuration is only required to be present in the *root* directory of each Spring.Web web application (i.e. in the `web.config` file present in the top level virtual directory of an ASP.NEt web application).

The above XML configuration snippet will direct the ASP.NET infrastructure to use Spring.NET's page factory, which will in turn create instances of the appropriate `.aspx` `Page`, (possibly) inject dependencies into said `Page` (as required), and then forward the handling of the request to said `Page`.

After the Spring.Web page factory is configured, you will also need to define a root application context by adding a Spring.NET configuration section to that same `web.config` file. The final configuration file should look a little like this (your exact configuration will no doubt vary in particulars)...

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

    <configSections>
        <sectionGroup name="spring">
            <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
        </sectionGroup>
    </configSections>
    <spring>
        <context type="Spring.Context.Support.WebApplicationContext, Spring.Web">
            <resource uri="~/Config/CommonObjects.xml"/>
            <resource uri="~/Config/CommonPages.xml"/>
        </context>
    </spring>

    <system.web>
        <httpHandlers>
            <add verb="*" path="*.aspx" type="Spring.Web.Support.PageHandlerFactory, Spring.Web"/>
        </httpHandlers>
    </system.web>

</configuration>
```

There are a few important points that need to be noted with regard to the above configuration...

1. You must define a custom configuration section handler for the `spring/context` element. If you use Spring.NET for many applications on the same web server, it might be easier to move the whole definition of the Spring.NET section group to your `machine.config` file.

2. Within the <spring> element you will need to define a root context and specify the `Type` of the container that is be be instantiated as `Spring.Context.Support.WebApplicationContext`. This will ensure that all of the features provided by Spring.Web are handled properly (such as request-scoped object definitions).

3. The resources that contain the object definitions that will be used within the web application (such as service or busniness tier objects) then need to be specified as child elements within the <context> element. Object definition resources can be fully-qualified paths or URLs, or non-qualified, as in the example above. Non-qualified resources will be loaded using the default resource type for the context, which for the `WebApplicationContext` is the `WebResource` type.

4. Please note that the object definition resources do not all have to be the same resource type (e.g. all `file://`, all `http://`, all `assembly://`, etc). This means that you can load some object definitions from resources embedded directly within application assemblies (`assembly://`), while continuing to load other object definitions from web resources that can be easily more easily edited.

13.2.2. Context Hierarchy

ASP.NET provides a hierarchical configuration mechanism by allowing application developers to override configuration settings specified at a higher level in the web application directory hierarchy with configuration settings specified at the lower level.

For example, a web application's root `web.config` file overrides settings from the (lower level) `machine.config` file. In the same fashion, settings specified within the `web.config` file within a subdirectory of a web application will override settings from the root `web.config` and so on. Lower level `web.config` files can also add settings of their own that were not previously defined anywhere.

Spring.Web leverages this ASP.NET feature to provide support for a context hierarchy. Your lower level Web.config files can be used to add new object definitions or to override existing ones per virtual directory.

What this means to application developers is that one can easily componentize an application by creating a virtual directory per component and creating a custom context for each component that contains the necessary configuration info for that particular context. The configuration for a lower level component will generally contain only those definitions for the pages that the component consists of and (possibly) overrides for some of the definitions from the root context (for example, menus).

Because each such lower level component will usually contain only a few object definitions, application developers are encouraged to embed those object definitions directly into the `web.config` for the lower level context instead of relying on an external resource containing object definitions. This is easily accomplished by creating a component `web.config` similar to the following one:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>

  <configSections>
    <sectionGroup name="spring">
       <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core"/>
    </sectionGroup>
  </configSections>

  <spring>
    <context type="Spring.Context.Support.WebApplicationContext, Spring.Web">
        <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
        <object type="MyPage.aspx" parent="basePage">
              <property name="MyRootService" ref="myServiceDefinedInRootContext"/>
              <property name="MyLocalService" ref="myServiceDefinedLocally"/>
              <property name="Results">
                 <!-- ... -->
              </property>
        </object>
        <object id="myServiceDefinedLocally" type="MyCompany.MyProject.Services.MyServiceImpl, MyAssembly"/>
    </objects>
  </spring>
</configuration>
```

The `<context/>` element seen above (contained within the `<spring/>` element) simply tells the Spring.NET infrastructure code to load (its) object definitions from the `spring/objects` section of the configuration file.

Needless to say, you can (and should) avoid the need to specify `<configSections/>` element by moving the configuration handler definition for the `<objects>` element to a higher level (root) `Web.config` file, or even to the level of the `machine.config` file if Spring.NET is to be used for multiple applications on the same server.

A very important point to be aware of is that this component-level context can reference definitions from it's parent context(s). Basically, if a referenced object definition is not found in the current context, Spring.NET will search all the ancestor contexts in the context hierarchy until it finds said object definition (or ultimately fails and throws an exception).

13.3. Dependency Injection for ASP.NET pages

Spring.Web builds on top of the featureset and capabilities of ASP.NET; one example of this can be seen the way that Spring.Web has used the code-behind class of the `Page` mechanism to satisfy the `Controller` portion of the MVC architectural pattern. In MVC-based (web) applications, the `Controller` is typically a thin wrapper around one or more service objects... in the specific case of Spring.Web, the Spring.NET team realized that it was very important that service object dependencies be easily injected into `Page Controllers`. Accordingly, Spring.Web provides first class support for dependency injection in ASP.NET `Pages`. This allows application developers to inject any required service object dependencies (and indeed any other dependencies) into their `Pages` using standard Spring.NET configuration instead of having to rely on custom service locators or manual object lookups in a Spring.NET application context.

Once an application developer has configured the Spring.NET web application context, said developer can easily create object definitions for the pages that compose that web application:

```
<objects>

  <object id="masterPage" type="~/Master.aspx" />
  <object id="basePage" abstract="true">
      <property name="Master" ref="masterPage"/>
  </object>
  <object type="Login.aspx">
      <property name="Authenticator" ref="authenticationService"/>
  </object>
  <object type="Default.aspx" parent="basePage"/>

</objects>
```

This example contains four definitions:

1. The first definition specifies the `Page` that is to be used as a Master Page.
2. The second definition is an abstract definition for the base page that many other pages in the application will inherit from. In this case, the definition simply specifies which page is to be referenced as the master page.
3. The third definition defines a login page that neither inherits from the base page nor references the master page. What it does show is how to inject a service object dependency into a page instance (the `authenticationService` is defined elsewhere).
4. The final definition defines a default application page. In this case it simply inherits from the base page in order to inherit the master page dependency, but apart from that it doesn't need any additional dependency injection configuration.

One thing that slightly differentiates the configuration of ASP.NET pages from the configuration of other .NET classes is in the value passed to the `type` attribute. As can be seen in the above configuration snippet the `type` name is actually the (path to the) `.aspx` file for the `Page`, relative to the directory context it is defined in. In the case of the above example, those definitions are in the root context so `Login.aspx` and `Default.aspx` also must be in the root of the web application's virtual directory. The master page is defined using an absolute path because it could conceivably be referenced from child contexts that are defined within subdirectories of the web application.

The astute reader may have noticed that the definitions for the `Login` and `Default` pages don't specify either of the `id` and `name` attributes. This is is marked contrast to typical object definitions in Spring.NET, where the `id` or `name` attributes are typically mandatory (although not always, as in the case of inner object definitions). This is actually intentional, because in the case of Spring.Web `Page Controller` instances one typically wants to use the name of the `.aspx` file name as the identifier. If an `id` is not specified, the Spring.Web infrastructure will simply use the name of the `.aspx` file as the object identifier (minus any leading path information, and minus the file extension too).

Of course, nothing prevents an application developer from specifying an `id` or `name` value explicitly; one use case when the explicit naming might be useful is when one wants to expose the same page multiple times using a slightly different configuration (Add / Edit pages for example).

13.3.1. Injecting Dependencies into Controls

Spring.Web also allows application developers to inject dependencies into controls (both user controls and standard controls) that are contained within a page. This can be accomplished in two ways - either globally for all the controls of a particular `Type`, or locally, for a specific control within a page.

In order to inject dependencies globally, the fully qualified `Type` name of the control needs to be used as the object definition identifier; please note that the assembly name that usually has to be supplied with fully qualified `Type` names in Spring.NET object definitions must not be specified.

```
<object id="MyProject.MyControl" abstract="true">
   <!-- inject dependencies here... -->
</object>
```

In order to inject dependencies locally, the control's `UniqueID` needs to used as the object definition identifier:

```
<object id="myContainerControl:myTargetControl" abstract="true">
   <!-- inject dependencies here... -->
</object>
```

In either case, be sure to mark the object definition as `abstract` (by adding `abstract="true"` to the attribute list of the `<object/>` element).

The easiest way to obtain a control's `UniqueID` is by turning tracing on for the application and looking for the `UniqueID` in the control hierarchy section of the resulting page trace. Yes, this is... unfortunate.

Please note that both global and local definitions for the same control can be specified, in which case the resulting definitions will be merged, with values from the local definition overriding values from the global definition as necessary.

Due to context scoping issues, application developers will probably want to define global control definitions in the root context so that dependencies are injected consistently throughout the web application. In contrast, local control definitions are better defined within the child context for the component, in order to avoid naming conflicts between different components.

The Limitations of Control Injection

Unfortunately, it is currently impossible to inject dependencies into controls before their `Init` event is fired, which means objects supplied via dependency injection cannot be used (referenced) within the control's `OnInit` method (or in any event handler for the `Init` event). This is because dependencies are injected **after** the `Init` event has fired, but **before** the `Load` event has fired; what this means in practical terms is that the event handler for the `Load` event is the first place in the code where dependency injected objects and values can safely be used.

13.4. Master Pages

Support for master pages in Spring.Web is very similar to the support for master pages in ASP.NET 2.0.

The idea is that a web developer can define a layout template for the site as a master page and specify content place holders that other pages can then reference and populate. A sample master page (`Master.aspx`) could look like this:

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<%@ Page language="c#" Codebehind="Master.aspx.cs" AutoEventWireup="false" Inherits="ArtFair.Web.UI.Master" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
    <head>
        <title>Master Page</title>
        <link rel="stylesheet" type="text/css" href="<%= Context.Request.ApplicationPath %>/css/styles.css">
        <spring:ContentPlaceHolder id="head" runat="server"/>
    </head>
    <body>
        <form runat="server">
            <table cellPadding="3" width="100%" border="1">
                <tr>
                    <td colspan="2">
                        <spring:ContentPlaceHolder id="title" runat="server">
                            <!-- default title content -->
                        </spring:ContentPlaceHolder>
                    </td>
                </tr>
                <tr>
                    <td>
                        <spring:ContentPlaceHolder id="leftSidebar" runat="server">
                            <!-- default left side content -->
                        </spring:ContentPlaceHolder>
                    </td>
                    <td>
                        <spring:ContentPlaceHolder id="main" runat="server">
                            <!-- default main area content -->
                        </spring:ContentPlaceHolder>
                    </td>
                </tr>
            </table>
        </form>
    </body>
</html>
```

As you can see from the above code, the master page defines the overall layout for the page, in addition to four content placeholders that pages can override. The master page can also include default content within the placeholder that will be displayed if a derived page does not override the placeholder.

A page (Child.aspx) that uses this master page might look like this:

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<%@ Page language="c#" Codebehind="Child.aspx.cs" AutoEventWireup="false" Inherits="ArtFair.Web.UI.Forms.Child"
<html>
    <body>

        <spring:Content id="leftSidebarContent" contentPlaceholderId="leftSidebar" runat="server">
            <!-- left sidebar content -->
        </spring:Content>

        <spring:Content id="mainContent" contentPlaceholderId="main" runat="server">
            <!-- main area content -->
        </spring:Content>

    </body>
</html>
```

The `<spring:Content/>` control in the above example uses the `contentPlaceholderId` attribute (property) to specify exactly which placeholder from the master page is to be overridden. Because this particular page does not define content elements for the head and title place holders, they will be displayed using the default content supplied by the master page.

Both the `ContentPlaceHolder` and `Content` controls can contain any valid ASP.NET markup: HTML, standard ASP.NET controls, user controls, etc.

VS.NET 2003 issue

Technically, the `<html>` and `<body>` tags from the previous example are not strictly necessary because they are already defined in the master page. However, if these tags are omitted, then Visual Studio.NET.2003 will complain about a schema and IntelliSense won't work, so it's much easier to work in the HTML view if those tags are included. They will be ignored when the page is rendered.

13.4.1. Linking child pages to their master

The `Spring.Web.UI.Page` class exposes a property called `Master` that can be used to supply a reference to the child page's owning master page. There is also a property called `MasterFile` that allows one to specify the master page using the file name of said master page.

The recommended way to do this is by leveraging the Spring.NET IoC container and creating definitions similar to the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<objects>

  <object id="masterPage" type="~/Master.aspx" />

  <object id="basePage" abstract="true">
      <property name="Master" ref="masterPage"/>
  </object>

  <object type="Child.aspx" parent="basePage">
      <!-- inject other objects that page needs -->
  </object>

</objects>
```

This approach alows application developers to change the master page being used for a number of pages within a web application. Of course, the master page can still be overridden on a per context or page basis by specifiying the `Master` or `MasterFile` properties directly.

13.5. Bidirectional data binding

A problem with the existing data binding support in ASP.NET is that that it is one-way only. It allows application developers to bind page controls to the data model and display information from said data model, but it doesn't allow for the extraction of values from the controls when the form is submitted.

Spring.Web adds such bidirectional data binding to ASP.NET using custom attributes that application developers can use to describe bindings, and also by providing data binding logic within a base `Page` class.

Please note that in order to take advantage of the bidirectional data binding support providd by Spring.Web, an application developer *will* have to couple his or her view layer to Spring.Web; this is because the bidirectional data binding support *requires* extending a Spring.Web `Page` (and not the usual `System.Web.UI.Page` class).

Spring.Web data binding is very easy to use. Application developers need only attribute control declarations within the `Page` that is to benefit from Spring.Web's bidirectional data binding support; this is perhaps best illustrated by an example:

```
public class UserRegistration : Spring.Web.UI.Page
    {
        [Binding("Text", "UserInfo.Email")]
        protected TextBox email;

        [Binding("Text", "UserInfo.Password")]
        protected TextBox password;

        protected TextBox passwordConfirmation;

        [Binding("Text", "UserInfo.FullName")]
        protected TextBox name;

        [Binding("Text", "UserInfo.Address.Street1")]
        protected TextBox street1;

        [Binding("Text", "UserInfo.Address.Street2")]
        protected TextBox street2;

        [Binding("Text", "UserInfo.Address.City")]
        protected TextBox city;

        [Binding("Text", "UserInfo.Address.State")]
        protected TextBox state;

        [Binding("Text", "UserInfo.Address.PostalCode")]
        protected TextBox postalCode;

        [Binding("Text", "UserInfo.Address.Country")]
        protected TextBox country;

        private User user;

        // in this example, this User property is the 'data model'...
        public User UserInfo
        {
            get { return user; }
            set { user = value; }
        }

        // the rest of the class definition omitted...
    }
```

The first parameter to the `Binding` attribute is the name of the control property that is being bound. The second argument is an object navigation expression that will be used both to get and set the value in the underlying data model. For more information on the supported object navigation expression syntax please do refer to Chapter 7, *Expression Evaluation*.

The `Binding` attribute also has a third (optional) parameter, `OneWay`. If the `OneWay` value is set to `true` (it is `false` by default), Spring.Web will *not* attempt to update the underlying data model with the value of the bound control's property. This is useful when a control is bound to a calculated read-only property in a data model.

Another (again optional) parameter is `Format`, which allows an application developer to specify how the boun value must be formatted for display. This parameter is usually used in conjunction with the `OneWay` attribute to provide custom formatting for date or numeric values. Any of the usual format expressions supported by the `String.Format` method can be used.

13.5.1. Type Conversion

The Spring.Web data binder will attempt to convert types when copying values from controls to data model (and vice versa). If the `Format` parameter is specified in an applied `Binding` attribute, the Spring.Web data binder will use the value of the said `Format` to convert the data model value to a `String`. Otherwise, the Spring.Web data binder will rely on the standard .NET `TypeConverter` mechanism for conversion.

13.5.2. Data Binding Events

Spring.Web's base `Page` class adds two events to the standard .NET page lifecycle - `DataBound` and `DataUnbound`.

The `DataUnbound` event is fired after the data model has been updated using values from the controls. It is fired right after the `Load` event and only on postbacks, because it doesn't make sense to update the data model using the controls' initial values.

The `DataBound` is fired after controls have been updated using values from the data model. This happens right before the `PreRender` event.

The fact that data model is updated immediately after the `Load` event and that controls are updated right before the `PreRender` event means that your event handlers will be able to work with a correctly updated data model, as they execute after the `Load` event, and that any changes you make to the data model within event handlers will be reflected in the controls immediately afterwards, as they (the controls) are updated prior to the actual rendering.

13.6. Localization

While recognizing that the .NET framework has excellent support for localization, the support within ASP.NET 1.x is somewhat incomplete.

Every `.aspx` page in an ASP.NET project has a resource file associated with it, but those resources are never used (by the current ASP.NEt infrastructure). ASP.NET 2.0 will change that and allow application developers to use local resources for pages. In the meantime, the Spring.NET team built support for using local pages resources into Spring.Web thus allowing application developers to start using ASP.NET 2.0-like page resources immediately.

Spring.Web supports several different approaches to localization within a web application, which can be mixed and matched as appropriate. Both push and pull mechanisms are supported, as well as the fallback to globally defined resources when a local resource cannot be found. Spring.Web also provides support for user culture management and image localization, which are described in the later sections.

Tip

Introductory material covering ASP.NET Globalization and Localization can be found at the following URLs; Globalization Architecture for ASP.NET and Localization Practices for ASP.NET 2.0 by Michele Leroux Bustamante.

13.6.1. Automatic Localization Using Localizers ("Push" Localization)

The central idea behind 'push' localization is that an application developer should be able to specify localization resources in the resource file for the page and have those resources automatically applied to the user controls on the page by the framework. For example, an application developer could define a page such as `UserRegistration.aspx`...

```
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<%@ Page language="c#" Codebehind="UserRegistration.aspx.cs"
    AutoEventWireup="false" Inherits="ArtFair.Web.UI.Forms.UserRegistration" %>
<html>
    <body>
        <spring:Content id="mainContent" contentPlaceholderId="main" runat="server">
            <div align="right">
                <asp:LinkButton ID="english" Runat="server" CommandArgument="en-US">English</asp:LinkButton>&nbs
                <asp:LinkButton ID="serbian" Runat="server" CommandArgument="sr-SP-Latn">Srpski</asp:LinkButton>
            </div>
            <table>
                <tr>
                    <td><asp:Label id="emailLabel" Runat="server"/></td>
                    <td><asp:TextBox id="email" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="passwordLabel" Runat="server"/></td>
                    <td><asp:TextBox id="password" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="passwordConfirmationLabel" Runat="server"/></td>
                    <td><asp:TextBox id="passwordConfirmation" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="nameLabel" Runat="server"/></td>
                    <td><asp:TextBox id="name" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="street1Label" Runat="server"/></td>
                    <td><asp:TextBox id="street1" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="street2Label" Runat="server"/></td>
                    <td><asp:TextBox id="street2" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="cityLabel" Runat="server"/></td>
                    <td><asp:TextBox id="city" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="stateLabel" Runat="server"/></td>
                    <td><asp:TextBox id="state" Runat="server" Width="30px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="postalCodeLabel" Runat="server"/></td>
                    <td><asp:TextBox id="postalCode" Runat="server" Width="60px"/></td>
                </tr>
                <tr>
                    <td><asp:Label id="countryLabel" Runat="server"/></td>
                    <td><asp:TextBox id="country" Runat="server" Width="150px"/></td>
                </tr>
                <tr>
                    <td colspan="2">
                        <asp:Button id="saveButton" Runat="server"/>&nbsp;
                        <asp:Button id="cancelButton" Runat="server"/>
                    </td>
                </tr>
            </table>
        </spring:Content>
    </body>
</html>
```

A close inspection of the above `.aspx` code reveals that none of the `Label` or `Button` controls have had a value assigned to the `Text` property. The values of the `Text` property for these controls are stored in the local resource file (of the page) using the following convention to identify the resource (string).

```
$this.controlId.propertyName
```

The corresponding local resource file, `UserRegistration.aspx.resx`, is shown below.

```
<root>
  <data name="$this.emailLabel.Text">
    <value>Email:</value>
  </data>
  <data name="$this.passwordLabel.Text">
    <value>Password:</value>
  </data>
  <data name="$this.passwordConfirmationLabel.Text">
    <value>Confirm password:</value>
  </data>
  <data name="$this.nameLabel.Text">
    <value>Full name:</value>
  </data>
  <data name="$this.street1Label.Text">
    <value>Street 1:</value>
  </data>
  <data name="$this.street2Label.Text">
    <value>Street 2:</value>
  </data>
  <data name="$this.cityLabel.Text">
    <value>City:</value>
  </data>
  <data name="$this.stateLabel.Text">
    <value>State:</value>
  </data>
  <data name="$this.postalCodeLabel.Text">
    <value>ZIP:</value>
  </data>
  <data name="$this.countryLabel.Text">
    <value>Country:</value>
  </data>
  <data name="$this.saveButton.Text">
    <value>$messageSource.save</value>
  </data>
  <data name="$this.cancelButton.Text">
    <value>$messageSource.cancel</value>
  </data>
</root>
```

The last two resource definitions require some additional explanation. In some cases it makes sense to apply a resource that is defined *globally* as opposed to locally. In this example, it makes better sense to define values for the `Save` and `Cancel` buttons globally as they will probably be used throughout the application.

The above example demonstrates how one can achieve that by defining a *resource redirection* expression as the value of a local resource by prefixing a global resource name with the following string.

```
$messageSource.
```

Taling the case of the above example, this will tell the localizer to use the `save` and `cancel` portions of the resource key as lookup keys to retrieve the actual values from a global message source. The important thing to remember is that one need only define a resource redirect once, typically in the invariant resource file – any lookup for a resource redirect will simply fall back to the invariant culture, and result in a global message source lookup using the correct culture.

Tip

To view the .resx file for a page, you may need to enable "Project/Show All Files" in Visual Studio.NET. When "Show All Files" is enabled, the .resx file appears like a "child" of the code-behind page.

When Visual Studio.NET creates the .resx file, it will include a `xds:schema`element and several `reshead` elements. Your data elements will follow the `reshead` elements. When working with the .resx files, you may want to chose "Open With" from the context menu and select the "Source Code" text editor.

13.6.2. Working with Localizers

In order to apply such resources automatically, a localizer needs to be injected into all such oages requiring this feature (typically accomplished using a base oage definition that other oages will inherit from). The injected localizer will inspect the resource file when the page is first requested, cache the resources that start with the `'$this'` marker string value, and apply the values to the controls that populate the page prior to the page being rendered.

A localizer is simply an object that implements the `Spring.Globalization.ILocalizer` interface. `Spring.Globalization.AbstractLocalizer` is provided as a convenient base class for localization: this class has one abstract method, `LoadResources`. This method must load and return a list of all the resources that must be automatically applied from the resource store.

Spring.NET ships with one concrete implementation of a localizer, `Spring.Globalization.Localizers.ResourceSetLocalizer`, that retrieves a list of resources to apply from the local resource file. Future releases of Spring.NET will provide other localizers that read resources from an XML file or even a flat text file that contains resource name-value pairs which will allow application developers to store resources within the files in a web application instead of as embedded resources in an assembly. Of course, if an application developer would rather store such resources in a database, he or she can write their own `ILocalizer` implementation that will load a list of resources to apply from a database.

As mentioned previously, one would typically configure the localizer to be used within an abstract base definition for those pages that require localization as shown below.

```
<object id="localizer" type="Spring.Globalization.Localizers.ResourceSetLocalizer, Spring.Core"/>

<object id="basePage" abstract="true">
    <description>
        Pages that reference this definition as their parent
        (see examples below) will automatically inherit following properties.
    </description>
    <property name="Localizer" ref="localizer"/>
</object>
```

Of course, nothing prevents an application developer from defining a different localizer for each page in the application; in any case, one can always override the localizer defined in a base (page) definition. Alternatively, if one does want any resources to be applied automatically one can completely omit the localizer definition.

One last thing to note is that Spring.NET `UserControl` instances will (by default) inherit the localizer and other localization settings from the page that they are contained within, but one can similarly also override that behavior using explicit dependency injection.

13.6.3. Applying Resources Manually ("Pull" Localization)

While automatic localization as described above works great for many form-like pages, it doesn't work nearly as well for the controls defined within any iterative controls because the IDs for such iterative controls are not fixed. It also doesn't work well in those cases where one needs to display the same resource multiple times within the same page. For example, think of the header columns for outgoing and return flights tables within the SpringAir application (see Chapter 19, *SpringAir - Reference Application*).

In these situations, one should use a pull-style mechanism for localization, which boils down to a simple `GetMessage` call as shown below.

```
<asp:Repeater id="outboundFlightList" Runat="server">
  <HeaderTemplate>
    <table border="0" width="90%" cellpadding="0" cellspacing="0" align="center" class="suggestedTable">
      <thead>
        <tr class="suggestedTableCaption">
          <th colspan="6">
            <%= GetMessage("outboundFlights") %>
          </th>
        </tr>
        <tr class="suggestedTableColnames">
          <th><%= GetMessage("flightNumber") %></th>
          <th><%= GetMessage("departureDate") %></th>
          <th><%= GetMessage("departureAirport") %></th>
          <th><%= GetMessage("destinationAirport") %></th>
          <th><%= GetMessage("aircraft") %></th>
          <th><%= GetMessage("seatPlan") %></th>
        </tr>
      </thead>
      <tbody>
  </HeaderTemplate>
```

The `GetMessage` method is available within both the `Spring.Web.UI.Page` and `Spring.Web.UI.UserControl` classes, and it will automatically fall back to a global message source lookup if a local resource is not found.

13.6.4. Localizing Images within a Web Application

Spring.Web provides an easy (and consistent) way to localize images within a web application. Unlike text resources, which can be stored within embedded resource files, XML files, or even a database, images in a typical web application are usually stored as files on the file system. Using a combination of directory naming conventions and a custom ASP.NET control, Spring.Web allows application developers to localize images within the page as easily as text resources.

The Spring.Web `Page` class exposes the `ImagesRoot` property, which is used to define the root directory where images are stored. The default value is 'Images', which means that the localizer expects to find an 'Images' directory within the application root, but one can set it to any value in the definition of the page.

In order to localize images, one needs to create a directory for each localized culture under the `ImagesRoot` directory as shown below.

```
/MyApp
   /Images
      /en
      /en-US
      /fr
      /fr-CA
      /sr-SP-Cyrl
      /sr-SP-Latn
      ...
```

Once an appropriate folder hierarchy in is place all one need do is put the localized images in the appropriate directories and make sure that different translations of the same image are named the same. In order to place a localized image on a page, one needs to use the `<spring:LocalizedImage>` as shown below.

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"
                AutoEventWireup="false" Inherits="SpringAir.Web.StandardTemplate" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <body>
    <spring:LocalizedImage id="logoImage" imageName="spring-air-logo.jpg" borderWidth="0" runat="server" />
  </body>
</html>
```

This control will find the most specific directory that contains an image with the specified name using standard localization fallback rules and the user's culture. For example, if the user's culture is `'en-US'`, the localizer will look for the `spring-air-logo.jpg` file in `Images/en-US`, then in `Images/en` and finally, if the image file has still not been found, in the root `Images` directory (which for all practical purposes serves as an invariant culture folder).

13.6.5. Global Resources

Global resources are (on a per-context basis) defined as a plain vanilla object definition using the reserved name of `'messageSource'`, which one can add to one's Spring.NET configuration file as shown below.

```
<object id="messageSource" type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
    <property name="ResourceManagers">
        <list>
            <value>MyApp.Web.Resources.Strings, MyApp.Web</value>
        </list>
    </property>
</object>
```

The global resources are cached within the Spring.NET `IApplicationContext` and are accessible through the Spring.NET `IMessageSource` interface.

The Spring.Web Page and UserControl classes have a reference to their owning IApplicationContext and it's associated IMessageSource. As such, they will automatically redirect resource lookups to a global message source if a local resource cannot be found.

Currently, the ResourceSetMessageSource is the only message source implementation that ships with Spring.NET.

13.6.6. User Culture Management

In addition to global and local resource management, Spring.Web also adds support for user culture management by exposing the current CultureInfo through the UserCulture property on the Page and UserControl classes.

The UserCulture property will simply delegate culture resolution to an implementation of Spring.Globalization.ICultureResolver interface. One can specify exactly which culture resolver to use by configuring the CultureResolver property of the Page class in the relevant object definition as shown below.

```
<object id="BasePage" abstract="true">
    <property name="Master" ref="MasterPage"/>
    <property name="CultureResolver">
        <object type="Spring.Globalization.Resolvers.CookieCultureResolver, Spring.Web"/>
    </property>
</object>
```

Several useful implementations of the ICultureResolver ship as part of Spring.Web, so it is unlikely that application developers will have to implement their own culture resolver. However, if one does have such a requirement, the resulting implementation should be fairly straightforward as there are only two methods that one need implement. The following sections discuss each available implementation of the ICultureResolver interface.

13.6.6.1. DefaultWebCultureResolver

This is default culture resolver implementation. It will be used if one does not specify a culture resolver for a page, or if one explicitly injects a DefaultWebCultureResolver into a page definition explicity. The latter case (explicit injection) is sometimes useful because it allows one to to specify a culture that should always be used by providing a value to the DefaultCulture property on the resolver.

The DefaultWebCultureResolver will first look at the DefaultCulture property and return its value if said property value is not null. If it is null, the DefaultWebCultureResolver will fall back to request header inspection, and finally, if no 'Accept-Lang' request headers are present it will return the UI culture of the currently executing thread.

13.6.6.2. RequestCultureResolver

This resolver works in a similar way to the DefaultWebCultureResolver with the exception that it always checks request headers *first*, and only then falls back to the value of the DefaultCulture property or the culture cound to the current thread.

13.6.6.3. SessionCultureResolver

This resolver will look for culture information in the user's session and return it if it finds one. If not, it will fall back to the behavior of the DefaultWebCultureResolver.

13.6.6.4. CookieCultureResolver

This resolver will look for culture information in a cookie, and return it if it finds one. If not, it will fall back to the behavior of the `DefaultWebCultureResolver`.

Warning

`CookieCultureResolver` will not work if your application uses `localhost` as the server URL, which is a typical setting in a development environment.

In order to work around this limitation you should use `SessionCultureResolver` during development and switch to `CookieCultureResolver` before you deploy application in a production. This is easily accomplished in Spring.Web (simply change the config file) but is something that you should be aware of.

13.6.7. Changing Cultures

In order to be able to change the culture application developers will need to use one of the culture resolvers that support culture changes, such as `SessionCultureResolver` or `CookieCultureResolver`. One could also write a custom `ICultureResolver` that will persist culture information in a database, as part of a user's profile.

Once that requirement is satisfied, all that one need do is to set the `UserCulture` property to a new `CultureInfo` object before the page is rendered. In the following `.aspx` example, there are two link buttons that can be used to change the user's culture. In the code-behind, this is all one need do to set the new culture. A code snippet for the code-behind file (`UserRegistration.aspx.cs`) is shown below.

```
protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.english.Command += new CommandEventHandler(this.SetLanguage);
    this.serbian.Command += new CommandEventHandler(this.SetLanguage);

    base.OnInit(e);
}

private void SetLanguage(object sender, CommandEventArgs e)
{
    this.UserCulture = new CultureInfo((string) e.CommandArgument);
}
```

13.7. Result Mapping

One of the problems evident in many ASP.NET applications is that there is no built-in way to externalize the flow of an application. The most common way of defining application flow is by hardcoding calls to the `Response.Redirect` and `Server.Transfer` methods within event handlers.

This approach is problematic because any changes to the flow of an application neccessitates code changes (with the attendant recompilation, testing, redeployment, etc). A much better way, and one that has been proven to work successfully in many MVC (Model-View-Controller) web frameworks is to provide the means to externalize the mapping of action results to target pages.

Spring.Web adds this functionality to ASP.NET by allowing one to define result mappings within the definition of a page, and to then simply use logical result names within event handlers to control application flow.

In Spring.Web, a logical result is encapsulated and defined by the `Result` class; because of this one can configure results just like any other object:

```
<objects>

    <object id="homePageResult" type="Spring.Web.Support.Result, Spring.Web">
        <property name="TargetPage" value="~/Default.aspx"/>
        <property name="Mode" value="Transfer"/>
        <property name="Parameters">
            <dictionary>
                <entry key="literal" value="My Text"/>
                <entry key="name" value="${UserInfo.FullName}"/>
                <entry key="host" value="${Request.UserHostName}"/>
            </dictionary>
        </property>
    </object>

    <object id="loginPageResult" type="Spring.Web.Support.Result, Spring.Web">
        <property name="TargetPage" value="Login.aspx"/>
        <property name="Mode" value="Redirect"/>
    </object>

    <object type="UserRegistration.aspx" parent="basePage">
        <property name="UserManager" ref="userManager"/>
        <property name="Results">
            <dictionary>
                <entry key="userSaved" value-ref="homePageResult"/>
                <entry key="cancel" value-ref="loginPageResult"/>
            </dictionary>
        </property>
    </object>

</objects>
```

The only property that you *must* supply a value for each and every result is the `TargetPage` property. The value of the `Mode` property can be either `Transfer` or `Redirect`, and defaults to `Transfer` if none is specified.

If one's target page requires parameters, one can define them using the `Parameters` dictionary property. One simply specifies either literal values or object navigation expressions for such parameter values; if one specifies an expression, this expression will be evaluated in the context of the page in which the result is being referenced... in the specific case of the above example, this means that any page that uses the `homePageResult` needs to expose a `UserInfo` property on the page class itself.

Parameters will be handled differently depending on the result mode. For redirect results, every parameter will be converted to a string, then URL encoded, and finally appended to a redirect query string. On the other hand, parameters for transfer results will be added to the `HttpContext.Items` collection before the request is transferred to the target page. This means that transfers are more flexible because any object can be passed as a parameter between pages. They are also more efficient because they don't require a round-trip to the client and back to the server, so transfer mode is recommended as the preferred result mode (it is also the current default).

The above example shows independent result object definitions, which are useful for global results such as a home- and login- page. `Result` definitions that are only going to be used by one page should be simply embedded within the definition of a page, either as inner object definitions or using a special shortcut notation for defining a result definition:

```
<object type="~/UI/Forms/UserRegistration.aspx" parent="basePage">
        <property name="UserManager">
            <ref object="userManager"/>
        </property>
        <property name="Results">
            <dictionary>
                <entry key="userSaved" value="redirect:UserRegistered.aspx?status=Registration Successful,user=$
                <entry key="cancel" value-ref="homePageResult"/>
            </dictionary>
        </property>
</object>
```

The short notation for the result must adhere to the following format...

```
[<mode>:]<targetPage>[?param1,param2,...,paramN]
```

There are two possible values for the `mode` value referred to in the above notation snippet; they are...

```
redirect
```

```
transfer
```

One thing to notice is that a comma is used instead of an ampersand to separate parameters; this is done so as to avoid the need for laborious ampersand escaping within an XML object definition. The use of the ampersand character is still supported if required, but one will then have to specify it using the well known & entity reference.

Once one has defined one's results, it is very simple to use them within the event handlers of one's pages (`UserRegistration.apsx.cs`)...

```
private void SaveUser(object sender, EventArgs e)
{
    UserManager.SaveUser(UserInfo);
    SetResult("userSaved");
}

public void Cancel(object sender, EventArgs e)
{
    SetResult("cancel");
}

protected override void OnInit(EventArgs e)
{
    InitializeComponent();

    this.saveButton.Click += new EventHandler(this.SaveUser);
    this.cancelButton.Click += new EventHandler(this.Cancel);

    base.OnInit(e);
}
```

One could of course further refactor the above example and use defined constants. This would be a good thing to do in the case of a logical result name such as "home" that is likely to be referenced by many pages.

13.8. Client-Side Scripting

ASP.NET has decent support for client-side scripting through the use of the `Page.RegisterClientScriptBlock` and `Page.RegisterStartupScript` methods.

However, neither of these two methods allows you to output a registered script markup within a `<head>` section of a page, which is (in many cases) exactly what you would like to do.

13.8.1. Registering Scripts within the head HTML section

Spring.Web adds several methods to enhance client-side scripting to the base `Spring.Web.UI.Page` class: `RegisterHeadScriptBlock` and `RegisterHeadScriptFile`, each with a few overrides. You can call these methods from your custom pages and controls in order to register script blocks and script files that must be included in the `<head>` section of the final HTML page.

The only additional thing that is required to make this work is that you use the `<spring:Head>` server-side control to define your `<head>` section instead of using the standard HTML `<head>` element. This is shown below.

```
<%@ Page language="c#" Codebehind="StandardTemplate.aspx.cs"
                AutoEventWireup="false" Inherits="SpringAir.Web.StandardTemplate" %>
<%@ Register TagPrefix="spring" Namespace="Spring.Web.UI.Controls" Assembly="Spring.Web" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
  <spring:Head runat="server" id="Head1">
    <title>
      <spring:ContentPlaceHolder id="title" runat="server">
        <%= GetMessage("default.title") %>
      </spring:ContentPlaceHolder>
    </title>
    <LINK href="<%= CssRoot %>/default.css" type="text/css" rel="stylesheet">
    <spring:ContentPlaceHolder id="head" runat="server"></spring:ContentPlaceHolder>
  </spring:Head>
  <body>
  ...
  </body>
</html>
```

The example above shows you how you would typically set-up a `<head>` section within a master page template in order to be able to change the title value and to add additional elements to the `<head>` section from the child pages using `<spring:ContentPlaceholder>` controls. However, only the `<spring:Head>` declaration is required in order for Spring.NET `Register*` scripts to work properly.

TODO : insert example

13.8.2. Adding CSS Definitions to the head Section

In a similar fashion, you can add references to CSS files, or even specific styles, directly to the `<head>` HTML section using `Page.RegisterStyle` and `Page.RegisterStyleFile` methods. The latter one simply allows you to include a reference to an external CSS file, while the former one allows you to define embedded style definitions by specifying the style name and definition as the parameters. The final list of style definitions registered this way will be rendered within the single embedded `style` section of the final HTML document.

TODO : insert example

13.8.3. Well-Known Directories

In order to make the manual inclusion of client-side scripts, CSS files and images easier, the Spring.Web `Page` class exposes several properties that help you reference such artifacts using absolute paths. This affords web application developers a great deal of convenince functionality straight out of the box if they stick to common conventions such as a web application (directory) structure..

These properties are `ScriptsRoot`, `CssRoot` and `ImagesRoot`. They have default values of `Scripts`, `CSS` and `Images`, which will work just fine if you create and use these directories in your web application root. However, if you prefer to place them somewhere else, you can always override default values by injecting new values into your page definitions (you will typically inject these values only in the base page definition, as they are normally shared by all the pages in the application). An example of such configuration is shown below:

```
<object id="basePage" abstract="true">
    <description>
        Convenience base page definition for all the pages.

        Pages that reference this definition as their parent (see the examples below)
        will automatically inherit following properties....

    </description>
    <property name="CssRoot" value="Web/CSS"/>
    <property name="ImagesRoot" value="Web/Images"/>
</object>
```

13.9. Wizards

TODO

13.10. Web Services Support

While the out-of-the-box support for web services in .NET is excellent, there are a few areas that the Spring.NET thought could use some improvement.

13.10.1. Server-side issues

One thing that the Spring.NET team didn't like much is that we had to have all these .asmx files lying around when all said files did was specify which class to instantiate to handle web service requests.

Second, the Spring.NET team also wanted to be able to use the Spring.NET IoC container to inject dependencies into our web service instances. Typically, a web service will rely on other objects, service objects for example, so being able to configure which service object implementation to use is very useful.

Last, but not least, the Spring.NET team did not like the fact that creating a web service is an implementation task. Most (although not all) services are best implemented as normal classes that use coarse-grained service interfaces, and the decision as to whether a particular service should be exposed as a remote object, web service, or even an enterprise (COM+) component, should only be a matter of configuration, and not implementation.

13.10.2. Client-side issues

On the client side, the main objection the Spring.NET team has is that client code becomes tied to a proxy *class*, and not to a service *interface*. Unless you make the proxy class implement the service interface manually, as described by Juval Lowy in his book "Programming .NET Components", application code will be less flexible and it becomes very difficult to plug in different service implementation in the case when one decides to use a new and improved web service implementation or a local service instead of a web service.

The goal for Spring.NET's web sevices support is to enable the easy generation of the client-side proxies that implement a specific service interface.

13.10.3. Removing the need for .asmx files

Unlike web pages, which use `.aspx` files to store presentation code, and code-behind classes for the logic, web services are completely implemented within the code-behind class. This means that .asmx files serve no useful purpose, and as such they should neither be necessary nor indeed required at all.

Spring.NET allows application developers to expose existing web services easily by registering a custom implementation of the `WebServiceHandlerFactory` class and by creating a standard Spring.NET object definition for the service.

By way of an example, consider the following web service...

```
namespace MyComany.MyApp.Services
{
    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService
    {
        [WebMethod]
        public string HelloWorld()
        {
            return "Hello World!";
        }
    }
}
```

This is just a standard class that has methods decorated with the `WebMethod` attribute and (at the class-level) the `WebService` attribute. Application developers can create this web service within Visual Studio.NET just like any other class.

All that one need to do in order to publish this web service is:

1. Register the `Spring.Web.Services.WebServiceFactoryHandler` as the HTTP handler for `*.asmx` requests within one's `web.config` file.

```
<system.web>
    <httpHandlers>
        <add verb="*" path="*.asmx" type="Spring.Web.Services.WebServiceHandlerFactory, Spring.Web"/>
    </httpHandlers>
</system.web>
```

Of course, one can register any other extension as well, but typically there is no need as Spring.NET's handler factory will behave exactly the same as a standard handler factory if said handler factory cannot find the object definition for the specified service name. In that case the handler factory will simply look for an .asmx file.

2. Create an object definition for one's web service.

```
<object name="HelloWorld.asmx" type="MyComany.MyApp.Services.HelloWorldService, MyAssembly" abstract="true"/>
```

Note that one is not absolutely required to make the web service object definition `abstract` (via the `abstract="true"` attribute), but this is a recommended best practice in order to avoid creating an unnecessary instance of the service. Because the .NET infrastructure creates instances of the target service object internally for each request, all Spring.NET needs to provide is the `System.Type` of the service class, which can be retrieved from the object definition even if it is amrked as `abstract`.

That's pretty much it – one can access this web service using the value specified for the `name` attribute of the object definition as the service name:

```
http://localhost/MyWebApp/HelloWorld.asmx
```

13.10.4. Injecting dependencies into web services

For arguments sake, let's say that we want to change the implementation of the `HelloWorld` method to make the returned message configurable.

One way to do it would be to use some kind of message locator to retrieve an appropriate message, but that locator needs to implemented. Also, it would certainly be an odd architecture that used dependency injection throughout the application to configure objects, but that resorted to the service locator approach when dealing with web services.

Ideally, one should be able to define a property for the message within one's web service class and have Spring.NET inject the message value into it:

```
namespace MyApp.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }

    [WebService(Namespace="http://myCompany/services")]
    public class HelloWorldService : IHelloWorld
    {
        private string message;
        public string Message
        {
            set { message = value; }
        }

        [WebMethod]
        public string HelloWorld()
        {
            return this.message;
        }
    }
}
```

The problem with standard SpringNET DI usage in this case is that Spring.NEt does not control the instantiation of the web service. This happens deep in the internals of the .NET framework, thus making it quite difficult to plug in the code that will perform the configuration.

The solution is to create a dynamic server-side proxy that will wrap the web service and configure it. That way, the .NET framework gets a reference to a proxy type from Spring.NET and instantiates it. The proxy then asks a Spring.NET application context for the actual web service instance that will process requests.

This proxying requires that one export the web service explicitly using the `Spring.Web.Services.WebServiceExporter` class; in the specific case of this example, one must also not forget to configure the `Message` property for said service:

```
<object id="HelloWorld" type="MyApp.Services.HelloWorldService, MyApp">
    <property name="Message" value="Hello, World!"/>
</object>

<object id="HelloWorldExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
    <property name="TargetName" value="HelloWorld"/>
</object>
```

The `WebServiceExporter` copies the existing web service and method attribute values to the proxy implementation (if indeed any are defined). Please note however that existing values can be overridden by setting properties on the `WebServiceExporter`.

Interface Requirements

In order to support some advanced usage scenarios, such as the ability to expose an AOP proxy as a web service (allowing the addition of AOP advices to web service methods), Spring.NET requires those objects that need to be exported as web services to implement a (service) interface.

Only methods that belong to an interface will be exported by the `WebServiceExporter`.

13.10.5. Exposing PONOs as Web Services

Now that we are generating a server-side proxy for the service, there is really no need for it to have all the attributes that web services need to have, such as `WebMethod`. Because .NET infrastructure code never really sees the "real" service, those attributes are redundant – the proxy needs to have them on its methods, because that's what .NET deals with, but they are not necessary on the target service's methods.

This means that we can safely remove the `WebService` and `WebMethod` attribute declarations from the service implementation, and what we are left with is a plain old .NET object (a PONO). The example above would still work, because the proxy generator will automatically add `WebMethod` attributes to all methods of the exported interfaces.

However, that is still not the ideal solution. You would looe information that the optional `WebService` and `WebMethod` attributes provide, such as service namespace, description, transaction mode, etc. One way to keep those values is to leave them within the service class and the proxy generator will simply copy them to the proxy class instead of creating empty ones, but that really does defeat the purpose.

A second, better way, is to set all the necessary values within the definoitn of the service exporter, like so...

```
<object id="HelloWorldExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
    <property name="TargetName" value="HelloWorld"/>
    <property name="Namespace" value="http://myCompany/services"/>
    <property name="Description" value="My exported HelloWorld web service"/>
    <property name="Methods">
        <dictionary>
            <entry key="HelloWorld">
                <object type="System.Web.Services.WebMethodAttribute, System.Web.Services">
                    <property name="Description" value="My Spring-configured HelloWorld method."/>
                    <property name="MessageName" value="ZdravoSvete"/>
                </object>
            </entry>
        </dictionary>
    </property>
</object>

// or, once configuration improvements are implemented...
<web:service targetName="HelloWorld" namespace="http://myCompany/services">
    <description>My exported HelloWorld web service.</description>
    <methods>
        <method name="HelloWorld" messageName="ZdravoSvete">
            <description>My Spring-configured HelloWorld method.</description>
        </method>
    </methods>
</web:service>
```

Based on the configuration above, Spring.NET will generate a web service proxy for all the interfaces implemented by a target and add attributes as necessary. This accomplishes the same goal while at the same time moving web service metadata from implementation class to configuration, which allows one to export pretty much *any* class as a web service.

One can also export only certain interfaces that a service class implements by setting the `Interfaces` property of the `WebServiceExporter`...

Distributed Objects Warning

Distributed Objects Warning

Just because you *can* export any object as a web service, doesn't mean that you *should*. Distributed computing principles still apply and you need to make sure that your services are not chatty and that arguments and return values are serializable.

You still need to exercise common sense when deciding whether to use web services (or remoting in general) at all, or if local service objects are all you need.

13.10.6. Exporting an AOP Proxy as a Web Service

It is often useful to be able to export an AOP proxy as a web service. For example, consider the case where you have a service that is wrapped with an AOP proxy that you want to access both locally and remotely (as a web service). The local client would simply obtain a reference to an AOP proxy directly, but any remote client needs to obtain a reference to an exported web service proxy, that delegates calls to an AOP proxy, that in turn delegates them to a target object while applying any configured AOP advice.

Effecting this setup is actually fairly straightforward; because an AOP proxy is an object just like any other object, all you need to do is set the `WebServiceExporter`'s `TargetName` property to the `id` (or indeed the `name` or `alias`) of the AOP proxy. The following code snippets show how to do this...

```
<object id="DebugAdvice" type="MyApp.AOP.DebugAdvice, MyApp"/>

<object id="TimerAdvice" type="MyApp.AOP.TimerAdvice, MyApp"/>

<object id="MyService" type="MyApp.Services.MyService, MyApp"/>

<object id="MyServiceProxy" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop">
    <property name="TargetName" value="MyService"/>
    <property name="IsSingleton" value="true"/>
    <property name="InterceptorNames">
        <list>
            <value>DebugAdvice</value>
            <value>TimerAdvice</value>
       </list>
    </property>
</object>

<object id="MyServiceExporter" type="Spring.Web.Services.WebServiceExporter, Spring.Web">
    <property name="TargetName" value="MyServiceProxy"/>
    <property name="Name" value="MyService"/>
    <property name="Namespace" value="http://myApp/webservices"/>
    <property name="Description" value="My web service"/>
    <property name="Interfaces">
        <list>
            <value>MyApp.Services.IMyService, MyApp</value>
        </list>
    </property>
</object>
```

That's it – every call to the methods of the exported web service will be intercepted by the target AOP proxy, which in turn will apply the configured debugging and timing advice to it.

13.10.7. Client-side Proxy Creation

The problem with the web-service proxy classes that are generated by VS.NET or the WSDL command line utility is that they don't implement a service interface. This tightly couples client code with web services and makes it impossible to change the implementation at a later date without modifying and recompiling the client.

Spring.NET provides a simple `IFactoryObject` implementation that will generate a *"proxy for proxy"* (however obtuse that may sound). Basically, the `Spring.Web.Services.WebServiceProxyFactory` class will create a proxy for the VS.NET- / WSDL-generated proxy that implements a specified service interface (thus solving the problem with the web-service proxy classes mentioned in the preceding paragraph).

At this point, an example may well be more illustrative in conveying what is happening; consider the following interface definition that we wish to expose as a web service...

```
namespace MyCompany.Services
{
    public interface IHelloWorld
    {
        string HelloWorld();
    }
}
```

In order to be able to reference a web service endpoint through this interface, you need to add a definition similar to the example shown below to your client's application context:

```
<object id="HelloWorld" type="Spring.Web.Services.WebServiceProxyFactory, Spring.Services">
    <property name="ProxyClass" value="MyCompany.WebServices.HelloWorld, MyClientApp"/>
    <property name="ServiceInterface" value="MyCompany.Services.IHelloWorld, MyServices"/>
</object>
```

What is important to notice is that the underlying implementation class for the web service does not have to implement the same `IHelloWorld` service interface... so long as matching methods with compliant signatures exist (a kind of duck typing), Spring.NET will be able to create a proxy and delegate method calls appropriately. If a matching method cannot be found, the Spring.NET infrastructure code will throw an exception.

That said, if you control both the client and the server it is probably a good idea to make sure that the web service class on the server implements the service interface, especially if you plan on exporting it using Spring.NET's `WebServiceExporter`, which requires an interface in order to work.

Chapter 14. Spring.NET miscellanea

14.1. Introduction

This chapter contains miscellanea information on features, goodies, caveats that does not belong to any paricular area.

14.2. PathMatcher

Note, Spring.Util.PathMatcher is currently only available in CVS, not the RC3 release. If you want to use these feature please get the code from CVS (instructions) or from the download section of the Spring.NET website that contains an .zip with the full CVS tree.

`Spring.Util.PathMatcher` provides `Ant/NAnt`-like path name matching features.

To do the match, you use the method:

```
public static bool Match(string pattern, string path)
```

If you want to decide if case is important or not use the method:

```
public static bool Match(string pattern, string path, bool ignoreCase)
```

14.2.1. General rules

To build your pattern, you use the `*`, `?` and `**` building blocks:

- `*`: matches any number of non slash characters;
- `?`: matches exactly 1 (one) non slash/dot character;
- `**`: matches any subdirectory, without taking care of the depth;

14.2.2. Matching filenames

A file name can be matched using the following notation:

```
foo?bar.*
```

matches:

```
fooAbar.txt
foo1bar.txt
foo_bar.txt
foo-bar.txt
```

does not match:

```
foo.bar.txt
foo/bar.txt
```

```
foo\bar.txt
```

The classical all files pattern:

```
*.*
```

matches:

```
foo.db
.db
foo
foo.bar.db
foo.db.db
db.db.db
```

does not match:

```
c:/
c:/foo.db
c:/foo
c:/.db
c:/foo.foo.db
//server/foo
```

14.2.3. Matching subdirectories

A directory name can be matched at any depth level using the following notation:

```
**/db/**
```

That pattern matches the following paths:

```
/db
//server/db
c:/db
c:/spring/app/db/foo.db
//Program Files/App/spaced dir/db/foo.db
/home/spring/spaced dir/db/v1/foo.db
```

but does not match these:

```
c:/spring/app/db-v1/foo.db
/home/spring/spaced dir/db-v1/foo.db
```

You can compose subdirectories to match like this:

```
**/bin/**/tmp/**
```

That pattern matches the following paths:

```
c:/spring/foo/bin/bar/tmp/a
c:/spring/foo/bin/tmp/a/b.c
```

but does not match these:

```
c:/spring/foo/bin/bar/temp/a
c:/tmp/foo/bin/bar/a/b.c
```

You can use more advanced patterns:

```
**/.spring-assemblies*/**
```

matches:

```
c:/.spring-assemblies
c:/.spring-assembliesabcd73xs
c:/app/.spring-assembliesabcd73xs
c:/app/.spring-assembliesabcd73xs/foo.dll
//server/app/.spring-assembliesabcd73xs
```

does not match:

```
c:/app/.spring-assemblie
```

14.2.4. Case does matter, slashes don't

.NET is expected to be a cross-platform development ... platform. So, `PathMatcher` will match taking care of the case of the pattern and the case of the path. For example:

```
**/db/**/*.DB
```

matches:

```
c:/spring/service/deploy/app/db/foo.DB
```

but does not match:

```
c:/spring/service/deploy/app/DB/foo.DB
c:spring/service/deploy/app/spaced dir/DB/foo.DB
//server/share/service/deploy/app/DB/backup/foo.db
```

If you do not matter about case, you should explicitly tell the `Pathmatcher`.

Back and forward slashes, in the very same cross-platform spirit, are not important:

```
spring/foo.bar
```

matches all the following paths:

```
c:\spring\foo.bar
c:/spring\foo.bar
c:/spring/foo.bar
/spring/foo.bar
\spring\foo.bar
```

Chapter 15. Visual Studio.NET Integration

15.1. XML Editing and Validation

(Available in 1.0)

Most of this section is well traveled territory for those familiar with editing XML files in their favorite XML editor. The XML configuration data that defines the objects that Spring will manage for you are validated against the Spring.NET XML Schema at runtime. The location of the XML configuration data to create an `IApplicationContext` can be any of the resource locations supported by Spring's `IResource` abstraction. (See Section 3.11, “The IResource abstraction” for more information.) To create an `IApplicationContext` using a "standalone" XML configuration file the custom configuration section in the standard .NET application configuration would read:

```
<spring>

  <context>
    <resource uri="file://objects.xml"/>
  </context>

</spring>
```

The XML document `objects.xml` should reference the Spring.NET XML schema as shown below

```
<?xml version="1.0" encoding="UTF-8"?>
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects.xsd
  <object id="..." type="...">
  ...
  </object>
  <object id="..." type="...">
  ...
  </object>
  ...
</objects>
```

To aid XML editors in associating the physical loation of the schema file with the xml document, the attribute `xsi:schemaLocation` is used. This also requires declaring the XMLSchema-instance namespace. The schema is also in the `doc/schema` directory of the distribution. The benefits of associating an XML document with a schema are to validate the document and to use editing features such as IntelliSense/content completion assistance.

The XML editor in VS.NET 2002/2003 *does not* recognize the `xsi:schemaLocation` as a hint to associate a schema with a XML document. Instead you should add the schema to either your VS.NET project or in your VS.NET installation directory. The VS.NET directory will be either

`C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml` for VS.NET 2003

or

`C:\Program Files\Microsoft Visual Studio .NET\Common7\Packages\schemas\xml` for VS.NET 2002

depending on your version of VS.NET. VS.NET 2005 *does* recognize the `xsi:schemaLocation`. If you would like to have a local fallback installation of the schema for use when you are not connected to the internet, the VS.NET 2005 directory for XML schemas is

```
C:\Program Files\Microsoft Visual Studio 8\Common7\Packages\schemas\xml
```

or similar.

It is typically easier to place the file in the well known location under the VS.NET installation directory than to copy the XSD file for each project you create. As simple aid in this task, you can use the the NAnt build file located in the `doc/schema` directory that comes with Spring and execute

```
nant
```

The default nant target will copy the file spring-object.xsd from the doc/schema directory to the appropriate VS.NET directory.

Once you have registered the schema with VS.NET you can adding only the namespace declaration to the objects element,

```
<objects xmlns="http://www.springframework.net">
```

will be enough to get IntelliSense and validation of the configuration file from within VS.NET. Alternatively, you can select the targetSchema property to be `Spring.NET Configuration` in the Property Sheet for the configuration file.

As shown in the section Section 3.7, “Interacting with the IObjectFactory” Spring.NET supports using .NET's application configuration file as the location to store the object defintions that will be managed by the object factory.

```
<configuration>

  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>

  <spring>

    <context>
      <resource uri="config://spring/objects"/>
    </context>

    <objects xmlns="http://www.springframework.net">
        ...
    </objects>

  </spring>

</configuration>
```

In this case VS.NET will still provide you with IntelliSense help but you will not be able to fully validate the document as the entire schema for App.config is not known the VS.NET (strange isn't that...) To be able to validate this document one would need to install the .NET Configuration File schema and and addition schema that incorporates the `<spring>` and `<context>` section in addition to the `<objects>` would need to be created.

Keep these tradeoffs in mind as you decide where to place the bulk of your configuration information. Conventional wisdom is do quick prototyping with App.config and use another IResource location, file or embedded assembly resource, for serious development.

15.2. Versions of XML Schema

The schema was updated from Spring 1.0.1 to 1.0.2 in order to support generics. The schema for version 1.0.1 is located under `http://www.springframework.net/xsd/1.0.1/` The schema for the latest version will always be located under `http://www.springframework.net/xsd/`

15.3. Integrated API help

As part of the installation process the API documentation for Spring.NET is registered with Visual Studio. There are two versions of the documentation, one for VS.NET 2002/2003 and the other for VS.NET 2005. They differ only in the format applied, VS.NET 2005 using the sexy new format. Enjoy!

Chapter 16. Quickstarts

16.1. Introduction

This chapter includes a grab bag of quickstart examples for using the Spring.NET framework.

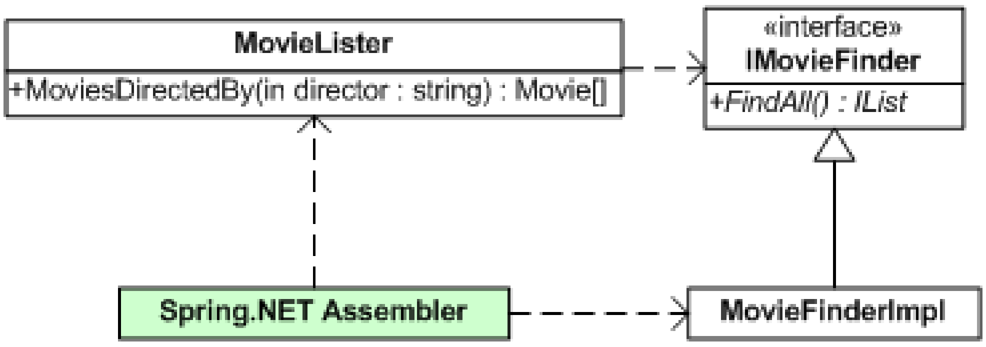
16.2. Movie Finder

The source material for this simple demonstration of Spring.NET's IoC features is lifted straight from Martin Fowler's article that discussed the ideas underpinning the IoC pattern. See Inversion of Control Containers and the Dependency Injection pattern for more information. The motivation for basing this quickstart example on said article is because the article is pretty widely known, and most people who are looking at IoC for the first time typically will have read the article (at the time of writing a simple Google search for 'IoC' yields the article in the first five results).

Fowler's article used the example of a search facility for movies to illustrate IoC and Dependency Injection (DI). The article described how a `MovieLister` object might receive a reference to an implementation of the `IMovieFinder` interface (using DI).

The `IMovieFinder` returns a list of all movies and the `MovieLister` filters this list to return an array of `Movie`objects that match a specified directors name. This example demonstrates how the Spring.NET IoC container can be used to supply an appropriate `IMovieFinder` implementation to an arbitrary `MovieLister` instance.

The C# code listings for the MovieFinder application can be found in the `examples/Spring/Spring.Examples.MovieFinder` directory off the top level directory of the Spring.NET distribution.



16.2.1. Getting Started - Movie Finder

The startup class for the MovieFinder example is the `MovieApp` class, which is an ordinary .NET class with a single application entry point...

```
using System;
namespace Spring.Examples.MovieFinder
{
  public class MovieApp
  {
    public static void Main ()
    {
    }
  }
}
```

What we want to do is get a reference to an instance of the `MovieFinder` class... since this is a Spring.NET example we'll get this reference from Spring.NET's IoC container, the `IApplicationContext`. There are a number of ways to get a reference to an `IApplicationContext` instance, but for this example we'll be using an `IApplicationContext` that is instantiated from a custom configuration section in a standard .NET application config file...

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <sectionGroup name="spring">
            <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
            <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>
    <spring>
        <context>
            <resource uri="config://spring/objects"/>
        </context>
        <objects>
            <description>An  example that demonstrates simple IoC features.</description>
        </objects>
    </spring>
</configuration>
```

The objects that will be used in the example application will be configured as XML `<object/>` elements nested inside the `<objects/>` element.

The body of the `Main` method in the `MovieApp` class can now be fleshed out a little further...

```

using System;
using Spring.Context;
...
    public static void Main ()
    {
      IApplicationContext ctx = ContextRegistry.GetContext();
    }
...
```

As can be seen in the above C# snippet, a `using` statements has been added to the `MovieApp` source. The `Spring.Context` namespace gives the application access to the `IApplicationContext` class that will serve as the primary means for the application to access the IoC container. The line of code...

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

... retrieves a fully configured `IApplicationContext` implementation that has been configured using the named `<objects/>` section from the application config file.

16.2.2. First Object Definition

As of yet, no objects have been defined in the application config file, so lets do that now. The very miminal XML definition for the `MovieLister` instance that we are going to use the application can be seen in the following XML snippet...

```
<objects>
    <object name="MyMovieLister"
      type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
    </object>
  </objects>
```

Notice that the full, assembly-qualified name of the `MovieLister` class has been specified in the `type` attribute of the object definition, and that the definition has been assigned the (unique) id of `MyMovieLister`.

Using this id, an instance of the object so defined can be retrieved from the `IApplicationContext` reference like so...

```
...
    public static void Main ()
    {
      IApplicationContext ctx = ContextRegistry.GetContext();
      MovieLister lister = (MovieLister) ctx.GetObject ("MyMovieLister");
    }
...
```

The `lister` instance has not yet had an appropriate implementation of the `IMovieFinder` interface injected into it. Attempting to use the `MoviesDirectedBy` method will most probably result in a nasty `NullReferenceException` since the `lister` instance does not yet have a reference to an `IMovieFinder`. The XML configuration for the `IMovieFinder` implementation that is going to be injected into the `lister` instance looks like this...

```
<objects>
    <object name="MyMovieFinder"
        type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
    </object>
</objects>
```

16.2.3. Setter Injection

What we want to do is inject the `IMovieFinder` instance identified by the `MyMovieFinder` id into the `MovieLister` instance identified by the `MyMovieLister` id, which can be accomplished using Setter Injection and the following XML...

```
<objects>
    <object name="MyMovieLister"
      type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
        <!-- using setter injection... -->
        <property name="movieFinder" ref="MyMovieFinder"/>
    </object>
    <object name="MyMovieFinder"
        type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
    </object>
</objects>
```

When the `MyMovieLister` object is retrieved from (i.e. instantiated by) the `IApplicationContext` in the application, the Spring.NET IoC container will inject the reference to the `MyMovieLister` object into the `MovieFinder` property of the `MyMovieLister` object. The `MovieFinder` object that is referenced in the application is then fully configured and ready to be used in the application to do what is does best... list movies by director.

```
...
    public static void Main ()
    {
      IApplicationContext ctx = ContextRegistry.GetContext();
      MovieLister lister = (MovieLister) ctx.GetObject ("MyMovieLister");
      Movie[] movies = lister.MoviesDirectedBy("Roberto Benigni");
      Console.WriteLine ("\nSearching for movie...\n");
      foreach (Movie movie in movies)
      {
          Console.WriteLine (
              string.Format ("Movie Title = '{0}', Director = '{1}'.",
              movie.Title, movie.Director));
      }
      Console.WriteLine ("\nMovieApp Done.\n\n");
    }
...
```

16.2.4. Constructor Injection

Let's define another implementation of the `IMovieFinder` interface in the application config file...

```
...
  <object name="AnotherMovieFinder"
      type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Spring.Examples.MovieFinder">
  </object>
...
```

This XML snippet describes an `IMovieFinder` implementation that uses a colon delimited text file as it's movie source. The C# source code for this class defines a single constructor that takes a `System.IO.FileInfo` as it's single constructor argument. As this object definition currently stands, attempting to get this object out of the `IApplicationContext` in the application with a line of code like so...

```
IMovieFinder finder = (IMovieFinder) ctx.GetObject ("AnotherMovieFinder");
```

will result in a fatal `Spring.Objects.Factory.ObjectCreationException`, because the `Spring.Examples.MovieFinder.ColonDelimitedMovieFinder` class does not have a default constructor that takes no arguments. If we want to use this implementation of the `IMovieFinder` interface, we will have to supply an appropriate constructor argument...

```
...
  <object name="AnotherMovieFinder"
      type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Spring.Examples.MovieFinder">
      <constructor-arg index="0" value="movies.txt"/>
  </object>
...
```

Unsurprisingly, the <constructor-arg/> element is used to supply constructor arguments to the constructors of managed objects. The Spring.NET IoC container uses the functionality offered by `System.ComponentModel.TypeConverter` specializations to convert the `movies.txt` string into an instance of the `System.IO.FileInfo` that is required by the single constructor of the `Spring.Examples.MovieFinder.ColonDelimitedMovieFinder` (see Section 4.3, "Type conversion" for more an in depth treatment concerning the automatic type conversion functionality offered by Spring.NET).

So now we have two implementations of the `IMovieFinder` interface that have been defined as distinct object definitions in the config file of the example application; if we wanted to, we could switch the implementation that the `MyMovieLister` object uses like so...

```
...
  <object name="MyMovieLister"
    type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
      <!-- lets use the colon delimited implementation instead -->
      <property name="movieFinder" ref="AnotherMovieFinder"/>
  </object>
  <object name="MyMovieFinder"
      type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
  </object>
  <object name="AnotherMovieFinder"
      type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Spring.Examples.MovieFinder">
      <constructor-arg index="0" value="movies.txt"/>
  </object>
...
```

Note that there is no need to recompile the application to effect this change of implementation... simply changing the application config file and then restarting the application will result in the Spring.NET IoC container injecting the colon delimited implementation of the `IMovieFinder` interface into the `MyMovieLister` object.

16.2.5. Summary

This example application is quite simple, and admittedly it doesn't do a whole lot. It does however demonstrate the basics of wiring together an object graph using an intuitive XML format. These simple features will get you through pretty much 80% of your object wiring needs. The remaining 20% of the available configuration options are there to cover corner cases such as factory methods, lazy initialization, and suchlike (all of the configuration options are described in detail in the Chapter 3, *Objects, Object Factories, and Application Contexts*).

16.2.6. Logging

Often enough the first use of Spring.NET is also a first introduction to log4net. To kick start your understanding of log4net this section gives a quick overview. The authoritative place for information on log4net is the log4net website. Other good online tutorials are Using log4net (OnDotNet article) and Quick and Dirty Guide to Configuring Log4Net For Web Applications. Spring.NET is using version 1.2.9 where as most of the documentation out there is for verison 1.2.0. There have been some changes between the two so always double check at the log4net web site for definitive information. Also note that we are investigating using a "commons" logging library so that Spring.NET will not be explicity tied to log4net but will be able to use other logging packages such as NLog and Microsoft enterprise logging application block.

The general usage pattern for log4net is to configure your loggers, (either in App/Web.config or a seperate file), initialize log4net in your main application, declare some loggers in code, and then log log log. (Sing along...) We are using App.config to configure the loggers. As such, we declare the log4net configuration section handler as shown below

```
<section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
```

The corresponding configuration section looks like this

```
<log4net>
  <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
     <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%date [%thread] %-5level %logger - %message%newline" />
     </layout>
  </appender>

  <!-- Set default logging level to DEBUG -->
  <root>
     <level value="DEBUG" />
     <appender-ref ref="ConsoleAppender" />
  </root>

  <!-- Set logging for Spring to INFO.  Logger names in Spring correspond to the namespace -->
  <logger name="Spring">
     <level value="INFO" />
  </logger>
</log4net>
```

The appender is the output sink - in this case the console. There are a large variety of output sinks such as files, databases, etc. Refer to the log4net Config Examples for more information. Of interest as well is the PatternLayout which defines exactly the information and format of what gets logged. Usually this is the date, thread, logging level, logger name, and then finally the logging message. Refer to PatternLayout Documentation for information on how to customize.

The logging name is up to you to decide when you declare the logger in code. In the case of this example we used the convention of giving the logging name the name of the fully qualified class name.

```
private static readonly ILog LOG = LogManager.GetLogger(typeof (MovieApp));
```

Other conventions are to give the same logger name across multiple classes that constitute a logical component or subsystem within the application, for example a data access layer. One tip in selecting the pattern layout is to shorten the logging name to only the last 2 parts of the fully qualified name to avoid the message sneaking off to the right too much (where can't see it) because of all the other information logged that precedes it. Shortening the logging name is done using the format %logger{2}.

To initialize the logging system add the following to the start of your application

```
XmlConfigurator.Configure();
```

Note that if you are using or reading information on version 1.2.0 this used to be called DOMConfigurator.Configure();

The logger sections associate logger names with logging levels and appenders. You have great flexibilty to mix and match names, levels, and appenders. In this case we have defined the root logger (using the special tag root) to be at the debug level and have an console sink. We can then specialize other loggers with different setting. In this case, loggers that start with "Spring" in their name are logged at the info level z`and also sent to the console. Setting the value of this logger from INFO to DEBUG will show you detailed logging information as the Spring container goes about its job of creating and configuring your objects. Coincidentally, the example code itself uses Spring in the logger name, so this logger also controls the output level you see from running MainApp. Finally, you are ready to use the simple logger api to log, i.e.

```
LOG.Info("Searching for movie...");
```

Logging exceptions is another common task, which can be done using the error level

```
try {
   //do work
{
catch (Exception e)
{
  LOG.Error("Movie Finder is broken.", e);
}
```

16.3. ApplicationContext and IMessageSource

16.3.1. Introduction

The example program `Spring.Examples.AppContext` shows the use the application context for text localization, retrieving objects contained in ResourceSets, and applying the values of embedded resource properties to an object. The values that are retrieved are displayed in a window.

The application context configuration file contains an object definition with the name `messageSource`. of the type `Spring.Context.Support.ResourceSetMessageSource` which implements the interface `IMessageSource`. This interface provides various methods for retrieving localized resources such as text and images as described in Section 3.14.2, “Using the IMessageSource”. When creating an instance of IApplicationContext, an object with the name 'messageSource' is searched for and used as the implementation for the context's IMessageSource functionality.

The `ResourceSetMessageSource` takes a list of ResourceManagers to define the collection of culture-specific resources. The ResourceManager can be contructed in two ways. The first way is to specifying a two part string consisting of the base resource name and the containing assembly. In this example there is an embedded resource file, Images.resx in the project. The second way is to use helper factory class `ResourceManagerFactoryObject` that takes a resource base name and assembly name as properties. This second way of specifying a ResourceManager is useful if you would like direct access to the ResourceManager in other parts of your application. In the example program an embedded resource file, MyResource.resx and a Spanish specific resource file, MyResources.es.resx are declared in this manner. The corresponding XML fragment is shown below

```
...
        <object name="messageSource" type="Spring.Context.Support.ResourceSetMessageSource, Spring.Core">
          <property name="resourceManagers">
            <list>
              <value>Spring.Examples.AppContext.Images, Spring.Examples.AppContext</value>
              <ref object="myResourceManager"/>
            </list>
          </property>
        </object>

        <object name="myResourceManager" type="Spring.Objects.Factory.Config.ResourceManagerFactoryObject, Sprin
          <property name="baseName">
            <value>Spring.Examples.AppContext.MyResource</value>
          </property>
          <property name="assemblyName">
            <value>Spring.Examples.AppContext</value>
          </property>
        </object>
...
```

The main application creates the application context and then retrieves various resources via their key names. In the code all the key names are declared as static fields in the class `Keys.` The resource file Images.resx contains image data under the key name `bubblechamber` (aka Keys.BUBBLECHAMBER). The code `Image image = ctx.GetResourceObject(Keys.BUBBLECHAMBER) as Image;` is used to retrieve the image from the context. The resource files MyResource.resx contains a text resource, `Hello {0} {1}` under the key name `HelloMessage` (aka Keys.HELLO_MESSAGE) that can be used for string text formatting purposes. The example code

```
string msg = ctx.GetMessage(Keys.HELLO_MESSAGE,
                            CultureInfo.CurrentCulture,
                            "Mr.", "Anderson");
```

retrieves the text string and replaces the placeholders in the string with the passed argument values resulting in the text, "Hello Mr. Anderson". The current culture is used to select the resource file MyResource.resx. If instead the Spanish culture is specified

```
CultureInfo spanishCultureInfo = new CultureInfo("es");
string esMsg = ctx.GetMessage(Keys.HELLO_MESSAGE,
                              spanishCultureInfo,
                              "Mr.", "Anderson");
```

Then the resource file MyResource.es.resx is used instead as in standard .NET localization. Spring is simply delegating to .NET ResourceManager to select the appropriate localized resource. The Spanish version of the resource differs from the English one in that the text under the key `HelloMessage` is `Hola {0} {1}` resulting in the text "`Hola Mr. Anderson`".

As you can see in this example, the title "Mr." should not be used in the case of the spanish localization. The title can be abstracted out into a key of its own, called `FemaleGreeting` (aka Keys.FEMALE_GREETING). The replacement value for the message argument {0} can then be made localization aware by wrapping the key in a convenience class DefaultMessageResolvable. The code

```
string[] codes = {Keys.FEMALE_GREETING};
DefaultMessageResolvable dmr = new DefaultMessageResolvable(codes, null);

msg = ctx.GetMessage(Keys.HELLO_MESSAGE,
                     CultureInfo.CurrentCulture,
                     dmr, "Anderson");
```

will assign msg the value, Hello Mrs. Anderson, since the value for the key `FemaleGreeting` in MyResource.resx is 'Mrs.' Similarly, the code

```
esMsg = ctx.GetMessage(Keys.HELLO_MESSAGE,
                       spanishCultureInfo,
                       dmr, "Anderson");
```

will assign esMsg the value, Hola Senora Anderson, since the value for the key `FemaleGreeting` in MyResource.es.resx is 'Senora'.

Localization can also apply to object and not just strings. The .NET 1.1 framework provides the utility class ComponentResourceManager that can apply multiple resource values to object properties in a performant manner. (VS.NET 2005 makes heavy use of this class in the code it generates for winform applications.) The example program has a simple class, Person, that has an integer property Age and a string property Name. The resource file, Person.resx contains key names that follow the pattern, person.<PropertyName>. In this case it contains person.Name and person.Age. The code to assign these resource values to an object is shown below

```
Person p = new Person();
ctx.ApplyResources(p, "person", CultureInfo.CurrentUICulture);
```

While you could also use the Spring itself to set the properties of these objects, the configuration of simple properties using Spring will not take into account localization. It may be convenient to combine approaches and use Spring to configure the Person's object references while using IApplicationContext inside an AfterPropertiesSet callback (see IInitializingObject) to set the Person's culture aware properties.

16.4. ApplicationContext and IEventRegistry

16.4.1. Introduction

The example program `Spring.Examples.EventRegistry` shows how to use the application context to wire .NET events in a loosely coupled manner.

Loosely coupled eventing is normally associated with Message Oriented Middleware (MOM) where a daemon process acts as a message broker between other independent processes. Processes communicate indirectly with each other by sending messages though the message broker. The process that initiates the communication is known as a publisher and the process that receives the message is known as the subscriber. By using an API specific to the middleware these processes register themselves as either publishers or subscribers with the message broker. The communication between the publisher and subscriber is considered loosley coupled because neither the publisher or subscriber has a direct reference to each other, the messages brokers acts as an intermediary between the two processes. The `IEventRegistry` is the analogue of the message broker as applied to .NET events. Publishers are classes that invoke a .NET event, subscribers are the classes that register interest in these events, and the messages sent between them are instances of System.EventArgs. The implementation of `IEventRegistry` determine the exact semantics of the notification style and coupling between subscribers and publishers.

The `IApplicationContext` interface extends the `IEventRegistry` interface and implementations of `IApplicationContext` delegate the event registry functionality to an instance of `Spring.Objects.Events.Support.EventRegistry`. `IEventRegistry` is a simple inteface with one publish method and two subscribe methods. Refer to Section 3.14.4, “Loosely coupled events” for a reminder of their signatures. The `Spring.Objects.Events.Support.EventRegistry` implementation is essentially a convenience to decouple the event wiring process between publisher and subscribers. In this implementation, after the event wiring is finished, publishers are directly coupled to the subscribers via the standard .NET eventing mechanisms. Alternate implementations could increase the decoupling futher by having the event registry subscribe to the events and be responsible for then notifying the subscribers.

In this example the class `MyClientEventArgs` is a subclass of `System.EventArgs` that defines a string property EventMessage. The class `MyEventPublisher` defines a public event with the delgate signature `void SimpleClientEvent( object sender, MyClientEventArgs args )` The method `void ClientMethodThatTriggersEvent1()` fires this event. On the subscribing side, the class `MyEventSubscriber` contains a method, `HandleClientEvents` that matches the delegate signature and has a boolean property which is set to true if this method is called.

The publisher and subscriber classes are defined in an application context configuration file but that is not required in order to participate with the event registry. The main program, `EventRegistryApp` creates the application context and asks it for an instance of `MyEventPublisher` The publisher is registered with the event registry via the call, `ctx.PublishEvents( publisher )`. The event registry keeps a reference to this publisher for later use to register any subscribers that match its event signature. Two subscribers are then created and one of them is wired to the publisher by calling the method `ctx.Subscribe( subscriber, typeof(MyEventPublisher) )` Specifying the type indicates that the subscriber should be registered only to events from objects of the type `MyEventPublisher`. This acts as a simple filtering mechanism on the subscriber.

The publisher then fires the event using normal .NET eventing semantics and the subscriber is called. The subscriber prints a message to the console and sets a state variable to indicate it has been called. The program then simply prints the state variable of the two subscribers, showing that only one of them (the one that registered with the event registry) was called.

16.5. Pooling example

The idea is to build an executor backed by a pool of `QueuedExecutor`: this will show how Spring.NET provides some useful low-level/high-quality reusable threading and pooling abstractions. This executor will provide parallel executions (in our case `grep`-like file scans). *Note: This example is not in the 1.0.0 release to its use of classes in the Spring.Threading namespace scheduled for release in Spring 1.1. To access ths example please get the code from CVS (instructions) or from the download section of the Spring.NET website that contains an .zip with the full CVS tree.*

Some information on `QueuedExecutor` is helpful to better understand the implementation and to possibly disagree with it. Keep in mind that the point is to show how to develop your own object-pool.

A `QueuedExecutor` is an executor where `IRunnable` instances are run serialy by a worker thread. When you `Execute` with a `QueuedExecutor`, your request is queued; at some point in the future your request will be taken and executed by the worker thread: in case of error the thread is terminated. However this executor recreates its worker thread as needed.

Last but not least, this executor can be shut down in a few different ways (please refer to the Spring.NET SDK documentation). Given its simplicity, it is very powerful.

The example project `Spring.Examples.Pool` provides an implementation of a pooled executor, backed by n instances of `Spring.Threading.QueuedExecutor`: please ignore the fact that `Spring.Threading` includes already a very different implementation of a `PooledExecutor`: here we wanto to use a pool of `QueuedExecutors`.

This executor will be used to implement a parallel recursive `grep`-like console executable.

16.5.1. Implementing `Spring.Pool.IPoolableObjectFactory`

In order to use the `SimplePool` implementation, the first thing to do is to implement the `IPoolableObjectFactory` interface. This interface is intended to be implemented by objects that can create the type of objects that should be pooled. The `SimplePool` will call the lifecycle methods on `IPoolableObjectFactory` interface (`MakeObject, ActivateObject, ValidateObject, PassivateObject, and DestroyObject`) as appropriate when the pool is created, objects are borrowed and returned to the pool, and when the pool is destroyed.

In our case, as already said, we want to to implement a pool of `QueuedExecutor`. Ok, here the declaration:

```
public class QueuedExecutorPoolableFactory : IPoolableObjectFactory
{
```

the first task a factory should do is to create objects:

```
object IPoolableObjectFactory.MakeObject()
{
    // to actually make this work as a pooled executor
    // use a bounded queue of capacity 1.
    // If we don't do this one of the queued executors
    // will accept all the queued IRunnables as, by default
    // its queue is unbounded, and the PooledExecutor
    // will happen to always run only one thread ...
    return new QueuedExecutor(new BoundedBuffer(1));
}
```

and should be also able to destroy them:

```
void IPoolableObjectFactory.DestroyObject(object o)
{
    // ah, self documenting code:
    // Here you can see that we decided to let the
    // executor process all the currently queued tasks.
    QueuedExecutor executor = o as QueuedExecutor;
    executor.ShutdownAfterProcessingCurrentlyQueuedTasks();
}
```

When an object is taken from the pool, to satisfy a client request, may be the object should be activated. We can possibly implement the activation like this:

```
void IPoolableObjectFactory.ActivateObject(object o)
{
    QueuedExecutor executor = o as QueuedExecutor;
    executor.Restart();
}
```

even if a `QueuedExecutor` restarts itself as needed and so a valid implementation could leave this method empty.

After activation, and before the pooled object can be succesfully returned to the client, it is validated (should the object be invalid, it will be discarded: this can lead to an empty unusable pool [8]). Here we check that the worker thread exists:

```
bool IPoolableObjectFactory.ValidateObject(object o)
{
    QueuedExecutor executor = o as QueuedExecutor;
    return executor.Thread != null;
```

[8]You may think that we can provide a smarter implementation and you are probably right. However, it is not so difficult to create a new pool in case the old one became unusable. It could not be your preferred choice but surely it leverages simplicity and object immutability

```
}
```

Passivation, symmetrical to activation, is the process a pooled object is subject to when the object is returned to the pool. In our case we simply do nothing:

```
void IPoolableObjectFactory.PassivateObject(object o)
{
}
```

At this point, creating a pool is simply a matter of creating an `SimplePool` as in:

```
pool = new SimplePool(new QueuedExecutorPoolableFactory(), size);
```

16.5.2. Being smart using pooled objects

Taking advantage of the `using` keyword seems to be very important in these `c#` days, so we implement a very simple helper (`PooledObjectHolder`) that can allow us to do things like:

```
using (PooledObjectHolder holder = PooledObjectHolder.UseFrom(pool))
{
    QueuedExecutor executor = (QueuedExecutor) holder.Pooled;
    executor.Execute(runnable);
}
```

without worrying about obtaining and returning an object from/to the pool.

Here is the implementation:

```
public class PooledObjectHolder : IDisposable
{
    IObjectPool pool;
    object pooled;

    /// <summary>
    /// Builds a new <see cref="PooledObjectHolder"/>
    /// trying to borrow an object form it
    /// </summary>
    /// <param name="pool"></param>
    private PooledObjectHolder(IObjectPool pool)
    {
        this.pool = pool;
        this.pooled = pool.BorrowObject();
    }

    /// <summary>
    /// Allow to access the borrowed pooled object
    /// </summary>
    public object Pooled
    {
        get
        {
            return pooled;
        }
    }

    /// <summary>
    /// Returns the borrowed object to the pool
    /// </summary>
    public void Dispose()
    {
        pool.ReturnObject(pooled);
    }
    /// <summary>
    /// Creates a new <see cref="PooledObjectHolder"/> for the
    /// given pool.
    /// </summary>
    public static PooledObjectHolder UseFrom(IObjectPool pool)
    {
        return new PooledObjectHolder(pool);
    }
}
```

Please don't forget to destroy all the pooled istances once you have finished! How? Well using something like this in `PooledQueuedExecutor`:

```
public void Stop ()
{
    // waits for all the grep-task to have been queued ...
    foreach (ISync sync in syncs)
    {
        sync.Acquire();
    }
    pool.Close();
}
```

16.5.3. Using the executor to do a parallel `grep`

The use of the just built executor is quite straigtforward but a little tricky if we want to really exploit the pool.

```
private PooledQueuedExecutor executor;

public ParallelGrep(int size)
{
    executor = new PooledQueuedExecutor(size);
}

public void Recurse(string startPath, string filePattern, string regexPattern)
{
    foreach (string file in Directory.GetFiles(startPath, filePattern))
    {
        executor.Execute(new Grep(file, regexPattern));
    }
    foreach (string directory in Directory.GetDirectories(startPath))
    {
        Recurse(directory, filePattern, regexPattern);
    }
}

public void Stop()
{
    executor.Stop();
}
```

```
public static void Main(string[] args)
{
    if (args.Length < 3)
    {
        Console.Out.WriteLine("usage: {0} regex directory file-pattern [pool-size]", Assembly.GetEntryAssembly()
        Environment.Exit(1);
    }

    string regexPattern = args[0];
    string startPath = args[1];
    string filePattern = args[2];
    int size = 10;
    try
    {
        size = Int32.Parse(args[3]);
    }
    catch
    {
    }
    Console.Out.WriteLine ("pool size {0}", size);

    ParallelGrep grep = new ParallelGrep(size);
    grep.Recurse(startPath, filePattern, regexPattern);
    grep.Stop();
}
```

16.6. AOP

Refer to Chapter 17, *AOP Guide*.

Chapter 17. AOP Guide

17.1. Introduction

This is an introductory guide to Aspect Oriented Programming (AOP) with Spring.NET.

This guide assumes little to no prior experience of having *used* Spring.NET AOP on the part of the reader. However, it *does* assume a certain familiarity with the terminology of AOP in general. It is probably better if you have read (or at least have skimmed through) the AOP section of the reference documentation beforehand, so that you are familiar with a) just what AOP is, b) what problems AOP is addressing, and c) what the AOP concepts of `advice`, `pointcut`, and `joinpoint` actually mean... this guide spends absolutely zero time defining those terms. Having said all that, if you are the kind of developer who learns best by example, then by all means follow along... you can always consult the reference documentation as the need arises (see Section 9.1.1, “AOP concepts”).

*The examples in this guide are **intentionally** simplistic. One of the core aims of this guide is to get you up and running with Spring.NET's flavor of AOP in as short a time as possible. Having to comprehend even a simple object model in order to understand the AOP examples would not be conducive to learning Spring.NET AOP. It is left as an exercise for the reader to take the concepts learned from this guide and apply them to his or her own code base. Again, having said all of that, this guide concludes with a number of cookbook-style AOP 'recipes' that illustrate the application of Spring.NET's AOP offering in a real world context; additionally, the Spring.NET reference application contains a number of Spring.NET AOP aspects particular to it's own domain model (see Chapter 19, SpringAir - Reference Application).*

17.2. The basics

This initial section introduces the basics of defining and then applying some simple advice.

17.2.1. Applying advice

Lets see (a very basic) example of using Spring.NET AOP. The following example code simply applies advice that writes the details of an advised method call to the system console. Admittedly, this is not a particularly compelling or even useful application of AOP, but having worked through the example, you will then hopefully be able to see how to apply your own custom advice to perform useful work (transaction management, auditing, security enforcement, thread safety, etc).

Before looking at the AOP code proper lets quickly look at the domain classes that are the target of the advice (in Spring.NET AOP terminology, an instance of the following class is going to be the *advised object*.

```
public interface ICommand
{
    object Execute(object context);
}

public class ServiceCommand : ICommand
{
    public object Execute(object context)
    {
        Console.Out.WriteLine("Service implementation : [{0}]", context);
        return null;
    }
}
```

Find below the advice that is going to be applied to the `object Execute(object context)` method of the `ServiceCommand` class. As you can see, this is an example of *around advice* (see Section 9.3.2, “Advice types”).

```
    public class ConsoleLoggingAroundAdvice : IMethodInterceptor
    {
        public object Invoke(IMethodInvocation invocation)
        {
            Console.Out.WriteLine("Advice executing; calling the advised method..."); ❶
            object returnValue = invocation.Proceed(); ❷ ❸
            Console.Out.WriteLine("Advice executed; advised method returned " + returnValue); ❹
            return returnValue; ❺
        }
    }
```

❸returnValue
❹returnValue
❺returnValue

So thus far we have three artifacts: an interface (`ICommand`); an implementation of said interface (`ServiceCommand`); and some (trivial) advice (encapsulated by the `ConsoleLoggingAroundAdvice` class). All that remains is to actually apply the `ConsoleLoggingAroundAdvice` advice to the invocation of the `Execute()` method of the `ServiceCommand` class. Lets look at how to effect this programmatically...

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingAroundAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute("This is the argument");
```

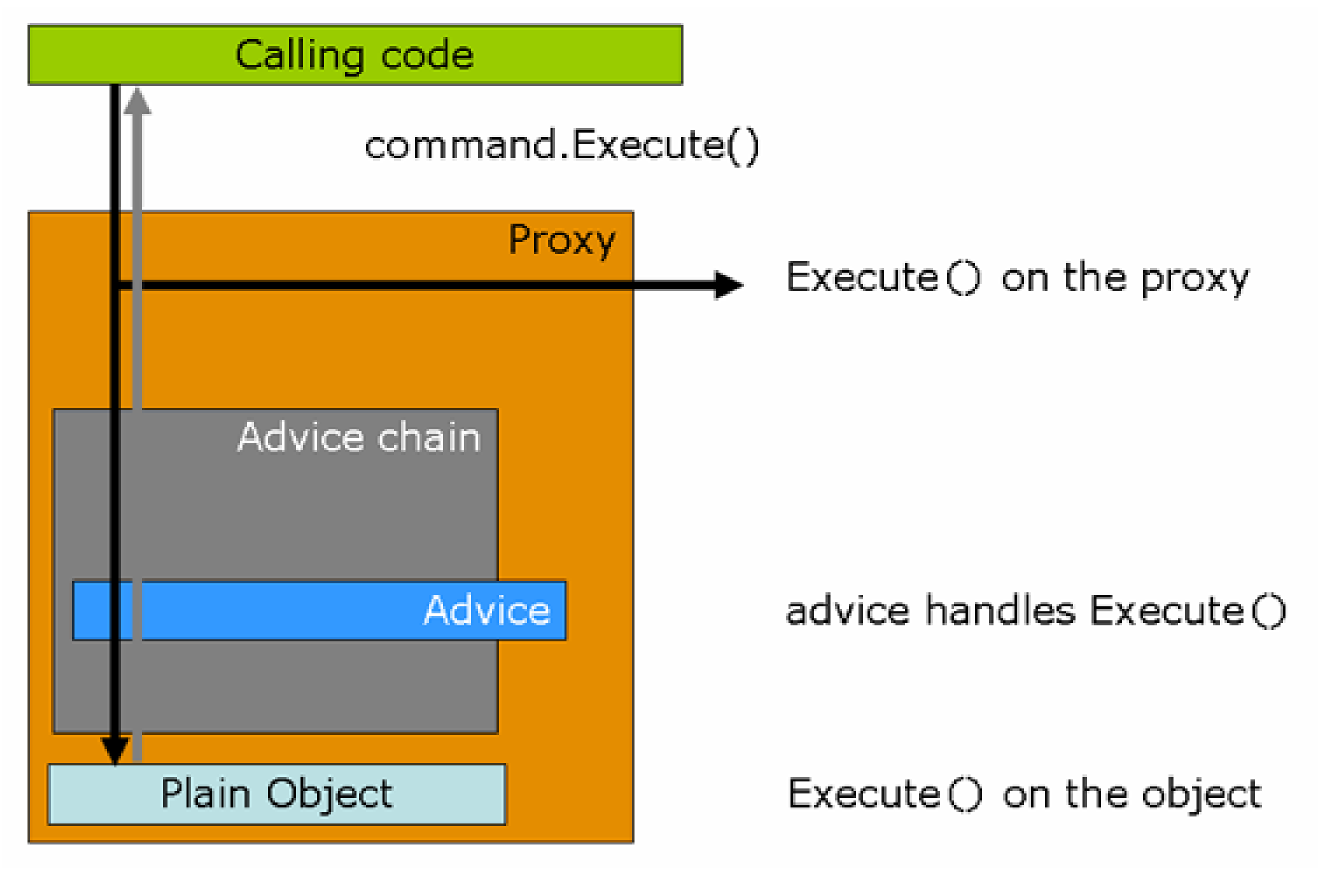
The result of executing the above snippet of code will look something like this...

```
    Advice executing; calling the advised method...
    Service implementation : [This is the argument]
    Advice executed; advised method returned
```

The output shows that the advice (the `Console.Out` statements from the `ConsoleLoggingAroundAdvice` was applied *around* the invocation of the advised method.

So what is happening here? The fact that the preceding code used a class called `ProxyFactory` may have clued you in. The constructor for the `ProxyFactory` class took as an argument the object that we wanted to advise (in this case, an instance of the `ServiceCommand` class). We then added some advice (a `ConsoleLoggingAroundAdvice` instance) using the `AddAdvice()` method of the `ProxyFactory` instance. We then called the `GetProxy()` method of the `ProxyFactory` instance which gave us a proxy... an (AOP) proxy that proxied the target object (the `ServiceCommand` instance), and called the advice (a single instance of the `ConsoleLoggingAroundAdvice` in this case). When we invoked the `Execute(object context)` method of the proxy, the advice was `'applied'` (executed), as can be seen from the attendant output.

The following image shows a graphical view of the flow of execution through a Spring.NET AOP proxy.



One thing to note here is that the AOP proxy that was returned from the call to the `GetProxy()` method of the `ProxyFactory` instance was cast to the `ICommand` interface that the `ServiceCommand` target object implemented. This is very important... currently, Spring.NET's AOP implementation mandates the use of an interface for advised objects. In short, this means that in order for your classes to leverage Spring.NET's AOP support, those classes that you wish to use with Spring.NET AOP **must** implement at least one interface. In practice this restriction is not as onerous as it sounds... in any case, it is *generally* good practice to program to interfaces anyway (support for applying advice to classes that do not implement any interfaces is planned for a future point release of Spring.NET AOP).

The remainder of this guide is concerned with fleshing out some of the finer details of Spring.NET AOP, but basically speaking, that's about it.

As a first example of fleshing out one of those finer details, find below some Spring.NET XML configuration that does *exactly* the same thing as the previous example; it should also be added that this declarative style approach to Spring.NET AOP is preferred to the programmatic style.

```
<object id="consoleLoggingAroundAdvice"
        type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>
<object id="myServiceObject" type="Spring.Aop.Framework.ProxyFactoryObject">
    <property name="target">
        <object id="myServiceObjectTarget"
            type="Spring.Examples.AopQuickStart.ServiceCommand"/>
    </property>
    <property name="interceptorNames">
        <list>
            <value>consoleLoggingAroundAdvice</value>
        </list>
    </property>
</object>
```

```
ICommand command = (ICommand) ctx["myServiceObject"];
command.Execute();
```

Some comments are warranted concerning the above XML configuration snippet. Firstly, note that the `ConsoleLoggingAroundAdvice` is itself a plain vanilla object, and is eligible for configuration just like any other class... if the advice itself needed to be injected with any dependencies, any such dependencies could be injected as normal.

Secondly, notice that the object definition corresponding to the object that is retrieved from the IoC container is a `ProxyFactoryObject`. The `ProxyFactoryObject` class is an implementation of the `IFactoryObject` interface; `IFactoryObject` implementations are treated specially by the Spring.NET IoC container... in this specific case, it is not a reference to the `ProxyFactoryObject` instance itself that is returned, but rather the object that the `ProxyFactoryObject` produces. In this case, it will be an advised instance of the `ServiceCommand` class.

Thirdly, notice that the target of the `ProxyFactoryObject` is an instance of the `ServiceCommand` class; this is the object that is going to be advised (i.e. invocations of its methods are going to be intercepted). This object instance is defined as an inner object definition... this is the preferred idiom for using the `ProxyFactoryObject`, as it means that other objects cannot acquire a reference to the raw object, but rather only the advised object.

Finally, notice that the advice that is to be applied to the target object is referred to by its object name in the list of the names of interceptors for the `ProxyFactoryObject`'s `interceptorNames` property. In this particular case, there is only one instance of advice being applied... the `ConsoleLoggingAroundAdvice` defined in an object definition of the same name. The reason for using a list of object names as opposed to references to the advice objects themselves is explained in the reference documentation...

'... if the `ProxyFactoryObject`'s singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, an independent instance would need to be returned, so it is necessary to be able to obtain an instance of the prototype from the context; holding a reference isn't sufficient.'

17.2.2. Using Pointcuts - the basics

The advice that was applied in the previous section was rather indiscriminate with regard to which methods on the advised object were to be advised... the `ConsoleLoggingAroundAdvice` simply intercepted **all** methods (that were part of an interface implementation) on the target object.

This is great for simple examples and suchlike, but not so great when you only want certain methods of an object to be advised. For example, you may only want those methods beginning with `'Start'` to be advised; or you may only want those methods that are called with specific runtime argument values to be advised; or you may only want those methods that are decorated with a `Lockable` attribute to be advised.

The mechanism that Spring.NET AOP uses to discriminate about where advice is applied (i.e. which method invocations are intercepted) is encapsulated by the `IPointcut` interface (see Section 9.2, “Pointcuts in Spring.NET”). Spring.NET provides many out-of-the-box implementations of the `IPointcut` interface... the implementation that is used if none is explicitly supplied (as was the case with the first example) is the canonical `TruePointcut` : as the name suggests, this pointcut always matches, and hence **all** methods that can be advised will be advised.

So let's change the configuration of the advice such that it is only applied to methods that contain the letters `'Do'`. We'll change the `ICommand` interface (and it's attendant implementation) to accomodate this...

```
    public interface ICommand
    {
        void Execute();

        void DoExecute();
    }

    public class ServiceCommand : ICommand
    {
        public void Execute()
        {
            Console.Out.WriteLine("Service implementation : Execute()...");
        }

        public void DoExecute()
        {
            Console.Out.WriteLine("Service implementation : DoExecute()...");
        }
    }
```

Please note that the advice itself (encapsulated within the `ConsoleLoggingAroundAdvice` class) does not need to change; we are changing *where* this advice is applied, and not the advice itself.

Programmatic configuration of the advice, taking into account the fact that we only want methods that contain the letters `'Do'` to be advised, looks like this...

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvisor(new DefaultPointcutAdvisor(
        new SdkRegularExpressionMethodPointcut("Do"),
        new ConsoleLoggingAroundAdvice()));
    ICommand command = (ICommand) factory.GetProxy();
    command.DoExecute();
```

The result of executing the above snippet of code will look something like this...

```
    Intercepted call : about to invoke next item in chain...
    Service implementation...
    Intercepted call : returned
```

The output indicates that the advice was applied around the invocation of the advised method, because the name of the method that was executed contained the letters `'Do'`. Try changing the pertinent code snippet to invoke the `Execute()` method, like so...

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvisor(
        new DefaultPointcutAdvisor(
            new SdkRegularExpressionMethodPointcut("Do"),
            new ConsoleLoggingAroundAdvice()));
    ICommand command = (ICommand) factory.GetProxy();

    // note that there is no 'Do' in this method name
    command.Execute();
```

Run the code snippet again; you will see that the advice will not be applied : the pointcut is not matched (the method name does not contain the letters `'Do'`), resulting in the following (unadvised) output...

```
Service implementation...
```

XML configuration that accomplishes exactly the same thing as the previous programmatic configuration example can be seen below...

```
    <object id="consoleLoggingAroundAdvice"
        type="Spring.Aop.Support.RegularExpressionMethodPointcutAdvisor">
        <property name="pattern" value="Do"/>
        <property name="advice">
            <object type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>
        </property>
    </object>
    <object id="myServiceObject"
            type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>consoleLoggingAroundAdvice</value>
            </list>
        </property>
    </object>
```

You'll will perhaps have noticed that this treatment of pointcuts introduced the concept of an `advisor` (see Section 9.4, “Advisors in Spring.NET”). An advisor is nothing more the composition of a pointcut (i.e. *where* advice is going to be applied), and the advice itself (i.e. *what* is going to happen at the interception point). The `consoleLoggingAroundAdvice` object defines an advisor that will apply the advice to all those methods of the advised object that match the pattern `'Do'` (the pointcut). The pattern to match against is supplied as a simple string value to the `pattern` property of the `RegularExpressionMethodPointcutAdvisor` class.

17.3. Going deeper

The first section should (hopefully) have demonstrated the basics of firstly defining advice, and secondly, of choosing where to apply that advice using the notion of a pointcut. Of course, there is a great deal more to Spring.NET AOP than the aforementioned single advice type and pointcut. This section continues the exploration of Spring.NET AOP, and describes the various advice and pointcuts that are available for you to use (yes, there is more than one type of advice and pointcut).

17.3.1. Other types of Advice

The advice that was demonstrated and explained in the preceding section is what is termed *'around advice'*. The name *'around advice'* is used because the advice is applied *around* the target method invocation. In the specific case of the `ConsoleLoggingAroundAdvice` advice that was defined previously, the target was made available to the advice as an `IMethodInvocation` object... a call was made to the `Console` class before the target was invoked, and a call was made to the `Console` class after the target method invocation was invoked. The advice surrounded the target, one could even say that the advice was totally 'around' the target... hence the name, *'around advice'*.

'around advice' provides one with the opportunity to do things both **before** the target gets a chance to do anything, and **after** the target has returned: one even gets a chance to inspect (and possibly even totally change) the return value.

Sometimes you don't need all that power though. If we stick with the example of the `ConsoleLoggingAroundAdvice` advice, what if one just wants to log the fact that a method was called? In that case one doesn't need to do anything *after* the target method invocation is to be invoked, nor do you need access to the return value of the target method invocation. In fact, you only want to do something *before* the target is to be invoked (in this case, print out a message to the system `Console` detailing the name of the method). In the tradition of good programming that says one should use only what one needs and no more, Spring.NET has another type of advice that one can use... if one only wants to do something *before* the target method invocation is invoked, why bother with having to manually call the `Proceed()` method? The most expedient solution simply is to use *'before advice'*.

17.3.1.1. Before advice

'before advice' is just that... it is advice that runs *before* the target method invocation is invoked. One does not get access to the target method invocation itself, and one cannot return a value... this is a good thing, because it means that you cannot inadvertently forget to call the `Proceed()` method on the target, and it also means that you cannot inadvertently forget to return the return value of the target method invocation. If you don't need to inspect or change the return value, or even do anything after the successful execution of the target method invocation, then *'before advice'* is just what you need.

'before advice' in Spring.NET is defined by the `IMethodBeforeAdvice` interface in the `Spring.Aop` namespace. Lets just dive in with an example... we'll use the same scenario as before to keep things simple. Let's define the *'before advice'* implementation first.

```
    public class ConsoleLoggingBeforeAdvice : IMethodBeforeAdvice
    {
        public void Before(MethodInfo method, object[] args, object target)
        {
            Console.Out.WriteLine("Intercepted call to this method : " + method.Name);
            Console.Out.WriteLine("    The target is               : " + target);
            Console.Out.WriteLine("    The arguments are           : ");
            if(args != null)
            {
                foreach (object arg in args)
                {
                    Console.Out.WriteLine("\t: " + arg);
                }
            }
        }
    }
```

Let's apply a single instance of the `ConsoleLoggingBeforeAdvice` advice to the invocation of the `Execute()` method of the `ServiceCommand`. What follows is programmatic configuration; as you can see, its pretty much identical to the previous version... the only difference is that we're using our new *'before advice'* (encapsulated as an instance of the `ConsoleLoggingBeforeAdvice` class).

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
    Intercepted call to this method : Execute
    The target is                   : Spring.Examples.AopQuickStart.ServiceCommand
    The arguments are               :
```

The output clearly indicates that the advice was applied **before** the invocation of the advised method. Notice that in contrast to *'around advice'*, with *'before advice'* there is no chance of forgetting to call the `Proceed()` method on the target, because one does not have access to the `IMethodInvocation` (as is the case with *'around advice'*)... similarly, you cannot forget to return the return value either.

If you can use *'before advice'*, then do so. The simpler programming model offered by *'before advice'* means that there is less to remember, and thus potentially less things to get wrong.

Here is the Spring.NET XML configuration for applying our *'before advice'* declaratively...

```
    <object id="beforeAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingBeforeAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>beforeAdvice</value>
            </list>
        </property>
    </object>
```

17.3.1.2. After advice

Just as *'before advice'* defines advice that executes **before** an advised target, *'after advice'* is advice that executes **after** a target has been executed.

'after advice' in Spring.NET is defined by the `IAfterReturningAdvice` interface in the `Spring.Aop` namespace. Again, lets just fire on ahead with an example... again, we'll use the same scenario as before to keep things simple.

```
    public class ConsoleLoggingAfterAdvice : IAfterReturningAdvice
    {
        public void AfterReturning(
            object returnValue, MethodInfo method, object[] args, object target)
        {
            Console.Out.WriteLine("This method call returned successfully : " + method.Name);
            Console.Out.WriteLine("    The target was                      : " + target);
            Console.Out.WriteLine("    The arguments were                  : ");
            if(args != null)
            {
                foreach (object arg in args)
                {
                    Console.Out.WriteLine("\t: " + arg);
                }
            }
            Console.Out.WriteLine("    The return value is                 : " + returnValue);
        }
    }
```

Let's apply a single instance of the `ConsoleLoggingAfterAdvice` advice to the invocation of the `Execute()` method of the `ServiceCommand`. What follows is programmatic configuration; as you can, its pretty much identical to the *'before advice'* version (which in turn was pretty much identical to the original *'around advice'* version)... the only real difference is that we're using our new *'after advice'* (encapsulated as an instance of the `ConsoleLoggingAfterAdvice` class).

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingAfterAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
    This method call returned successfully : Execute
    The target was                         : Spring.Examples.AopQuickStart.ServiceCommand
    The arguments were                     :
    The return value is                    : null
```

The output clearly indicates that the advice was applied **after** the invocation of the advised method. Again, it bears repeating that your real world development will actually have an advice implementation that does something useful after the invocation of an advised method. Notice that in contrast to *'around advice'*, with *'after advice'* there is no chance of forgetting to call the `Proceed()` method on the target, because just like *'before advice'* you don't have access to the `IMethodInvocation`... similarly, although you get access to the return value of the target, you cannot forget to return the return value either. You can however change the state of the return value, typically by setting some of its properties, or by calling methods on it.

The best-practice rule for *'after advice'* is much the same as it is for *'before advice'*; namely that if you can use *'after advice'*, then do so (in preference to using *'around advice'*). The simpler programming model offered by *'after advice'* means that there is less to remember, and thus less things to get potentially wrong.

A possible use case for *'after advice'* would include performing access control checks on the return value of an advised method invocation; consider the case of a service that returns a list of document URI's... depending on the identity of the (Windows) user that is running the program that is calling this service, one could strip out those URI's that contain sensitive data for which the user does not have sufficient priviliges to access. That is just one (real world) scenario... I'm sure you can think of plenty more that are a whole lot more relevant to your own development needs.

Here is the Spring.NET XML configuration for applying the *'after advice'* declaratively...

```
    <object id="afterAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingAfterAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>afterAdvice</value>
            </list>
        </property>
    </object>
```

17.3.1.3. Throws advice

So far we've covered *'around advice'*, *'before advice'*, and *'after advice'*... these advice types will see you through most if not all of your AOP needs. However, one of the remaining advice types that Spring.NET has in its locker is *'throws advice'*.

'throws advice' is advice that executes when an advised method invocation *throws* an exception.. hence the name. One basically applies the *'throws advice'* to a target object in much the same way as any of the previously mentioned advice types. If during the execution of ones application none of any of the advised methods throws an exception, then the *'throws advice'* will never execute. However, if during the execution of your application an advised method *does* throw an exception, then the *'throws advice'* will kick in and be executed. You can use *'throws advice'* to apply a common exception handling policy across the varoius objects in your application, or to perform logging of every exception thown by an advised method, or to alert (perhaps via email) the support team in the case of particularly of critical exceptions... the list of possible uses cases is of course endless.

The *'throws advice'* type in Spring.NET is defined by the `IThrowsAdvice` interface in the `Spring.Aop` namespace... basically, one defines on one's *'throws advice'* implementation class what types of exception are going to be handled. Lets take a quick look at the `IThrowsAdvice` interface...

```
    public interface IThrowsAdvice : IAdvice
    {
    }
```

Yes, that is really it... it is a marker interface that has no methods on it. You may be wondering how Spring.NET determines which methods to call to effect the running of one's *'throws advice'*. An example would perhaps be illustrative at this point, so here is some simple Spring.NET style *'throws advice'*...

```
    public class ConsoleLoggingThrowsAdvice : IThrowsAdvice
    {
        public void AfterThrowing(Exception ex)
        {
            Console.Out.WriteLine("Advised method threw this exception : " + ex);
        }
    }
```

Lets also change the implementation of the `Execute()` method of the `ServiceCommand` class such that it throws an exception. This will allow the advice encapsulated by the above `ConsoleLoggingThrowsAdvice` to kick in.

```
    public class ServiceCommand : ICommand
    {
        public void Execute()
        {
            throw new UnauthorizedAccessException();
        }
    }
```

Let's programmatically apply the *'throws advice'* (an instance of our `ConsoleLoggingThrowsAdvice`) to the invocation of the `Execute()` method of the above `ServiceCommand` class; to wit...

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

The result of executing the above snippet of code will look something like this...

```
    Advised method threw this exception : System.UnauthorizedAccessException:
    Attempted to perform an unauthorized operation.
```

As can be seen from the output, the `ConsoleLoggingThrowsAdvice` kicked in when the advised method invocation threw an exception. There are a number of things to note about the `ConsoleLoggingThrowsAdvice` advice class, so lets take them each in turn.

In Spring.NET, *'throws advice'* means that you have to define a class that implements the `IThrowsAdvice` interface. Then, for each type of exception that your *'throws advice'* is going to handle, you have to define a method with this signature...

```
    void AfterThrowing(Exception ex)
```

Basically, your exception handling method has to be named `AfterThrowing`. This name is important... your exception handling method (s) absolutely must be called `AfterThrowing`. If your handler method is not called `AfterThrowing`, then your *'throws advice'* will **never** be called, it's as simple as that. Currently, this naming restriction is not configurable (although it may well be opened up for configuration in the future).

Your exception handling method must (at the very least) declare a parameter that is an `Exception` type... this parameter can be the root `Exception` class (as in the case of the above example), or it can be an `Exception` subclass if you only want to handle certain types of exception. It is good practice to always make your exception handling methods have an `Exception` parameter that is the most specialized `Exception` type possible... i.e. if you are applying *'throws advice'* to a method that could only ever throw `ArgumentExceptions`, then declare the parameter of your exception handling method as...

```
    void AfterThrowing(ArgumentException ex)
```

Note that your exception handling method can have any return type, but returning any value from a Spring.NET *'throws advice'* method would be a waste of time... the Spring.NET AOP infrastructure will simply ignore the return value, so always define the return type of your exception handling methods to be `void`.

Finally, here is the Spring.NET XML configuration for applying the *'throws advice'* declaratively...

```
    <object id="throwsAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingThrowsAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>throwsAdvice</value>
            </list>
        </property>
    </object>
```

One thing that cannot be done using *'throws advice'* is exception swallowing. It is not possible to define an exception handling method in a *'throws advice'* implementation that will swallow any exception and prevent said exception from bubbling up the call stack. The nearest thing that one can do is define an exception handling method in a *'throws advice'* implementation that will wrap the handled exception in another exception; one would then throw the wrapped exception in the body of one's execption handling method. One can use this to implement some sort of exception translation or exception scrubbing policy, in which implementation specific exceptions (such as `SqlException` or `OracleException` exceptions being thrown by an advised data access object) get replaced with a business exception that has meaning to the service objects in one's business layer. A toy example of this type of *'throws advice'* can be seen below.

```
    public class DataAccessExceptionScrubbingThrowsAdvice : IThrowsAdvice
    {
        public void AfterThrowing (SqlException ex)
        {
            // business objects in higher level service layer need only deal with PersistenceException...
            throw new PersistenceException ("Cannot access persistent storage.", ex.StackTrace);
        }
    }
```

Spring.NET's data access library already has this kind of functionality (and is a whole lot more sophisticated)... the above example is merely being used for illustrative purposes.

This treatment of *'throws advice'*, and of Spring.NET's implementation of it is rather simplistic. *'throws advice'* features that have been omitted include the fact that one can define exception handling methods that permit access to the original object, method, and method arguments of the advised method invocation that threw the original exception. This is a quickstart guide though, and is not meant to be exhaustive... do consult the *'throws advice'* section of the reference documentation, which describes how to declare an exception handling method that gives one access to the above extra objects, and how to declare multiple exception handling methods on the same `IThrowsAdvice` implementation class (see Section 9.3.2.3, “Throws advice”).

17.3.1.4. Introductions (mixins)

In a nutshell, introductions are all about adding new state and behaviour to arbitrary objects... transparently and at runtime. Introductions (also called mixins) allow one to emulate multiple inheritance, typically with an eye towards applying crosscutting state and operations to a wide swathe of objects in your application that don't share the same inheritance hierarchy.

17.3.1.5. Layering advice

The examples shown so far have all demonstrated the application of a single advice instance to an advised object. Spring.NET's flavor of AOP would be pretty poor if one could only apply a single advice instance per advised object... it is perfectly valid to apply multiple advice to an advised object. For example, one might apply transactional advice to a service object, and also apply a security access checking advice to that same advised service object.

In the interests of keeping this section lean and tight, let's simply apply *all* of the advice types that have been previously described to a single advised object... in this first instance we'll just use the default pointcut which means that every possible joinpoint will be advised, and you'll be able to see that the various advice instances are applied in order.

Please do consult the class definitions for the following previously defined advice types to see exactly what each advice type implementation does... we're going to be using single instances of the `ConsoleLoggingAroundAdvice`, `ConsoleLoggingBeforeAdvice`, `ConsoleLoggingAfterAdvice`, and `ConsoleLoggingThrowsAdvice` advice to advise a single instance of the `ServiceCommand` class.

```
    ProxyFactory factory = new ProxyFactory(new ServiceCommand());
    factory.AddAdvice(new ConsoleLoggingBeforeAdvice());
    factory.AddAdvice(new ConsoleLoggingAfterAdvice());
    factory.AddAdvice(new ConsoleLoggingThrowsAdvice());
    factory.AddAdvice(new ConsoleLoggingAroundAdvice());
    ICommand command = (ICommand) factory.GetProxy();
    command.Execute();
```

Here is the Spring.NET XML configuration for declaratively applying multiple advice.

```
    <object id="throwsAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingThrowsAdvice"/>
    <object id="afterAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingAfterAdvice"/>
    <object id="beforeAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingBeforeAdvice"/>
    <object id="aroundAdvice"
            type="Spring.Examples.AopQuickStart.ConsoleLoggingAroundAdvice"/>

    <object id="myServiceObject"
        type="Spring.Aop.Framework.ProxyFactoryObject">
        <property name="target">
            <object id="myServiceObjectTarget"
                type="Spring.Examples.AopQuickStart.ServiceCommand"/>
        </property>
        <property name="interceptorNames">
            <list>
                <value>throwsAdvice</value>
                <value>afterAdvice</value>
                <value>beforeAdvice</value>
                <value>aroundAdvice</value>
            </list>
        </property>
    </object>
```

17.3.1.6. Configuring advice

In case it is not immediately apparent, remember that advice is just a plain old .NET object (a PONO); advice can have constructors that can take any number of parameters, and like any other .NET class, advice can have properties. What this means is that one can leverage the power of the Spring.NET IoC container to apply the IoC principle to one's advice, and in so doing reap all the benefits of Dependency Injection.

Consider the case of throws advice that needs to report (fatal) exceptions to a first line support centre. The throws advice could declare a dependency on a reporting service via a .NET property, and the Spring.NET container could dependency inject the reporting service dependency into the throws advice when it is being created; the reporting dependency might be a simple Log4NET wrapper, or a Windows EventLog wrapper, or a custom reporting exception reporting service that sends detailed emails concerning the fatal exception.

Also bear in mind the fact that Spring.NET's AOP implementation is quite independent of Spring.NET's IoC container. As you have seen, the various examples used in this have illustrated both programmatic and declarative AOP configuration (the latter being illustrated via Spring.NET's IoC XML configuration mechanism).

17.3.2. Using Attributes to define Pointcuts

17.4. The Spring.NET AOP Cookbook

The preceding treatment of Spring.NET AOP has (quite intentionally) been decidedly simple. The overarching aim was to convey the concepts of Spring.NET AOP... this section of the Spring.NET AOP guide contains a number of real world examples of the application of Spring.NET AOP.

17.4.1. Caching

This example illustrates one of the more common usages of AOP... caching.

Lets consider the scenario where we have some static reference data that needs to be kept around for the duration of an application. The data will almost never change over the uptime of an application, and it exists only in the database to satisfy referential integrity amongst the various relations in the database schema. An example of such static (and typically immutable) reference data would be a collection of `Country` objects (comprising a country name and a code). What we would like to do is suck in the collection of `Country` objects and then pin them in a cache. This saves us having to hit the back end database again and again every time we need to reference a country in our application (for example, to populate dropdown controls in a Windows Forms desktop application).

The Data Access Object (DAO) that will load the collection of `Country` objects is called `AdoCountryDao` (it is an implementation of the data-access-technology agnostic DAO interface called `ICountryDao`). The implementation of the `AdoCountryDao` is quite simple, in that every time the `FindAllCountries` instance method is called, an instance will query the database for an `IDataReader` and hydrate zero or more `Country` objects using the returned data.

```
    public class AdoCountryDao : ICountryDao
    {
        public IList FindAllCountries ()
        {
            // implementation elided for clarity...
            return countries;
        }
    }
```

Ideally, what we would like to have happen is for the results of the *first* call to the `FindAllCountries` instance method to be cached. We would also like to do this in a non-invasive way, because caching is something that we might want to apply at any number of points across the codebase of our application. So, to address what we have identified as a *cross cutting concern*, we can use Spring.NET AOP to implement the caching.

The mechanism that this example is going to use to identify (or pick out) areas in our application that we would like to apply caching to is a .NET `Attribute`. Spring.NET ships with a number of useful custom .NET `Attribute` implementations, one of which is the cunningly named `CacheAttribute`. In the specific case of this example, we are simply going to decorate the definition of the `FindAllCountries` instance method with the `CacheAttribute`.

```
public class AdoCountryDao : ICountryDao
{
    [Cache]
    public IList FindAllCountries ()
    {
        // implementation elided for clarity...
        return countries;
    }
}
```

The SpringAir reference application that is packaged as part of the Spring.NET distribution comes with a working example of caching applied using Spring.NET AOP (see Chapter 19, *SpringAir - Reference Application*).

17.4.2. Performance Monitoring

This recipe show how easy it is to instrument the classes and objects in an application for performance monitoring. The performance monitoring implementation uses one of the (many) Windows performance counters to display and track the performance data.

17.4.3. Retry Rules

This final recipe describes a simple (but really quite useful) aspect... retry logic. Using Spring.NET AOP, it is quite easy to surround an operation such as a method that opens a connection to a database with a (configurable) aspect that tries to obtain a database connection any number of times in the event of a failure.

17.5. Spring.NET AOP Best Practices

Spring.NET AOP is an 80% AOP solution, in that it only tries to solve the 80% of those cases where AOP is a good fit in a typical enterprise application. This final section of the Spring.NET AOP guide describes where Spring.NET AOP is typically useful (the 80%), as well as where Spring.NET AOP is not a good fit (the 20%).

Chapter 18. .NET Remoting Quick start

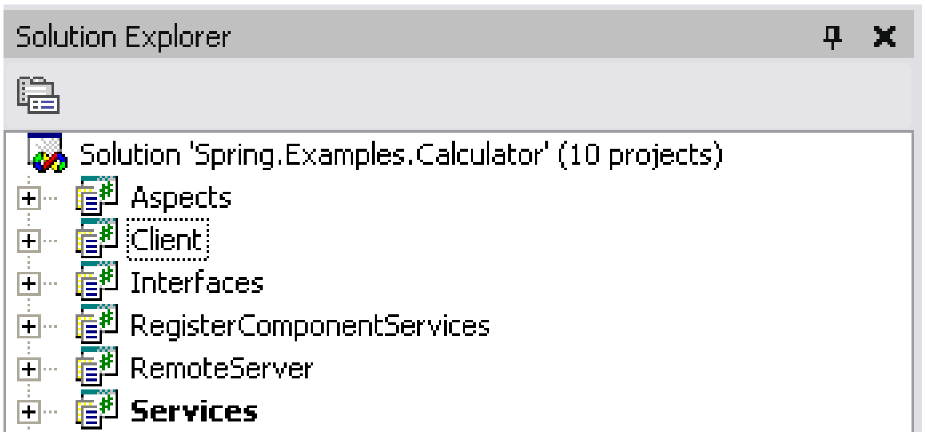
18.1. Introduction

This quickstart demonstrates the basic usage of Spring.NET's remoting infrastructure. The infrastructure classes are located in the `Spring.Services` assembly under the `Spring.Services.Remoting`namespace. The overall strategy is to export .NET objects on the server side as either CAO or SAO objects using `CaoExporter` or `SaoExporter` and obtain references to these objects on the client side using `CaoFactoryObject` and `SaoFactoryObject`. This quickstart does assume familiarity with .NET remoting on the part of the reader. If you are new to .NET remoting you may find the links to introductory remoting material presented at the conclusion of this quickstart of some help.

18.2. The Remoting Sample Project

As usual with quick start examples in Spring.NET, the classes used in the quickstart are intentionally simple. In the specific case of this remoting quickstart we are going to make a simple calculator that can be accessed remotely. The same calculator class will be exported in multiple ways reflecting the variety of .NET remoting options available (CAO, SAO-SingleCall, SAO-Singleton) and also the use of adding AOP advice to SAO hosted objects.

The example solution is located in the `examples\Spring\Spring.Examples.Calculator` directory and contains multiple projects.


Solution Explorer
Solution 'Spring.Examples.Calculator' (10 projects)
Aspects
Client
Interfaces
RegisterComponentServices
RemoteServer
Services

The `Interfaces` project contains the interface `ICalculator` that defines the basic operations of a calculator and another interface `IAdvancedCalculator` that adds support for memory storage for results. (woo hoo - big feature - HP-12C beware!) These interfaces are shown below. The `Services` project contains an implementation of the these interfaces, namely the classes `Calculator` and `AdvancedCalculator`. The purpose of the `AdvancedCalculator` implementation is to demonstrate the configuration of object state for SAO-singleton objects. Note that the calculator implementations *do not* inherit from the `MarshalByRefObject` class. The `Client` project contains the client application and the `RemoteServer` project contains a console application that will host a remoted instance of the `AdvancedCalculator` class. The `Aspects` project contains some logging advice that will be used to demonstrate the application of aspects to remoted SAO objects. `RegisterComponentServices` is related to enterprise service exporters and is not relevant for this quickstart.

```
public interface ICalculator
{
    int Add(int n1, int n2);
    int Substract(int n1, int n2);

    double Divide(int n1, int n2);

    int Multiply(int n1, int n2);
}
```

An extension of this interface that supports having a slot for calculator memory is shown below

```
public interface IAdvancedCalculator : ICalculator
{
        int GetMemory();

        void SetMemory(int memoryValue);

        void MemoryClear();

        void MemoryAdd(int num);
}
```

The structure of the VS.NET solution is a consequence of following the best practice of using interfaces to share type information between a .NET remoting client and server. The benefits of this approach are that the client does not need a reference to the assembly that contains the implementation class. Having the client reference the implementation assembly is undesirable for a variety of reasons. One reason being security since an untrusted client could potentially obtain the source code to the implementation since Intermediate Language (IL) code is easily reverse engineered. Another, more compelling, reason is to provide a greater decoupling between the client and server so the server can update its implementation of the interface in a manner that is quite transparent to the client; i.e. the client code need not change. Independent of .NET remoting best practices, using an interface to provide a service contract is just good object-oriented design. This lets the client choose another implementation unrelated to .NET Remoting, for example a local, test-stub or a web services implementation. One of the major benefits of using Spring.NET is that it reduces the cost of doing 'interface based programming' to almost nothing. As such, this best practice approach to .NET remoting fits naturally into the general approach to application development that Spring.NET encourages you to follow. Ok, with that barrage of OO design ranting finished, on to the implementation!

18.3. Implementation

The implementation of the calculators contained in the `Servies` project is quite straightforward. The only interesting methods are those that deal with the memory storage, which is the state that we will be configuring explicitly using constructor injection. A subset of the implementation is shown below.

```
public class Calculator : ICalculator
{

    public int Add(int n1, int n2)
    {
        return n1 + n2;
    }

    public int Substract(int n1, int n2)
    {
        return n1 - n2;
    }

    public double Divide(int n1, int n2)
    {
        return n1 / n2;
    }

    public int Multiply(int n1, int n2)
    {
        return n1 * n2;
    }

}

public class AdvancedCalculator : Calculator, IAdvancedCalculator
{

  private int memoryStore = 0;

  public AdvancedCalculator()
  {}

  public AdvancedCalculator(int initalMemory)
  {
    memoryStore = initalMemory;
  }

  public int GetMemory()
  {
    return memoryStore;
  }

  // other methods omitted in this listing...

}
```

The `RemotedCalculator` project hosts remoted objects inside a console application. The code is also quite simple and shown below

```
public static void Main(string[] args)
{
    try
    {
        RemotingConfiguration.Configure("RemoteServer.exe.config"); ❶

        IApplicationContext ctx = ContextRegistry.GetContext(); ❷

        Console.Out.WriteLine("Server listening...");
    }
    catch (Exception e)
    {
        Console.Out.WriteLine(e);
    }
    finally
    {
        Console.Out.WriteLine("--- Press <return> to quit ---");
        Console.ReadLine();
    }
}
```

❶❷*ust*

The configuration of the .NET remoting channels is done using the standard `system.runtime.remoting` configuration section inside the .NET configuration file of the application (`App.config`). In this case we are using the `tcp` channel on port `8005`.

```
<system.runtime.remoting>
  <application>
    <channels>
      <channel ref="tcp" port="8005" />
    </channels>
  </application>
</system.runtime.remoting>
```

The objects created in Spring's application context are shown below. Multiple resource files are used to export these objects under various remoting configurations. The AOP advice used in this example is a simple Log4Net based around advice.

```
<spring>
        <context>
                <resource uri="config://spring/objects" />
                <resource uri="assembly://RemoteServer/RemoteServer.Config/cao.xml" />
                <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleCall.xml" />
                <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleCall-aop.xml" />
                <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleton.xml" />
                <resource uri="assembly://RemoteServer/RemoteServer.Config/saoSingleton-aop.xml" />
        </context>
        <objects xmlns="http://www.springframework.net">
                <description>Definitions of objects to be exported.</description>

                <object id="Log4NetLoggingAroundAdvice" type="Aspects.Logging.Log4NetLoggingAroundAdvice, Aspect
                        <property name="Level" value="Debug" />
                </object>

                <object id="singletonCalculator" type="Services.AdvancedCalculator, Services">
                        <constructor-arg type="int" value="217"/>
                </object>

                <object id="singletonCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop
                        <property name="target" ref="singletonCalculator"/>
                        <property name="interceptorNames">
                                <list>
                                        <value>Log4NetLoggingAroundAdvice</value>
                                </list>
                        </property>
                </object>

                <object id="prototypeCalculator" type="Services.AdvancedCalculator, Services" singleton="false">
                        <constructor-arg type="int" value="217"/>
                </object>

                <object id="prototypeCalculatorWeaved" type="Spring.Aop.Framework.ProxyFactoryObject, Spring.Aop
                        <property name="targetSource">
                                <object type="Spring.Aop.Target.PrototypeTargetSource, Spring.Aop">
                                        <property name="TargetObjectName" value="prototypeCalculator"/>
                                </object>
                        </property>
                        <property name="interceptorNames">
                                <list>
                                        <value>Log4NetLoggingAroundAdvice</value>
                                </list>
                        </property>
                </object>

        </objects>
</spring>
```

The declaration of the calculator instance, `singletonCalculator` for example, and the setting of any property values and / or object references is done as you would normally do for any object declared in the Spring.NET configuration file. To expose the calculator objects as .NET remoted objects the exporter `Spring.Remoting.CaoExporter` is used for CAO objects and `Spring.Remoting.SaoExporter` is used for SAO objeccts. Both exporters require the setting of a `TargetName` property that refers to the name of the object in Spring's IoC container that will be remoted. The semantics of SAO-SingleCall and CAO behavior are achieved by exorting a target object that is declared as a "prototype" (i.e. singleton=false). For SAO objects, the `ServiceName` property defines the the name of the service as it will appear in the URL that clients use to locate the remote object. To set the remoting lifetime of the objects to be infinite, the property `Infinte` is set to true.

The configuration for the exporting a SAO-Singleton is shown below.

```
<object name="saoSingletonCalculator" type="Spring.Remoting.SaoExporter, Spring.Services">
  <property name="TargetName" value="singletonCalculator" />
  <property name="ServiceName" value="RemotedSaoSingletonCalculator" />
  <property name="Infinite" value="true" />
</object>
```

This will result in the remote object being identified by the URL `tcp://localhost:8005/RemotedSaoSingletonCalculator`. The use of `SaoExporter` and `CaoExporter` for other configuration are similar, look at the configuration files in the RemoteServer project files for more information.

On the client side, the client application will connect a specific type of remote calculator service, object, ask it for it's current memory value, which is pre-configured to `217`, then perform a simple addition. As in the case of the server, the channel configuration is done using the standard .NET Remoting configuration section of the .NET application configuration file (`App.config`), as can been seen below.

```
<system.runtime.remoting>
    <application>
            <channels>
                    <channel ref="tcp"/>
            </channels>
    </application>
</system.runtime.remoting>
```

The client implementation code is shown below.

```
public static void Main(string[] args)
{
        try
        {
                pause();

                RemotingConfiguration.Configure("Client.exe.config");

                IApplicationContext ctx = ContextRegistry.GetContext();

                Console.Out.WriteLine("Get Calculator...");
                IAdvancedCalculator firstCalc = (IAdvancedCalculator) ctx.GetObject("calculatorService");
                Console.Out.WriteLine("Memory = " + firstCalc.GetMemory());
                firstCalc.MemoryAdd(2);
                Console.Out.WriteLine("Memory + 2 = " + firstCalc.GetMemory());

                Console.Out.WriteLine("Get Calculator...");
                IAdvancedCalculator secondCalc = (IAdvancedCalculator) ctx.GetObject("calculatorService");
                Console.Out.WriteLine("Memory = " + secondCalc.GetMemory());
        }
        catch (Exception e)
        {
                Console.Out.WriteLine(e);
        }
        finally
        {
                Console.Out.WriteLine("--- Press <return> to quit ---");
                Console.ReadLine();
        }
}
```

Note that the client application code is not aware that it is using a remote object. The `pause()` method simply waits until the `Return` key is pressed on the console so that the client doesn't make a request to the server before the server has had a chance to start. The standard configuration and initialization of the .NET remoting infrastructure is done before the creation of the Spring.NET IoC container. The configuration of the client application is structed in such a way that one can easily switch implementations of the `calculatorService` retrieved from the application context. In more complex applications the calcuator service would be a dependency on another object in your application, say in a workflow processing layer. The following listing shows a configuration for use of a local implementation and then several remote implementations. The same Exporter approach can be used to create Web Services and Serviced Components (EnterpriseServices) of the calcualtor object but are not discussed in this QuickStart.

```

<spring>
        <context>
                <!-- Only one at a time ! -->

                <!-- ================================== -->
                <!-- In process (local) implementations -->
                <!-- ================================== -->
                <resource uri="assembly://Client/Client.Config.InProcess/inProcess.xml" />

                <!-- ======================== -->
                <!-- Remoting implementations -->
                <!-- ======================== -->

                <!-- Make sure 'RemoteServer' console application is running and listening. -->

                <!-- <resource uri="assembly://Client/Client.Config.Remoting/cao.xml" /> -->
                <!-- <resource uri="assembly://Client/Client.Config.Remoting/cao-ctor.xml" /> -->
                <!-- <resource uri="assembly://Client/Client.Config.Remoting/saoSingleton.xml" /> -->
                <!-- <resource uri="assembly://Client/Client.Config.Remoting/saoSingleton-aop.xml" /> -->
                <!-- <resource uri="assembly://Client/Client.Config.Remoting/saoSingleCall.xml" /> -->
                <!-- <resource uri="assembly://Client/Client.Config.Remoting/saoSingleCall-aop.xml" /> -->

                <!-- =========================== -->
                <!-- Web Service implementations -->
                <!-- =========================== -->
                <!-- Make sure 'http://localhost/SpringCalculator/' web application is running -->
                <!-- <resource uri="assembly://Client/Client.Config.WebServices/webServices.xml" /> -->
                <!-- <resource uri="assembly://Client/Client.Config.WebServices/webServices-aop.xml" /> -->

                <!-- ================================= -->
                <!-- EnterpriseService implementations -->
                <!-- ================================= -->
                <!-- Make sure you register components with 'RegisterComponentServices' console application.  --
                <!-- <resource uri="assembly://Client/Client.Config.EnterpriseServices/enterpriseServices.xml" /
        </context>
</spring>
```

The inProcess.xml configuration file creates an instance of AdvancedCalculator directly

```

<objects xmlns="http://www.springframework.net">

  <description>inProcess</description>

  <object id="calculatorService" type="Services.AdvancedCalculator, Services" />

</objects>
```

Factory classes are used to create a client side reference to the .NET remoting implementations. For SAO objects use the `SaoFactoryObject` class and for CAO objects use the `CaoFactoryObject` class. The configuration for obtaining a reference to the previously exported SAO singleton implementation is shown below

```

<objects xmlns="http://www.springframework.net">

        <description>saoSingleton</description>

        <object id="calculatorService" type="Spring.Remoting.SaoFactoryObject, Spring.Services">
                <property name="ServiceInterface" value="Interfaces.IAdvancedCalculator, Interfaces" />
                <property name="ServiceUrl" value="tcp://localhost:8005/RemotedSaoSingletonCalculator" />
        </object>

</objects>
```

You must specify the property `ServiceInterface` as well as the location of the remote object via the `ServiceUrl` property. The property replacement facilities of Spring.NET can be leveraged here to make it easy to configure the URL value based on environment variable settings, a standard .NET configuration section, or an external property file. This is useful to easily switch between test, QA, and production (yea baby!) environments. An example of how this would be expressed is...

```
<property name="ServiceUrl" value="${protocol}://${host}:${port}/RemotedSaoSingletonCalculator" />
```

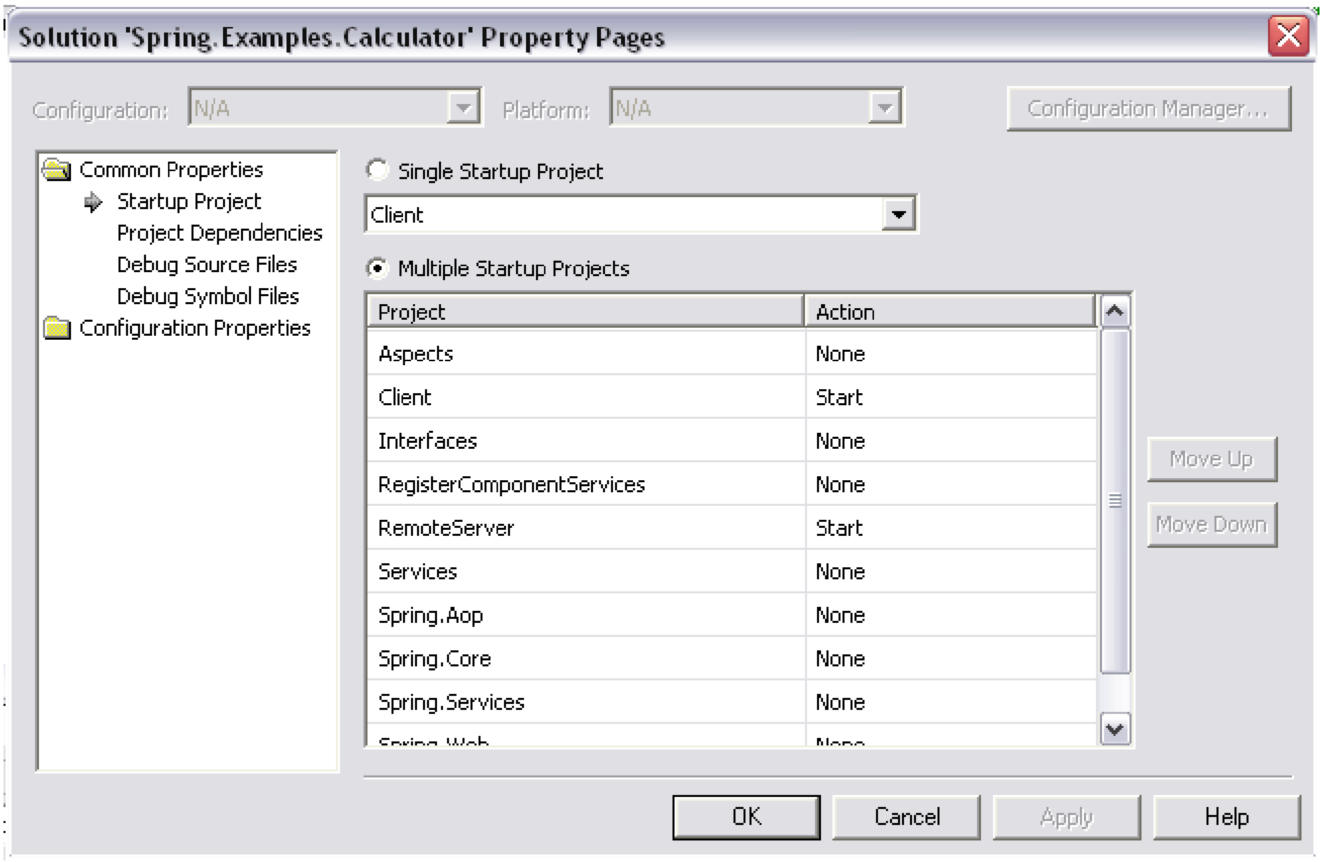
The property values in this example are defined elsewhere; refer to Section 3.9.1, “The PropertyPlaceholderConfigurer” for additional information. As mentioned previously, more important in terms of configuration flexibility is the fact that now you can swap out different implementations (.NET remoting based or otherwise) of this interface by making a simple change to the configuration file.

The configuration for obtaining a reference to the previously exported CAO implementation is shown below

```
<objects xmlns="http://www.springframework.net">

        <description>cao</description>

        <object id="calculatorService" type="Spring.Remoting.CaoFactoryObject, Spring.Services">
                <property name="RemoteTargetName" value="prototypeCalculator" />
                <property name="ServiceUrl" value="tcp://localhost:8005" />
        </object>

</objects>
```

18.4. Running the application

Now that we have had a walk though of the implementation and configuration it is finally time to run the application (if you haven't yet pulled the trigger). Be sure to set up VS.NET to run multiple applications on startup as shown below.



Project	Action
Aspects	None
Client	Start
Interfaces	None
RegisterComponentServices	None
RemoteServer	Start
Services	None
Spring.Aop	None
Spring.Core	None
Spring.Services	None
Spring.Web	None

Running the solution yields the following output in the server window... TO BE DONE...

18.5. Additional Resources

.NET remoting is a huge topic. Some introductory articles on .NET remoting can be found online at MSDN.

Ingo Rammer is also a very good authority on .NET remoting, and the .NET Remoting FAQ (link below) which is maintained by Ingo is chock full of useful information.

- An Introduction to Microsoft .NET Remoting Framework
- Microsoft .NET Remoting: A Technical Overview
- Advanced .NET Remoting (authored by Ingo Rammer)
- .NET Remoting FAQ

Chapter 19. SpringAir - Reference Application

19.1. Introduction

This chapter describes the reference application for Spring.NET.

19.2. The Architecture

...

19.3. The Implementation

...

19.3.1. The Domain Layer

...

19.3.2. The Service Layer

...

19.3.3. The Web Layer

...

19.4. Summary

...

Chapter 20. Spring.NET for Java Developers

20.1. Introduction

This chapter is to help Java developers get their sea legs using Spring.NET. It is not intended to be a comprehensive comparison between .NET and Java. Rather, it highlights the day-to-day differences you will experience when you start to use Spring.NET.

20.2. Beans to Objects

There are some simple name changes, basically everywhere you saw the word 'bean' you will now see the word 'object'. A comparison of a simple Spring configuration file highlights these small name changes. Here is the application.xml file for the sample MovieFinder application in Spring.Java

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   <bean id="MyMovieLister" class="MovieFinder.MovieLister">
      <property name="finder" ref="MyMovieFinder"/>
   </bean>
   <bean id="MyMovieFinder" class="MovieFinder.SimpleMovieFinder"/>
</beans>
```

Here is the corresponding file in Spring.NET

```
<objects xmlns="http://www.springframework.net"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.net http://www.springframework.net/xsd/spring-objects.xsc
  <object name="MyMovieLister"
          type="Spring.Examples.MovieFinder.MovieLister, Spring.Examples.MovieFinder">
        <property name="movieFinder" ref="MyMovieFinder"/>
  </object>
  <object name="MyMovieFinder"
          type="Spring.Examples.MovieFinder.SimpleMovieFinder, Spring.Examples.MovieFinder"/>
</objects>
```

As you can easily see the <beans> and <bean> elements are replaced by <objects> and <object> elements. The class definition in Spring.Java contains the fully qualified class name. The Spring.NET version also contains the fully qualified classname but in addition specifies the name of the assembly where that type is located. This is necessary since .NET does not have a 'classpath' concept. Assembly names in .NET can have up to four parts to describe the exact version.

The other XML Schema elements in Spring.NET are the same as in Spring.Java's DTD except for specifying string based key value pairs. In Java this is represented by the java.util.Properties class and the xml element is name <props> as shown below

```

<property name="people">
  <props>
    <prop key="PennAndTeller">The magic property</prop>
    <prop key="GeorgeCarlin">The funny property</prop>
  </props>
</property>
```

In .NET the analogous class is System.Collections.Specialized.NameValueCollection and is represented by the xml element <name-values>. The listing of the elements also follows the .NET convention of application configuration files using the <add> element with 'key' and 'value' attributes. This is show below

```

<property name="people">
```

```
  <name-values>
    <add key="PennAndTeller" value="The magic property"/>
    <add key="GeorgeCarlin" value="The funny property"/>
  </name-values>
</property>
```

20.3. PropertyEditors to TypeConverters

PropertyEditors from the java.beans package provide the ability to convert from a string to an instance of a Java class and vica-versa. For example, to set a string array property, a comma delimited string can be used. The Java class that provides this functionality is the appropriately named StringArrayPropertyEditor. In .NET, TypeConverters from the System.ComponentModel namespace provide the same functionality. The type conversion functionality in .NET also allows for TypeConverters to be explicitly registered with a data type. This allows for transparent setting of complex object properties. However, some classes in the .NET framework do not support the style of conversion we are used to from Spring.Java, such as setting of a string[] with a comma delimited string. The type converter, StringArrayConverter in the Spring.Objects.TypeConverters namespace is therefore explicitly registered with Spring.NET in order to provide this functionality. As in the case of Spring.Java, Spring.NET allows user defined type converters to be registered. However, if you are creating a custom type in .NET, using the standard .NET mechanisms for type conversion is the preferred approach.

20.4. ResourceBundle-ResourceManager

20.5. Exceptions

Exceptions in Java can either be checked or unchecked. .NET supports only unchecked exceptions. Spring.Java prefers the use of unchecked exceptions, frequently making conversions from checked to unchecked exceptions. In this respect Spring.Java is similar to the default behavior of .NET

20.6. Application Configuration

In Spring.Java it is very common to create an ObjectFactory or ApplicaitonContext from an external XML configuration file This functionality is also provided in Spring.NET. However, in .NET the System.Configuration namespace provides support for managing application configuration information. The functionality in this namespace depends on the availability of specially named files: Web.config for ASP.NET applications and <MyExe>.exe.config for WinForms and console applications. <MyExe> is the name of your executable. As part of the compilation process, if you have a file name App.config in the root of your project, the compiler will rename the file to <MyExe>.exe.config and place it into the runtime executable folder.

These application configuration files are XML based and contain configuration sections that can be referenced by name to retrieve custom configuration objects. In order to inform the .NET configuration system how to create a custom configuration object from one of these sections, an implementation of the interface, IConfigurationSectionHandler, needs to be registered. Spring.NET provides two implementations, one to create an IApplicationContext from a `<context>` section and another to configure the context with object definitions containedin an `<objects>` section. The `<context>` section is very powerful and expressive. It provides full support for locating all `IResource` via uri syntax and hierarchical contexts without coding or using more verbose XML as would be required in the current version of spring.Java

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<configuration>

    <configSections>
        <sectionGroup name="spring">
            <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
            <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
        </sectionGroup>
    </configSections>

    <spring>

        <context>
            <resource uri="config://spring/objects"/>
        </context>

        <objects>
            <description>An  example that demonstrates simple IoC features.</description>
            <object name="MyMovieLister" type="Spring.Examples.MovieFinder.MovieLister, MovieFinder">
                <property name="movieFinder" ref="AnotherMovieFinder"/>
            </object>
            <object name="MyMovieFinder" type="Spring.Examples.MovieFinder.SimpleMovieFinder, MovieFinder"/>
            <!--
            An IMovieFinder implementation that uses a text file as it's movie source...
            -->
            <object name="AnotherMovieFinder" type="Spring.Examples.MovieFinder.ColonDelimitedMovieFinder, Movie
                <constructor-arg index="0" value="movies.txt"/>
            </object>
        </objects>

    </spring>

</configuration>
```

The <configSections> and <section> elements are a standard part of the .NET application configuration file. These elements are used to register an instance of IConfigurationSecitonHandler and associate it with another xml element in the file, in this case the <context> and <objects> elements.

The following code segment is used to retrieve the IApplicationContext from the .NET application configuration file.

```
IApplicationContext ctx
          = ConfigurationUtils.GetSection("spring/context") as IApplicationContext;
```

In order to enforce the usage of the named configuration section `spring/context` the preferred instantiation mechanism is via the use of the registry class ContextRegistry as shown below

```
IApplicationContext ctx = ContextRegistry.GetContext();
```

20.7. AOP Framework

20.7.1. Cannot specify target name at the end of interceptorNames for ProxyFactoryObject

When configuring the list of interceptor names on a `ProxyFactoryObject` instance (or object definition), one *cannot* specify the name of the target (i.e. the object being proxied) at the end of the list of interceptor names. This shortcut *is* valid in Spring Java, where the `ProxyFactoryBean` will automatically detect this, and use the last name in the interceptor names list as the target of the `ProxyFactoryBean`. The following configuration, which would be valid in Spring Java (barring the obvious element name changes), is **not** valid in Spring.NET (so don't do it).

```
<?xml version="1.0" encoding="utf-8" ?>
    <objects>
        <object id="target" type="Spring.Objects.TestObject">
            <property name="name" value="Bingo"/>
        </object>

        <object id="nopInterceptor" type="Spring.Aop.Interceptor.NopInterceptor"/>

        <object id="prototypeTarget" type="Spring.Aop.Framework.ProxyFactoryObject">
            <property name="interceptorNames" value="nopInterceptor,target"/> <!-- not valid! -->
        </object>
    </objects>
```

In Spring.NET, the `InterceptorNames` property of the `ProxyFactoryObject` can *only* be used to specify the names of interceptors. Use the `TargetName` property to specify the name of the target object that is to be proxied.

The main reason for not supporting exactly the same style of configuration as Spring Java is because this 'feature' is regarded as a legacy holdover from Rod Johnson's initial Spring AOP implementation, and is currently only kept as-is (in Spring Java) for reasons of backward compatibility.

Appendix A. Spring.NET's `spring-objects.xsd`

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns="http://www.springframework.net" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:vs="http://s

    <xs:annotation>

        <xs:documentation>

            Spring Objects XML Schema Definition

            Based on Spring Beans DTD, authored by Rod Johnson &amp; Juergen Hoeller


            Author: Griffin Caprio


            This defines a simple and consistent way of creating a namespace

            of managed objects configured by a Spring XmlObjectFactory.

            This document type is used by most Spring functionality, including

            web application contexts, which are based on object factories.


            Each object element in this document defines an object.

            Typically the object type (System.Type is specified, along with plain vanilla

            object properties.


            Object instances can be "singletons" (shared instances) or "prototypes"

            (independent instances).


            References among objects are supported, i.e. setting an object property

            to refer to another object in the same factory or an ancestor factory.


            As alternative to object references, "inner object definitions" can be used.

            Singleton flags and names of such "inner object" are always ignored:

            Inner object are anonymous prototypes.


            There is also support for lists, dictionaries, and sets.

        </xs:documentation>

    </xs:annotation>

    <xs:annotation>

        <xs:documentation>Defines a base type for any required string.  Defines a string with a minimum length c

    </xs:annotation>

    <xs:simpleType name="nonNullString">
```

```
        <xs:restriction base="xs:string">
            <xs:minLength value="1"/>
        </xs:restriction>
    </xs:simpleType>
    <xs:annotation>
        <xs:documentation>
            Element containing informative text describing the purpose of the enclosing
            element. Always optional.
            Used primarily for user documentation of XML object definition documents.
        </xs:documentation>
    </xs:annotation>
    <xs:simpleType name="description">
        <xs:restriction base="nonNullString"/>
    </xs:simpleType>
    <xs:complexType name="valueObject">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="type" type="nonNullString" use="optional"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <!--
         Defines a reference to another object in this factory or an external
         factory (parent or included factory).
    -->
    <xs:complexType name="objectReference">
        <xs:attribute name="object" type="nonNullString" use="optional"/>
        <xs:attribute name="local" type="xs:IDREF" use="optional"/>
        <xs:attribute name="parent" type="nonNullString" use="optional"/>
        <!--
             References must specify a name of the target object.
             The "object" attribute can reference any name from any object in the context,
             to be checked at runtime.
             Local references, using the "local" attribute, have to use object ids;
             they can be checked by this DTD, thus should be preferred for references
             within the same object factory XML file.
        -->
    </xs:complexType>
    <!-- Defines a reference to another object or a type. -->
    <xs:complexType name="objectOrClassReference">
        <xs:attribute name="object" type="nonNullString" use="optional"/>
        <xs:attribute name="local" type="xs:IDREF" use="optional"/>
        <xs:attribute name="type" type="nonNullString" use="optional"/>
    </xs:complexType>
    <xs:group name="objectList">
        <xs:sequence>
            <xs:element name="description" type="description" minOccurs="0"/>
            <xs:choice>
                <xs:element name="object" type="vanillaObject"/>
                <!--
                     Defines a reference to another object in this factory or an external
                     factory (parent or included factory).
                -->
                <xs:element name="ref" type="objectReference"/>
                <!--
                     Defines a string property value, which must also be the id of another
                     object in this factory or an external factory (parent or included factory).
                     While a regular 'value' element could instead be used for the same effect,
                     using idref in this case allows validation of local object ids by the xml
                     parser, and name completion by helper tools.
                -->
                <xs:element name="idref" type="objectReference"/>
                <!--
                     A objectList can contain multiple inner object, ref, collection, or value elements.
                     Lists are untyped, pending generics support, although references will be
                     strongly typed.
                     A objectList can also map to an array type. The necessary conversion
                     is automatically performed by AbstractObjectFactory.
                -->
                <xs:element name="list">
                    <xs:complexType>
                        <xs:group ref="objectList" minOccurs="0" maxOccurs="unbounded"/>
                    </xs:complexType>
                </xs:element>
                <!--
                     A set can contain multiple inner object, ref, collection, or value elements.
                     Sets are untyped, pending generics support, although references will be
                     strongly typed.
                -->
                <xs:element name="set">
                    <xs:complexType>
                        <xs:group ref="objectList" minOccurs="0" maxOccurs="unbounded"/>
                    </xs:complexType>
                </xs:element>
                <!--
                     A Spring map is a mapping from a string key to object (a .NET IDictionary).
                     Maps may be empty.
                -->
                <xs:element name="dictionary" type="objectMap"/>
                <!--
                     Name-values elements differ from map elements in that values must be strings.
                     Name-values may be empty.
                -->
                <xs:element name="name-values" type="objectNameValues"/>
                <!--
                     Contains a string representation of a property value.
                     The property may be a string, or may be converted to the
                     required type using the System.ComponentModel.TypeConverter
                     machinery. This makes it possible for application developers
                     to write custom TypeConverter implementations that can
                     convert strings to objects.

                     Note that this is recommended for simple objects only.
                     Configure more complex objects by setting properties to references
                     to other objects.
                -->
                <xs:element name="value" type="valueObject"/>
                <!--
                     Denotes a .NET null value. Necessary because an empty "value" tag
                     will resolve to an empty String, which will not be resolved to a
                     null value unless a special TypeConverter does so.
                -->
                <xs:element name="null"/>
            </xs:choice>
        </xs:sequence>
    </xs:group>
    <xs:complexType name="objectNameValues">
```

```
        <xs:sequence>
            <!--
                 The "value" attribute is the string value of the property. The "key"
                 attribute is the name of the property.
            -->
            <xs:element name="add" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType mixed="true">
                    <xs:attribute name="key" type="nonNullString" use="required"/>
                    <xs:attribute name="value" use="required" type="xs:string"/>
                    <!-- Each property element must specify both the key and value. -->
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="importElement">
        <xs:attribute name="resource" type="nonNullString" use="required"/>
    </xs:complexType>
    <xs:complexType name="aliasElement">
        <xs:attribute name="name" type="nonNullString" use="required"/>
        <xs:attribute name="alias" type="nonNullString" use="required"/>
    </xs:complexType>
    <xs:complexType name="objectMap">
        <xs:sequence>
            <xs:element type="mapEntryElement" name="entry" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="mapEntryElement">
        <xs:sequence>
            <xs:element type="mapKeyElement" name="key" minOccurs="0" maxOccurs="1"/>
            <xs:group ref="objectList" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="key" type="nonNullString" use="optional"/>
        <xs:attribute name="value" type="nonNullString" use="optional"/>
        <xs:attribute name="key-ref" type="nonNullString" use="optional"/>
        <xs:attribute name="value-ref" type="nonNullString" use="optional"/>
    </xs:complexType>
    <xs:complexType name="mapKeyElement">
        <xs:group ref="objectList" minOccurs="1"/>
    </xs:complexType>
    <xs:annotation>
        <xs:documentation>Defines constructor argument.</xs:documentation>
    </xs:annotation>
    <xs:complexType name="constructorArgument">
        <xs:group ref="objectList" minOccurs="0"/>
        <xs:attribute name="name" type="nonNullString" use="optional"/>
        <xs:attribute name="index" type="nonNullString" use="optional"/>
        <xs:attribute name="type" type="nonNullString" use="optional"/>
        <xs:attribute name="value" type="nonNullString" use="optional"/>
        <xs:attribute name="ref" type="nonNullString" use="optional"/>
        <!--
              The constructor-arg tag can have an optional index attribute,
              to specify the exact index in the constructor argument list. Only needed
              to avoid ambiguities, e.g. in case of 2 arguments of the same type.
        -->
        <!--
              The constructor-arg tag can have an optional named parameter attribute,
              to specify a named parameter in the constructor argument list.
        -->
        <!--
              The constructor-arg tag can have an optional type attribute,
              to specify the exact type of the constructor argument. Only needed
              to avoid ambiguities, e.g. in case of 2 single argument constructors
              that can both be converted from a String.
        -->
    </xs:complexType>
    <xs:annotation>
        <xs:documentation>Defines property.</xs:documentation>
    </xs:annotation>
    <xs:complexType name="property">
        <xs:group ref="objectList" minOccurs="0"/>
        <!-- The property name attribute is the name of the objects property. -->
        <xs:attribute name="name" type="nonNullString" use="required"/>
        <xs:attribute name="value" type="nonNullString" use="optional"/>
        <xs:attribute name="ref" type="nonNullString" use="optional"/>
    </xs:complexType>
    <xs:annotation>
        <xs:documentation>Defines a single named object.</xs:documentation>
    </xs:annotation>
    <xs:complexType name="vanillaObject">
        <xs:sequence>
            <xs:element name="description" type="description" minOccurs="0" maxOccurs="1"/>
            <!--
                 Object definitions can specify zero or more constructor arguments.
                 They correspond to either a specific index of the constructor argument list
                 or are supposed to be matched generically by type.
                 This is an alternative to "autowire constructor".
            -->
            <xs:element name="constructor-arg" type="constructorArgument" minOccurs="0" maxOccurs="unbounded"/>
            <!--
                 Object definitions can have zero or more properties.
                 Spring supports primitives, references to other objects in the same or
                 related factories, lists, dictionaries and properties.
            -->
            <xs:element name="property" type="property" minOccurs="0" maxOccurs="unbounded"/>
            <!-- Object definitions can have zero or more subscriptions. -->
            <xs:element name="listener" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ref" type="objectOrClassReference" minOccurs="0" maxOccurs="unbounded"
                    </xs:sequence>
                    <xs:attribute name="event" type="nonNullString" use="optional"/>
                    <xs:attribute name="method" type="nonNullString" use="required"/>
                    <!-- The event(s) the object is interested in. -->
                    <!-- The name or name pattern of the method that will handle the event(s). -->
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="optional"/>
        <xs:attribute name="name" type="nonNullString" use="optional"/>
        <xs:attribute name="type" type="nonNullString" use="optional"/>
        <xs:attribute name="parent" type="nonNullString" use="optional"/>
        <xs:attribute name="abstract" type="xs:boolean" use="optional" default="false"/>
        <xs:attribute name="singleton" type="xs:boolean" use="optional" default="true"/>
        <xs:attribute name="scope" use="optional" default="application">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="application"/>
                    <xs:enumeration value="session"/>
                    <xs:enumeration value="request"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="lazy-init" use="optional" default="default">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="true"/>
                    <xs:enumeration value="false"/>
                    <xs:enumeration value="default"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="autowire" use="optional" default="default">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="no"/>
                    <xs:enumeration value="byName"/>
                    <xs:enumeration value="byType"/>
                    <xs:enumeration value="constructor"/>
                    <xs:enumeration value="autodetect"/>
                    <xs:enumeration value="default"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="dependency-check" use="optional" default="default">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="none"/>
                    <xs:enumeration value="objects"/>
                    <xs:enumeration value="simple"/>
                    <xs:enumeration value="all"/>
                    <xs:enumeration value="default"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="depends-on" type="nonNullString" use="optional"/>
        <xs:attribute name="init-method" type="nonNullString" use="optional"/>
```

```
        <xs:attribute name="destroy-method" type="nonNullString" use="optional"/>
        <xs:attribute name="factory-method" type="nonNullString" use="optional"/>
        <xs:attribute name="factory-object" type="nonNullString" use="optional"/>
        <!--
             Objects can be identified by an id, to enable reference checking.
             There are constraints on a valid XML id: if you want to reference your object
             in .NET code using a name that's illegal as an XML id, use the optional
             "name" attribute. If neither given, the object type name is used as id.
        -->
        <!--
             Optional. Can be used to create one or more aliases illegal in an id.
             Multiple aliases can be separated by any number of spaces or commas.
        -->
        <!--
             Each object definition must specify the full, assembly qualified of the type,
             or the name of the parent object from which the type can be worked out.

             Note that a child object definition that references a parent will just
             add respectively override property values and be able to change the
             singleton status. It will inherit all of the parent's other parameters
             like lazy initialization or autowire settings.
        -->
        <!--
             Is this object "abstract", i.e. not meant to be instantiated itself but
             rather just serving as parent for concrete child object definitions?
             Default is false. Specify true to tell the object factory to not try to
             instantiate that particular object in any case.
        -->
        <!--
             Is this object a "singleton" (one shared instance, which will
             be returned by all calls to GetObject() with the id),
             or a "prototype" (independent instance resulting from each call to
             getObject(). Default is singleton.

             Singletons are most commonly used, and are ideal for multi-threaded
             service objects.
        -->
        <!--
             Optional attribute controlling the scope of singleton instances. It is
             only applicable to ASP.Net web applications and it has no effect on prototype
             objects. Applications other than ASP.Net web applications simply ignore this attribute.
             It has 3 possible values:
             1. "application"
             Default object scope. Objects defined with application scope will behave like
             traditional singleton objects. Same instance will be returned from every call
             to IApplicationContext.GetObject()

             2. "session"
             Objects with this scope will be stored within user's HTTP session. Session scope
             is typically used for objects such as shopping cart, user profile, etc.

             3. "request"
             Object with this scope will be initialized for each HTTP request, but unlike with prototype
             objects, same instance will be returned from all calls to IApplicationContext.GetObject()
             within the same HTTP request. For example, if one ASP page forwards request to another using
             Server.Transfer method, they can easily share the state by configuring dependency to the same
             request-scoped object.
        -->
        <!--
             If this object should be lazily initialized.
             If false, it will get instantiated on startup by object factories
             that perform eager initialization of singletons.
        -->
        <!--
             Optional attribute controlling whether to "autowire" object properties.
             This is an automagical process in which object references don't need to be coded
             explicitly in the XML object definition file, but Spring works out dependencies.

             There are 5 modes:

             1. "no"
             The traditional Spring default. No automagical wiring. Object references
             must be defined in the XML file via the <ref> element. We recommend this
             in most cases as it makes documentation more explicit.

             2. "byName"
             Autowiring by property name. If a object of class Cat exposes a dog property,
             Spring will try to set this to the value of the object "dog" in the current factory.
```

```
          3. "byType"
          Autowiring if there is exactly one object of the property type in the object factory.
          If there is more than one, a fatal error is raised, and you can't use byType
          autowiring for that object. If there is none, nothing special happens - use
          dependency-check="objects" to raise an error in that case.

          4. "constructor"
          Analogous to "byType" for constructor arguments. If there isn't exactly one object
          of the constructor argument type in the object factory, a fatal error is raised.

          5. "autodetect"
          Chooses "constructor" or "byType" through introspection of the object class.
          If a default constructor is found, "byType" gets applied.

          The latter two are similar to PicoContainer and make object factories simple to
          configure for small namespaces, but doesn't work as well as standard Spring
          behaviour for bigger applications.

          Note that explicit dependencies, i.e. "property" and "constructor-arg" elements,
          always override autowiring. Autowire behaviour can be combined with dependency
          checking, which will be performed after all autowiring has been completed.
     -->
     <!--
          Optional attribute controlling whether to check whether all this
          objects dependencies, expressed in its properties, are satisfied.
          Default is no dependency checking.

          "simple" type dependency checking includes primitives and String
          "object" includes collaborators (other objects in the factory)
          "all" includes both types of dependency checking
     -->
     <!--
          The names of the objects that this object depends on being initialized.
          The object factory will guarantee that these objects get initialized before.

          Note that dependencies are normally expressed through object properties or
          constructor arguments. This property should just be necessary for other kinds
          of dependencies like statics (*ugh*) or database preparation on startup.
```

```
        -->
        <!--
             Optional attribute for the name of the custom initialization method
             to invoke after setting object properties. The method must have no arguments,
             but may throw any exception.
        -->
        <!--
             Optional attribute for the name of the custom destroy method to invoke
             on object factory shutdown. The method must have no arguments,
             but may throw any exception. Note: Only invoked on singleton objects!
        -->
    </xs:complexType>
    <xs:annotation>
        <xs:documentation>The document root.  At least one object definition is required.</xs:documentation>
    </xs:annotation>
    <xs:element name="objects">
        <xs:complexType>
            <xs:choice maxOccurs="unbounded">
                <xs:element name="description" type="description" minOccurs="0" maxOccurs="1"/>
                <xs:element name="import" type="importElement" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="alias" type="aliasElement" minOccurs="0" maxOccurs="unbounded"/>
                <xs:element name="object" type="vanillaObject" minOccurs="0" maxOccurs="unbounded"/>
            </xs:choice>
            <xs:attribute name="default-lazy-init" type="xs:boolean" use="optional" default="false"/>
            <xs:attribute name="default-dependency-check" use="optional" default="none">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="none"/>
                        <xs:enumeration value="objects"/>
                        <xs:enumeration value="simple"/>
                        <xs:enumeration value="all"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="default-autowire" use="optional" default="no">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="no"/>
                        <xs:enumeration value="byName"/>
                        <xs:enumeration value="byType"/>

                        <xs:enumeration value="constructor"/>

                        <xs:enumeration value="autodetect"/>

                    </xs:restriction>

                </xs:simpleType>

            </xs:attribute>

            <!--

                 Default values for all object definitions. Can be overridden at

                 the "object" level. See those attribute definitions for details.

            -->

        </xs:complexType>

    </xs:element>

</xs:schema>
```