إلى قارئ هذا الكتاب ، تحية طيبة وبعد ...

لقد أصبحنا نعيش في عالم يعج بالأبحاث والكتب والمعلومات، وأصبح العلم معياراً حقيقياً لتفاضل الأمم والدول والمؤسسات والأشخاص على حدٍّ سواء، وقد أمسى بدوره حلاً شبه وحيدٍ لأكثر مشاكل العالم حدة وخطورة، فالبيئة تبحث عن حلول، وصحة الإنسان تبحث عن حلول، والموارد التي تشكل حاجة أساسية للإنسان تبحث عن حلول كذلك، والطاقة والغذاء والماء جميعها تحديات يقف العلم في وجهها الآن ويحاول أن يجد الحلول لها. فأين نحن من هذا العلم ؟ وأين هو منا؟

نسعى في موقع عالم الإلكترون www.4electron.com لأن نوفر بين أيدي كل من حمل على عاتقه مسيرة درب تملؤه التحديات ما نستطيع من أدوات تساعده في هذا الدرب، من مواضيع علمية، ومراجع أجنبية بأحدث إصداراتها، وساحات لتبادل الآراء والأفكار العلمية والمرتبطة بحياتنا الهندسية، وشروحٍ لأهم برمجيات الحاسب التي تتداخل مع تطبيقات الحياة الأكاديمية والعملية، ولكننا نتوقع في نفس الوقت أن نجد بين الطلاب والمهندسين والباحثين من يسعى مثلنا لتحقيق النفع والفائدة للجميع، ويحلم أن يكون عضواً في مجتمعٍ يساهم بتحقيق بيئة خصبة للمواهب والإبداعات والتألق، فهل تحلم بذلك ؟

حاول أن تساهم بفكرة، بومضة من خواطر تفكيرك العلمي، بفائدة رأيتها في إحدى المواضيع العلمية، بجانب مضيء لمحته خلف ثنايا مفهوم هندسي ما. تأكد بأنك ستلتمس الفائدة في كل خطوة تخطوها، وترى غيرك يخطوها معك ...

أخي القارئ، نرجو أن يكون هذا الكتاب مقدمة لمشاركتك في عالمنا العلمي التعاوني، وسيكون موقعكم عالم الإلكترون ww.4electron.com بكل الإمكانيات المتوفرة لديه جاهزاً على الدوام لأن يحقق البيئة والواقع الذي يبحث عنه كل باحث أو طالب في علوم الهندسة، ويسعى فيه للإفادة كل ساعٍ ، فأهلاً وسهلاً بكم .

مع تحيات إدارة الموقع وفريق عمله

www.4electron.com

CSLA .NET Version 2.1 Handbook

ROCKFORD LHOTKA

CSLA .NET Version 2.1 Handbook

Revision 2

Editor:	Teresa Lhotka
Technical reviewer:	Brant Estes

The source code for this book (CSLA .NET 2.1.3) is available at http://www.lhotka.net/cslanet.

Acknowledgements

Neither this book, nor CSLA .NET version 2.1, would have been possible without support from Magenic Technologies. Magenic is the premier .NET development company in the US, and is a Microsoft Gold Certified Partner.

You can reach Magenic at http://www.magenic.com.

CSLA .NET has attracted a community of very thoughtful, intelligent and dedicated people. You can find many of them at http://forums.lhotka.net.

The bug fixes and feature enhancements described in this book come, in no small part, through the encouragement and feedback provided by this stellar community. Thank you all!

Special thanks to Andrés Villanueva (Xal), who provided a great deal of feedback and help with testing.

And thank you to Chris Russi, who created the new CSLA .NET logo graphics such as the one on this page.

About the Author

Rockford Lhotka is the author of numerous books, including *Expert VB 2005 Business Objects* and *Expert C# 2005 Business Objects*. He is a Microsoft regional director, a Microsoft MVP, and an INETA speaker. Rockford speaks at many conferences and user groups around the world and is a columnist for MSDN Online. Rockford is the principal technology evangelist for Magenic Technologies, one of the nation's premiere Microsoft gold certified partners dedicated to solving today's most challenging business problems using 100-percent Microsoft tools and technology.

Contents

List of Tables

List of Figures

Introduction

Welcome to the CSLA .NET Version 2.1 Handbook. This book covers the features, enhancements and changes made to the CSLA .NET framework during the creation of version 2.1.

> **Note:** In the process of writing this book some changes were made to the framework. The exact version of the framework that corresponds to this book is version 2.1.3, which you can download from http://www.lhotka.net/cslanet/download.aspx.

Before Reading this Book

CSLA .NET version 2.0 is part of the *Expert VB 2005 Business Objects* and *Expert C# 2005 Business Objects* books, published by Apress (http://www.apress.com). These books are ISBN 1590596315 and ISBN 1590596323 respectively.

This Handbook assumes that you have read and are familiar with the content from the *Expert VB 2005 Business Objects* or *Expert C# 2005 Business Objects* book.

> **Note:** CSLA stands for Component-based, Scalable, Logical Architecture

The CSLA .NET framework is licensed according to the license at http://www.lhotka.net/cslanet/license.aspx.

CSLA .NET version 2.1 is an evolutionary step forward from version 2.0. It includes some bug fixes and minor enhancements made in versions 2.0.1 through 2.0.3, plus a set of more substantial changes to the framework.

Organization of the Book

The enhancements made in version 2.1 are fairly wide-ranging, and so they affect many parts of the framework itself, and enable a number of new capabilities for developers building applications based on business objects. Some enhancements affect a single class, others affect many classes. Some framework classes have been changed due to multiple enhancements.

To provide some level of order, this book is organized around feature enhancements. Each enhancement or change will first be discussed in terms of its impact on CSLA .NET, and then in terms of how it can be used when building applications.

In each section, you can choose whether to read the details behind the change, or skip through to the discussion on how to use the enhancement in your application development.

Breaking Changes from CSLA .NET version 2.0

Where possible, I have attempted to avoid breaking existing code based on version 2.0, but there are some cases where breaking changes were required. Table 1 lists the breaking changes and their likely severity.

Summary	Severity
The calling semantics for `DataPortal.Create<T>()` and `DataPortal.Fetch<T>()` (with no criteria at all) are different. They now invoke `DataPortal_Create()` and `DataPortal_Fetch()` (with no criteria parameter) respectively.	**affects everyone**
The `Overridable/virtual DataPortal_Create()` methods declared in `BusinessBase` and `BusinessListBase` have changed their signature.	**affects virtually everyone**
Per-type validation rules requires code changes when moving from 2.0 to 2.1.	**affects virtually everyone**
Per-type authorization rules may require code changes when moving from 2.0 to 2.1.	**may affect** your code
`Csla.DataPortalException` now includes the original exception message text in its message text to assist in debugging.	**unlikely** to break your code
The `Parent` property in `BusinessBase` is now of type `IParent`.	**unlikely** to break your code
The `RunLocal` attribute is no longer inherited from base class methods when the methods are overridden by a subclass.	**unlikely** to break your code
The order in which `OnDeserialized()` is called has changed.	**unlikely** to break your code

Table 1. Breaking changes in version 2.1

Each of these breaking changes flows from a specific enhancement or change made to the framework for version 2.1. I will discuss the nature of the breaking change along with the enhancement later in the book.

Known Issues with version 2.1

There is one known issue where version 2.1does not function as expected Table 2 identifies the known issue with version 2.1.

Class	Summary
`CslaDataSource`	You can not add a `CslaDataSource` to a page by choosing to "add a new data source" from within the `GridView` or `DetailsView` controls. Attempting to do this will result in an exception that prevents the control from displaying properly. I have been unable to resolve this issue, but there is a viable workaround. You must manually add a `CslaDataSource` control to your page, either using drag-and-drop from the Toolbox, or by typing the tag into the page. At this point you can configure the assembly/type information in the data source control. You can then choose this data source control as the data source for your `GridView` or `DetailsView` control.

Table 2. Known issue with version 2.1

I am continuing to research this issue. As I learn more, I hope to resolve this issue to provide full integration of the control into the Visual Studio environment.

Summary of Changes and Enhancements

While CSLA .NET version 2.1 is an evolutionary update from version 2.0, it does include some substantial changes, which involve parts of the CSLA .NET framework code, and enable some powerful capabilities for your business development efforts.

At a high level, the changes can be grouped into a set of functional enhancements as listed in Table 3.

Enhancement	Summary
Validation rules	Enhancements for performance, reduction of memory usage and new features and capabilities
Authorization rules	Enhancements for performance and reduction of memory usage
`FilteredBindingList`	A new class that allows you to create a filtered view of any list or collection
`EditableRootListBase`	A new base class that allows you to create a collection of editable root objects (objects derived from `BusinessBase`)
Changes to `Csla.Core`	Various changes and additions to the `Csla.Core` namespace to support the other enhancements listed here, and to enable new scenarios for business and UI developers
`LocalContext`	A new property on `ApplicationContext` to allow you to more easily pass global values to all data access code on a server
Data portal	Address consistency issues with the data portal that were introduced in version 2.0.2, and minor enhancements to the Remoting channel
`SmartDate`	Provide simpler and more explicit syntax for creating `SmartDate` objects, and enable more flexible formatting of date values
`CslaDataSource`	Add support for collections that provide paging and sorting functionality
Miscellaneous	Many minor enhancements and bug fixes to various pre-existing features

Table 3. Functional enhancements in version 2.1

The rest of this book will address the changes in each of these functional areas.

Validation Rules

Perhaps the single biggest set of changes in version 2.1 involve validation rules processing. These changes improve performance, reduce memory consumption and add new capabilities in terms of how broken validation rules can be expressed.

CSLA .NET 2.0 includes a validation rules processing mechanism where each validation rule is implemented as a method. The method signature of each of the *rule methods* is defined by the `Csla.Validation.RuleHandler` delegate. The following is an example of a simple rule method:

```
Private Function MyRuleMethod( _
  ByVal target As Object, ByVal e As RuleArgs) As Boolean

  Dim result As Boolean

  If <condition is met> Then
    result = True

  Else
    e.Description = "Human readable description"
    result = False
  End If

  Return result

End Function
```

When a business object is created, an `AddBusinessRules()` method is called, allowing the object to associate rule methods with properties. For instance:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule( _
    AddressOf MyRuleMethod, "MyProperty")

End Sub
```

Rules for a property are automatically checked when the `PropertyHasChanged()` method is called within a property `Set` block. You can also explicitly check the rules for a property by calling `ValidationRules.CheckRules(propertyName)`, or for all properties by calling `ValidationRules.CheckRules()`.

While this mechanism works very well, it has some drawbacks and limitations which version 2.1 seeks to address. Table 4 lists the new concepts introduced in version 2.1.

Concept	Description
Per-type rules	Rather than maintaining a list of rules methods for each property in each object instance, you can now maintain a list of rules for each property at a class, or type, level. This increases performance and reduces memory usage, because the associations between rules and properties are stored just once for all object instances. The concept of per-instance rule methods remains in the framework, but is no longer the default (or recommended) approach.
Dependant properties	In your `AddBusinessRules()` method, you can now call `ValidationRules.AddDependantProperty()` to indicate that one property depends on another. In practical terms, this means that when validation rules are checked for one property, the rules for the dependant property are also checked.
Rule severity	Within a rule method, you can now specify the severity of a rule if it is broken. The severity can be one of `Error`, `Warning` or `Information`.
Rule priority	When associating a rule method with a property, you can now specify a priority for the rule. When the rules for a property are checked, they are checked in priority order (starting with priority 0 and counting up).
Short-circuiting	Short-circuiting allows you to stop the processing of rules for a property under certain conditions. The most direct technique is for the rule method to set `e.StopProcessing` to `True`. Also, when using priority-based rules, you can set a threshold so rules below a certain priority will *only* be processed if no previous rule has been broken.
Strongly-typed rule methods	Using generics, you can now define a rule method that accepts strongly typed parameters for `target` and arguments (`e`).
Retrieve rules for an object	You can now retrieve a list of the rules associated with the properties of an object. This is exposed as a `Protected` method in `BusinessBase`, and it returns an array of `String` values with this format: `rule://ruleMethod/propertyName?arg1=x&arg2=y`
Retrieve array of broken rules	The `BrokenRulesCollection` class now has a `ToArray()` method that returns an array of `String` values containing the human readable descriptions of all broken rules in the object.

Table 4. New validation concepts in version 2.1

Implementing these changes required the addition of some new classes to the CSLA .NET framework, and changes to a number of existing classes. As much as possible, I preserved backward compatibility with existing, version 2.0, code, but there are some breaking changes as a result of these enhancements.

Before discussing how to use these enhancements, let's walk through the changes to the CSLA .NET framework itself.

Framework Changes

Enhancing the validation rule processing in CSLA .NET involved changing and adding a number of classes. Here is a list of changed classes or types:

- `BusinessBase` (from `Csla.Core`)
- `BrokenRule`
- `BrokenRulesCollection`
- `CommonRules`
- `RuleArgs`
- `RuleHandler`
- `RuleMethod`
- `ValidationRules`

And here is a list of new classes or types:

- `RuleSeverity`
- `RulesList`
- `SharedValidationRules`
- `ValidationRulesManager`

As you can see, virtually every class in the `Csla.Validation` namespace was affected by these changes. Let's walk through each functional enhancement and examine the changes required to implement each. Keep in mind that some of these changes are interrelated, so you may see some unfamiliar code in some earlier sections. This code will be more fully explained later.

Implementing Per-Type Validation Rules

Version 2.1 adds the concept of per-type rule methods, while retaining per-instance rule methods. However, the default behavior is now to use per-type validation rules, which means that `Csla.Core.BusinessBase.AddBusinessRules()` is now used to add per-type rules. Similarly, the `ValidationRules.AddRule()` method now adds per-type rules, rather than per-instance rules.

Changes to BusinessBase

To retain the per-instance rule support, a new method has been added to `Csla.Core.BusinessBase`:

```
    Protected Overridable Sub AddInstanceBusinessRules()

    End Sub
```

The constructor has been enhanced to always call the new `AddInstanceBusinessRules()` method, but to only call `AddBusinessRules()` if per-type rules haven't already been established:

```
    Protected Sub New()

      Initialize()
      AddInstanceBusinessRules()
      If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
        SyncLock Me.GetType
          If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
            AddBusinessRules()
          End If
        End SyncLock
      End If
      AddInstanceAuthorizationRules()
      If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
        SyncLock Me.GetType
          If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
            AddAuthorizationRules()
          End If
        End SyncLock
      End If

    End Sub
```

What this means, in practice, is that `AddBusinessRules()` is called just one time during the life of an `AppDomain`. Once the per-type rules are set up, they are retained by the `SharedValidationRules` object for the lifetime of the application.

The `SyncLock` statement ensures that two threads can't try to invoke `AddBusinessRules()` simultaneously. The odds of this occurring would be very small regardless, but this code structure ensures that it won't happen.

ValidationRulesManager Class

Before getting into the details of `SharedValidationRules`, we need to discuss the new `ValidationRulesManager` class. Since business objects may now have two lists of rule methods (one per-type, the other per-instance), a new type of object was required to manage these individual lists of rules. This is the purpose behind the `ValidationRulesManager` class.

`ValidationRulesManager` is a container for a list of rule methods associated with each of the business object's properties. This is implemented using a `Dictionary` object:

```
  Friend Class ValidationRulesManager

    Private mRulesList As _
      Generic.Dictionary(Of String, RulesList)

  End Class
```

Each item in the `Dictionary` is a `RulesList` object, which is a list of the rule methods associated with the specified property. The key value for the `Dictionary` is the property name.

This `Dictionary` object is created on-demand to minimize memory consumption and overhead. The `RulesDictionary` property implements this behavior:

```
Friend ReadOnly Property RulesDictionary() As _
             Generic.Dictionary(Of String, RulesList)
  Get
    If mRulesList Is Nothing Then
      mRulesList = New Generic.Dictionary(Of String, RulesList)
    End If
    Return mRulesList
  End Get
End Property
```

More interesting and complex, is the `GetRulesForProperty()` method. This method is responsible for finding and returning the `RulesList` object containing the rules for a specified property. However, there are two different scenarios under which this method might be called: adding rules to the property, or retrieving the rules for the property.

When the first rule is added for a property, a new `RulesList` object must be created to maintain the list of rules. On the other hand, when retrieving rules for a property; if there are no rules then `Nothing` is returned, and no `RulesList` object should be created. This behavior is controlled by the `createList` parameter:

```
Friend Function GetRulesForProperty( _
  ByVal propertyName As String, _
  ByVal createList As Boolean) As RulesList

  ' get the list (if any) from the dictionary
  Dim list As RulesList = Nothing
  If RulesDictionary.ContainsKey(propertyName) Then
    list = RulesDictionary.Item(propertyName)
  End If

  If createList AndAlso list Is Nothing Then
    ' there is no list for this name - create one
    list = New RulesList
    RulesDictionary.Add(propertyName, list)
  End If
  Return list

End Function
```

In either case, an attempt to retrieve any existing `RulesList` object is made by checking to see if the `Dictionary` contains a key corresponding to the property name. If such a list exists, it is returned. If no such list exists, and if `createList` is `True`, then a new `RulesList` object is created and added to the `Dictionary`.

The goal is to avoid creating `RulesList` objects or `Dictionary` entries where possible. If there are no rules for a property, then that property should incur no overhead in terms of memory consumption or object creation.

Obviously, there must be a way to add rules for a property. While this could be handled by any code calling `ValidationRulesManager`, I chose to implement `AddRule()` methods directly in `ValidationRulesManager` to centrally implement the behavior. The first `AddRule()` method adds simple rule methods:

```
Public Sub AddRule( _
  ByVal handler As RuleHandler, ByVal args As RuleArgs, ByVal priority As Integer)

  ' get the list of rules for the property
  Dim list As List(Of IRuleMethod) = _
    GetRulesForProperty(args.PropertyName, True).GetList(False)

  ' we have the list, add our new rule
  list.Add(New RuleMethod(handler, args, priority))

End Sub
```

Notice how `GetRulesForProperty()` is called, passing `True` for the `createList` parameter.

There's also an overload of `AddRule()` to handle strongly-typed rule methods:

```
Public Sub AddRule(Of T, R As RuleArgs)( _
  ByVal handler As RuleHandler(Of T, R), ByVal args As R, ByVal priority As Integer)

  ' get the list of rules for the property
  Dim list As List(Of IRuleMethod) = _
    GetRulesForProperty(args.PropertyName, True).GetList(False)

  ' we have the list, add our new rule
  list.Add(New RuleMethod(Of T, R)(handler, args, priority))

End Sub
```

It also turns out that the `RulesList` object is responsible for maintaining the list of other properties that are dependant on the current property. There's a `RulesList` object for each property with rules, and that `RulesList` maintains the names of any other properties whose rules should be checked any time *this* property's rules are checked.

Again, I chose to encapsulate the behavior of associating a dependant property using a method in the `ValidationRulesManager` class:

```
Public Sub AddDependantProperty( _
  ByVal propertyName As String, ByVal dependantPropertyName As String)

  ' get the list of rules for the property
  Dim list As List(Of String) = _
    GetRulesForProperty(propertyName, True).GetDependancyList(True)

  ' we have the list, add the dependency
  list.Add(dependantPropertyName)

End Sub
```

Though these `AddRule()` and `AddDependantProperty()` methods are scoped as `Public`, remember that `ValidationRulesManager` itself is scoped as `Friend`. All of this functionality exists for internal use by CSLA .NET, not directly by business or UI code.

RulesList Class

Each `RulesList` object exists to manage the list of rule methods and dependant properties for a given property. To maintain this data, each `RulesList` object keeps a `List` of rule methods and a `List` of dependant property names. It also maintains a flag indicating whether the list of rules have yet been sorted by priority:

```
Friend Class RulesList

  Private mList As New List(Of IRuleMethod)
  Private mSorted As Boolean
  Private mDependantProperties As List(Of String)

End Class
```

You've already seen how `ValidationRulesManager` calls an `Add()` method on `RulesList` to add a new rule method to the list of rules for a property. The `Add()` method not only adds the item, but also sets `mSorted` to `False`, because adding a new item to the list potentially upsets any pre-existing sort:

```
  Public Sub Add(ByVal item As IRuleMethod)

    mList.Add(item)
    mSorted = False

  End Sub
```

`ValidationRulesManager` also includes code that calls a `GetList()` method to get the list of rules contained within the `RulesList` object. This `GetList()` method is called in two different scenarios: one is when rules are being added to the list, the other is when the rules are being invoked by the `ValidationRules.CheckRules()` method. In this latter case, in order to implement rule priorities, the rules must be sorted by priority within the list. The functionality to support sorting will be discussed later in this book. To control whether the list should be sorted before being returned, the `applySort` parameter is used:

```
  Public Function GetList(ByVal applySort As Boolean) As List(Of IRuleMethod)

    If applySort AndAlso Not mSorted Then
      mList.Sort()
      mSorted = True
    End If
    Return mList

  End Function
```

The `mSorted` field is used to avoid re-sorting the list in the case that it has already been sorted. In normal usage, all rules are added when an object is first created, and then the `CheckRules()` method is called numerous times after that point. The `mSorted` field is an optimization to ensure that the contents of the list are only sorted when needed.

`ValidationRulesManager` calls a `GetDependancyList()` method, which returns the `List` of property names that are dependant on this property. This `List` object is created on-demand by the `GetDependancyList()` method:

```
  Public Function GetDependancyList(ByVal create As Boolean) As List(Of String)

    If mDependantProperties Is Nothing AndAlso create Then
      mDependantProperties = New List(Of String)
    End If
    Return mDependantProperties

  End Function
```

`ValidationRulesManager`, combined with `RulesList`, provide a powerful storage mechanism for the rule methods and dependant properties associated with each property in a

business object. A normal business object will have a per-type `ValidationRulesManager` object, and some objects may have a per-instance `ValidationRulesManager` object instead of, or in addition to, the per-type object.

SharedValidationRules Module

The `SharedValidationRules` type exists to maintain the per-type `ValidationRulesManager` objects for all business objects in the application. The word "shared" is used in the type name, because per-type rules are shared across all instances of a given type of business object.

The `ValidationRulesManager` objects are maintained in a `Dictionary`, keyed by the type of each business object in the application:

```
Friend Module SharedValidationRules

  Private mManagers As New Dictionary(Of Type, ValidationRulesManager)

End Module
```

The `BusinessBase` class uses `SharedValidationRules` to retrieve and manage the list of rule methods and dependant properties for each type of business object. The `SharedValidationRules` module implements a `GetManager()` method to allow retrieval of the appropriate `ValidationRulesManager` object for a specific business object type:

```
  Friend Function GetManager( _
    ByVal objectType As Type, ByVal create As Boolean) As ValidationRulesManager

    Dim result As ValidationRulesManager = Nothing
    If Not mManagers.TryGetValue(objectType, result) AndAlso create Then
      SyncLock mManagers
        result = New ValidationRulesManager
        mManagers.Add(objectType, result)
      End SyncLock
    End If
    Return result

  End Function
```

As with the previous classes, you can see that the `ValidationRulesManager` for a business object type is only created on-demand. The `create` parameter is used by `BusinessBase` to differentiate between calls to `GetManager()` for the purpose of adding new rules (in which case `create` is `True`); and retrieving rules for the `CheckRules()` implementation (in which case `create` is `False`).

`BusinessBase` also calls a `RulesExistFor()` method to determine whether per-type rules do exist for the business object type. This method simply calls the `ContainsKey()` method of the `Dictionary` object to determine if a `ValidationRulesManager` exists for the specified type.

Notice the use of the `SyncLock` statement in the `GetManager()` method. Because this method is `Shared` (due to being in a `Module`), it should be made threadsafe. In the case that multiple threads call `GetManager()` at the same time, `SyncLock` will ensure that only one thread at a time will execute the critical code in the method.

Changes to ValidationRules

The `ValidationRulesManager` and `RulesList` classes provide the basis for managing both per-type and per-instance rules. The `SharedValidationRules` module uses `ValidationRulesManager` to maintain the per-type rules. The per-instance rules, however, are still managed directly by the `ValidationRules` class, and so it has been changed to support the new per-instance model, as well as the per-type model.

To begin with, the declaration of the field to hold the per-instance rules is changed:

```
' reference to per-instance rules manager for this object
<NonSerialized()> _
Private mInstanceRules As ValidationRulesManager
' reference to per-type rules manager for this object
<NonSerialized()> _
Private mTypeRules As ValidationRulesManager
' reference to the active set of rules for this object
<NonSerialized()> _
Private mRulesToCheck As ValidationRulesManager
```

The `mRulesList` field is redefined as `mInstanceRules`, which is now of type `ValidationRulesManager`.

Additionally, the `mTypeRules` field is used to maintain a direct reference to the per-type `ValidationRulesManager` for this business object. Technically this isn't necessary, because you can always call `SharedValidationRules.GetManager()` to get the rules for a type, but there's some overhead to that call. Storing the reference in an instance field is a minor optimization of the process.

Finally, the `mRulesToCheck` field maintains a reference to the active set of rules used by the `CheckRules()` method. This is required because I chose to keep the per-instance rules concept while adding the per-type support. A business developer might choose to only use per-type rules, or only per-instance rules. Or they might choose to use some of each, within the same object.

Notice that these fields are marked with the `NonSerialized` attribute. This ensures that their contents won't be serialized if the business object is converted to a byte stream; to be moved across the network, for instance.

It is less expensive to recreate the property-rule associations for per-instance rules than it is to serialize and deserialize all this information.

Because the rule associations for per-type rules exsist at the AppDomain level, it makes no sense to serialize those associations as part of the object's state.

RulesToCheck Method

To optimize retrieval of the correct set of validation rules, the `RulesToCheck()` method evaluates the environment and returns the appropriate `ValidationRulesManager`, depending on what types of rules have been defined for the business object:

```
Private ReadOnly Property RulesToCheck() As ValidationRulesManager
  Get
    If mRulesToCheck Is Nothing Then
      Dim instanceRules As ValidationRulesManager = GetInstanceRules(False)
      Dim typeRules As ValidationRulesManager = GetTypeRules(False)
      If instanceRules Is Nothing Then
        If typeRules Is Nothing Then
```

```
            mRulesToCheck = Nothing

          Else
            mRulesToCheck = typeRules
          End If

        ElseIf typeRules Is Nothing Then
          mRulesToCheck = instanceRules

        Else
          ' both have values - consolidate into instance rules
          mRulesToCheck = instanceRules
          For Each de As Generic.KeyValuePair(Of String, RulesList) In _
                                                  typeRules.RulesDictionary
            Dim instanceList As List(Of IRuleMethod) = _
              mRulesToCheck.GetRulesForProperty(de.Key, True).GetList(False)
            instanceList.AddRange(de.Value.GetList(False))
          Next
        End If
      End If
      Return mRulesToCheck
    End Get
  End Property
```

The `mRulesToCheck` field is used as a cache to avoid the overhead of performing this evaluation more than once. If `mRulesToCheck` is `Nothing`, then the evaluation occurs. Otherwise, the pre-existing value is returned.

To determine which rules need checking, the method first retrieves the `ValidationRulesManager` objects for both per-type and per-instance rules. Notice that in both cases the parameter value is `False`, indicating that no `ValidationRulesManager` object should be created due to this operation. In other words, if no rules exist for this object, the result is `Nothing`.

Then the resulting values are evaluated. Table 5 lists the possible outcomes.

Per-type Rules	Per-instance Rules	Result
`Nothing`	`Nothing`	`Nothing`
Contains rules	`Nothing`	Per-type rules
`Nothing`	Contains rules	Per-instance rules
Contains rules	Contains rules	Consolidated list of per-type and per-instance rules

Table 5. Possible results of RulesToCheck method

The only complex part of the process occurs when both per-type and per-instance rules exist. In this case, the two lists must be merged into one, primarily to support the concepts of rule priority and short-circuiting, which I'll discuss later. As you can imagine, to get priority-sorted rules, all the rules for a property must be in a consolidated list; regardless of whether the association is per-type or per-instance.

> **Note:** Due to the overhead involved in merging per-type and per-instance rules into a consolidated list, I recommend you avoid using both per-type and per-instance rules if at all possible.

To avoid creating extra objects, the existing per-instance `ValidationRulesManager` object becomes the repository for *all* the object's validation rules. The rule methods from the per-type `ValidationRulesManager` are merged into the per-instance object:

```
            ' both have values - consolidate into instance rules
            mRulesToCheck = instanceRules
            For Each de As Generic.KeyValuePair(Of String, RulesList) In _
                                                  typeRules.RulesDictionary
              Dim instanceList As List(Of IRuleMethod) = _
                mRulesToCheck.GetRulesForProperty(de.Key, True).GetList(False)
              instanceList.AddRange(de.Value.GetList(False))
            Next
```

To do this, the code loops through each entry in the per-type object's `Dictionary`, copying the rule method objects from the per-type object into the corresponding per-instance object. The end result is that the per-instance `ValidationRulesManager` contains all the rules for the object.

Getting Instance and Type Rules

The `RulesToCheck()` method makes use of a couple helper methods implemented in `ValidationRules`: `GetInstanceRules()` and `GetTypeRules()`:

```
    Private Function GetInstanceRules( _
      ByVal createObject As Boolean) As ValidationRulesManager

      If mInstanceRules Is Nothing Then
        If createObject Then
          mInstanceRules = New ValidationRulesManager
        End If
      End If
      Return mInstanceRules

    End Function

    Private Function GetTypeRules( _
      ByVal createObject As Boolean) As ValidationRulesManager

      If mTypeRules Is Nothing Then
        mTypeRules = SharedValidationRules.GetManager(mTarget.GetType, createObject)
      End If
      Return mTypeRules

    End Function
```

Again, the creation of the `ValidationRulesManager` objects are controlled by a parameter, and the objects are only created on-demand.

Adding Per-Instance Validation Rules

Per-type rules are now the default, and so `ValidationRules.AddRule()` now adds a per-type rule. This means that the pre-existing `AddRule()` methods had to be renamed. They are now named `AddInstanceRule()`.

The behavior of `AddInstanceRule()` is the same as the `AddRule()` methods from version 2.0. The result is that a business developer who wants to use per-instance rules must override the `AddInstanceBusinessRules()` method.

> **Note:** I recommend avoiding per-instance rules if possible. Per-type rules provide performance and memory consumption benefits, and should be the preferred solution.

In `AddInstanceBusinessRules()`, they must call `AddInstanceRule()` methods to associate rule methods with properties:

```
Protected Overrides Sub AddInstanceBusinessRules()

  ValidationRules.AddInstanceRule( _
    AddressOf MyRuleMethod, "MyPropertyName")

End Sub
```

This is the exact same behavior as in version 2.0, but the methods have been renamed.

Adding Per-Type Validation Rules

The `AddBusinessRules()` method is now used to add per-type rules. This method is not called on each object creation, but is typically only called on the *first* object created. Remember that all rules associated with properties in this method are shared across all instances of the business object type.

Within `AddBusinessRules()`, the business developer calls the `AddRule()` method to associate rule methods with individual properties:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule( _
    AddressOf MyRuleMethod, "MyPropertyName")

End Sub
```

There are some extra restrictions on per-type rule methods. Remember that they are shared across all instances of the business object type, and so they *can not* be instance methods of your business object. They *can be* `Shared` methods in any class, methods in a `Module` or even instance methods of some other object.

Though you can't verify the rule methods at compile time, it is possible to verify them at runtime. The `ValidateHandler()` method performs a check to ensure that the rule method is not an instance method of the business object type:

```
Private Function ValidateHandler( _
  ByVal method As System.Reflection.MethodInfo) As Boolean

  If Not method.IsStatic AndAlso method.DeclaringType.Equals(mTarget.GetType) Then
    Throw New InvalidOperationException( _
      String.Format("{0}: {1}", _
      My.Resources.InvalidRuleMethodException, method.Name))
  End If
  Return True

End Function
```

Other overloads of `ValidateHandler()` exist, though they all delegate to this one. For instance, here's a simple overload:

```
Private Function ValidateHandler(ByVal handler As RuleHandler) As Boolean

  Return ValidateHandler(handler.Method)

End Function
```

Using this method, the `AddRule()` methods can then ensure that only valid rule methods are associated with the object's properties. There are several `AddRule()` overloads; the following is the most frequently used implementation:

```
Public Sub AddRule( _
  ByVal handler As RuleHandler, ByVal propertyName As String)

  ValidateHandler(handler)
  GetTypeRules(True).AddRule(handler, New RuleArgs(propertyName), 0)

End Sub
```

Once the rule method has been validated, the `ValidationRulesManager` object containing the per-type rules for this business object type is retrieved (and created if necessary). The rule method is then added to that `ValidationRulesManager` to establish the association between the rule method and the business object's property.

Checking Validation Rules

The trigger for running the validation rule methods is the same in version 2.1 as it was in version 2.0: the business object calls `ValidationRules.CheckRules()`, or `PropertyHasChanged()`. Within the framework, however, the process of invoking the rule methods is changed to accommodate per-type rules, as well as rule priority and short-circuiting. As I discuss the changes to the `CheckRules()` methods in this section, changes due to the other features will be discussed later.

The `CheckRules()` method has two `Public` overloads:

- `CheckRules(String)`
  checks rules for a single property
- `CheckRules()`
  checks rules for all properties

Either way, there is a `List` of validation rule method delegates that must be invoked on a per-property basis. At the most basic level, a `Private` overload of `CheckRules()` implements this behavior. Since that method's code is primarily concerned with rule priorities and short-circuiting, I will discuss the details later.

There are also two `Private` overloads of `CheckRules()`, which are used to organize the code in a reusable manner. These overloads are:

- `CheckRules(ValidationRulesManager, String)`
  used to implement dependant properties
- `CheckRules(List(Of IRuleMethod))`
  used to implement rule priorities and short-circuiting

The `Public` overload of `CheckRules()` that executes rules for a single property contains code that is primarily focused on implementing property dependant properties. It is important

to note, however, that it calls the `RulesToCheck()` method I discussed earlier in order to get the correct `ValidationRulesManager` for the object:

```
    ' get the rules dictionary
    Dim rules As ValidationRulesManager = RulesToCheck
```

If this object is not `Nothing`, then the list of rule methods for the specific property is retrieved:

```
      ' get the rules list for this property
      Dim rulesList As RulesList = rules.GetRulesForProperty(propertyName, False)
```

The rule methods contained in this `RulesList` object correspond to the specified property, and it is the rules from this list that are passed to the non-`Public` overload of `CheckRules()` to be executed.

The `Public` overload of `CheckRules()` that runs the rules for all properties is comparatively simple, since it can delegate the hard work:

```
  Public Sub CheckRules()

    Dim rules As ValidationRulesManager = RulesToCheck
    If rules IsNot Nothing Then
      For Each de As Generic.KeyValuePair(Of String, RulesList) In _
                                                rules.RulesDictionary
        CheckRules(de.Value.GetList(True))
      Next
    End If

  End Sub
```

Again, the `RulesToCheck()` method is used to retrieve the appropriate `ValidationRulesManager` object that contains the rules for this business object. If the result is not `Nothing`, the code loops through the items in the `Dictionary` contained by the `ValidationRulesManager` object. Each entry is a `RulesList` object that contains the rules for a property. A `Private` overload of `CheckRules()` is called on each `RulesList` object to invoke those rules, using rule priorities and short-circuiting.

At this point, you have seen the changes to `BusinessBase` and `ValidationRules` necessary to implement per-type validation rules. These two classes make use of the new `SharedValidationRules`, `ValidationRulesManager` and `RulesList` classes to provide support for both the new per-type and the older per-instance behaviors.

Implementing Dependant Properties

Support for dependant properties is a new feature of version 2.1. Many business objects have business rules that span multiple properties of the object, where a change to one property's value can cause another property's business rules to be invalid. Obviously, detecting that some other property's rules have become invalid requires running the rule methods of that other property. The dependant property support in version 2.1 addresses this need.

A business developer can add code to the `AddBusinessRules()` method to define dependant properties. A list of dependant properties is maintained for each property on the object, and when `CheckRules()` is called for a specific property, the rules for any dependant

properties related to that property are invoked after that specific property's rules have been executed.

ValidationRulesManager Class

You've already seen most of the code in the new `ValidationRulesManager` class. This class implements a method to encapsulate the process of associating a dependant property with a business object property:

```
Public Sub AddDependantProperty( _
  ByVal propertyName As String, ByVal dependantPropertyName As String)

  ' get the list of rules for the property
  Dim list As List(Of String) = _
    GetRulesForProperty(propertyName, True).GetDependancyList(True)

  ' we have the list, add the dependency
  list.Add(dependantPropertyName)

End Sub
```

This method first gets the `RulesList` object corresponding to the specified property. It then gets the list of dependant properties contained within that `RulesList` object, and adds the new property name to the list.

As with adding a new rule method, when adding a dependant property, the code creates an instance of a `RulesList` object if it doesn't already exist.

While the `ValidationRules` class could include the code to get the right `RulesList` object, retrieve the dependency list and add the item, this method simplifies and encapsulates that process, keeping the code in `ValidationRules` simpler and easier to maintain.

RulesList Class

Each property that has rules or dependencies will have a corresponding `RulesList` object. It is the job of this `RulesList` object to maintain the list of rule methods, and the list of dependant properties. You've already seen the declaration for the `mDependantProperties` field, which is a simple `List(Of String)`.

The `GetDepedencyList()` method is used by `ValidationRulesManager` to retrieve this list. As with most of the other objects in `Csla.Validation`, this list object is only created on-demand:

```
Public Function GetDependancyList(ByVal create As Boolean) As List(Of String)

  If mDependantProperties Is Nothing AndAlso create Then
    mDependantProperties = New List(Of String)
  End If
  Return mDependantProperties

End Function
```

The create parameter is used to control whether an instance of the `List(Of String)` should be created. The object is only created when a new dependency is being added, and not in the case that dependencies are being used by the `ValidationRules.CheckRules()` implementation.

Changes to ValidationRules

The bulk of the changes to support dependant properties are in `ValidationRules`. This class now contains a `Public` method used by the business developer to add dependant properties, as well as substantial changes to the `CheckRules()` method to invoke the rule methods for any dependant properties.

Adding Dependant Properties

The `AddDependantProperty()` method is called from within the business object's `AddBusinessRules()` method to add a dependant property. For example:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddDependantProperty( _
    "PropertyName", "DependantPropertyName")

End Sub
```

This indicates that when the validation rules for `PropertyName` are checked, the validation rules for `DependantPropertyName` should also be checked. A property can have any number of dependant properties. Additionally, two properties may be dependant on each other.

Here's the `AddDependantProperty()` method itself:

```
Public Sub AddDependantProperty( _
  ByVal propertyName As String, ByVal dependantPropertyName As String)

  GetTypeRules(True).AddDependantProperty(propertyName, dependantPropertyName)

End Sub
```

It first gets the per-type `ValidationRulesManager` object for the current business object type, creating it if it doesn't exist. Then the `AddDependantProperty()` method is called to add the property name to the appropriate `RulesList` object.

> **Note:** Dependant properties are stored at a per-type level only. There is no provision for setting up dependencies at a per-instance level.

Checking Validation Rules

When `CheckRules()` is called to check the rules for all properties of the object, there's no need to worry about dependant properties. They are all getting checked anyway. But when `CheckRules()` is called to check the rules of a specific property, any dependant property's rules must also be checked.

The per-property `CheckRules()` implementation looks like this (with the dependency-related lines highlighted):

```
Public Sub CheckRules(ByVal propertyName As String)

  ' get the rules dictionary
  Dim rules As ValidationRulesManager = RulesToCheck
  If rules IsNot Nothing Then
    ' get the rules list for this property
    Dim rulesList As RulesList = rules.GetRulesForProperty(propertyName, False)
    If rulesList IsNot Nothing Then
```

```
          ' get the actual list of rules (sorted by priority)
          Dim list As List(Of IRuleMethod) = rulesList.GetList(True)
          If list IsNot Nothing Then
            CheckRules(list)
          End If
          Dim dependancies As List(Of String) = rulesList.GetDependancyList(False)
          If dependancies IsNot Nothing Then
            For i As Integer = 0 To dependancies.Count - 1
              Dim dependantProperty As String = dependancies(i)
              CheckRules(rules, dependantProperty)
            Next
          End If
        End If
      End If

    End Sub
```

The `GetList()` method of `RulesList` is called to retrieve the list of rule methods for the specified property. If that list is not `Nothing`, the rules are invoked by calling a `Private` overload of `CheckRules()`. That `Private` overload contains code to implement rule priorities and short-circuiting, and I'll discuss it later. For now it is enough to know that the rules in the list are invoked.

Then the list of dependant property names is retrieved from the `RulesList` object. Notice that the parameter value `False` is passed to `GetDependancyList()`, so no objects are created. If they don't exist, `Nothing` will be returned.

A `For...Each` loop is then used to go through the list of dependant property names, calling a `Private` overload of `CheckRules()` to execute each property's rules. On the surface it seems that you could just do a recursive call to `CheckRules(String)`, but it is important to remember that properties can be dependant on each other. Such a recursive call could result in an infinite loop and, eventually, a stack overflow exception.

The `Private` overload of `CheckRules()` executes the rules for the dependant property. The previous code has already gone through the work of retrieving the `ValidationRulesManager` for the current business object, so that is passed in as a parameter to optimize the process:

```
    Private Sub CheckRules( _
      ByVal rules As ValidationRulesManager, ByVal propertyName As String)

      ' get the rules list for this property
      Dim rulesList As RulesList = rules.GetRulesForProperty(propertyName, False)
      If rulesList IsNot Nothing Then
        ' get the actual list of rules (sorted by priority)
        Dim list As List(Of IRuleMethod) = rulesList.GetList(True)
        If list IsNot Nothing Then
          CheckRules(list)
        End If
      End If

    End Sub
```

Using the provided `ValidationRulesManager` object, this method retrieves the `RulesList` object for the dependant property, and then gets the list of rule methods from that `RulesList` object. Assuming these objects are not `Nothing`, the `Private` overload of `CheckRules()` is called to execute this property's rules using rule priorities and short-circuiting.

Notice that this method is not recursive. Property dependency goes just one level deep, so a property that is dependant on another property that is in turn dependant on *another* property

will *not* trigger the rules to be invoked for all three properties. Only the original property and its immediate dependant property's validation rules will be checked.

At this point, you can see how a business developer uses `ValidationRules.AddDependantProperty()` to set up property dependencies. Those dependencies are stored in a `RulesList` object, and are used by `ValidationRules.CheckRules()` when the rules are checked for a specific property.

Implementing Rule Severity

The next enhancement to validation rules processing is the addition of rule severities. In version 2.0, all rules had the same severity, but many applications have the need for different levels of severity. For instance, some rules may require user notification, but shouldn't stop an object from being saved (`Information` or `Warning` severities), while other rules should stop the object from being saved (`Error` severity).

RuleSeverity Type

The severity levels supported by CSLA .NET version 2.1 are defined by the `RuleSeverity` type:

```
Public Enum RuleSeverity
  [Error]
  Warning
  Information
End Enum
```

This type is used in the implementation of rule severities.

Changes to RuleArgs

Each rule method can set the severity of the rule as part of its processing. This is important, because some rules might have different levels of "being invalid" based on different conditions. By allowing the rule method itself to indicate the severity of the result, you have a lot of flexibility in how severities are used.

A rule method is always passed a parameter derived from the `RuleArgs` type. In version 2.1, `RuleArgs` now includes a `Severity` property, and corresponding instance field:

```
Public Class RuleArgs

  ' ...

  Private mSeverity As RuleSeverity = RuleSeverity.Error

  ' ...

  Public Property Severity() As RuleSeverity
    Get
      Return mSeverity
    End Get
    Set(ByVal value As RuleSeverity)
      mSeverity = value
    End Set
  End Property

  ' ...

End Class
```

The default severity is `Error`, to match the behavior of version 2.0. Within a rule method, the business developer can set the `Severity` property if another severity is required. For instance:

```
Private Function MyRuleMethod( _
  ByVal target As Object, ByVal e As RuleArgs) As Boolean

  If <condition is met> Then
    Return True

  Else
    e.Description = "Human readable description"
    e.Severity = RuleSeverity.Warning
    Return False
  End If
End Function
```

This `Severity` property value is used by the `ValidationRules`, `BrokenRule` and `BrokenRulesCollection` classes to record, store and retrieve broken rules by severity.

Changes to BrokenRule

If a rule is broken, `ValidationRules.CheckRules()` adds it to the `BrokenRulesCollection`, with the rule's details contained in a `BrokenRule` object. This is true regardless of the rule's severity, but the severity is maintained as part of the information about the broken rule.

The `BrokenRule` class has a field to store the severity value, and a property so other code can examine the value:

```
<Serializable()> _
Public Class BrokenRule
  Private mRuleName As String
  Private mDescription As String
  Private mProperty As String
  Private mSeverity As RuleSeverity

  ' ...

  Public ReadOnly Property Severity() As RuleSeverity
    Get
      Return mSeverity
    End Get
  End Property

End Class
```

The constructor also includes code to deal with the value:

```
  Friend Sub New(ByVal rule As IRuleMethod)
    mRuleName = rule.RuleName
    mDescription = rule.RuleArgs.Description
    mProperty = rule.RuleArgs.PropertyName
    mSeverity = rule.RuleArgs.Severity
  End Sub
```

With `BrokenRule` storing the severity value, `BrokenRulesCollection` can provide some useful behaviors to filter out various types of broken rule.

Changes to BrokenRulesCollection

The `BrokenRulesCollection` contains a list of `BrokenRule` objects, each one corresponding to a rule method that has returned `False`, along with a human readable description and a severity value. This class has been enhanced to provide some filtering capabilities, so it is possible to retrieve all broken rules, or only those of a specific severity.

Perhaps the biggest change, however, is in how `BrokenRulesCollection` interacts with the `IsValid` property from `ValidationRules`. The new `Information` and `Warning` severities don't cause an object to be invalid. That is reserved for `Error` severity only. So where `ValidationRules.IsValid` used to just check to see if any rules were broken, it now must check to see if any `Error` severity rules are broken.

Severity Counters

To efficiently support this concept, `BrokenRulesCollection` maintains a counter of the number of broken rules in each severity:

```
<Serializable()> _
Public Class BrokenRulesCollection
  Inherits Core.ReadOnlyBindingList(Of BrokenRule)

  Private mErrorCount As Integer
  Private mWarningCount As Integer
  Private mInfoCount As Integer

  ' ...

  Public ReadOnly Property ErrorCount() As Integer
    Get
      Return mErrorCount
    End Get
  End Property

  Public ReadOnly Property WarningCount() As Integer
    Get
      Return mWarningCount
    End Get
  End Property

  Public ReadOnly Property InformationCount() As Integer
    Get
      Return mInfoCount
    End Get
  End Property

  ' ...

End Class
```

As rules are added and removed from the collection, the values are incremented and decremented accordingly. In the `Add()` method the value is incremented:

```
    Friend Overloads Sub Add(ByVal rule As IRuleMethod)

      Remove(rule)

      IsReadOnly = False
      Dim item As New BrokenRule(rule)
      IncrementCount(item)
      Add(item)
      IsReadOnly = True

    End Sub
```

The `IncrementCount()` helper method takes care of the details:

```
    Private Sub IncrementCount(ByVal item As BrokenRule)

      Select Case item.Severity
        Case RuleSeverity.Error
          mErrorCount += 1
        Case RuleSeverity.Warning
          mWarningCount += 1
        Case Else
          mInfoCount += 1
      End Select

    End Sub
```

Similarly, the `Remove()` method calls a `DecrementCount()` helper method:

```
    Private Sub DecrementCount(ByVal item As BrokenRule)

      Select Case item.Severity
        Case RuleSeverity.Error
          mErrorCount -= 1
        Case RuleSeverity.Warning
          mWarningCount -= 1
        Case Else
          mInfoCount -= 1
      End Select

    End Sub
```

The end result is that the three counts are kept up to date as broken rules are added and removed from the collection. This allows the count values to be returned quickly. The values are totaled, without the need to scan through the collection each time the count is required.

Retrieving Rules

The collection implements a `GetFirstBrokenRule()` method, which is intended to return the first broken rule for a specified property. In version 2.1, this method is altered to only look at `Error` severity rules to preserve backward compatibility with version 2.0:

```
    Public Function GetFirstBrokenRule(ByVal [property] As String) As BrokenRule

      Return GetFirstMessage([property], RuleSeverity.Error)

    End Function
```

Notice that this method makes use of a new method: `GetFirstMessage()`. This new method is `Public`, and allows any calling code to retrieve the human-readable description for the first broken rule of any severity:

```
  Public Function GetFirstMessage(ByVal [property] As String) As BrokenRule

    For Each item As BrokenRule In Me
      If item.Property = [property] Then
        Return item
      End If
    Next
    Return Nothing

  End Function

  Public Function GetFirstMessage( _
    ByVal [property] As String, ByVal severity As RuleSeverity) As BrokenRule

    For Each item As BrokenRule In Me
      If item.Property = [property] AndAlso item.Severity = severity Then
        Return item
      End If
    Next
    Return Nothing

  End Function
```

There are two overloads of this method; one allows the caller to retrieve the first message regardless of severity, the other filters the result to match a specific severity.

There is also a new overload of the `ToString()` method, which filters the results based on severity:

```
  Public Overloads Function ToString(ByVal severity As RuleSeverity) As String

    Dim result As New System.Text.StringBuilder()
    Dim item As BrokenRule
    Dim first As Boolean = True

    For Each item In Me
      If item.Severity = severity Then
        If first Then
          first = False
        Else
          result.Append(Environment.NewLine)
        End If
        result.Append(item.Description)
      End If
    Next
    Return result.ToString

  End Function
```

The pre-existing `ToString()` method is unchanged, returning all rule descriptions regardless of severity.

Changes to ValidationRules

The `ValidationRules` class implements an `IsValid` property. As I discussed earlier, the way `BrokenRulesCollection` expresses validity has changed, and the `IsValid` implementation was changed accordingly:

```
    Friend ReadOnly Property IsValid() As Boolean
      Get
        Return BrokenRulesList.ErrorCount = 0
      End Get
    End Property
```

Rather than relying on the `Count` property as was done in version 2.0, `IsValid` now checks the `ErrorCount` property, thus ignoring any `Information` or `Warning` severity rules.

Changes to BusinessBase

Finally, `BusinessBase` implements the `System.ComponentModel.IDataErrorInfo` interface, which defines an Error property. This property has been modified to call the `BrokenRulesCollection.ToString()` overload so only `Error` severity broken rules are returned:

```
    Private ReadOnly Property [Error]() As String _
      Implements System.ComponentModel.IDataErrorInfo.Error
      Get
        If Not IsValid Then
          Return ValidationRules.GetBrokenRules.ToString(Validation.RuleSeverity.Error)

        Else
          Return ""
        End If
      End Get
    End Property
```

You should now understand how a rule method can set the severity of a broken rule, and how that severity is recorded in the corresponding `BrokenRule` object. You've seen how the `BrokenRulesCollection` exposes filtered views of the broken rules, and how the `IsValid` functionality has been enhanced so only `Error` severity rules make an object invalid.

Implementing Rule Priority

In version 2.1, rules may be assigned a priority, which is 0 or greater. Rule methods are invoked in priority order, starting with 0 and climbing to successively higher values. In other words, the higher the number, the later the rule will be executed. Within a priority, the order in which rules are invoked is non-deterministic. What this means, is that you can't control the order in which priority 0 rules are invoked, but if you set a rule to priority 1 you know it will be invoked after *all* priority 0 rules are complete.

The rule priority feature is designed primarily to support the concept of short-circuiting, which is discussed later. By invoking rules in priority order, the framework provides business developers with the ability to ensure that some rules are invoked before others.

Changes to ValidationRules

The priority of a rule is set when the rule is associated with a property through the `AddInstanceRule()` or `AddRule()` methods. To support this concept, these methods have overloads that accept the priority value. For example:

```
Public Sub AddInstanceRule( _
  ByVal handler As RuleHandler, ByVal propertyName As String, _
  ByVal priority As Integer)

  GetInstanceRules(True).AddRule(handler, New RuleArgs(propertyName), priority)

End Sub
```

And

```
Public Sub AddRule( _
  ByVal handler As RuleHandler, ByVal propertyName As String, _
  ByVal priority As Integer)

  ValidateHandler(handler)
  GetTypeRules(True).AddRule(handler, New RuleArgs(propertyName), priority)

End Sub
```

There are numerous overloads of each method, and I'm not going to list them all here. The important thing to recognize is that each of these overloads calls the `AddRule()` method in the appropriate `ValidationRulesManager` object. That priority value is stored along with each rule method in the `ValidationRulesManager` object.

Later, in the `CheckRules()` method, this priority value is used to ensure that the rule methods are returned in a sorted order. This is triggered by the highlighted line of code shown here:

```
Public Sub CheckRules(ByVal propertyName As String)

  ' get the rules dictionary
  Dim rules As ValidationRulesManager = RulesToCheck
  If rules IsNot Nothing Then
    ' get the rules list for this property
    Dim rulesList As RulesList = rules.GetRulesForProperty(propertyName, False)
    If rulesList IsNot Nothing Then
      ' get the actual list of rules (sorted by priority)
      Dim list As List(Of IRuleMethod) = rulesList.GetList(True)
      If list IsNot Nothing Then
        CheckRules(list)
      End If
      Dim dependancies As List(Of String) = rulesList.GetDependancyList(False)
      If dependancies IsNot Nothing Then
        For i As Integer = 0 To dependancies.Count - 1
          Dim dependantProperty As String = dependancies(i)
          CheckRules(rules, dependantProperty)
        Next
      End If
    End If
  End If

End Sub
```

As you can see, it is the `GetList()` method from the `RulesList` class that actually performs the sort operation.

Changes to ValidationRulesManager

The `ValidationRulesManager` class implements `AddRule()` methods that are called from `ValidationRules`. There are two overloads for this method, and both accept a `priority` parameter:

```
Public Sub AddRule(ByVal handler As RuleHandler, ByVal args As RuleArgs, _
  ByVal priority As Integer)

  ' get the list of rules for the property
  Dim list As List(Of IRuleMethod) = _
    GetRulesForProperty(args.PropertyName, True).GetList(False)

  ' we have the list, add our new rule
  list.Add(New RuleMethod(handler, args, priority))

End Sub

Public Sub AddRule(Of T, R As RuleArgs)( _
  ByVal handler As RuleHandler(Of T, R), ByVal args As R, ByVal priority As Integer)

  ' get the list of rules for the property
  Dim list As List(Of IRuleMethod) = _
    GetRulesForProperty(args.PropertyName, True).GetList(False)

  ' we have the list, add our new rule
  list.Add(New RuleMethod(Of T, R)(handler, args, priority))

End Sub
```

The `priority` parameter value is used during the construction of the `RuleMethod` object that contains details about the rule. The `RulesList` class contains the code to sort the rules based on this value.

Changes to RuleMethod

In the `RuleMethod` class, the priority value is maintained in a field, and exposed through a property.

```
Private mPriority As Integer

Public ReadOnly Property Priority() As Integer Implements IRuleMethod.Priority
  Get
    Return mPriority
  End Get
End Property
```

`RuleMethod` also implements the `IComparable` interface, and the priority value is used in the implementation of the `CompareTo()` methods:

```
Private Function CompareTo(ByVal obj As Object) As Integer _
  Implements System.IComparable.CompareTo

  Return Priority.CompareTo(CType(obj, IRuleMethod).Priority)

End Function

Private Function CompareTo1(ByVal other As IRuleMethod) As Integer _
  Implements System.IComparable(Of IRuleMethod).CompareTo

  Return Priority.CompareTo(other.Priority)

End Function
```

By implementing `CompareTo()` based on the priority value, the built-in capability of .NET to sort a list can be used by `RulesList` to sort the objects by priority.

The same implementation exists in `RuleMethod(Of T, R)`.

Changes to RulesList

The `RulesList` object contains the list of rules to be evaluated by the `ValidationRules.CheckRules()` method. Earlier you saw how the `CheckRules()` method calls a `GetList()` method to get the sorted list of rules to invoke. The `GetList()` method implements the sorting:

```
    Public Function GetList(ByVal applySort As Boolean) As List(Of IRuleMethod)

      If applySort AndAlso Not mSorted Then
        SyncLock mList
          If applySort AndAlso Not mSorted Then
            mList.Sort()
            mSorted = True
          End If
        End SyncLock
      End If
      Return mList

    End Function
```

The `GetList()` method is called not only by `CheckRules()`, but also when new rules are being added to the list. Since sorting is an expensive operation, a parameter is used to control whether sorting should occur when `GetList()` is called. As rules are added, no sorting is requested, but when `CheckRules()` calls this method, it indicates that it wants a sorted result.

Notice that `mList`, which is a `List(Of IRuleMethod)`, is directly sorted by calling the `Sort()` method. This is possible because the `RuleMethod` classes implement `IComparable`, and use the priority value to implement the `CompareTo()` methods.

As an optimization, the code keeps track of whether the list has been sorted by using a Boolean field. Any time items are added to the list, this field is set to `False`, and after the sort is complete it is set to `True`. The result is that, with normal usage, the list of rules is only sorted one time.The pre-sorted result is returned on all subsequent calls.

At this point, you should understand that a business developer can choose to specify a priority for each rule when calling `AddRule()` or `AddInstanceRule()`. This priority value is maintained by the `RuleMethod` objects, and is used by `RulesList` to perform the sort. The `CheckRules()` method then invokes the methods in order, starting with priority 0 and climbing from there.

Implementing Short-Circuiting

Short-circuiting is a feature that stops the processing of rules part-way through. The result is that not all rules for a property are invoked. There are two ways to short-circuit rule processing for a property: a rule method can explicitly stop the processing, or CSLA .NET can be told to stop processing rules if any previous (higher priority) rule has already returned `False`.

This feature is useful, because it allows the business developer to check all the inexpensive, easily checked, rules first, and only invoke expensive rules (such as those that

might hit the database) if all previous rules were satisfied (returned `True`). The rule priority feature discussed earlier is a key part of this capability, because it allows the business developer to control the order in which rules are invoked.

Changes to ValidationRules

The `ValidationRules` object is responsible for invoking the rule methods. If a rule method returns `False`, then the rule is considered broken and is added to the business object's `BrokenRulesCollection`. This is handled in the parameterless `Private` overload of `CheckRules()`. While this method is primarily concerned with implementing rule priorities and short-circuiting, it does have the ultimate responsibility for recording whether a rule is broken or not.

It also turns out that there's an intersection between the short-circuiting behavior and rule severity. It is important to realize that only `Error` severity rules can trigger short-circuiting. `Information` and `Warning` severities don't stop the processing of subsequent rule methods.

The lines of code highlighted here are used to implement priority-based short-circuiting:

```
    Private Sub CheckRules(ByVal list As List(Of IRuleMethod))

      Dim previousRuleBroken As Boolean
      Dim shortCircuited As Boolean

      For index As Integer = 0 To list.Count - 1
        Dim rule As IRuleMethod = list(index)
        ' see if short-circuiting should kick in
        If Not shortCircuited AndAlso _
            (previousRuleBroken AndAlso _
             rule.Priority > mProcessThroughPriority) Then
          shortCircuited = True
        End If

        If shortCircuited Then
          ' we're short-circuited, so just remove
          ' all remaining broken rule entries
          BrokenRulesList.Remove(rule)

        Else
          ' we're not short-circuited, so check rule
          If rule.Invoke(mTarget) Then
            ' the rule is not broken
            BrokenRulesList.Remove(rule)

          Else
            ' the rule is broken
            BrokenRulesList.Add(rule)
            Dim args As RuleArgs = rule.RuleArgs
            If args.Severity = RuleSeverity.Error Then
              previousRuleBroken = True
            End If
          End If
          If args.StopProcessing Then
            shortCircuited = True
          End If
        End If
      Next

    End Sub
```

Due to this, the `CheckRules()` method must examine the severity contained in the `RuleArgs` parameter as it comes back from the rule method:

```
            If rule.RuleArgs.Severity = RuleSeverity.Error Then
              previousRuleBroken = True
            End If
```

The `previousRuleBroken` field is used to keep track of whether any rule evaluated thus far has returned `False`. This value is then used to trigger the short-circuiting process itself, by setting the `shortCircuited` field to `True`:

```
        ' see if short-circuiting should kick in
        If Not shortCircuited AndAlso _
            (previousRuleBroken AndAlso _
             rule.Priority > mProcessThroughPriority) Then
          shortCircuited = True
        End If
```

If `shortCircuited` is `True`, then normal processing of rule methods is suspended, and instead, all subsequent rule entries are simply removed from the list of broken rules:

```
        If shortCircuited Then
          ' we're short-circuited, so just remove
          ' all remaining broken rule entries
          BrokenRulesList.Remove(rule)
```

The entries are removed because there is no way to know if the rules are broken or not. Remember, once `shortCircuited` is set to `True` no further rule methods are invoked. Rather than assume all the unchecked rules are broken (which could be very misleading for the end user), the code assumes that all unchecked rules are not broken, so their descriptions do not appear to the end user as issues to be resolved.

It is also possible for a rule method to directly cause short-circuiting to occur. To do this, a rule method sets `e.StopProcessing` to `True`. This value is used by the `CheckRules()` method to set the `shortCircuited` field to `True`, causing the same short-circuiting behavior as with the priority-based scheme:

```
          If rule.RuleArgs.StopProcessing Then
              shortCircuited = True
            End If
```

Notice that this check occurs regardless of whether the rule method returns `True` or `False`. Even an unbroken rule can stop the processing of subsequent rules by setting `StopProcessing` to `True`.

Changes to RuleArgs

The `RuleArgs` class now includes a `StopProcessing` property for use by a rule method that wants to immediately trigger short-circuiting:

```
    Private mStopProcessing As Boolean

    Public Property StopProcessing() As Boolean
      Get
        Return mStopProcessing
      End Get
      Set(ByVal value As Boolean)
        mStopProcessing = value
      End Set
    End Property
```

At this point, you should understand how the `CheckRules()` method has been enhanced to stop the processing of rules once short-circuiting has been triggered. Short-circuiting can be triggered through a priority-based threshold scheme, or explicitly by a rule method setting the `StopProcessing` property of a `RuleArgs` object to `True`.

Implementing Strongly-typed Rule Methods

In CSLA .NET 2.0, rule methods conform to the `RuleHandler` delegate signature:

```
Public Delegate Function RuleHandler( _
  ByVal target As Object, ByVal e As RuleArgs) As Boolean
```

This signature accepts parameters of type `Object` and `RuleArgs`, allowing the use of any type of object as a target, and any subclass of `RuleArgs` as a parameter. The drawback to this approach is that it is often necessary to cast the `target` or `e` parameters before using them, which requires extra code and can lead to runtime type mismatch exceptions.

CSLA .NET 2.1 enhances the way rule methods are implemented to allow for strongly typed parameters to the methods. This is done by defining a second delegate signature using generics, and by adding a new generic `RuleMethod` class to store these strongly typed method references. This also requires altering the `RulesList` class to maintain a list of `IRuleMethod`, rather than `RuleMethod`.

Generic RuleHandler Delegate

The `RuleHandler` delegate in CSLA .NET 2.0 uses basic polymorphic types as parameters. That definition is retained in version 2.1, but a new delegate definition is required to support strongly typed parameters:

```
Public Delegate Function RuleHandler(Of T, R As RuleArgs)( _
  ByVal target As T, ByVal e As R) As Boolean
```

Using this new delegate, it is possible to specify the types for both the `target` and `e` parameters during development, so the compiler can check those types during compilation.

IRuleMethod Interface

The `RuleMethod` object is used to store a reference, along with metadata, for a rule method defined by the `RuleHandler` delegate. In version 2.1, a new generic `RuleMethod` class must be added to maintain a reference to the new generic `RuleHandler` delegate. In order to provide polymorphic use of both `RuleMethod` types, an `IRuleMethod` interface is required:

```
Friend Interface IRuleMethod
  ReadOnly Property Priority() As Integer
  ReadOnly Property RuleName() As String
  ReadOnly Property RuleArgs() As RuleArgs
  Function Invoke(ByVal target As Object) As Boolean
End Interface
```

This interface is required because generic types are not polymorphic. The only ways to make a generic type be polymorphic are for it to inherit from a non-generic base class or implement a non-generic interface. This non-generic interface can be implemented by both the original `RuleMethod` and new generic `RuleMethod` classes so they can be used

interchangeably (polymorphically) through this interface. The interface will be used by `RulesList` so it can store either type of rule reference.

Changes to RuleMethod

The existing version 2.0 `RuleMethod` class must be enhanced to implement the new `IRuleMethod` interface. Since the interface defines the same methods that were already implemented by `RuleMethod`, this is a simple process. The `Implements` keyword is used to indicate that the class implements the interface:

```
Friend Class RuleMethod

  Implements IRuleMethod
```

And the `Implements` clause is used on the existing methods to link them to the interface. For example:

```
  Public ReadOnly Property Priority() As Integer Implements IRuleMethod.Priority
    Get
      Return mPriority
    End Get
  End Property
```

The other methods are altered in a similar manner.

Generic RuleMethod Type

CSLA .NET 2.1 includes a new `RuleMethod` class. This class is virtually identical to the existing `RuleMethod` class, except that this one accepts generic type parameters and uses them to define the `mHandler` field that references the rule method delegate. The class definition and field declarations are:

```
Friend Class RuleMethod(Of T, R As RuleArgs)

  Implements IRuleMethod
  Implements IComparable
  Implements IComparable(Of IRuleMethod)

  Private mHandler As RuleHandler(Of T, R)
  Private mRuleName As String = ""
  Private mArgs As R
  Private mPriority As Integer
```

Notice the generic type parameters and how they are used to declare the `mHandler` field. The generic type parameters are also used in the implementation of various methods within the class. For instance, the `RuleArgs` property returns a value of type `R`:

```
  Public ReadOnly Property RuleArgs() As R
    Get
      Return mArgs
    End Get
  End Property
```

Of course the `IRuleMethod` interface requires a return type of `RuleArgs`, so the interface implementation is separate:

```
Private ReadOnly Property IRuleMethod_RuleArgs() As RuleArgs _
  Implements IRuleMethod.RuleArgs
  Get
    Return RuleArgs
  End Get
End Property
```

A similar technique is used to provide a generic overload of the `Invoke()` method. The result is a `RuleMethod` object that can maintain a reference to a rule method with strongly typed parameters.

Changes to ValidationRules

The `ValidationRules` class now includes generic overloads for `AddRules()` (and `AddInstanceRules()`) so it is possible to specify the types of the `target` and `RuleArgs` parameters. A number of overloads have been added, and I won't list them all here. An example of an overload is:

```
Public Sub AddInstanceRule(Of T)( _
  ByVal handler As RuleHandler(Of T, RuleArgs), ByVal propertyName As String)

  GetInstanceRules(True).AddRule(Of T, RuleArgs) _
    (handler, New RuleArgs(propertyName), 0)

End Sub
```

This overload only specifies the type of the `target` parameter. Another example illustrates how both the `target` and `RuleArgs` parameter are typed:

```
Public Sub AddRule(Of T, R As RuleArgs)( _
  ByVal handler As RuleHandler(Of T, R), ByVal args As R)

  ValidateHandler(handler)
  GetTypeRules(True).AddRule(handler, args, 0)

End Sub
```

Using these overloads, a business developer can get strong typing on one or both of their rule method parameters.

You should now understand that a business developer can associate either loosely typed or strongly typed rule methods with the properties of a business object. Strongly typed rule methods avoid the need for casting the `target` and `e` parameter values within the rule method, and provide for compile-time type checking.

Implementing Rule Retrieval

CSLA .NET 2.1 implements a new feature that allows a business developer to retrieve the list of rule methods that have been added to an object. This list includes all the per-type and per-instance rule methods associated with all the properties of the object.

Keep in mind that it is possible for a rule method to be associated with a property more than one time. In such a case, it is likely that the arguments passed through the `RuleArgs` parameter are different, and must be used to distinguish between the two different rule method instances. This means that the list of rule methods returned for an object must include not only the name of the rule method, and the associated property name, but also must include the parameter values passed to the method.

I chose to represent the rule methods using the URI format. For instance, a rule will appear as:

```
rule://methodName/propertyName?arg1=value&arg2=value
```

I had two reasons for choosing the URI format. First, this format can clearly express all the information about a rule method, including the method name, the property associated with the rule and all the arguments passed to the rule method. Second, by conforming to the URI format, the `System.Uri` class in the .NET framework can be used to easily parse these values. This makes it relatively easy for a business or UI developer to retrieve any given part of the URI without having to manually parse the string.

Changes to ValidationRules

The `ValidationRules` class now includes a `GetRuleDescriptions()` method, which returns an array of `String` values, with each string representing a rule method. This code is relatively simple, since each `RuleMethod` object is responsible for generating its own text representation:

```
  Public Function GetRuleDescriptions() As String()

    Dim result As New List(Of String)
    Dim rules As ValidationRulesManager = RulesToCheck
    For Each de As Generic.KeyValuePair(Of String, RulesList) In rules.RulesDictionary
      Dim list As List(Of IRuleMethod) = de.Value.GetList(False)
      For i As Integer = 0 To list.Count - 1
        Dim rule As IRuleMethod = list(i)
        result.Add(CObj(rule).ToString)
      Next
    Next
    Return result.ToArray

  End Function
```

This code loops through all the properties in the object that have rules, and then loops through the rules associated with each property. Notice the use of the `RulesToCheck()` helper method, which returns the consolidated list of per-type and per-instance rules for this particular business object. This method was discussed earlier in the book.

The real work occurs in the `RuleMethod` and `RuleArgs` classes, which are responsible for generating the URI text representation.

Changes to RuleMethod

Both the generic and non-generic `RuleMethod` classes maintain a `mRuleName` field, which stores the text representation of the rule. The value of this field is returned from the `ToString()` method in the `RuleMethod` class.

The `mRuleName` field value is set in the constructor:

```
    Public Sub New(ByVal handler As RuleHandler, _
      ByVal args As RuleArgs)

      mHandler = handler
      mArgs = args
      mRuleName = _
        String.Format("rule://{0}/{1}", mHandler.Method.Name, mArgs.ToString)

    End Sub
```

Notice how the name of the rule method itself is combined with the `ToString()` value from the `RuleArgs` object to create the URI text result. This is important, because it places a constraint on the implementation of any `RuleArgs.ToString()` method implementation. In the standard `RuleArgs` class, only the property name is returned. However, any subclass of `RuleArgs` *must* return a text fragment in the following format:

```
propertyName?arg1=value&arg2=value
```

If this is not done, then the resulting text value will not be a properly formatted URI value.

It should now be clear how a business object can call the `GetRuleDescriptions()` method to retrieve an array of URI-formatted text values, each entry representing a rule that has been associated with a property of the business object.

Implementing BrokenRulesCollection.ToArray

The `BrokenRulesCollection` includes a `ToString()` override, which returns the human-readable descriptions of the broken rules for the object as a single text value. Sometimes it is more valuable to have the broken rule descriptions returned as an individual text value for each rule. The `ToArray()` methods provide this capability:

```
    Public Function ToArray() As String()

      Dim result As New List(Of String)
      For Each item As BrokenRule In Me
        result.Add(item.Description)
      Next
      Return result.ToArray

    End Function

    Public Function ToArray(ByVal severity As RuleSeverity) As String()

      Dim result As New List(Of String)
      For Each item As BrokenRule In Me
        If item.Severity = severity Then
          result.Add(item.Description)
        End If
      Next
      Return result.ToArray

    End Function
```

There are two overloads, one that returns all rule descriptions, and one where the list is filtered by severity. In either case, the human-readable text descriptions are placed into an array of `String` values, with one entry per broken rule.

Using the Enhancements

CSLA .NET version 2.1 includes substantial enhancements to the way validation rules are associated with a business object, how they are processed, and how the results can be retrieved. In many cases only minor code changes are required to move from version 2.0 to 2.1, though there are exceptions. Some of the features in version 2.1 are entirely new, and you'll need to change your code to exploit them.

Using Per-Type Validation Rules

In CSLA .NET 2.1, business rules may be associated with a business object at the type or instance level. Per-type rules are associated with *all* business objects of a given type, while per-instance rules are associated with one specific instance of a business object.

Per-type rules are far more efficient in their use of memory, and offer performance benefits because the association of rules to properties only occurs once per AppDomain rather than as each object is created. Typically, this means the association occurs once during the lifetime of the application.

Per-instance rules provide more flexibility, because these rules are associated with the object's properties as each object is created. You can write code to change the way the rules are associated with the object based on the specific object being created. This results in more memory consumption and slower performance, because the list of rules is maintained and created as each business object is instantiated.

When creating a business object, you can now override `AddBusinessRules()` and `AddInstanceBusinessRules()`.

Associating Rule Methods with Properties

The `AddBusinessRules()` method is called only once per AppDomain for each type of business object. In this method, you can call `ValidationRules.AddRule()` to associate rule methods with properties of your business object. These rule methods will then be associated with the properties of *all* business objects of that type.

A typical `AddBusinessRule()` method might look like this:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule( _
    AddressOf Csla.Validation.CommonRules.StringRequired, "Name")

End Sub
```

This associates the `StringRequired` rule method from `CommonRules` with the object's `Name` property. While this code looks the same as it did in version 2.0, the results are quite different. This `AddBusinessRules()` method will typically only be called once during the lifetime of the application, and the rule association that's set up here is applied to all instances of the business object.

If you want to associate a rule method with a property only for a specific object instance, you should override `AddInstanceBusinessRules()`. Such an override might look like this:

```
Protected Overrides Sub AddInstanceBusinessRules()

  ValidationRules.AddInstanceRule( _
    AddressOf Csla.Validation.CommonRules.StringRequired, "City")

End Sub
```

This associates the `StringRequired` rule method with the object's `City` property. Notice the use of the `AddInstanceRule()` method, rather than `AddRule()`. This indicates that the association should be added only for this particular object instance, rather than all objects of this type.

> Make sure to only call `AddRule()` within `AddBusinessRules()`, and `AddInstanceRule()` within `AddInstanceBusinessRules()`.

Implementing Per-Type Rule Methods

There are some restrictions on per-type rule methods. Remember that these rule methods are shared across all instances of a given business object type. This means that some rule method implementations from version 2.0 will not work as per-type rule methods in 2.1, while others will work fine.

Per-type rule methods may be one of:

1. A `Shared` method in any class (including the business class itself)
2. Any method in a `Module` (as these are effectively `Shared` methods)
3. An instance method from another object (not the business object itself)

In fact, the only methods that *can't* be used as a per-type rule method are instance methods of your business object itself.

> If you are converting from version 2.0 to 2.1 and you have rule methods implemented as instance methods of your business object you can either change them to `Shared` methods or you can use `AddInstanceBusinessRules()`to associate them with your properties.

For instance, the methods in `Csla.Validation.CommonRules` are all `Shared` methods, so they can be used as per-type rule methods. Similarly, any `Shared` methods in your business class can be used, because they are automatically available across all instances of the type.

It is important to remember that these rule methods will be invoked for all instances of your business object. Due to this, these methods *must* use the `target` parameter to retrieve the data values to be validated.

If you are implementing widely-used rules that are common to many types of object you should follow the pattern used in `Csla.ValidationRules.CommonRules`. You can find an explanation of that code in *Expert VB 2005 Business Objects* (ISBN 1590596315).

If you are implementing rules specific to your business object type, you can implement the method like this:

```
<Serializable()> _
Public Class Attendee
  Inherits BusinessBase(Of Attendee)

  Private mAge As Integer
  Private mDrinkingBadge As Boolean

  ' other code goes here

  Protected Overrides Sub AddBusinessRules()

    ValidationRules.AddRule( _
      AddressOf AllowedToDrink(Of Attendee), "DrinkingBadge")

  End Sub

  Private Shared Function AllowedToDrink(Of T)( _
    ByVal target As T, ByVal e As Validation.RuleArgs) As Boolean

    If target.mAge < 21 AndAlso target.mDrinkingBadge Then
      e.Description = "Can not drink if under 21"
      Return False

    Else
      Return True
    End If

  End Function

  ' other code goes here

End Class
```

This code works because .NET allows `Shared` methods in a class to access the `Private` fields of an instance of that type. Since `target` is an instance of `Attendee`, and the `AllowedToDrink()` method is implemented in the `Attendee` class, its code is allowed to access the `Private` fields of the `Attendee` object.

Using Dependant Properties

Many business objects have properties that are interdependent, where changing one property should trigger re-validation of other properties on the object, along with validating the property that was changed. If your object has properties whose rules should be checked because a *different* property was changed, then you should use dependant properties.

You set up property dependencies in the `AddBusinessRules()` method as you implement your business object. For instance:

```
  Protected Overrides Sub AddBusinessRules()

    ' call ValidationRules.AddRules() here

    ValidationRules.AddDependantProperty("StartDate", "EndDate")

  End Sub
```

In this example any time the rules for `StartDate` are checked, the rules for `EndDate` will also be checked. The `EndDate` property is dependant on `StartDate`. The processing of rules for both properties will be triggered by a call to `PropertyHasChanged()` for `StartDate`, or an explicit call to `ValidationRules.CheckRules("StartDate")`.

The relationship is not automatically bi-directional. In other words, checking the rules for `EndDate` *will not* cause the rules for `StartDate` to be checked. If you want that to happen, you can use an overload of `AddDependantProperty()` to indicate that `StartDate` is also dependant on `EndDate`:

```
Protected Overrides Sub AddBusinessRules()

  ' call ValidationRules.AddRules() here

  ValidationRules.AddDependantProperty("StartDate", "EndDate", True)

End Sub
```

With this change, both properties are now dependant on the other property, so checking the rules for either property will cause both sets of rules to be checked.

You can also make multiple properties dependant on a single property. For example:

```
Protected Overrides Sub AddBusinessRules()

  ' call ValidationRules.AddRules() here

  ValidationRules.AddDependantProperty("StartDate", "EndDate", True)
  ValidationRules.AddDependantProperty("EndDate", "CloseDate")

End Sub
```

In this case, both `StartDate` and `CloseDate` are dependant on `EndDate`. So when the rules are checked for `EndDate`, they will also be checked for both `StartDate` and `CloseDate`.

It is important to realize that dependant properties are established *only* in `AddBusinessRules()`, and so are per-type. All instances of your business object will have the same property dependencies. Even if you are using per-instance rules, you must establish the dependencies in `AddBusinessRules()`. However, you should also know that dependant properties affect both per-type and per-instance rules.

Using Rule Severity

CSLA .NET 2.1 introduces the concept of rule severity, where a broken rule method can indicate the severity of its result. Table 6 lists the possible severities.

Severity	Description
`Error`	The broken rule means the object is not valid, and the result should appear in the UI as a validation error.
`Warning`	The broken rule means the object is valid, but this property has an issue that should be addressed.
`Information`	The broken rule means the object is valid, but there is something the user should know about this property.

Table 6. Rule severity definitions

The severity of a rule is set within the rule method itself. This allows your code in the rule method to determine the appropriate severity for the rule failure, allowing for a great deal of flexibility. For example, a rule method could look like this:

```
Private Shared Function CreditLimitCheck(Of T As SalesOrder)( _
  ByVal target As T, ByVal e As Validation.RuleArgs) As Boolean

  Dim cust As Customer = target.GetCustomer
  If target.TotalAmount > cust.CreditLimit Then
    e.Description = "Credit limit exceeded"
    e.Severity = Validation.RuleSeverity.Error
    Return False

  ElseIf target.TotalAmount > cust.CreditLimit * .9 Then
    e.Description = "Nearing credit limit"
    e.Severity = Validation.RuleSeverity.Warning
    Return False

  ElseIf target.TotalAmount > cust.CreditLimit * .5 Then
    e.Description = "Exceeding 50% of credit limit"
    e.Severity = Validation.RuleSeverity.Information
    Return False

  Else
    Return True
  End If

End Function
```

In this example, the rule's logic checks a credit limit value. If the limit is exceeded the result is an error, while if it is just under the limit a warning is issued. If the amount exceeds 50% of the limit then an informational message is returned.

Remember, only the `Error` result causes the object to be considered invalid. `Warning` and `Information` results do not cause `IsValid` to return `False`, and will not prevent the `Save()` method from saving the object. You may choose to override `Save()` to alter this default behavior, if desired.

Using Rule Priorities

Sometimes it is important to control which rule methods are executed first, and which are executed later in the process. CSLA .NET 2.1 allows you to control the order of execution through the use of rule priorities. As you associate rule methods to properties, you can choose to provide a priority for that rule. This is supported through overloads of `AddRule()` and `AddInstanceRule()`, and so it affects your code in the `AddBusinessRules()` and `AddInstanceBusinessRules()` methods.

Priorities start at 0 (zero), which is the highest priority. A priority of 1 is the next lowest, followed by priority 2, priority 3 and so on. All rules of priority 0 are invoked before any priority 1 rules will be invoked. Within a given priority the order of the rules is nondeterministic, meaning that you can not count on the order in which they will be invoked. The default priority for rules is priority 0.

The most common use for rule priorities is to enable short-circuiting, which I'll discuss next. The idea is that you can run low-cost rules first, and only invoke expensive rules if none of the inexpensive rules fail. For example, there's no sense going to the database to validate some value, if that value is required, but is currently blank. The following code ensures that the required-field check runs before the database lookup:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule( _
    AddressOf Csla.Validation.CommonRules.StringRequired, "CreditCode")
  ValidationRules.AddRule( _
    AddressOf VerifyCreditCode, "CreditCode", 1)

End Sub
```

The bolded call to `AddRule()` is specifying that this rule should run at priority 1, after all priority 0 rules have run. Since 0 is the default, the `StringRequired` rule will run first, followed by the `VerifyCreditCode` rule.

It is important to realize that the per-type and per-instance rule lists are merged *before* sorting by priority. This means that *all* priority 0 rules (per-type and per-instance) are run before any per-type or per-instance priority 1 rules will be invoked.

Again, the primary purpose behind rule priorities is to support short-circuiting, so let's talk about that feature.

Using Short-Circuiting

The short-circuiting feature allows you to stop the processing of rule methods for a property in the middle of the process. Some of the rule methods will have been invoked, and any remaining rule methods are not invoked once short-circuiting occurs. The goal of short-circuiting is to allow you to invoke less expensive rule methods first, and only invoke more expensive rule methods if all previous rules were satisfied.

This feature is strongly linked to the concept of rule priorities as discussed earlier. For short-circuiting to work, you must be able to control the order in which the rule methods are invoked.

Short-circuiting stops the processing of rules for the current property only. If the property has dependant properties, their rules will be processed even if short-circuiting has occurred. Similarly, if `ValidationRules.CheckRules()` is used to check all rules for all properties, short-circuiting won't stop the overall process; it will only stop the processing of subsequent rules for each individual property.

Short-circuiting can be used in two ways. The most common scenario is to run all rules up to a certain priority, and then only run lower-priority rules if no previous rule methods have returned `False` with a severity of `Error`. Another option is that a rule method can explicitly cause short-circuiting through code, based on the logic in that method. Let's look at each approach.

Short-Circuiting by Priority

When short-circuiting by priority, you must use the overloads of `AddRule()` and `AddInstanceRule()` to set the priority of your rule methods. Remember that priority 0 is the default, and that larger priorities are executed after lower priorities.

`ValidationRules` exposes a property, `ProcessThroughPriority`, that controls when short-circuiting will have an effect. All rule methods at `ProcessThroughPriority` or smaller will be invoked. Rule methods with priorities greater than `ProcessThroughPriority` will only be invoked if no previous rule method has returned `False` with a severity of `Error`.

Notice that rule methods returning `False` with severity of `Warning` or `Information` *do not* trigger short-circuiting.

So using the previous example from the rule priority discussion, the `AddBusinessRules()` method sets rule priorities:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule( _
    AddressOf Csla.Validation.CommonRules.StringRequired, "CreditCode")
  ValidationRules.AddRule( _
    AddressOf VerifyCreditCode, "CreditCode", 1)

  ValidationRules.ProcessThroughPriority = 0

End Sub
```

In this case, I have explicitly set the `ProcessThroughPriority` to `0`, which is the default. This means that all priority 0 rule methods will be invoked, regardless of success or failure. But rules at priority 1 or higher will only be invoked if no prior rule has returned `False` with a severity if `Error`.

For this example the `VerifyCreditCode` rule method will not be invoked if the `StringRequired` rule returns `False`. This meets the goal of running less expensive rules first and only running more expensive rules if the inexpensive ones are satisfied.

Explicit Short-Circuiting

Another approach to short-circuiting is to have a rule method explicitly indicate that no subsequent rules should be invoked for this property. Typically, you'll use rule priorities to control the order in which rule methods are invoked, so you can predict which rules won't be executed when short-circuiting occurs.

Within a rule method, you can choose to stop the processing of all subsequent rule methods for the current property. To do this, the rule method should set `e.StopProcessing` to `True`. For example:

```
Private Shared Function IdExists(Of T As Customer)( _
  ByVal target As T, ByVal e As RuleArgs) As Boolean

  Dim result As Boolean = IdExistsInTable(target.mId)
  If result Then
    e.Description = "Id already exists in database"
    e.StopProcessing = True
    Return False

  Else
    Return True
  End If
End Function
```

This rule method checks to see if the id value already exists in the database. If it does, the method returns `False`, along with a description. But it also sets `StopProcessing` to `True`, ensuring that no subsequent rule methods will be invoked for this property.

Short-circuiting, either using the priority threshold or explicitly stopping the processing, can be used to gain substantial performance benefits for properties that have both expensive and inexpensive rule methods.

Using Strongly-typed Rule Methods

If you've been reading through the previous sections on using the new validation features, you've seen examples of strongly typed rule methods, but I haven't walked through their use from end to end.

Strongly-typed rule methods allow you to use the compiler to help you debug your code, rather than waiting for casting exceptions at runtime. They also allow you to avoid having to manually cast the `target` and `RuleArgs` parameter values in your rule methods.

Using strongly-typed rule methods is a two part process. First, you should use the generic syntax when defining your rule method itself. Second, you must use the generic overloads of `AddRule()` and `AddInstanceRule()` when associating your rule methods with the properties of your business object.

Defining Strongly-typed Rule Methods

You can choose to define only the type of the `target` parameter, or also the type of the `RuleArgs` parameter. The following rule method defines only the type of the `target` parameter:

```
Private Shared Function MaxCredit(Of T As Customer)( _
  ByVal target As T, ByVal e As RuleArgs) As Boolean

  If target.mCreditLimit > 10000 Then
    e.Description = "Maximum credit limit exceeded"
    Return False

  Else
    Return True
  End If
End Function
```

This syntax may appear a bit odd at first, because the method isn't directly defining the generic type. Instead, a generic constraint is used to require that `T` be of type `Customer`. This constraint allows the code inside the method to treat `T` as though it were of type `Customer`, and requires any code invoking this method to provide a parameter of type `Customer` (or a subclass of `Customer`).

This next example defines the types of both parameters:

```
Private Shared Function MaxCredit(Of T As Customer, R As MaxCreditRuleArgs)( _
  ByVal target As T, ByVal e As R) As Boolean

  If target.mCreditLimit > e.MaxCredit Then
    e.Description = "Maximum credit limit exceeded"
    Return False

  Else
    Return True
  End If
End Function
```

The same technique is used, so both the `T` and `R` type parameters are constrained to specific types, and the `target` and `e` parameters are defined by those type parameters. Within the method, the target and e values are strongly typed, so any properties or methods defined on `Customer` and `MaxCreditRuleArgs` are available for use.

In either case, notice that you don't need to write any code in the method to cast the parameter values. The need to cast is avoided through the use of generics.

Adding Strongly-typed Rule Methods to your Objects

When you have rule methods defined using the generic syntax shown above you need to use generic overloads of `AddRule()` and `AddInstanceRule()` when associating those methods with your business object's properties.

The following example is used for a rule method that only defines the type of the `target` parameter:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule(Of Customer)( _
    AddressOf MaxCredit, "RequestedCredit")

End Sub
```

If both the `target` and `e` parameter types are specified by the rule method, then you need to use code like the following:

```
Protected Overrides Sub AddBusinessRules()

  ValidationRules.AddRule(Of Customer, MaxCreditRuleArgs)( _
    AddressOf MaxCredit, _
    New MaxCreditRuleArgs("RequestedCredit", 10000))

End Sub
```

The primary reason for using strongly-typed rule methods is that the compiler can help you find parameter typing issues at compile time. This is simpler and more reliable than trying to find type conversion exceptions at runtime during testing.

Retrieving Rule Information

The `ValidationRules` class now implements a `GetRuleDescriptions()` method that returns an array of `String`. Each item in the array represents a rule method that has been associated with a property of your business object. Both per-type and per-instance rules are returned in this array.

The returned information can be useful for generating documentation about the rules used in each object, or by a UI developer to help automate the creation of the UI. In particular, you could use this information in a Web Forms UI framework to help automate the association of validation controls to other UI controls to mirror some of the validation rules in your business objects.

The `ValidationRules` class is a `Protected` member of `BusinessBase`, and so the `GetRuleDescriptions()` method can only be called from code within your business objects themselves. If you want to expose this information publicly, to the UI for example, you'll need to implement your own `Public` method for that purpose.

Each item in the array is a `String` in URI format, with the following structure:

```
rule://methodName/propertyName?arg1=value&arg2=value
```

Table 7 lists the parts of the structure.

URI Part	Example	Description
Scheme	`rule://`	Prefix indicating this is a business rule.
Host	`methodName`	The name of the rule method.
LocalPath	`propertyName`	The name of the business object property associated with the rule.
Query	`arg1=value&arg2=value`	A list of extra arguments and their values as provided to the rule method when `AddRule()` or `AddInstanceRule()` was called with a custom `RuleArgs` subclass.

Table 7. Parts of the rule:// URI format

One primary reason the items are in a URI format is so you can use the built-in functionality of `System.Uri` to parse the values. The following example code shows how to extract each part of the URI as listed in Table 7:

```
Dim rules() As String = ValidationRules.GetRuleDescriptions

Dim uri As New System.Uri(rules(0))

Dim scheme As String = uri.GetLeftPart(UriPartial.Scheme)
Dim methodName As String = uri.Host
Dim propertyName As String = uri.LocalPath.Substring(1)
Dim query As String = uri.Query
```

Using the `GetRuleDescriptions()` method, along with `System.Uri` for parsing, allows you to gain relatively detailed information about the rules associated with the properties of your business object.

Retrieving Broken Rules in an Array

The `BusinessBase` class in CSLA .NET exposes a `BrokenRulesCollection` property, which means that all of your editable business objects automatically expose this property. The purpose of this property is to allow the UI developer to get a list of the human-readable descriptions of all broken rules in your business object.

In CSLA .NET 2.1, a `ToArray()` method has been added to the `BrokenRulesCollection` class. That way, the UI developer can easily retrieve an array of all the broken rule descriptions for a business object. This means the UI can contain code like the following:

```
Dim cust As Customer = Customer.GetCustomer(42)
Dim brokenRules() As String = cust.BrokenRulesCollection.ToArray
For Each rule As String In brokenRules
  ' do something with the text description
Next
```

This example returns a list of all broken rule descriptions, of all severities. The `ToArray()` method has one overload, which allows you to restrict the results to a specific severity: `Error`, `Warning` or `Information`. For example:

```
    Dim brokenRules() As String = _
      Me.BrokenRulesCollection.ToArray(Validation.RuleSeverity.Information)
```

In this case, only the `Information` severity broken rule descriptions are returned.

At this point, you should understand both the implementation and usage of the new validation rules features in CSLA .NET 2.1. Next let's discuss the changes to authorization rules.

Authorization Rules

The authorization rules support in CSLA .NET has been enhanced in version 2.1 to support the concept of per-type rules, much like you've already seen with validation rules. The version 2.0 concept of per-instance rules is still available. So if your objects need to have different sets of roles, on an object-by-object basis, that is possible.

CSLA .NET 2.1 also adds a new interface, `IAuthorizeReadWrite`, to standardize how authorization is exposed to the UI developer. This interface provides a clear mechanism by which the UI can ask a business object whether the current user is authorized to read or write each property on the object.

For most objects you'll typically use the new per-type authorization support, because it requires less memory and increases performance. The reason for this is that the list of roles authorized to read and write each property are loaded just once per AppDomain, rather than once per object instance. In most cases, this means that the roles are loaded once and are cached for the lifetime of the application.

CSLA .NET 2.0 maintains the list of roles that are allowed or denied read and write access to each property on a per-object basis. The `AddAuthorizationRules()` method in your business object was called as each object instance was created.

In CSLA .NET 2.1, this behavior has changed. The `AddAuthorizationRules()` method is now called once per AppDomain, so each of your business objects only loads this role information a single time. In most cases, you will not need to change your existing code, because any existing `AddAuthorizationRules()` method will continue to work as it did. It just won't be called as often.

Loading per-type authorization rules looks like this:

```
Protected Overrides Sub AddAuthorizationRules()

  AuthorizationRules.DenyRead("Name", "Guest")
  AuthorizationRules.AllowRead("Name", "User", "Supervisor")
  AuthorizationRules.DenyWrite("Name", "User")
  AuthorizationRules.AllowWrite("Name", "Supervisor")

End Sub
```

If you do have conditional code to load different sets of roles for different objects of the same type, then you'll need to move that code to a new `AddInstanceAuthorizationRules()` method, and make use of the new instance methods on the `AuthorizationRules` object. For instance, loading per-instance roles looks like this:

```
Protected Overrides Sub AddInstanceAuthorizationRules()

  AuthorizationRules.InstanceDenyRead("Name", "Guest")
  AuthorizationRules.InstanceAllowRead("Name", "User", "Supervisor")
  AuthorizationRules.InstanceDenyWrite("Name", "User")
  AuthorizationRules.InstanceAllowWrite("Name", "Supervisor")

End Sub
```

The changes to the CSLA .NET framework to support this per-type and per-instance concept are not unlike the changes to validation rules. Fortunately, authorization rules are a simpler concept and so the code changes aren't as complex.

Framework Changes

Enhancing the authorization rules processing in CSLA .NET involved changing and adding a number of classes. Here is a list of changed classes or types:

- `BusinessBase` (from `Csla.Core`)
- `ReadOnlyBase` (from `Csla`)
- `AuthorizationRules`
- `ReadWriteAuthorization` (from `Csla.Windows`)

And here is a list of new classes or types:

- `AuthorizationRulesManager`
- `IAuthorizeReadWrite`
- `SharedAuthorizationRules`

As you can see, most of the classes in `Csla.Security` were affected by these changes. Let's walk through the changes to the framework code, and then I'll discuss how to use these changes in your business classes.

Implementing Per-Type Authorization Rules

Version 2.1 adds the concept of per-type authorization rules, while retaining support for per-instance rules. The default behavior is now to use per-type authorization rules, which means that the `AddAuthorizationRules()` methods in the CSLA .NET base classes are now used to add per-type authorization rules. Similarly, the methods on `AuthorizationRules` to allow or deny read and write access to properties are now used to define per-type roles.

Changes to BusinessBase

`BusinessBase` has been enhanced to not only implement `AddAuthorizationRules()`, but also `AddInstanceAuthorizationRules()`:

```
Protected Overridable Sub AddInstanceAuthorizationRules()

End Sub
```

This method is invoked at appropriate points during the business object's lifecycle, when the roles need to be associated with the object's properties. This occurs when the object is created. The per-type rules are also established through the constructor, if they haven't been previously loaded into the current AppDomain:

```
Protected Sub New()

  Initialize()
  AddInstanceBusinessRules()
  If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
    SyncLock Me.GetType
```

```
        If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
          AddBusinessRules()
        End If
      End SyncLock
    End If
    AddInstanceAuthorizationRules()
    If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
      SyncLock Me.GetType
        If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
          AddAuthorizationRules()
        End If
      End SyncLock
    End If

  End Sub
```

Notice that `AddInstanceAuthorizationRules()` is called any time an object is created, but `AddAuthorizationRules()` is only called if the per-type rules haven't already been initialized. The `SharedAuthorizationRules` class is responsible for maintaining all the per-type rules for all business object types that have been loaded in the AppDomain.

In CSLA .NET 2.0, the authorization rules were maintained in an instance field within the object. In version 2.1, the per-type rules are maintained in `SharedAuthorizationRules`, but the per-instance rules are still maintained in a `Private` field within each business object. However, the declaration of this `mAuthorizationRules` field has been changed to include the `NonSerialized` attribute:

```
  <NotUndoable()> _
  <NonSerialized()> _
  Private mAuthorizationRules As Security.AuthorizationRules
```

`BusinessBase` now reloads the per-instance authorization roles when an object is deserialized, rather than including that role information in the serialized byte stream. This decreases the size of the byte stream, making use of a remote data portal more efficient.

As a side-effect of this change, the object needs to re-load the per-instance authorization rules any time the object is deserialized. This is handled by the `OnDeserializedHandler()` method in `BusinessBase`:

```
  <OnDeserialized()> _
  Private Sub OnDeserializedHandler(ByVal context As StreamingContext)

    OnDeserialized(context)
    ValidationRules.SetTarget(Me)
    AddInstanceBusinessRules()
    If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
      SyncLock Me.GetType
        If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
          AddBusinessRules()
        End If
      End SyncLock
    End If
    AddInstanceAuthorizationRules()
    If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
      SyncLock Me.GetType
        If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
          AddAuthorizationRules()
        End If
      End SyncLock
    End If

  End Sub
```

The code also checks to make sure the per-type rules exist, and it loads them if they are needed. This is important because the first object retrieved from a remote data portal call could be the object that initializes the per-type rules on a client workstation, and that initialization would occur due to deserialization of the object from the data portal.

Changes to ReadOnlyBase

Read-only objects also have authorization rules, and the changes to `ReadOnlyBase` are very similar to those in `BusinessBase`. Again, an `AddInstanceAuthorizationRules()` method is defined, and the methods to add authorization rules are invoked in the constructor and on deserialization.

Since the code changes to `ReadOnlyBase` are so similar to those in `BusinessBase`, I won't repeat them here.

Changes to AuthorizationRules

The `AuthorizationRules` class has extensive changes to handle per-type and per-instance rules. This is the class that is used by business developers as they write code in their business classes to set up and use authorization rules, and so it is the primary point of entry to the authorization rules functionality.

Caching the AuthorizationRuleManager Objects

Because `BusinessBase` and `ReadOnlyBase` no longer allow serialization of the `AuthorizationRules` object, this object is no longer marked with the `Serializable` attribute. Additionally, `AuthorizationRules` now maintains a reference to both the per-type and per-instance role lists for the business object:

```
Public Class AuthorizationRules

  Private mBusinessObjectType As Type
  Private mTypeRules As AuthorizationRulesManager
  Private mInstanceRules As AuthorizationRulesManager
```

Though `SharedAuthorizationRules` is responsible for maintaining the list of per-type rules for all business objects, it is more efficient to retrieve that list once and cache the reference directly in each business object. This is the purpose behind the `mTypeRules` field. The `mInstanceRules` field maintains a reference to the list of per-instance rules for the current business object.

The class includes a couple helper properties to initialize these fields. This means that no code in the rest of the class uses these fields directly, but rather all access is through the helper properties:

```
  Private ReadOnly Property InstanceRules() _
    As AuthorizationRulesManager
    Get
      If mInstanceRules Is Nothing Then
        mInstanceRules = New AuthorizationRulesManager
      End If
      Return mInstanceRules
    End Get
  End Property

  Private ReadOnly Property TypeRules() _
```

```
      As AuthorizationRulesManager
      Get
        If mTypeRules Is Nothing Then
          mTypeRules = SharedAuthorizationRules.GetManager(mBusinessObjectType, True)
        End If
        Return mTypeRules
      End Get
    End Property
```

Notice how both properties handle the creation or initialization of the field, caching the result after the first load. In the case of `InstanceRules` this is merely a convenience, but in the case of `TypeRules` this is done as an optimization to avoid a `Dictionary` lookup every time the per-type rules are needed.

Adding Authorization Rules

To see how these helper properties are used, let's look at the `AllowRead()` method. This method is used to add a per-type authorization rule to the object:

```
    Public Sub AllowRead( _
      ByVal propertyName As String, ByVal ParamArray roles() As String)

      Dim currentRoles As RolesForProperty = TypeRules.GetRolesForProperty(propertyName)
      For Each item As String In roles
        currentRoles.ReadAllowed.Add(item)
      Next

    End Sub
```

Notice how the `TypeRules` property is used to retrieve the list of roles for the specified property, so the new role can be added to that list. The `RolesForProperty` object is unchanged from version 2.0, and you can get details about that class from *Expert VB 2005 Business Objects* (ISBN 1590596315).

The `AllowWrite()`, `DenyRead()` and `DenyWrite()` methods follow this same pattern.

Per-instance rules are added using a parallel set of methods. For example, here's the `InstanceAllowRead()` method:

```
    Public Sub InstanceAllowRead( _
      ByVal propertyName As String, ByVal ParamArray roles() As String)

      Dim currentRoles As RolesForProperty = _
        InstanceRules.GetRolesForProperty(propertyName)
      For Each item As String In roles
        currentRoles.ReadAllowed.Add(item)
      Next

    End Sub
```

In this case, the `InstanceRules` helper property is used to retrieve the list of per-instance rules, so the role can be added to the specified property. The `InstanceAllowWrite()`, `InstanceDenyRead()` and `InstanceDenyWrite()` methods follow this same pattern.

Checking Authorization Rules

Finally, `AuthorizationRules` implements a set of methods that check the authorization rules, such as `HasReadAllowedRoles()` and `IsReadAllowed()`. These methods now check both the per-type and per-instance role lists. For example, here's the `HasReadAllowedRoles()` method:

```
  Public Function HasReadAllowedRoles( _
    ByVal propertyName As String) As Boolean

    Dim result As Boolean
    If InstanceRules.GetRolesForProperty(propertyName).ReadAllowed.Count > 0 Then
      result = True

    Else
      result = TypeRules.GetRolesForProperty(propertyName).ReadAllowed.Count > 0
    End If

    Return result

  End Function
```

Notice that both the per-type and per-instance lists are checked. Similarly, `IsReadAllowed()` checks both lists:

```
  Public Function IsReadAllowed(ByVal propertyName As String) As Boolean

    Dim result As Boolean
    Dim user As System.Security.Principal.IPrincipal = ApplicationContext.User
    If InstanceRules.GetRolesForProperty(propertyName).IsReadAllowed(user) Then
      result = True

    Else
      result = TypeRules.GetRolesForProperty(propertyName).IsReadAllowed(user)
    End If
    Return result

  End Function
```

There are comparable methods to check for denied read, allowed write and denied write. These methods are used by `BusinessBase` and `ReadOnly` base to implement their `CanReadProperty()` and `CanWriteProperty()` methods, but all the changes from version 2.0 to 2.1 are encapsulated here in `AuthorizationRules`.

AuthorizationRulesManager Class

The per-type and per-instance authorization rules are now stored in an instance of `AuthorizationRulesManager`. This object is responsible for organizing the lists of roles associated with each property of the business object.

To do this, `AuthorizationRulesManager` maintains a `Dictionary`, where the key is the name of the property, and the value is a `RolesForProperty` object that maintains the list of allowed and denied roles for reading and writing to that property.

```
  Private mRules As Dictionary(Of String, RolesForProperty)

  Friend ReadOnly Property RulesList() _
    As Dictionary(Of String, RolesForProperty)
    Get
      If mRules Is Nothing Then
        mRules = New Dictionary(Of String, RolesForProperty)
      End If
      Return mRules
    End Get
  End Property
```

The `AuthorizationRulesManager` then implements a method to allow retrieval of the role data. This method simply returns the `RolesForProperty` object associated with the specified property name:

```
  Friend Function GetRolesForProperty( _
    ByVal propertyName As String) As RolesForProperty

    Dim currentRoles As RolesForProperty = Nothing
    If Not RulesList.ContainsKey(propertyName) Then
      currentRoles = New RolesForProperty
      RulesList.Add(propertyName, currentRoles)

    Else
      currentRoles = RulesList.Item(propertyName)
    End If
    Return currentRoles

  End Function
```

You've already seen how this method is used by `AuthorizationRules` as it implements the methods like `HasReadAllowedRoles()` and `IsReadAllowed()`.

SharedAuthorizationRules Class

The final class needed to implement per-type authorization rules is the `SharedAuthorizationRules` class. As you've seen in the previous code, this type is responsible for maintaining all the per-type authorization rules for all business object types in the AppDomain.

This object maintains a `Dictionary`, keyed by business object type, that contains the `AuthorizationRulesManager` object with each business object's per-type rules. This `Dictionary` is a `Shared` field, meaning it is global to the AppDomain:

```
Friend Module SharedAuthorizationRules

  Private mManagers As New Dictionary(Of Type, AuthorizationRulesManager)
```

Remember that all fields and methods in a `Module` are automatically `Shared`.

The `GetManager()` method is used to retrieve the `AuthorizationRulesManager` for a specific business object type:

```
  Friend Function GetManager(ByVal objectType As Type, ByVal create As Boolean) _
    As AuthorizationRulesManager

    Dim result As AuthorizationRulesManager = Nothing
    If Not mManagers.TryGetValue(objectType, result) AndAlso create Then
      SyncLock mManagers
        result = New AuthorizationRulesManager
        mManagers.Add(objectType, result)
      End SyncLock
    End If
    Return result

  End Function
```

This method is implemented in much the same manner as the `GetManager()` method in the `Csla.Validation.SharedValidationRules` class. The `AuthorizationRules` object uses this

method to retrieve the appropriate `AuthorizationRulesManager` for the business object when it needs access to the per-type rules.

Notice the use of the `SyncLock` statement in the `GetManager()` method. Because this method is `Shared` (due to being in a `Module`), it should be made threadsafe. In the case that multiple threads call `GetManager()` at the same time, `SyncLock` will ensure that only one thread at a time will execute the critical code in the method.

There's also a `RulesExistFor()` method that is used by `BusinessBase` and `ReadOnlyBase` to determine whether per-type rules have been initialized for a specified business type:

```
  Public Function RulesExistFor(ByVal objectType As Type) As Boolean

    Return mManagers.ContainsKey(objectType)

  End Function
```

At this point, you should understand how the per-type and per-instance authorization rules are implemented. The `BusinessBase` and `ReadOnlyBase` classes allow the business developer to define both per-type and per-instance rules. The `AuthorizationRules` class manages both sets of rules for each business object, relying on `AuthorizationRulesManager` objects to maintain the detailed information on a per-property basis. And the `SharedAuthorizationRules` object manages all the per-type rules for all objects, caching them and making them available to all code in the AppDomain.

Implementing IAuthorizeReadWrite

The Microsoft .NET framework defines `System.ComponentModel.IDataErrorInfo` to provide a standardized way for UI code to ask objects whether any validation rules are currently broken. This interface is used by Windows Forms data binding to power the `ErrorProvider` control for example.

Unfortunately, there is no equivalent interface for standardizing per-property authorization. In version 2.1, CSLA .NET introduces its own interface for this purpose, making it easier to develop UI components and frameworks that can interact with business objects in a standardized manner.

IAuthorizeReadWrite Interface

The `Csla.Security.IAuthorizeReadWrite` interface provides a standard way for UI code to ask a business object if the current user is authorized to read or write to each property of the object:

```
Public Interface IAuthorizeReadWrite
  Function CanWriteProperty(ByVal propertyName As String) As Boolean
  Function CanReadProperty(ByVal propertyName As String) As Boolean
End Interface
```

The UI can use the information provided by this interface to provide visual cues to the user as to what they can expect to do with each data element.

For this to work, your business objects must implement this interface. You don't need to worry about this detail in your business classes, as the interface is implemented on your behalf in `BusinessBase` and `ReadOnlyBase`.

Changes to BusinessBase and ReadOnlyBase

Both `BusinessBase` and `ReadOnlyBase` implement `IAuthorizeReadWrite`, relying in the pre-existing authorization rules implementation to do the hard work. In fact, BusinessBase already implemented `CanReadProperty()` and `CanWriteProperty()` methods in CSLA .NET 2.0, so those methods now simply implement the interface:

```
  Public Overridable Function CanReadProperty( _
    ByVal propertyName As String) As Boolean _
    Implements Csla.Security.IAuthorizeReadWrite.CanReadProperty
```

And

```
  Public Overridable Function CanWriteProperty( _
    ByVal propertyName As String) As Boolean _
    Implements Csla.Security.IAuthorizeReadWrite.CanWriteProperty
```

The same is true for `CanReadProperty()` in `ReadOnlyBase`, but in version 2.0, `ReadOnlyBase` didn't implement `CanWriteProperty()` at all. Now it must provide an implementation, though it simply returns `False` at all times:

```
Private Function CanWriteProperty(ByVal propertyName As String) As Boolean _
  Implements Security.IAuthorizeReadWrite.CanWriteProperty

  Return False

End Function
```

Since read-only objects should only have read-only properties, there shouldn't be a case where the user is authorized to write to a read-only property.

With these changes, all CSLA .NET objects support this new interface and can be accessed in a standardized manner from UI code. The CSLA .NET framework does include one UI helper that is impacted by this change: `Csla.Windows.ReadWriteAuthorization`.

Changes to the ReadWriteAuthorization Control

The `ReadWriteAuthorization` control is a Windows Forms extender control that helps simplify authorization logic in a Windows Forms detail form. It automatically sets the `ReadOnly` property on controls like `TextBox` based on the authorization information provided by the business object property to which that control is bound through data binding. For complete details about this control please refer to *Expert VB 2005 Business Objects* (ISBN 1590596315).

In CSLA .NET 2.1, the `ReadWriteAuthorization` control has been enhanced to use the new `IAuthorizeReadWrite` interface when querying the business object to get authorization information. This impacts the implementation of the `ApplyAuthorizationRules()` method:

```
  Private Sub ApplyAuthorizationRules(ByVal control As Control)

    For Each binding As Binding In control.DataBindings
      ' get the BindingSource if appropriate
      If TypeOf binding.DataSource Is BindingSource Then
        Dim bs As BindingSource = CType(binding.DataSource, BindingSource)
        ' get the BusinessObject if appropriate
        Dim ds As Csla.Security.IAuthorizeReadWrite = _
          TryCast(bs.DataSource, Csla.Security.IAuthorizeReadWrite)
```

```
          If ds IsNot Nothing Then
            ' get the object property name
            Dim propertyName As String = _
              binding.BindingMemberInfo.BindingField

            ApplyReadRules(control, binding, _
              ds.CanReadProperty(propertyName))
            ApplyWriteRules(control, binding, _
              ds.CanWriteProperty(propertyName))
          End If
        End If
      Next

    End Sub
```

The lines of code using the new interface are highlighted for clarity.

If you compare this code to the original version 2.0 code, you'll see that this implementation is much simpler, and avoids the need to check for and cast the type to either `BusinessBase` or `ReadOnlyBase`. This change not only makes the code easier to read, but it means that the `ReadWriteAuthorization` control will automatically support any future object types that implement `IAuthorizeReadWrite`.

Using the Enhancements

The per-type enhancements to the authorization rules support in CSLA .NET are often transparent to existing code. In most cases, you can follow the same coding approach you used in version 2.0. However, you'll get better performance and less consumption of memory.

The `IAuthorizeReadWrite` interface has no impact on your business code at all. It exists entirely to help support the creation of UI frameworks and components.

Using Per-Type Authorization Rules

In CSLA .NET 2.1, authorization rules may be associated with a business object at the type or instance level. Per-type rules are associated with *all* business objects of a given type, while per-instance rules are associated with one specific instance of a business object.

Per-type rules are far more efficient in their use of memory, and offer performance benefits because the association of rules to properties only occurs once per AppDomain rather than as each object is created. Typically, this means the association occurs once during the lifetime of the application.

Per-instance rules provide more flexibility, because these rules are associated with the object's properties as each object is created. You can write code to change the way the rules are associated with the object based on the specific object being created. This results in more memory consumption and slower performance, because the list of rules is maintained and created as each business object is instantiated.

When creating a business object, you can now override either `AddAuthorizationRules()` or `AddInstanceAuthorizationRules()`.

Associating Rule Methods with Properties

The `AddAuthorizationRules()` method is called only once per AppDomain for each type of business object. In this method, you can call the `AllowRead()`, `AllowWrite()`, `DenyRead()`

and `DenyWrite()` methods on `AuthorizationRules` to associate roles with the properties of your business object. These roles are then used by CSLA .NET to provide authorization for the properties of all business objects of that type.

A typical `AddAuthorizationRules()` method might look like this:

```
Protected Overrides Sub AddAuthorizationRules()

  AuthorizationRules.DenyRead("Name", "Guest")
  AuthorizationRules.AllowRead("Name", "User", "Supervisor")
  AuthorizationRules.DenyWrite("Name", "User")
  AuthorizationRules.AllowWrite("Name", "Supervisor")

End Sub
```

It is also possible to load different authorization rules for each object instance. These are called per-instance rules and are configured in an `AddInstanceAuthorizationRules()` method. Such a method might look like this:

```
Protected Overrides Sub AddInstanceAuthorizationRules()

  AuthorizationRules.InstanceDenyRead("Name", "Guest")
  AuthorizationRules.InstanceAllowRead("Name", "User", "Supervisor")
  AuthorizationRules.InstanceDenyWrite("Name", "User")
  AuthorizationRules.InstanceAllowWrite("Name", "Supervisor")

End Sub
```

In general terms, you should avoid using per-instance rules in favor of the more efficient per-type rules.

Using IAuthorizeReadWrite

The `IAuthorizeReadWrite` interface is designed to support UI framework and component authors. If you are building a UI framework or component, you can use this interface to standardize how you interact with any CSLA .NET business object.

Given a reference to a business object, you can simply cast the object to `IAuthorizeReadWrite` to use the standard methods on that interface:

```
Dim cust As Customer = Customer.GetCustomer(42)

Dim auth As IAuthorizeReadWrite = CType(cust, IAuthorizeReadWrite)

Dim canRead As Boolean = auth.CanReadProperty("Name")
Dim canWrite As Boolean = auth.CanWriteProperty("Name")
```

You can use this technique as appropriate when creating your UI frameworks and components.

You should have a good understanding of the new per-type and `IAuthorizeReadWrite` features of authorization within CSLA .NET 2.1, including both their implementation and usage in your business objects.

FilteredBindingList

CSLA .NET 2.0 includes the `SortedBindingList` class, which provides an editable, sorted, view of any `IList(Of T)` collection type. Since arrays and most list and collection types implement `IList(Of T)`, `SortedBindingList` allows you to get a live sorted view of almost any list in .NET.

The new `FilteredBindingList` class provides the same kind of live view against any `IList(Of T)`, and provides the ability to filter the contents of that list. As with `SortedBindingList`, applying a filter doesn't alter the original list at all, it merely alters the view. However, adding or removing items from the filtered list immediately adds or removes the item from the original list.

Because `FilteredBindingList` and `SortedBindingList` both implement `IList(Of T)`, they are composable. This means you can take an array or list, use `FilteredBindingList` to get a filtered view, and then use `SortedBindingList` to get a sorted view of the filtered view.

The default filter is a simple wildcard match, but `FilteredBindingList` is extensible and you can provide your own filter algorithm. A filter is merely a method that matches a delegate method signature, and you pass a reference to that delegate into `FilteredBindingList`.

Framework Changes

Three types have been added to CSLA .NET to support the `FilteredBindingList`:

- `FilteredBindingList`
- `FilterProvider`
- `DefaultFilter`

Obviously, most of the work occurs in `FilteredBindingList` itself. `FilterProvider` defines the delegate signature for any filter provider, and `DefaultFilter` implements the default matching filter provided by CSLA .NET.

Implementing FilteredBindingList

`FilteredBindingList` depends on the `FilterProvider` delegate to do its work. I'll cover that, along with the default filter, first. Then I'll discuss `FilteredBindingList` itself.

FilterProvider Delegate

When filtering a list, each item in the list must be checked to see if it meets the filter condition. To do this, `FilteredBindingList` loops through all the items in the original list, calling a `Boolean` method to evaluate each item. If the item matches the filter condition this method should return `True`, and `FilteredBindingList` will include that item in the filtered view.

This filter method must conform to a specific method signature, defined by the `FilterProvider` delegate:

```
Public Delegate Function FilterProvider( _
  ByVal item As Object, ByVal filter As Object) As Boolean
```

The `item` parameter is the item to be evaluated, and it comes from the original list. The `filter` parameter could be any criteria required by the filter provider method. In the case of `DefaultFilter`, this is a simple text value that is compared to the item with a wildcard match.

The filter provider method must evaluate the item to determine if it meets the filter criteria, and return `True` if the item should be included in the filtered view.

DefaultFilter

CSLA .NET includes a default filter provider method that does a simple text-based wildcard match against each item in the original list. The `DefaultFilter` class implements a single method, `Filter`, that conforms to the `FilterProvider` delegate signature:

```
Friend Class DefaultFilter

  Public Shared Function Filter( _
    ByVal item As Object, ByVal filterValue As Object) As Boolean

    Dim result As Boolean = False

    If Not item Is Nothing AndAlso Not filterValue Is Nothing Then
      result = CStr(item).Contains(CStr(filterValue))
    End If

    Return result

  End Function

End Class
```

This method simply converts both the `item` and `filterValue` parameters to `String` values and uses the `Contains()` method to find out if the item's text representation contains the filter value.

The important thing is that the method returns `True` for items that meet the filter criteria, and `False` for items that don't meet the criteria. You can use this method as a template for creating other, more specialized, filters to meet your needs.

FilteredBindingList Class

The `FilteredBindingList` class contains a reference to the original `IList(Of T)` object, and provides a filtered view of the items in that original list. To do this, it maintains a list of the index values for the items in the original list that meet the filter criteria based on the filter provider method.

`FilteredBindingList` is a long and somewhat complex class, primarily because it directly implements a number of collection interfaces. These include:

- `IList(Of T)`
- `IBindingList`
- `IEnumerable(Of T)`
- `ICancelAddNew`

These are the same interfaces implemented by `SortedBindingList`, and so I am not going to cover all the code in great detail. You can refer to *Expert VB 2005 Business Objects* (ISBN 1590596315) for a more lengthy discussion on how these interfaces are implemented as a wrapper around the original list object.

I also recommend referring to the book for details regarding the event interactions between the original list and the filtered view. As items are added, removed or changed in either the original list or the view; events are raised and handled by both lists. This interaction is relatively complex, but is required to provide the ability to directly manipulate the data in the filtered list as though it were a normal list object, even though it is really just a wrapper around another list object.

Basic Implementation

There are some important differences between `SortedBindingList` and `FilteredBindingList` that need to be discussed. One of the most important is somewhat subtle: `SortedBindingList` always has the same number of items as the original list, while `FilteredBindingList` often has a different number of items. This simple fact complicates the implementation of the wrapper class in various ways, starting with the `Count` property:

```
Public ReadOnly Property Count() As Integer _
    Implements IList(Of T).Count, IBindingList.Count
  Get
    If mFiltered Then
      Return mFilterIndex.Count
    Else
      Return mList.Count
    End If
  End Get
End Property
```

Where `SortedBindingList` always delegates this call to the original list, `FilteredBindingList` must return only the number of items in its filtered list. Looking at this code, you can infer much about how `FilteredBindingList` does its work. The `mList` field contains a reference to the original list, while `mFilterIndex` is the list of original index values for all items meeting the filter criteria.

There are two important helper methods that are used throughout the implementation: `OriginalIndex()` and `FilteredIndex()`. These helper methods translate index values to and from the original index and the filtered index. In other words, the original list could have 10 items, and the filtered view may only show the last 5. This means that an original index of 0 doesn't exist in the filtered view at all. A filtered index of 0 translates to an original index of 5, while an original index of 9 translates to a filtered index of 4.

```
Private Function OriginalIndex(ByVal filteredIndex As Integer) As Integer
  Return mFilterIndex(filteredIndex).BaseIndex
End Function

Private Function FilteredIndex(ByVal originalIndex As Integer) As Integer

  Dim result As Integer = -1
  If mFiltered Then
    Dim index As Integer = 0
    Do While index < mFilterIndex.Count
      If mFilterIndex(index).BaseIndex = originalIndex Then
        result = index
        Exit Do
      End If
      index += 1
    Loop

  Else
    result = originalIndex
  End If
  Return result

End Function
```

The `OriginalIndex()` method is relatively straightforward, because the `mFilterIndex` field maintains a cross-reference table that maps filtered index values back to the original indexes. All that's required here is to find the entry in the filtered index and retrieve the original (base) index stored at that location in the index.

The `FilteredIndex()` method is a bit more complex because there's no index to directly translate original index values to their filtered counterparts. Instead, the code loops through `mFilterIndex` looking for a match between the requested index and the original index stored in the filtered index. Notice that if a match isn't found, then a value of `-1` is returned from the method to indicate that the original item isn't included in the filtered view.

Applying the Filter

The most interesting part of `FilteredBindingList` is applying the filter itself. This process is initiated through the `ApplyFilter()` method:

```
Public Sub ApplyFilter(ByVal propertyName As String, ByVal filter As Object)

  mFilterBy = Nothing

  If (Not String.IsNullOrEmpty(propertyName)) Then
    Dim itemType As Type = GetType(T)
    For Each prop As PropertyDescriptor In TypeDescriptor.GetProperties(itemType)
      If prop.Name = propertyName Then
        mFilterBy = prop
        Exit For
      End If
    Next prop
  End If

  ApplyFilter(mFilterBy, filter)

End Sub

Public Sub ApplyFilter(ByVal [property] As PropertyDescriptor, ByVal filter As Object)

  mFilterBy = [property]
  mFilter = filter
  DoFilter()
```

```
  End Sub
```

This method has two overloads for parity with the `ApplySort()` method in `SortedBindingList`. The first takes a `String` value to identify the property on which to sort, while the second takes a `PropertyDescriptor`. Notice that these methods merely ensure that the `PropertyDescriptor` and filter criteria are stored in the appropriate fields before `DoFilter()` is invoked. The `DoFilter()` method does the actual work:

```
  Private Sub DoFilter()

    Dim index As Integer = 0
    mFilterIndex.Clear()

    If mProvider Is Nothing Then
      mProvider = AddressOf DefaultFilter.Filter
    End If

    If mFilterBy Is Nothing Then
      For Each obj As T In mList
        If mProvider.Invoke(obj, mFilter) Then
          mFilterIndex.Add(New ListItem(obj, index))
        End If
        index += 1
      Next obj
    Else
      For Each obj As T In mList
        Dim tmp As Object = mFilterBy.GetValue(obj)
        If mProvider.Invoke(tmp, mFilter) Then
          mFilterIndex.Add(New ListItem(tmp, index))
        End If
        index += 1
      Next obj
    End If

    mFiltered = True

    OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, 0))

  End Sub
```

This method ensures that `mProvider` is set to a valid filter provider delegate. The business developer may have set this value when creating an instance of `FilteredBindingList`, or by setting the `FilterProvider` property. However, if they did neither then the value would be `Nothing`, and so here it is set to use the default filter method I discussed earlier.

When `ApplyFilter()` is called, the filter property could be passed as `Nothing`. In that case, the object itself is passed to the filter provider method along with the filter criteria:

```
      For Each obj As T In mList
        If mProvider.Invoke(obj, mFilter) Then
          mFilterIndex.Add(New ListItem(obj, index))
        End If
        index += 1
      Next obj
```

This process is repeated for each item in the original list, resulting in `mFilterIndex` containing an entry for each item that meets the filter criteria.

On the other hand, if the filter should run against a specified property, then that property value is retrieved from the object and the value is then passed to the filter provider method:

```
      For Each obj As T In mList
        Dim tmp As Object = mFilterBy.GetValue(obj)
        If mProvider.Invoke(tmp, mFilter) Then
          mFilterIndex.Add(New ListItem(tmp, index))
        End If
        index += 1
      Next obj
```

The basic process is the same, as the code loops through all items in the original list, adding an entry to `mFilterIndex` for each matching element.

Either way, the end result is that `mFilterIndex` contains a list of items that match the filtered criteria. This list is used by the rest of the implementation to provide the filtered view. For instance, the `For...Each` statement uses an enumerator to loop through the items in the list. When the list is filtered, a special enumerator is returned to the `For...Each` code generated by the compiler:

```
Public Function GetEnumerator() As IEnumerator(Of T) _
  Implements IEnumerable(Of T).GetEnumerator

  If mFiltered Then
    Return New FilteredEnumerator(mList, mFilterIndex)
  Else
    Return mList.GetEnumerator()
  End If

End Function
```

This `FilteredEnumerator` returns the items in the filtered index, rather than all the items in the original list. Similarly, the `Item` property gets or sets the item corresponding to the filtered index location:

```
Default Public Property Item(ByVal index As Integer) As T _
    Implements IList(Of T).Item
  Get
    If mFiltered Then
      Dim src As Integer = OriginalIndex(index)
      Return mList(src)
    Else
      Return mList(index)
    End If
  End Get
  Set(ByVal value As T)
    If mFiltered Then
      mList(OriginalIndex(index)) = value
    Else
      mList(index) = value
    End If
  End Set
End Property
```

Notice the use of the `OriginalIndex()` helper method to translate the filtered index location back to the original list location, so the item can be retrieved from the original list. This reinforces the idea that the `FilteredBindingList` doesn't directly contain the items, but rather, it delegates all that work to the original list.

Removing the Filter

Along with the `ApplyFilter()` methods, there's also a `RemoveFilter()` method to remove any filter:

```
Public Sub RemoveFilter()

  UnDoFilter()

End Sub
```

The `UndoFilter()` method is the counterpart to `DoFilter()`, removing the filter and resetting all the fields to default values:

```
Private Sub UnDoFilter()

  mFilterIndex.Clear()
  mFilterBy = Nothing
  mFilter = Nothing
  mFiltered = False

  OnListChanged(New ListChangedEventArgs(ListChangedType.Reset, 0))

End Sub
```

It also raises the `ListChanged` event to tell data binding that the list has changed, so any UI controls can refresh their display accordingly.

Adding Items to a Filtered List

`FilteredBindingList` allows items to be added or removed from the filtered view, and those items are directly added or removed from the original list. Removing items is simple enough, as they are removed from the original list and the filtered view. Adding items is a bit more complex, because the item being added might not meet the filter criteria. While the item must be added to the original list, it isn't so clear whether it should also be added to the filtered view.

I opted to leave all added items in the filtered view, even if they don't meet the filter criteria. The reason is the user experience for in-place editing within a grid control. If the user adds an item into a grid control that is bound to a filtered list, the user probably expects that item to remain in the grid. If `FilteredBindingList` immediately removes the new item from the view, the user would see the row disappear, even though it was added to the original list, and that could lead to serious confusion.

This is implemented in the `SourceChanged()` method, which is where `FilteredBindingList` is notified that the original list has changed. Remember that any item added to the list is actually added to the original list, and the filtered view finds out about this through a `ListChanged` event, which is handled by the `SourceChanged()` method. This method contains several sections to handle different events, including the case that a new item was added to the original list:

```
        Case ListChangedType.ItemAdded
          listIndex = e.NewIndex
          ' add new value to index
          newItem = mList(listIndex)
          If Not mFilterBy Is Nothing Then
            newKey = mFilterBy.GetValue(newItem)
          Else
            newKey = newItem
          End If
          mFilterIndex.Add(New ListItem(newKey, listIndex))
          filteredIndexValue = mFilterIndex.Count - 1
          ' raise event
          OnListChanged( _
```

```
            New ListChangedEventArgs(e.ListChangedType, filteredIndexValue))
```

This code contains many of the elements of the `DoFilter()` method I discussed earlier. It determines whether the filter is applied to a specific property or not, and sets a `newKey` value to the key value for the newly added item. It then adds a new entry to the filtered index corresponding to this new item in the original list.

Notice that it does not invoke the filter provider method. The newly added item is added at the end of the filtered view regardless of whether it matches the filter criteria, so there's no reason to invoke the filter provider method at all. This approach provides a user doing in-place editing in a data bound grid control with a predictable and expected experience.

At this point, you should understand that the `FilteredBindingList` merely maintains a filtered cross-reference index so it can provide a filtered view of the original list. Where possible, it delegates all work to the original list, though it often must translate index values between the filtered position and the original position of each item. The class also directly implements some properties and methods, like `Count`, to provide the illusion of being an actual collection rather than just a wrapper over another collection.

Using the Enhancements

While the implementation of `FilteredBindingList` is quite complex, using a filtered list is quite straightforward. Remember that `FilteredBindingList` looks and works like any `BindingList(Of T)` collection object, and so it can be used anywhere you would have otherwise used a `BindingList(Of T)`.

However, if you call the `ApplyFilter()` method, you can get a filtered view of the list, and that's the value of this object. In some cases you may need to build your own filter provider method, because a simple `Contains()` check may be insufficient for your needs.

I'll walk through the basic use of the object first, and then discuss how you can create and use your own filter provider method. Then I'll discuss how you can use `FilteredBindingList` and `SortedBindingList` together to create a filtered and sorted view of a list.

Using FilteredBindingList

The `FilteredBindingList` class itself is very similar in concept to the `SortedBindingList` already in CSLA .NET. It merely contains a reference to the original list object, and provides a filtered view of the items in that original list.

Creating an instance of `FilteredBindingList` requires that you already have an original list that contains the items you want to filter. This list can be as simple as an array, or as complex as a `BindingList(Of T)` or a business collection derived from `BusinessListBase(Of T,C)` or `ReadOnlyListBase(Of T,C)`. Here's a simple example using an array of `String` values:

```
    Dim originalList() As String = {"Rocky", "John", "Fred"}

    Dim filteredList As New FilteredBindingList(Of String)(originalList)

    filteredList.ApplyFilter("", "J")
```

The result of this code is that `filteredList` will have a `Count` of `1`, and it will contain only the item `John`, because that's the only element containing the letter `J`.

Here's another example, using a collection of `Customer` objects:

```
    Dim customers As CustomerList = CustomerList.GetCustomers()

    Dim filteredList As New FilteredBindingList(Of Customer)(customers)

    filteredList.ApplyFilter("Name", "J")
```

The result of this code will be only `Customer` objects with `Name` properties that contain the letter `J`. Notice that the `FilteredBindingList` is of type `Customer`, not `CustomerList`. This follows the same pattern as `BindingList(Of T)`, where the type parameter specifies the type of the items in the list.

These examples show two ways of calling `ApplyFilter()`, with and without a property name on which to filter. If `Nothing` or an empty `String` is passed as the property name, then the filter runs against the entire object. When a property name is passed to the method, then only that property value is used by the filter.

There's also an overload of `ApplyFilter()` that accepts a `PropertyDescriptor` instead of the property name. This overload exists for parity with `SortedBindingList`, and isn't used in most scenarios.

Creating a Custom Filter

The previous examples use the default filter provider method, which does a simple `Contains()` check to see if the specified text value is in the target object or property. You may have more sophisticated requirements for your filter criteria.

In that case, you'll need to create your own filter provider method. Typically, this will be a `Shared` method, or a method in a `Module`, though it can be any method you can use with the `AddressOf` operator. The primary requirement is that the method signature conform to the `FilterProvider` delegate discussed earlier. The method must return `True` for items that should be included in the filtered view, and `False` for those that should not. Here's the basic structure of a filter provider method:

```
Public Module MyCustomFilter

  Public Function Filter(ByVal item As Object, ByVal filterValue As Object) As Boolean

    If <condition is met> Then
      Return True

    Else
      Return False
    End If

  End Function

End Module
```

The `<condition is met>` part of the code is where you come in. You need to implement code here to check the value of the `item` parameter based on your rules, along with any criteria provided through the `filterValue` parameter.

Remember that the `item` parameter could be the value of a specific property, or it could be a reference to the actual business object. And keep in mind that the `filterValue` parameter is of type `Object`, and so it could be any value you'd like, even a complex object.

You can use this custom filter method by passing it into the `FilteredBindingList` in various ways. For example, when you create an instance of the list you can provide a reference to your method:

```
Dim list As New List(Of String)

Dim filteredList As New FilteredBindingList(Of String)( _
  list, AddressOf MyCustomFilter.Filter)

filteredList.ApplyFilter(...)
```

Another approach is to set the `FilterProvider` property:

```
Dim list As New List(Of String)

Dim filteredList As New FilteredBindingList(Of String)(list)
filteredList.FilterProvider = AddressOf MyCustomFilter.Filter

filteredList.ApplyFilter(...)
```

Or you can pass the custom filter method as a parameter to the `ApplyFilter()` method:

```
Dim list As New List(Of String)

Dim filteredList As New FilteredBindingList(Of String)(list)

filteredList.ApplyFilter("", AddressOf MyCustomFilter.Filter)
```

In each case the `FilteredBindingList` will use your custom filter method to filter the contents of the list.

Combining FilteredBindingList with SortedBindingList

One of the most exciting features of `FilteredBindingList` is that it can be applied against any `IList(Of T)`. The same is true of `SortedBindingList`. And both of these objects implement `IList(Of T)` themselves, which means they can be applied to each other. The result is that you can take an original list, apply a filter to it, and then apply a sort to that filtered result:

```
Dim originalList() As String = {"Rocky", "John", "Fred", "Joe"}

Dim filteredList As New FilteredBindingList(Of String)(originalList)

filteredList.ApplyFilter("", "J")

Dim sortedList As New SortedBindingList(Of String)(filteredList)

sortedList.ApplySort("", ComponentModel.ListSortDirection.Ascending)
```

The result of this code is that `sortedList` has a `Count` of `2`, containing `John` and `Joe`, but sorted in ascending order. Keep in mind that `filteredList` also has a `Count` of `2`, and it contains `John` and `Joe`, but not in sorted order, and `originalList` contains all four original items.

Of course neither `sortedList` nor `filteredList` really contain any items at all, they merely contain sorted and filtered indexes back to the items contained in `originalList`.

You should now understand how to use `FilteredBindingList`, including how to create your own filter provider methods and how you can combine it with `SortedBindingList` to create filtered and sorted views of an original list object.

EditableRootListBase

CSLA .NET 2.0 supports three types of collection; through the `BusinessListBase`, `ReadOnlyListBase` and `NameValueListBase` classes. Only `BusinessListBase` is designed to support adding, removing and editing of the items in the collection, and it requires that the objects it contains be editable child objects. That means objects that inherit from `BusinessBase`, where those objects call `MarkAsChild()` in their constructor.

The process of using a `BusinessListBase`-derived object is that you retrieve the collection, you interact with the collection and the items it contains, and then you save the collection:

```
' get the the collection
Dim codes As CodeList = CodeList.GetList

' edit the data in the collection
codes(0).Name = "New value"
codes(1).Name = "Another value"

' save all the changes
codes = codes.Save
```

This last step, saving the collection, is when any changes to the collection and its child objects are committed to your database. All changes are typically saved as a transactional unit.

In some cases more dynamic behavior is desired, so the changes to each item in the collection can be saved immediately, rather than waiting to save all the changes in a single `Save()` call at the end. This is the purpose behind the `EditableRootListBase` class: to support this more granular editing process:

```
' get the collection
Dim codes As CodeList = CodeList.GetList

' edit the first item
codes(0).Name = "New value"
codes.SaveItem(0)

' edit the second item
codes(1).Name = "Another value"
codes.SaveItem(1)
```

While this code accomplishes the same result as the prior example, the way it works is very different. In this case, each item is individually saved to the database right after the value has been edited. Rather than all changes being saved within the context of a single call to the data portal, and within a single transaction, this new approach uses separate calls to the data portal and separate transactions for each save operation.

Behind the scenes the implementation of the “child” objects is different as well. `EditableRootListBase` is designed to contain editable root objects, rather than editable child objects. In other words, it contains objects that inherit from `BusinessBase` that do *not* call `MarkAsChild()` in their constructor.

It is also the case that `EditableRootList` base tightly interacts with Windows Forms data binding for in-place editing within a grid control. The result is that edits to an item in the collection are automatically saved as the user moves out of a row in the grid control. This

includes both adding and editing of items. Also, if the user deletes an item in the grid control, that item is automatically deleted from the database, as soon as it is removed from the grid control.

Framework Changes

Implementing `EditableRootListBase` requires the addition of an interface, and some alterations to `BusinessBase` to support some of the automatic interaction with data binding through the new collection type. The following classes are changed:

- `BusinessBase`

And the following are new types:

- `EditableRootListBase`
- `IParent` (from `Csla.Core`)

Implementing EditableRootListBase

In this chapter I'll discuss only the `EditableRootListBase` class itself. The `IParent` interface, and the changes it requires in `BusinessBase` are discussed in the Csla.Core chapter later in the book.

EditableRootListBase Class

This new collection type inherits from the `ExtendedBindingList` class, which I'll discuss in the `Csla.Core` chapter later in the book. For now it is enough to understand that `ExtendedBindingList` inherits from `BindingList(Of T)`, which means that `EditableRootListBase` essentially inherits from `BindingList(Of T)` as well:

```
<Serializable()> _
Public MustInherit Class EditableRootListBase( _
    Of T As {Core.IEditableBusinessObject, Core.ISavable})
  Inherits Core.ExtendedBindingList(Of T)

  Implements Core.IParent
```

Like all CSLA .NET base classes, the `Serializable` attribute is used to indicate that this is a mobile object.

Also notice the constraints on the type parameter, `T`. `EditableRootListBase` will only contain objects that implement both the `IEditableBusinessObject` and `ISavable` interfaces as defined in `Csla.Core`. The result is that the collection can only contain editable root objects.

Finally, the class implements the `IParent` interface. This interface will be covered in detail in the `Csla.Core` chapter later in the book, but for now you should know that this interface enables interaction between an object and its container, or parent.

`EditableRootListBase` is organized into a set of code regions:

- SaveItem Methods
- Insert, Remove, Clear
- IParent Members
- Cascade Child Events
- Serialization Notification
- Data Access

Each region implements a key part of the functionality in the object. Several of these regions should seem familiar, as they are the same as you'd find in other CSLA .NET base classes. Others are unique to this particular class.

Let's walk through the code in each region.

SaveItem Methods Region

This region contains the code to save individual items in the collection. There are two overloads of the `SaveItem()` method, and they are both public so a UI developer can call them if needed. As you'll see, it is also the case that `SaveItem()` is automatically called due to the tight integration with Windows Forms data binding.

The first overload is a convenience, allowing the saving of an item by reference:

```
Public Sub SaveItem(ByVal item As T)

  SaveItem(IndexOf(item))

End Sub
```

The real work happens in the other overload:

```
Public Overridable Sub SaveItem(ByVal index As Integer)

  Dim raiseEvents As Boolean = Me.RaiseListChangedEvents
  Me.RaiseListChangedEvents = False

  mActivelySaving = True
  Dim item As T = Me.Item(index)
  Dim editLevel As Integer = item.EditLevel
  ' commit all changes
  For tmp As Integer = 1 To editLevel
    item.AcceptChanges()
  Next
  Try
    ' do the save
    Me.Item(index) = DirectCast(item.Save, T)

  Finally
    ' restore edit level to previous level
    For tmp As Integer = 1 To editLevel
      item.CopyState()
    Next

    mActivelySaving = False
    Me.RaiseListChangedEvents = raiseEvents
  End Try
  Me.OnListChanged(New ListChangedEventArgs(ListChangedType.ItemChanged, index))
```

```
  End Sub
```

The primary responsibility of this method is to call the `Save()` method on the item to be saved. It does this through the `ISavable` interface, and so can work with any editable root object:

```
    Try
      ' do the save
      Me.Item(index) = DirectCast(item.Save, T)
```

Remember that the type parameter, `T`, is constrained by the `ISavable` interface, so any field of type `T` is guaranteed to have a `Save()` method.

Notice that the result of the `Save()` call is used to replace the item in the list. This means that the collection automatically ends up containing a reference to the result of `Save()`, and the old reference is discarded.

Before trying to do the save, `RaiseListChangedEvents` is set to `False` to prevent any events raised by the child object during the save process from triggering `ListChanged` events back to the UI. Without this step the UI could receive numerous changed events during the save operation, causing UI flicker and possibly resulting in bugs that could be hard to find.

The `mActivelySaving` field is used to indicate that the child item is in the process of being saved. As you'll see, this is important because it is used to suppress the handling of some events that are raised during the process. If those events aren't suppressed, an infinite loop and stack overflow could result.

The most complex issue addressed in this code deals with the edit level of the child object.

Remember that CSLA .NET business objects support n-level undo capabilities. Windows Forms data binding often automatically triggers this behavior, especially if you bind the collection to a grid control. The result is that a child object, as the user edits that object, will be at edit level 1 or higher.

This is desirable, because it supports the idea that the user might press `ESC` to cancel changes to that row of data, and n-level undo can roll the object back to its previous state.

However, an object can only be saved if it is at edit level 0. This means that the edit level must be brought down to 0 before `Save()` can be called:

```
    Dim editLevel As Integer = item.EditLevel
    ' commit all changes
    For tmp As Integer = 1 To editLevel
      item.AcceptChanges()
    Next
```

However, once the `Save()` method has been called, the edit level must be restored to its original value, or data binding will fail to work properly:

```
      ' restore edit level to previous level
      For tmp As Integer = 1 To editLevel
        item.CopyState()
      Next
```

The issue would be that data binding would expect the object to be in an editable state, and if we don't restore the edit level there'd be a mismatch between data binding's expectation and the object's actual state.

The final step is to raise a `ListChanged` event to indicate that the item has changed:

```
    Me.OnListChanged(New ListChangedEventArgs(ListChangedType.ItemChanged, index))
```

Remember that the `Save()` call replaced the original item with a new object reference, so it is important that any consumers of the collection, such as data binding, know to refresh their references and update the display of any information.

Insert, Remove, Clear Region

As items are added to the collection, they must be made aware of their new parent. This is done by calling the `SetParent()` method on the newly added object. The `InsertItem()` method is automatically called when an item is inserted or added to the collection, so it is a natural place to take care of this detail:

```
Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As T)

  item.SetParent(Me)
  MyBase.InsertItem(index, item)

End Sub
```

Removing an item from the collection is a bit more complex, because removing an item from the collection means deleting it from the database as well. This is done by marking the object for deletion and then saving the object; using the deferred deletion support already in CSLA .NET.

The `RemoveItem()` method is automatically called when an item is being removed from the collection:

```
Protected Overrides Sub RemoveItem(ByVal index As Integer)

  ' delete item from database
  Dim item As T = Me.Item(index)

  ' only delete/save the item if it is not new
  If Not item.IsNew Then
    item.Delete()
    SaveItem(index)
  End If

  ' disconnect event handler if necessary
  Dim c As System.ComponentModel.INotifyPropertyChanged = _
    TryCast(item, System.ComponentModel.INotifyPropertyChanged)
  If c IsNot Nothing Then
    RemoveHandler c.PropertyChanged, AddressOf Child_PropertyChanged
  End If

  MyBase.RemoveItem(index)

End Sub
```

If the item being removed from the collection is a new object, then `IsNew` will return `True`. In that case, the object's data doesn't yet exist in the database, so there's no need to try and

delete it. However, if `IsNew` is `False` then the data exists in the database so the object needs to be deleted:

```
    If Not item.IsNew Then
      item.Delete()
      SaveItem(index)
    End If
```

The next bit of code removes any event handler hooked up to the item's `PropertyChanged` event. Normally `BindingList(Of T)` automatically handles this event hookup, but if the collection is serialized and deserialized, then the automatic hookup doesn't occur and an event handler must be set up explicitly as discussed in the Serialization Notification Region discussion later in this chapter.

If you manually set up an event handler, it is important to remove that event handler when you are done with the object, and that's what happens here:

```
    ' disconnect event handler if necessary
    Dim c As System.ComponentModel.INotifyPropertyChanged = _
      TryCast(item, System.ComponentModel.INotifyPropertyChanged)
    If c IsNot Nothing Then
      RemoveHandler c.PropertyChanged, AddressOf Child_PropertyChanged
    End If
```

If the item can be cast to `INotifyPropertyChanged`, the event handler is removed. If no event handler was established the removal does no work, and doesn't fail.

IParent Members Region

The `IParent` interface defined in `Csla.Core` formalizes the responsibilities of any object that contains other objects. It requires that the parent object in this case `EditableRootListBase`, handles the case where the child object's `ApplyEdit()` method has been called, and when the child wishes to be removed from its parent.

The `ApplyEditChild()` method is called by a child object when its `ApplyEdit()` method has been called, so the parent knows that the child's edit level has been reduced by one. This is important for `EditableRootListBase`, because when one if its items' edit level reaches 0, that item should be automatically saved:

```
  Private Sub ApplyEditChild( _
    ByVal child As Core.IEditableBusinessObject) _
    Implements Core.IParent.ApplyEditChild

    If Not mActivelySaving AndAlso child.EditLevel = 0 Then
      SaveItem(CType(child, T))
    End If

  End Sub
```

Notice the use of the `mActivelySaving` field, as well as the check for the edit level. The reason for this is that the `SaveItem()` method I discussed earlier may try to lower the edit level to 0. In that case, this `ApplyEditChild()` method should not also try to trigger a save of the item or the result would be two save attempts on the same child object.

The complexity comes because there are two ways to trigger the saving of an item: manually, by calling `SaveItem()`, or automatically through data binding when the edit level

of a child item hits 0. To make things even more interesting, you could manually trigger this edit level process in your code as well. For instance:

```
' get the collection
Dim codes As CodeList = CodeList.GetList

' edit the first item
codes(0).BeginEdit
codes(0).Name = "New value"
codes(0).ApplyEdit
```

That last line of code calls `ApplyEdit()`, lowering the edit level from 1 to 0. As a result the child item calls the `ApplyEditChild()` method in the collection, triggering a save operation. This is exactly what data binding does on your behalf when editing the collection in a Windows Forms grid control.

It is also possible, when using in-place editing in a grid control, for data binding to trigger the removal of a new child object. However, we've already overridden the `RemoveItem()` method in the collection, so the removal of child items is handled. Due to this, the `RemoveChild()` method does no work:

```
Private Sub RemoveChild( _
  ByVal child As Core.IEditableBusinessObject) Implements Core.IParent.RemoveChild

  ' do nothing, removal of a child is handled by
  ' the RemoveItem override

End Sub
```

Of course some implementation of this method is required by the `IParent` interface, even if it is an empty implementation.

Cascade Child Events Region

As I mentioned earlier, `BindingList(Of T)` normally handles the `PropertyChanged` events from any child objects in the collection. However, if the collection is serialized and deserialized then those event handlers don't get automatically reestablished. To overcome this, I manually handle the event using the following handler:

```
Private Sub Child_PropertyChanged(ByVal sender As Object, _
  ByVal e As System.ComponentModel.PropertyChangedEventArgs)

  For index As Integer = 0 To Count - 1
    If ReferenceEquals(Me(index), sender) Then
      OnListChanged(New System.ComponentModel.ListChangedEventArgs( _
        ComponentModel.ListChangedType.ItemChanged, index))
      Exit For
    End If
  Next

End Sub
```

This code raises a `ListChanged` event to indicate that the specified item in the collection has changed. This handler is hooked up to the child object in the Serialization Notification region.

Serialization Notification Region

Like all the other CSLA .NET base classes, `EditableRootListBase` implements a method so it is notified by the `BinaryFormatter` when the object has been deserialized. When this happens, an `Overridable` method named `OnDeserialized()` is called, allowing business classes to also be notified that the object has been deserialized.

In the case of `EditableRootListBase`, a little extra work is required to manually hook up a handler for any `PropertyChanged` events raised by the child objects in the collection:

```
<OnDeserialized()> _
Private Sub OnDeserializedHandler(ByVal context As StreamingContext)

  OnDeserialized(context)
  For Each child As Core.IEditableBusinessObject In Me
    child.SetParent(Me)
    Dim c As System.ComponentModel.INotifyPropertyChanged = _
      TryCast(child, System.ComponentModel.INotifyPropertyChanged)
    If c IsNot Nothing Then
      AddHandler c.PropertyChanged, AddressOf Child_PropertyChanged
    End If
  Next

End Sub
```

The code simply loops through all the items in the collection, and adds an event handler for `PropertyChanged` if the child item implements the `INotifyPropertyChanged` interface.

Data Access Region

All the CSLA .NET base classes implement the data access methods required by the data portal. Some of these methods are `Private` and merely throw exceptions when called, because those specific data access operations are not supported. For instance, a read-only object only supports the `DataPortal.Fetch()` operation, and all other operations result in an exception.

While `EditableRootListBase` is technically not a read-only object, the collection itself can not be saved or deleted. Remember that each individual child object in the collection must be an editable root object, and thus is responsible for implementing its own insert, update and delete operations.

This means that the only data portal the method `EditableRootListBase` allows to be overridden in a business subclass is `DataPortal_Fetch()`:

```
Protected Overridable Sub DataPortal_Fetch(ByVal criteria As Object)
  Throw New NotSupportedException(My.Resources.FetchNotSupportedException)
End Sub
```

The default behavior is to throw an exception, with the goal of forcing the business developer to override this method to implement their specific data access code. A business class must override this method with an implementation that loads all the editable root child objects into the collection based on the supplied criteria.

This completes the `EditableRootListBase` class. It should now be clear that a business developer can create a new type of collection that immediately inserts, updates and deletes its child objects rather than deferring all those changes until the collection itself is saved.

Using the Enhancements

The `EditableRootListBase` class offers an alternative to `BusinessListBase` when building collections. In many ways this new type of collection is similar to the Dynaset concept from Visual Basic 3.0 and DAO, in that changes to items in the collection are immediately committed to the underlying database.

EditableRootListBase Class Template

All `EditableRootListBase`-derived business collections follow a basic structure. The class includes a standard set of regions:

- Authorization Rules
- Factory Methods
- Data Access

Because `EditableRootListBase` already does the majority of the work, not a lot of code is required in the business subclass. Here's the code template:

```
Imports System.Data.SqlClient

<Serializable()> _
Public Class DynamicRootList
  Inherits EditableRootListBase(Of EditableRoot)

#Region " Authorization Rules "

  Public Shared Function CanGetObject() As Boolean
    ' TODO: customize to check user role
    Return ApplicationContext.User.IsInRole("")
  End Function

  Public Shared Function CanEditObject() As Boolean
    ' TODO: customize to check user role
    Return ApplicationContext.User.IsInRole("")
  End Function

#End Region

#Region " Factory Methods "

  Protected Overrides Function AddNewCore() As Object

    Dim item As EditableRoot = EditableRoot.NewEditableRoot
    Add(item)
    Return item

  End Function

  Public Shared Function NewDynamicRootList() As DynamicRootList
    Return New DynamicRootList()
  End Function

  Public Shared Function GetDynamicRootList() As DynamicRootList
    Return DataPortal.Fetch(Of DynamicRootList)()
  End Function

  Private Sub New()

    Me.AllowEdit = True
    Me.AllowNew = True
    Me.AllowRemove = True
```

```
  End Sub

#End Region

#Region " Data Access "

  Private Overloads Sub DataPortal_Fetch()

    ' TODO: load values
    RaiseListChangedEvents = False
    Using dr As SqlDataReader = Nothing
      While dr.Read
        Add(EditableRoot.GetEditableRoot(dr))
      End While
    End Using
    RaiseListChangedEvents = True

  End Sub

#End Region

End Class
```

This template illustrates how to create a collection that contains a type called `EditableRoot`, which would derive from `BusinessBase(Of T)`.

Altering the EditableRoot Template

This `EditableRoot` class is a standard editable root business object, with one exception: the `GetEditableRoot()` factory method is not a typical factory. The `GetEditableRoot()` factory method in the `EditableRoot` class looks like this:

```
  Friend Shared Function GetEditableRoot(ByVal dr As SqlDataReader) As EditableRoot

    Return New EditableRoot(dr)

  End Function
```

The constructor called in this code looks like this:

```
  Private Sub New(ByVal dr As SqlDataReader)

    Fetch(dr)

  End Sub
```

And the `Fetch()` method called by the constructor, located in the *Data Access* region, looks like this:

```
  Private Sub Fetch(ByVal dr As SqlDataReader)

    ' load object fields from data reader
    ' mName = dr.GetString("Name")
    MarkOld

  End Sub
```

This code should seem familiar, because it is the same pattern used in the editable child object template. What I'm doing here is changing only the retrieval code for the editable root object so it acts like a child object. All the rest of the editable root code remains the same; including the validation, authorization and data access code.

Using EditableRootListBase

Using the `EditableRootListBase` class requires that you create two business classes: the editable root to be contained in the collection, and the collection itself. To illustrate how to use this base class, I'll create a simple editable root, followed by the collection.

Creating an Editable Root

For illustration purposes, the following is the skeleton of a very simple editable root object, modified slightly so it has a child-style factory method as discussed earlier.

```
<Serializable()> _
Public Class Person
  Inherits BusinessBase(Of Person)

#Region " Business Methods "

  Private mId As Integer
  Public Property Id() As Integer
    <System.Runtime.CompilerServices.MethodImpl( _
      Runtime.CompilerServices.MethodImplOptions.NoInlining)> _
    Get
      CanReadProperty(True)
      Return mId
    End Get
    <System.Runtime.CompilerServices.MethodImpl( _
      Runtime.CompilerServices.MethodImplOptions.NoInlining)> _
    Set(ByVal value As Integer)
      CanWriteProperty(True)
      If Not mId.Equals(value) Then
        mId = value
        PropertyHasChanged()
      End If
    End Set
  End Property

  Private mName As String = ""
  Public Property Name() As String
    <System.Runtime.CompilerServices.MethodImpl( _
      Runtime.CompilerServices.MethodImplOptions.NoInlining)> _
    Get
      CanReadProperty(True)
      Return mName
    End Get
    <System.Runtime.CompilerServices.MethodImpl( _
      Runtime.CompilerServices.MethodImplOptions.NoInlining)> _
    Set(ByVal value As String)
      CanWriteProperty(True)
      If Not mName.Equals(value) Then
        mName = value
        PropertyHasChanged()
      End If
    End Set
  End Property

  Protected Overrides Function GetIdValue() As Object
    Return mId
  End Function

#End Region

#Region " Validation Rules "

#End Region

#Region " Authorization Rules "
```

```
#End Region

#Region " Factory Methods "

  Public Shared Function NewPerson() As Person
    Return DataPortal.Create(Of Person)()
  End Function

  Friend Shared Function GetPerson(ByVal dr As SafeDataReader) As Person
    Return New Person(dr)
  End Function

  Private Sub New()
    ' require use of factory methods
  End Sub

  Private Sub New(ByVal dr As SafeDataReader)
    Fetch(dr)
  End Sub

#End Region

#Region " Data Access "

  Private Shared lastId As Integer

  <RunLocal()> _
  Protected Overrides Sub DataPortal_Create()

    ' set a temporary id value
    lastId -= 1
    mId = lastId

  End Sub

  Private Sub Fetch(ByVal dr As SafeDataReader)

    mId = dr.GetInt32("Id")
    mName = dr.GetString("Name")
    MarkOld()

  End Sub

  Protected Overrides Sub DataPortal_Insert()

    ' insert data here
    Debug.WriteLine(String.Format("Insert object {0}", mId))

  End Sub

  Protected Overrides Sub DataPortal_Update()

    ' update data here
    Debug.WriteLine(String.Format("Update object {0}", mId))

  End Sub

  Protected Overrides Sub DataPortal_DeleteSelf()

    ' delete data here
    Debug.WriteLine(String.Format("Delete object {0}", mId))

  End Sub

#End Region

End Class
```

Though I'm not showing the implementation of the `DataPortal_XYZ` methods, you can see that all of them are implemented except for `DataPortal_Fetch()`. Notice too, that the

`GetPerson()` factory method is scoped as `Friend`, and that it ultimately calls the `Fetch()` method.

It is also important to note that the class does not call `MarkAsChild()`. The object should be an editable root, not a child. Additionally, the `Fetch()` method explicitly calls `MarkOld()`, which is required because the data portal is not being used to load this object with data.

Creating a Dynamic Collection

With the `Person` class complete, it is possible to create a dynamic list of `Person` objects by using the `EditableRootListBase` class:

```
<Serializable()> _
Public Class PersonList
  Inherits EditableRootListBase(Of Person)

#Region " Authorization Rules "

#End Region

#Region " Factory Methods "

  Protected Overrides Function AddNewCore() As Object

    Dim item As Person = Person.NewPerson
    Add(item)
    Return item

  End Function

  Public Shared Function GetList() As PersonList

    Return DataPortal.Fetch(Of PersonList)()

  End Function

  Private Sub New()

    Me.AllowEdit = True
    Me.AllowNew = True
    Me.AllowRemove = True

  End Sub

#End Region

#Region " Data Access "

  Private Overloads Sub DataPortal_Fetch()

    Me.RaiseListChangedEvents = False

    Dim dr As SafeDataReader = Nothing
    ' load data reader from database
    While dr.Read
      Add(Person.GetPerson(dr))
    End While

    Me.RaiseListChangedEvents = True

  End Sub

#End Region

End Class
```

This code is primarily focused on loading the collection with the appropriate editable root objects. While I'm using a parameterless `DataPortal.Fetch()` call, you could pass a criteria object as a parameter through to `DataPortal_Fetch()` if you need to filter the data that is loaded. The options for using the data portal here are the same as with any other CSLA .NET object.

Notice that the `DataPortal_Fetch()` method is responsible for retrieving the data from the database and passing the data reader object to the `GetPerson()` factory method, thus creating a `Person` object for each row of data from the database.

All the inserting, updating and deleting is automatically handled by `EditableRootListBase`, and by the `Person` object itself, so your collection code remains very simple.

I am also overriding the `AddNewCore()` method to enable in-place adding of new items by data binding in a grid control. This is a standard implementation of `AddNewCore()`; where a new item is created, added to the list and returned as a result of the method.

Interacting with the Dynamic Collection

Once you have a dynamic collection of editable root objects, you can interact with it through code or using data binding. While this collection style is designed primarily to support Windows Forms data binding with in-place editing in a grid control, you may find other scenarios where it is useful to you as well.

Using Data Binding

Using the collection with data binding is similar to using any other collection type, other than that you don't need to write any code to save the changes to the data. If you have a Windows Form with a grid control and associated `BindingSource` control, you'd set up the data binding like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load

  Me.PersonListBindingSource.DataSource = PersonList.GetList

End Sub
```

No other code is required in the form, because the collection directly commits any inserts, updates or deletes to the database as they occur in the grid control itself.

Using Manual Code with N-level Undo

You can also interact with the collection through code. If you choose to use n-level undo on the items in the collection then the same automatic behaviors used by data binding will be invoked on your behalf. For example, the following code edits an item, saving the results to the database:

```
Dim list As PersonList = PersonList.GetList

list(0).BeginEdit
list(0).Name = "John"
list(0).ApplyEdit
```

The following code inserts a new item:

```
Dim list As PersonList = PersonList.GetList

Dim newPerson As Person = list.AddNew

newPerson.BeginEdit()
newPerson.Id = 42
newPerson.Name = "Alice"
newPerson.ApplyEdit()
```

And the following code deletes an item:

```
Dim list As PersonList = PersonList.GetList

list.RemoveAt(0)
```

In each case the changes to the item are immediately committed to the database. This is automatically handled by the `EditableRootListBase` class itself.

Using Manual Code with Explicit SaveItem

If you don't want to use n-level undo, you can make explicit calls to the `SaveItem()` method on the collection, forcing individual items to be saved to the database. The following code is an example of editing an item:

```
Dim list As PersonList = PersonList.GetList

list(0).Name = "John"
list.SaveItem(0)
```

The following inserts a new item:

```
Dim list As PersonList = PersonList.GetList

Dim newPerson As Person = list.AddNew

newPerson.Id = 42
newPerson.Name = "Alice"
list.SaveItem(newPerson)
```

Notice the use of the `SaveItem()` overload that accepts an object reference as a parameter. This is simpler than trying to determine the index position of the newly added item, especially when the code already has a reference to that new item.

Finally, the following removes an existing item:

```
Dim list As PersonList = PersonList.GetList

list.RemoveAt(0)
```

Regardless of which technique you use to interact with your collection, you can see that `EditableRootListBase` simplifies the process of creating a dynamic collection that performs immediate updates of the database as the collection and its items are changed in your application.

Csla.Core Interfaces and Types

A number of changes have been made in `Csla.Core`, including the addition of some new base classes and interfaces, along with some restructuring of some existing interfaces. Most of these changes have been made in support of other changes in the framework. Other changes were made to support either UI or business developer scenarios, such as the `ISavable` interface which makes it easier to build a UI framework.

Table 8 lists the changes to the `Csla.Core` namespace.

Change	Description
`ExtendedBindingList`	Extends `BindingList(Of T)` to add a `RemovingItem` event to lists.
`IExtendedBindingList`	Interface used in the implementation of `ExtendedBindingList`.
`RemovingItemEventArgs`	`EventArgs` subclass used in the implementation of `ExtendedBindingList`.
`ISavable`	Provides a standard mechanism by which any editable root object can be saved.
`SavedEventArgs`	`EventArgs` subclass used in the implementation of `ISavable`.
`IParent`	Formalizes the concept of a parent object that contains child objects.
`IEditableBusinessObject`	Enhanced to interact properly with the new interfaces added in version 2.1.
`IReportTotalRowCount`	Defines a property used to implement paged collections. Used by `CslaDataSource`.

Table 8. Changes to Csla.Core.

Framework Changes

These changes can be grouped together into some broader themes:

- Raising a `RemovingItem` event from collections
- Standardizing the save process for editable objects
- Standardizing the parent-child relationship between objects
- Enabling paged collections

I'll walk through each of these in turn.

Implementing ExtendedBindingList

The `ExtendedBindingList` class inherits from `BindingList(Of T)` and adds a `RemovingItem` event. All the CSLA .NET collection base classes now inherit from `ExtendedBindingList` rather than `BindingList(Of T)`, and so gain this event as part of their interface. This includes:

- `BusinessListBase`
- `ReadOnlyListBase`
- `EditableRootListBase`

CSLA .NET collections already raise a `ListChanged` event because they inherit from `BindingList(Of T)`. Unfortunately, the `ListChanged` event is raised *after* an item has been removed. This makes it impossible to do anything with the removed item.

The purpose behind the `RemovingItem` event is to notify listeners that an item is in the process of being removed from the collection, such that the event handler can get a reference to the item being removed.

Declaring events in serializable objects is challenging, because the event might be handled by a method on a non-serializable object, such as a Windows Form or Web Page. In that case, when attempting to serialize the object you'll get an exception indicating that you are trying to serialize a non-serializable object.

The reason for this is that an event handler, behind the scenes, causes your object to maintain a reference to the object handling the event. During serialization, the `BinaryFormatter` traces all your object references, including these event references, so it can serialize those objects as well.

To overcome this issue, you must use a block structure to declare your events in a manner that is safe for serialization, and this is what `ExtendedBindingList` does with the `RemovingItem` event. The object reference is provided through the `RemovingItemEventArgs` parameter object that is provided with the event.

RemovingItemEventArgs Class

The `RemovingItemEventArgs` class is a subclass of `EventArgs` and follows the standard pattern for an `EventArgs` parameter object. This object will be passed to any method handling the `RemovingItem` event, and it provides that method with a reference to the item being removed from the collection:

```
Public Class RemovingItemEventArgs
  Inherits EventArgs

  Private mRemovingItem As Object

  Public ReadOnly Property RemovingItem() As Object
    Get
      Return mRemovingItem
    End Get
  End Property

  Public Sub New(ByVal removingItem As Object)
    mRemovingItem = removingItem
  End Sub
```

```
  End Class
```

The item reference is set in the constructor, and is provided to the event handler through a read-only property.

ExtendedBindingList Class

The `ExtendedBindingList` class inherits from `BindingList(Of T)` in the `System.ComponentModel` namespace, and extends that base class by adding the `RemovingItem` event:

```
  <Serializable()> _
  Public Class ExtendedBindingList(Of T)
    Inherits BindingList(Of T)

    Implements IExtendedBindingList

#Region " RemovingItem event "

    <NonSerialized()> _
    Private mNonSerializableHandlers As EventHandler(Of RemovingItemEventArgs)
    Private mSerializableHandlers As EventHandler(Of RemovingItemEventArgs)

    Public Custom Event RemovingItem As EventHandler(Of RemovingItemEventArgs) _
          Implements IExtendedBindingList.RemovingItem
      AddHandler(ByVal value As EventHandler(Of RemovingItemEventArgs))
        If value.Method.IsPublic AndAlso _
           (value.Method.DeclaringType.IsSerializable OrElse value.Method.IsStatic) Then
          mSerializableHandlers = _
            CType(System.Delegate.Combine( _
              mSerializableHandlers, value), EventHandler(Of RemovingItemEventArgs))

        Else
          mNonSerializableHandlers = _
            CType(System.Delegate.Combine( _
              mNonSerializableHandlers, value), EventHandler(Of RemovingItemEventArgs))
        End If
      End AddHandler

      RemoveHandler(ByVal value As EventHandler(Of RemovingItemEventArgs))
        If value.Method.IsPublic AndAlso _
           (value.Method.DeclaringType.IsSerializable OrElse value.Method.IsStatic) Then
          mSerializableHandlers = _
            CType(System.Delegate.Remove( _
              mSerializableHandlers, value), EventHandler(Of RemovingItemEventArgs))
        Else
          mNonSerializableHandlers = _
            CType(System.Delegate.Remove( _
              mNonSerializableHandlers, value), EventHandler(Of RemovingItemEventArgs))
        End If
      End RemoveHandler

      RaiseEvent(ByVal sender As System.Object, ByVal e As RemovingItemEventArgs)
        If mNonSerializableHandlers IsNot Nothing Then
          mNonSerializableHandlers.Invoke(sender, e)
        End If
        If mSerializableHandlers IsNot Nothing Then
          mSerializableHandlers.Invoke(sender, e)
        End If
      End RaiseEvent
    End Event

    <EditorBrowsable(EditorBrowsableState.Advanced)> _
    Protected Sub OnRemovingItem(ByVal removedItem As T)

      RaiseEvent RemovingItem(Me, New RemovingItemEventArgs(removedItem))
```

```
    End Sub

#End Region

    Protected Overrides Sub RemoveItem(ByVal index As Integer)
      OnRemovingItem(Me(index))
      MyBase.RemoveItem(index)
    End Sub

  End Class
```

The event is declared using a block structure, meaning that the code directly implements the adding and removing of event handlers. To store the references to the event handlers, delegate fields are used. Notice how they are declared:

```
    <NonSerialized()> _
    Private mNonSerializableHandlers As EventHandler(Of RemovingItemEventArgs)
    Private mSerializableHandlers As EventHandler(Of RemovingItemEventArgs)
```

The first is declared with the `NonSerialized` attribute, indicating that the `BinaryFormatter` should not attempt to serialize the objects referenced by this delegate. The second is a normal delegate declaration, similar to the one the compiler would have created for a normal event.

The code to add and remove handlers then checks to see if the handler of the event is an instance method of a non-serializable object. If that is the case then the `NonSerialized` delegate is used to store the handler reference, otherwise the normal delegate is used. For instance, here's the check used when adding a handler:

```
        If value.Method.IsPublic AndAlso _
           (value.Method.DeclaringType.IsSerializable OrElse value.Method.IsStatic) Then
```

The event is raised when an item is being removed from the collection. The `RemoveItem()` method is automatically invoked during the remove process. By overloading that method I can raise the `RemovingItem` event while the item being removed is still available:

```
    Protected Overrides Sub RemoveItem(ByVal index As Integer)
      OnRemovingItem(Me(index))
      MyBase.RemoveItem(index)
    End Sub
```

Notice that the reference to the object being removed is passed as a parameter to the `OnRemovingItem()` method. The `OnRemovingItem()` method follows the standard .NET pattern for raising events:

```
    <EditorBrowsable(EditorBrowsableState.Advanced)> _
    Protected Sub OnRemovingItem(ByVal removedItem As T)

      RaiseEvent RemovingItem(Me, New RemovingItemEventArgs(removedItem))

    End Sub
```

This method creates an instance of the `RemovingItemEventArgs` object, passing in the reference to the item being removed, so that reference will be available to all event handlers listening for this event.

While `BindingList(Of T)` is very powerful, the `RemovingItem` event is a useful extension to the base functionality it provides.

Implementing ISavable

Editable root objects in CSLA .NET implement a `Save()` method. This includes objects that inherit from both `BusinessBase` and `BusinessListBase`. In version 2.0 there was no common interface between both types of savable object, which made it very difficult to implement consistent UI frameworks that could save any editable root object.

The `ISavable` interface has been added to formalize the concept of a savable object, which really means an editable root object. Not only does `ISavable` define a common `Save()` method, but it defines a new event: `Saved`.

The `Saved` event is raised after an object has successfully saved itself by calling the data portal. This event follows the standard `EventHandler` pattern, passing two parameters to the event handler: a reference to the sender and a `SavedEventArgs` parameter. This `SavedEventArgs` parameter contains a reference to the new object that will be returned as a result of the `Save()` method call.

This event is intended to address the complexity that occurs when your business object is referenced in numerous locations throughout your application; by multiple forms in the UI, for instance. If you call `Save()` on the object in one location, all the other places where that object is referenced must be updated to use the new object returned as a result of `Save()`. In the past, you needed to implement some notification mechanism so your code could know to update those references.

The `Saved` event provides a solution because it is a standard, centralized, event that provides this notification. Any code holding a reference to a business object can handle the `Saved` event. That code will be notified when that object has been saved. The code can then update its reference to use the new object returned as a result of the `Save()` call.

ISavable Interface

The `ISavable` interface itself is straightforward:

```
Public Interface ISavable
  Function Save() As Object
  Event Saved As EventHandler(Of SavedEventArgs)
End Interface
```

Any class implementing this interface can be clearly saved, and will notify listeners once it has been saved.

SavedEventArgs Class

The `SavedEventArgs` class is a standard `EventArgs` subclass that provides a reference to the new object returned as a result of the `Save()` method:

```
Public Class SavedEventArgs
  Inherits EventArgs

  Private mNewObject As Object

  Public ReadOnly Property NewObject() As Object
```

```
    Get
      Return mNewObject
    End Get
  End Property

  Public Sub New(ByVal newObject As Object)
    mNewObject = newObject
  End Sub

End Class
```

The new object reference is passed in a parameter to the constructor, and is provided to event handlers as a read-only property.

Changes to BusinessBase and BusinessListBase

The `ISavable` interface is implemented by both `BusinessBase` and `BusinessListBase`. Both are base classes designed to support the creation of editable root objects.

Implementing the Save Method

The `Save()` method is easily implemented, because both classes already have `Save()` methods to which the interface implementation can delegate the work:

```
Private Function ISavable_Save() As Object Implements Core.ISavable.Save

  Return Save()

End Function
```

Remember that the previous `Save()` methods return type `T`, which is the type of the business object itself. The interface must return type `Object`, which poses no problem because any type can cast to `Object`.

Implementing the Saved Event

The `Saved` event implementation is somewhat complex. The issue is the same as with the `RemovingItem` event discussed earlier in the section on `ExtendedBindingList`: events require special declaration in a serializable object. I'm not going to repeat the details here, as the basic solution is the same as in `BindableBase` and `ExtendedBindingList`. The `Saved` event is declared using a block structure, and the delegate fields holding the references to event handlers are separate for serializable and non-serializable objects.

However, there's one key difference due to the way n-level undo works. Both `BindableBase` and `ExtendedBindingList` sit in the inheritance hierarchy above the point at which the `IUndoableObject` interface is implemented; and that is the point at which n-level undo stop processing fields in your objects.

`BusinessBase` and `BusinessListBase` are *lower* in the inheritance hierarchy than the class that implements `IUndoableObject`. Due to this, n-level undo will attempt to take a snapshot of any fields in these two classes, and that includes the delegate fields that reference the event handlers. If n-level undo were to try and take snapshots of these fields, a serialization exception would be the result.

To avoid that issue, the fields must have the `NotUndoable` attribute:

```
<NonSerialized()> _
<NotUndoable()> _
Private mNonSerializableSavedHandlers As EventHandler(Of Csla.Core.SavedEventArgs)
<NotUndoable()> _
Private mSerializableSavedHandlers As EventHandler(Of Csla.Core.SavedEventArgs)
```

Notice that only the first delegate is marked as `NonSerialized`, but both are marked as `NotUndoable`. The result is that they are totally ignored by n-level undo, and they behave properly when the object is serialized.

With the `Saved` event properly declared in both `BusinessBase` and `BusinessListBase`, all that remains is to raise the event at the appropriate point during the save process. The highlighted line of code shows where the event is raised in the `Save()` method:

```
Public Overridable Function Save() As T

  If Me.IsChild Then
    Throw New NotSupportedException( _
      My.Resources.NoSaveChildException)
  End If

  If EditLevel > 0 Then
    Throw New Validation.ValidationException( _
      My.Resources.NoSaveEditingException)
  End If

  If Not IsValid Then
    Throw New Validation.ValidationException( _
      My.Resources.NoSaveInvalidException)
  End If

  Dim result As T
  If IsDirty Then
    result = DirectCast(DataPortal.Update(Me), T)
  Else
    result = DirectCast(Me, T)
  End If

  OnSaved(result)
  Return result

End Function
```

The `OnSaved()` method raises the event:

```
<EditorBrowsable(EditorBrowsableState.Advanced)> _
Protected Sub OnSaved(ByVal newObject As T)

  RaiseEvent Saved(Me, New Csla.Core.SavedEventArgs(newObject))

End Sub
```

It creates an instance of `SavedEventArgs` to provide a reference to the result of the `Save()` method to all event handlers. Notice that the sender parameter is the *original* object that was saved, so an event handler has access to both the old and new object references.

`ISavable` provides a standard and powerful way to save objects and be notified when they've been saved. This combination can be very useful in the creation of UI frameworks or reusable base classes for forms or pages.

Implementing IParent

In CSLA .NET 2.0, the only parent for an editable child object was an object implementing the `IEditableCollection` interface; which really meant `BusinessListBase`. This turned out to be too limiting, because other objects could contain child objects as well, including the new `EditableRootListBase` collection type. I chose to generalize the concept of being a parent object through the `IParent` interface.

A number of classes had to change due to the introduction of the `IParent` interface:

- `IEditableBusinessObject`
- `BusinessBase`
- `BusinessListBase`

And as you've already seen, the new `EditableRootListBase` class makes use of the `IParent` interface.

IParent Interface

The `IParent` interface defines only the methods a child requires of its parent:

```
Public Interface IParent

  Sub RemoveChild(ByVal child As Core.IEditableBusinessObject)
  Sub ApplyEditChild(ByVal child As Core.IEditableBusinessObject)

End Interface
```

The `RemoveChild()` method is called when a child wants to be removed from its parent. This method is required by data binding; specifically the `System.ComponentModel.IEditableObject` interface defined by Microsoft. The way `IEditableObject` works, it is possible for data binding to notify a child object that it should remove itself from its collection. That child object then needs a way to ask the collection to remove the child object, and this is the purpose behind the `RemoveChild()` method.

The `ApplyEditChild()` method is called each time a child's `ApplyEdit()` method has completed. A parent object can use this method to be notified as its child objects have their changes applied. This method was added specifically to support the functionality of `EditableRootListBase` as discussed earlier in this book, but you may find it useful in other scenarios as well.

Changes to IEditableBusinessObject

Throughout CSLA .NET, all parent reference fields and methods have been changed to use the `IParent` interface type. This starts with the `SetParent()` method defined in the `IEditableBusinessObject` interface:

```
  Sub SetParent(ByVal parent As IParent)
```

Since the `BusinessListBase` class now implements `IParent`, there are no existing code breaks due to this change. However, with this change, there is now much more flexibility in terms of what objects can be used as parents of other objects.

Changes to BusinessBase

The `BusinessBase` class has had a reference to its parent object for some time. That parent reference is now of type `IParent`:

```
    <NotUndoable()> _
    <NonSerialized()> _
    Private mParent As Core.IParent

    <EditorBrowsable(EditorBrowsableState.Advanced)> _
    Protected ReadOnly Property Parent() As Core.IParent
      Get
        Return mParent
      End Get
    End Property
```

And of course, due to the change in `IEditableBusinessObject`, the `SetParent()` method now accepts a parameter of type `IParent`:

```
    Friend Sub SetParent(ByVal parent As Core.IParent) _
      Implements IEditableBusinessObject.SetParent

      mParent = parent

    End Sub
```

For the most part, the changes to `BusinessBase` are not significant. However, if you have pre-existing code that relies on the type of the `Parent` property you may have to change some of your code in response to this update.

Changes to BusinessListBase

`BusinessListBase` now implements the `IParent` interface, meaning that it implements both the `ApplyEditChild()` and `RemoveChild()` methods. The `ApplyEditChild()` method isn't needed for `BusinessListBase` to do its work, so the method is an empty implementation:

```
  Protected Overridable Sub EditChildComplete( _
    ByVal child As Core.IEditableBusinessObject) _
    Implements Core.IParent.ApplyEditChild

    ' do nothing, we don't really care
    ' when a child has its edits applied
  End Sub
```

In version 2.0, the `RemoveChild()` method was already implemented as part of `IEditableCollection`, and so it is merely changed to implement the `IParent` method:

```
  Private Sub RemoveChild(ByVal child As Core.IEditableBusinessObject) _
    Implements Core.IEditableCollection.RemoveChild, IParent.RemoveChild

    Remove(DirectCast(child, C))

  End Sub
```

My primary motivation behind creating the `IParent` interface was to enable the new `EditableRootListBase` class. There's no doubt however, that this new interface provides more clarity around the parent-child relationship, and makes it easier to create new types of parent object going forward.

Implementing IReportTotalRowCount

In web applications, it is a common requirement to page the data being returned to the browser. Ideally, however, you'd also page the data coming from the database, so only the specific data displayed on the page is actually retrieved from the database itself.

There are also some cases where paged data is required in Windows Forms applications, though that is less common. The basic structure of the problem and solution is the same in Windows as in the web: only the data displayed to the user should be retrieved from the database.

The only real trick to doing this is that you also need to know the total number of rows of data available. Even if you are returning a paged view of 10 items, you still need to know that there are 10,000 items in total. The reason this is required is that the UI needs to give the user appropriate visual cues so the user has an idea how much data there is in total, and where the current page of data is in relation to the start and end of the available data.

Web Forms data binding is designed to support the concept of paging, but there was no practical way to tap into this capability in CSLA .NET 2.0. The introduction of the `IReportTotalRowCount` interface allows you to create paged collection objects (based on `BusinessListBase` or `ReadOnlyListBase`) that can work with Web Forms data binding.

Some changes to the `CslaDataSource` control were required as well, and they are discussed later in the book, along with more details on how you can implement `IReportTotalRowCount` to build collections that support paging.

IReportTotalRowCount Interface

The new interface merely defines a `TotalRowCount` property:

```
Public Interface IReportTotalRowCount
  ReadOnly Property TotalRowCount() As Integer
End Interface
```

When you want to build a paged collection, you should implement this interface and return the total number of rows of data available through this property. The collection might only contain a fraction of the total number of rows available, but this property allows the UI to determine the total possible number of rows.

Using the Enhancements

The majority of the enhancements to `Csla.Core` are designed to support other, more public, enhancements in CSLA .NET itself. Only `ExtendedBindingList` and `ISavable` are designed for direct use by a UI or business developer, and so I'll discuss how to use them here.

The `IReportTotalRowCount` interface is also designed for use by a business developer, but I'll discuss its use later in the book in the chapter on the `CslaDataSource` control.

Using ExtendedBindingList

The `ExtendedBindingList` class inherits from `BindingList(Of T)` and adds a `RemovingItem` event to the pre-existing collection functionality provided by `BindingList`. You can use `ExtendedBindingList` anywhere you'd have used `BindingList` in the past. For instance:

```
Private WithEvents list As New ExtendedBindingList(Of String)
```

Then you can handle the `RemovingItem` event, as well as the `ListChanged` and `AddingNew` events provided by `BindingList` itself:

```
  Private Sub list_AddingNew( _
    ByVal sender As Object, ByVal e As System.ComponentModel.AddingNewEventArgs) _
    Handles list.AddingNew

  End Sub

  Private Sub list_ListChanged( _
    ByVal sender As Object, ByVal e As System.ComponentModel.ListChangedEventArgs) _
    Handles list.ListChanged

  End Sub

  Private Sub list_RemovingItem( _
    ByVal sender As Object, ByVal e As Csla.Core.RemovingItemEventArgs) _
    Handles list.RemovingItem

  End Sub
```

Within your `RemovingItem` event handler method, you can use `e.RemovingItem` to get a reference to the item being removed from the collection. You may use this capability to remove references to the item, or manipulate the item itself as it is being removed from the list.

Remember that `BusinessListBase`, `ReadOnlyListBase` and `EditableRootListBase` all inherit from `ExtendedBindingList`, and so already raise the `RemovingItem` event automatically.

Using ISavable

The `ISavable` interface is designed primarily to support the creation of UI frameworks or similar components. Using this interface, you can create reusable code that can save any editable root object. If you have a reference to an editable root object, you can also be notified when that object has been saved. That way, you can update your reference to use the result of the `Save()` operation.

Since there are many approaches to building UI frameworks and components, I'll illustrate the basic use of `ISavable` here, and you can determine how to apply the concept into your UI as you choose. Given a reference to an editable root object, you can write a reusable method to save the object like this:

```
Private mObject As Csla.Core.ISavable

Public Sub SaveObject()

  Try
    mObject = DirectCast(mObject.Save, Csla.Core.ISavable)

  Catch ex As Csla.DataPortalException
    ' process normal data exceptions here

  Catch ex As Exception
    ' process unexpected exceptions here

  End Try
```

```
  End Sub
```

It is also the case that any object implementing `ISavable` will raise the `Saved` event. You can handle that event to be notified when an object has been saved. Using this technique, you could replace the previous code with the following:

```
Private WithEvents mObject As Csla.Core.ISavable

Public Sub SaveObject()

  Try
    mObject.Save()

  Catch ex As Csla.DataPortalException
    ' process normal data exceptions here

  Catch ex As Exception
    ' process unexpected exceptions here

  End Try

End Sub

Private Sub mObject_Saved( _
  ByVal sender As Object, ByVal e As Csla.Core.SavedEventArgs) _
  Handles mObject.Saved

  mObject = DirectCast(e.NewObject, Csla.Core.ISavable)

End Sub
```

Using this second approach, the `SaveObject()` method no longer updates `mObject` to use the result of the `Save()` method call. Instead, the `Saved` event handler updates the `mObject` reference. Since `Saved` is only raised if the `Save()` operation succeeds, the reference is only updated in the case that a new object is returned as a result of the operation.

Remember that when using the local data portal you should still clone the business object before attempting the save. In that case, the `SaveObject()` method should call the `Save()` method like this:

```
    DirectCast(DirectCast(mObject, ICloneable).Clone, Csla.Core.ISavable).Save()
```

The business object is cloned, and then `Save()` is called on the clone. This way, if there's an exception thrown during the save operation the original `mObject` reference will still point to the original, unchanged, business object. If you don't do this, it is possible that the business object will have been changed in memory during the update process, and it would then be left in an indeterminate state, resulting in unpredictable results for the user.

This cloning step is only required if you are using a local data portal. If you are using a remote data portal the object is automatically cloned across the network to the application server, and so you don't need to worry about this detail.

The `ISavable` interface enables the creation of UI frameworks and components, and simplifies the process of updating references to objects after they've been saved. It is designed for use by both UI and business developers.

At this point, you should have a good understanding of the changes made to the `Csla.Core` namespace to both support the other enhancements to CSLA .NET and to provide new capabilities for UI and business developers.

LocalContext

CSLA .NET 2.0 introduced the `ApplicationContext` object, which provides a centralized location to store and access various application context data. Table 9 lists the types of information available through `ApplicationContext` in CSLA .NET 2.1.

Information	Description
`User`	A reference to the current user principal that can be safely used in both ASP.NET and non-ASP.NET environments.
`ExecutionLocation`	A value indicating whether your code is currently executing on the client or an application server.
`GlobalContext`	A `Dictionary` of values available on both client and server. This data is automatically moved to and from the application server by the data portal.
`ClientContext`	A `Dictionary` of values available on both client and server. This data is automatically moved from the client to the server by the data portal; but not from the server back to the client.
`LocalContext`	A `Dictionary` of values available to your code. This data is *not* moved between client and server by the data portal. The client and server have their own separate `Dictionary` objects.

Table 9. Information available through ApplicationContext

Of these, the only new feature in CSLA .NET version 2.1 is `LocalContext`.

`LocalContext` is similar to `GlobalContext` and `ClientContext`, in that it provides a `Dictionary` that is globally available to all your code, in both your business objects and UI. However, the `LocalContext` object is not moved across the network by the data portal. This means that the client and application server both have their own separate `LocalContext` objects.

Framework Changes

Implementing `LocalContext` requires changes only to the `ApplicationContext` class, which is in the `Csla\DataPortal` folder.

Implementing LocalContext

Like `GlobalContext` and `ClientContext`, the `LocalContext` object is designed to be available to all code for the current user on the client or server, whether the code is executing in ASP.NET or not. Remember that most server environments are shared by many concurrent user requests, and so the context objects can not be stored at the `AppDomain` level. All values

stored at the AppDomain level are shared by all users of the AppDomain, which would be a problem on any application server.

This rules out the use of Shared fields or using the AppDomain object itself. The only safe way to share global data for a user is by putting it on the current Thread object. There is some complexity introduced by ASP.NET, because the data stored on the Thread object isn't guaranteed to be consistently available in that environment. When running in ASP.NET, this type of data should be stored in the current HttpContext object.

Changes to ApplicationContext

When implementing LocalContext, I followed the same basic technique used with GlobalContext and ClientContext as discussed in *Expert VB 2005 Business Objects* (ISBN 1590596315). The following code was added to ApplicationContext:

```
Private Const mLocalContextName As String = "Csla.LocalContext"

Public ReadOnly Property LocalContext() As HybridDictionary
  Get
    Dim ctx As HybridDictionary = GetLocalContext()
    If ctx Is Nothing Then
      ctx = New HybridDictionary
      SetLocalContext(ctx)
    End If
    Return ctx
  End Get
End Property

Private Function GetLocalContext() As HybridDictionary

  If HttpContext.Current Is Nothing Then
    Dim slot As System.LocalDataStoreSlot = _
      Thread.GetNamedDataSlot(mLocalContextName)
    Return CType(Thread.GetData(slot), HybridDictionary)

  Else
    Return CType(HttpContext.Current.Items(mLocalContextName), HybridDictionary)
  End If

End Function

Private Sub SetLocalContext(ByVal localContext As HybridDictionary)

  If HttpContext.Current Is Nothing Then
    Dim slot As System.LocalDataStoreSlot = _
      Thread.GetNamedDataSlot(mLocalContextName)
    Threading.Thread.SetData(slot, localContext)

  Else
    HttpContext.Current.Items(mLocalContextName) = localContext
  End If

End Sub
```

As you can see, the code detects whether it is running in ASP.NET or not based on whether HttpContext.Current is Nothing. When running in ASP.NET, the Dictionary is stored in the HtppContext. Otherwise, it is stored using thread local storage on the Thread object.

The result is transparent to anyone using LocalContext: the Dictionary is safely available to all code on the current thread regardless of whether the code is running in ASP.NET or not.

Because LocalContext doesn't get moved to or from the server by the data portal there's no need to worry about writing code to move the object across the network. However, if the data portal is running on an application server, it always calls ApplicationContext.Clear() as the data portal request completes, ensuring that one user's context data isn't accidentally made available to the next user on that server thread. This applies to LocalContext as well, and so the Clear() method must also clear the LocalContext value:

```
Public Sub Clear()

  SetContext(Nothing, Nothing)
  SetLocalContext(Nothing)

End Sub
```

By setting the value to Nothing, the code ensures that the next user on this thread will not have access to the previous user's context data.

Using the Enhancements

The LocalContext object is a Dictionary that can contain any values you wish to make globally available to your code. These values are not shared between the client and the application server (if you are using a remote data portal), they are local to the specific environment.

On the application server, LocalContext exists only for the duration of the current data portal call. When the current data portal call completes, the LocalContext object on the server is discarded.

In a Windows Forms client, the LocalContext exists as long as the client process is running. This means that the context data is available for the lifetime of the application. In a Web Forms client, the LocalContext exists for the duration of the current page request, typically a fraction of a second. If you need longer-lived context data in a Web Forms application you should use the ASP.NET Session object instead.

The primary motivation for adding LocalContext to CSLA .NET is to provide an easy way to share database connection or transaction objects across all your data access code.

While System.Transactions offers some important benefits over Enterprise Services in terms of performance, it still invokes the Distributed Transaction Coordinator (DTC) as soon as you open more than one database connection. This includes opening a second database connection to the same database, even with the same connection string. As soon as the DTC is invoked, you incur a substantial performance penalty of at least 15%; just like you do when using Enterprise Services transactions.

To avoid this issue, you must open *one* database connection and share it between all your objects as they interact with the database. Typically, you'll open this connection in your editable root object, and then pass the connection to all the child collections and objects of that root object.

While you can pass this connection object as a parameter to all your Friend Update(), Insert() and Delete() methods, it is sometimes simpler to just make the connection global, and this is the purpose behind LocalContext.

Using LocalContext

While you may find other uses for `LocalContext`, my motivation for adding it to CSLA .NET is to provide global access to a database connection object from an editable root object down through its child objects. That's what I'll demonstrate here.

In your editable root object you implement the standard `DataPortal_XYZ` methods. When implementing `DataPortal_Insert()` and `DataPortal_Update()`, you'll typically also call your child collections or objects and ask them to insert or update their own data. When using either `Manual` or `TransactionScope` transactions, you'll often want to pass your database connection object or transaction object from the parent object down to its child objects.

Using TransactionScope Transactions

When using `LocalContext` along with `TransactionScope` style transactions, your editable root data portal methods would look like this:

```
<Transactional(TransactionalTypes.TransactionScope)> _
Protected Overrides Sub DataPortal_Insert()

  Using cn As SqlConnection = New SqlConnection
    cn.Open()
    ApplicationContext.LocalContext("cn") = cn
    ' insert root object data here
    mChildren.Update()
    ApplicationContext.LocalContext.Remove("cn")
  End Using

End Sub

<Transactional(TransactionalTypes.TransactionScope)> _
Protected Overrides Sub DataPortal_Update()

  Using cn As SqlConnection = New SqlConnection
    cn.Open()
    ApplicationContext.LocalContext("cn") = cn
    ' update root object data here
    mChildren.Update()
    ApplicationContext.LocalContext.Remove("cn")
  End Using

End Sub
```

The highlighted lines of code show the use of `LocalContext` to make the connection object globally available. Notice that before leaving the `Using` block, I remove the connection object from `LocalContext` so it doesn't accidentally get used after it has been disposed, as that would result in an exception.

Following this pattern, the child objects updated by the call to `mChildren.Update()` method call can simply reuse the existing connection object. For example, the `Insert()` and `Update()` methods in a child object would look like this:

```
Friend Sub Insert()

  Dim cn As SqlConnection = _
    DirectCast(ApplicationContext.LocalContext("cn"), SqlConnection)
  ' insert the child data using the connection

End Sub

Friend Sub Update()
```

```
    Dim cn As SqlConnection = _
      DirectCast(ApplicationContext.LocalContext("cn"), SqlConnection)
    ' update the child data using the connection

  End Sub
```

The highlighted lines of code show how the existing connection object is retrieved from `LocalContext` so it can be used by your code in the methods.

By reusing the same connection object to insert and update all child objects, you avoid opening more than one connection. Using this technique, `System.Transactions` won't invoke the DTC and so you'll get much better performance, without much increase in complexity.

Using Manual Transactions

If you are using `Manual` transactions you'll typically be creating and using your own database transaction object. In that case, your editable root methods would look like this:

```
  <Transactional(TransactionalTypes.TransactionScope)> _
  Protected Overrides Sub DataPortal_Insert()

    Using cn As SqlConnection = New SqlConnection
      cn.Open()
      Using tr As SqlTransaction = cn.BeginTransaction
        ApplicationContext.LocalContext("tr") = tr
        ' insert root object data here
        mChildren.Update()
        ApplicationContext.LocalContext.Remove("tr")
        tr.Commit()
      End Using
    End Using

  End Sub

  <Transactional(TransactionalTypes.TransactionScope)> _
  Protected Overrides Sub DataPortal_Update()

    Using cn As SqlConnection = New SqlConnection
      cn.Open()
      Using tr As SqlTransaction = cn.BeginTransaction
        ApplicationContext.LocalContext("tr") = tr
        ' update root object data here
        mChildren.Update()
        ApplicationContext.LocalContext.Remove("tr")
        tr.Commit()
      End Using
    End Using

  End Sub
```

Notice that it is the `SqlTransaction` object which is made global, not the connection object. The reason for this is that the connection object is a property of the transaction object, and so making the transaction object globally available also makes the connection globally available.

You can implement the `Insert()` and `Update()` methods in your child objects like this:

```
  Friend Sub Insert()

    Dim tr As SqlTransaction = _
      DirectCast(ApplicationContext.LocalContext("tr"), SqlTransaction)
    Dim cn As SqlConnection = tr.Connection
    ' insert the child data using the connection and transaction
```

```
  End Sub

  Friend Sub Update()

    Dim tr As SqlTransaction = _
      DirectCast(ApplicationContext.LocalContext("tr"), SqlTransaction)
    Dim cn As SqlConnection = tr.Connection
    ' update the child data using the connection and transaction

  End Sub
```

Notice how the `SqlTransaction` object is retrieved from `LocalContext`, and then the `SqlConnection` object is retrieved from the transaction object. The result is that you have easy access to both objects and can use them to set up your `SqlCommand` object to implement the insert or update operation.

Whether using `TransactionScope` or `Manual` transactions, `LocalContext` provides an easy way to share the connection or transaction objects between the root object and its child objects.

Data Portal

The data portal is one of the most complex parts of the CSLA .NET framework. It enables the concept of mobile objects, and acts as a channel adapter, hiding the underlying network technologies (if any) used to communicate with the "server-side" data access code.

One of the more interesting features of the data portal is that it allows client-side code to make a call like this:

```
Return DataPortal.Fetch(Of Person)(New Criteria(id))
```

And that line of code results in the creation of an empty `Person` object on the server, and the following method on that object is invoked by the data portal:

```
Private Overloads Sub DataPortal_Fetch(ByVal criteria As Criteria)

End Sub
```

Essentially, the `DataPortal.Fetch()` call is a call to `DataPortal_Fetch()`. Even the parameter value is passed through from the client to server code. Notice the use of strongly typed parameters through the process. This was a key addition to the data portal in CSLA .NET 2.0.

Unfortunately, when I implemented CSLA .NET 2.0, I didn't completely emulate normal calling semantics. Specifically, those for the passing of no parameter value at all. Worse, I allowed the use of no parameter for `DataPortal.Create()`, but not for `DataPortal_Fetch()`. This caused some confusion due to the inconsistency. Table 10 shows the calling patterns used in version 2.0.

Client-side	Server-side
`DataPortal.Create(Of Person)()`	`DataPortal_Create( _` `  ByVal criteria As Object)`
`DataPortal.Create(Of Person)(Nothing)`	`DataPortal_Create( _` `  ByVal criteria As Object)`
`DataPortal.Create(Of Person) _` `  (New Criteria())`	`DataPortal_Create( _` `  ByVal criteria As Criteria)`
`DataPortal.Fetch(Of Person)(Nothing)`	`DataPortal_Fetch( _` `  ByVal criteria As Object)`
`DataPortal.Fetch(Of Person) _` `  (New Criteria())`	`DataPortal_Fetch( _` `  ByVal criteria As Criteria)`

Table 10. Data portal method calling semantics in version 2.0

Notice that the `Create()` method allows no parameter, while the `Fetch()` method does not. Worse, `Create()` with no parameter calls the same method as `Create()` with a parameter of `Nothing`. In versions 2.0.1 and higher I tried various solutions, but ultimately realized that the only correct answer was to properly emulate normal .NET calling conventions.

In CSLA .NET version 2.1, the calling patterns follow normal conventions as shown in Table 11.

Client-side	Server-side
`DataPortal.Create(Of Person)()`	`DataPortal_Create()`
`DataPortal.Create(Of Person)(Nothing)`	`DataPortal_Create( _` `  ByVal criteria As Object)`
`DataPortal.Create(Of Person) _` `  (New Criteria())`	`DataPortal_Create( _` `  ByVal criteria As Criteria)`
`DataPortal.Fetch(Of Person)()`	`DataPortal_Fetch()`
`DataPortal.Fetch(Of Person)(Nothing)`	`DataPortal_Fetch( _` `  ByVal criteria As Object)`
`DataPortal.Fetch(Of Person) _` `  (New Criteria())`	`DataPortal_Fetch( _` `  ByVal criteria As Criteria)`

Table 11. Data portal method calling semantics in version 2.1

Notice that there's now parity between `Create()` and `Fetch()`.Also notice that when the client-side code passes no parameter, the server-side `DataPortal_XYZ` method accepts no parameter. This follows the normal method calling semantics you'd expect and makes the implementation of these methods more intuitive.

Framework Changes

Changing the data portal is always challenging, because one of my primary goals whenever changing CSLA .NET is to preserve backward compatibility as much as possible. In the case of the data portal, this not only means trying to not break business object implementations of factory methods and `DataPortal_XYZ` methods, but also I don't want to break any custom data portal channels people are using to communicate with their application servers.

Obviously, changing the calling semantics of the `DataPortal_XYZ` methods must have some impact on business object implementations, and that's unavoidable. After consulting with the participants of the CSLA .NET discussion forum at http://forums.lhotka.net, I decided to make the breaking change at this time, so as to avoid continued confusion going into the future.

However, I was able to avoid making changes to the `IDataPortalProxy` and `IDataPortalServer` interfaces. This should mean that existing data portal channels are unaffected by these changes. The drawback to this approach is that my implementation is not as elegant as I would prefer, and so I'm choosing to lose some elegance to gain some backward compatibility.

It is important to recognize that `IDataPortalServer` *did change* in version 2.0.2. Specifically, the method signature for `Fetch()` was changed to include a parameter explicitly indicating the type of the business object to be retrieved. This change made `Fetch()` more closely mirror `Create()` in this regard, and allows the `Fetch()` method to be called with no criteria parameter.

Implementing the Data Portal Changes

A number of classes needed to be changed to support the new data portal functionality, including:

- `MethodCaller`
- `Client\DataPortal`
- `Server\DataPortal`
- `Server\SimpleDataPortal`

The `MethodCaller` class contains the utility methods that find and invoke the appropriate methods, and so the bulk of the changes occurred to the code in that class. The other three classes have less significant changes, adapting to the new methods implemented in `MethodCaller`.

Changes to MethodCaller

The `MethodCaller` class now has four methods to support the calling semantics of the data portal methods:

- `FindMethod()`
- `GetMethod()`
- `GetCreateMethod()`
- `GetFetchMethod()`

Let's look at each method.

FindMethod

There are two overloads of the `FindMethod()` method, each responsible for finding a method matching a set of criteria. One looks for a method with a specific name and a specific set of parameter types. The other is less restrictive, looking for a specific name and the right *number* of parameters. These two methods are used by the `GetMethod()` methods in `MethodCaller` as they locate the method requested by `GetCreateMethod()` and `GetFetchMethod()`.

FindMethod with Matching Parameters

The first `FindMethod()` method is responsible for finding a method matching the supplied method name that also accepts parameters of the correct types. The reflection support in .NET already does most of the work in this regard, but inheritance makes things slightly more complex.

The reason is, that the same method can be implemented by multiple classes in an inheritance hierarchy. When making the `GetMethod()` reflection call in such a case, you can get an exception indicating there's an ambiguous result. This can be avoided by restricting the reflection call to only look at a specific type. Then you can loop up through the inheritance hierarchy, examining each type in turn until you reach the top of the hierarchy:

```
Public Function FindMethod( _
  ByVal objType As Type, _
  ByVal method As String, _
  ByVal types As Type()) As MethodInfo

  Dim info As MethodInfo = Nothing
  Do
```

```
    ' find for a strongly typed match
    info = objType.GetMethod(method, oneLevelFlags, Nothing, types, Nothing)
    If info IsNot Nothing Then
      Exit Do ' match found
    End If

    objType = objType.BaseType
  Loop While objType IsNot Nothing

  Return info

End Function
```

The `oneLevelFlags` field used in the `GetMethod()` call looks like this:

```
Private Const oneLevelFlags As BindingFlags = _
    BindingFlags.DeclaredOnly Or _
    BindingFlags.Instance Or _
    BindingFlags.Public Or _
    BindingFlags.NonPublic
```

This set of flags restricts the reflection call so it returns only instance methods declared directly by the specified type. If a matching method is located, it is returned. If not, the result is `Nothing`; to indicate that no match was found.

FindMethod with Matching Parameter Count

The second `FindMethod()` method is responsible for finding a method matching the supplied method name and number of parameters. It does not look at the parameter types, just the number of parameters, and it returns the first match it finds:

```
Public Function FindMethod( _
  ByVal objType As Type, _
  ByVal method As String, _
  ByVal parameterCount As Integer) As MethodInfo

  ' walk up the inheritance hierarchy looking
  ' for a method with the right number of
  ' parameters
  Dim result As MethodInfo = Nothing
  Dim currentType As Type = objType
  Do
    Dim info As MethodInfo = _
      currentType.GetMethod(method, oneLevelFlags)
    If info IsNot Nothing Then
      If info.GetParameters.Length = parameterCount Then
        ' got a match so use it
        result = info
        Exit Do
      End If
    End If
    currentType = currentType.BaseType
  Loop Until currentType Is Nothing

  Return result

End Function
```

Remember that the same method could be implemented in multiple classes in an inheritance hierarchy. Due to this, `FindMethod()` starts at the end of the hierarchy and works its way back toward the top looking for a match. This way, it will return any overridden versions of a method first, which is the behavior you'd expect when using inheritance.

In the end, this method will return the first matching method it finds, or `Nothing` if no matching method can be found.

GetMethod

The `GetMethod()` method was part of CSLA .NET 2.0. It has been altered in version 2.1 to get methods that have no parameters, as well as those that have parameters. It also does a better job of falling back to find matching methods in the case that the parameter types can't be matched:

```
  Public Function GetMethod(ByVal objectType As Type, _
    ByVal method As String, ByVal ParamArray parameters() As Object) _
    As MethodInfo

    Dim result As MethodInfo = Nothing

    ' put all param types into an array of Type
    Dim types As New List(Of Type)
    For Each item As Object In parameters
      If item Is Nothing Then
        types.Add(GetType(Object))

      Else
        types.Add(item.GetType)
      End If
    Next

    ' first see if there's a matching method
    ' where all params match types
    result = FindMethod(objectType, method, types.ToArray)

    If result Is Nothing Then
      ' no match found - so look for any method
      ' with the right number of parameters
      result = FindMethod(objectType, method, parameters.Length)
    End If

    ' no strongly typed match found, get default
    If result Is Nothing Then
      Try
        result = objectType.GetMethod(method, allLevelFlags)

      Catch ex As AmbiguousMatchException
        Dim methods() As MethodInfo = objectType.GetMethods
        For Each m As MethodInfo In methods
          If m.Name = method AndAlso _
          m.GetParameters.Length = parameters.Length Then
            result = m
            Exit For
          End If
        Next
        If result Is Nothing Then
          Throw
        End If
      End Try
    End If

    Return result

  End Function
```

Notice the use of both `FindMethod()` overloads. First, an attempt is made to find a matching method that has the exact right parameter types:

```
    result = FindMethod(objectType, method, types.ToArray)
```

If that fails, an attempt is made to find a method with the right number of parameters, even if the types don't exactly match:

```
      result = FindMethod(objectType, method, parameters.Length)
```

This second attempt will catch cases where a parameter's type is a subclass of the method's type, or where the parameter is `Nothing` or of type `Object` and the method expects a strongly typed parameter. It will also find route a strongly typed parameter value to a method expecting parameters of type `Object`, such as the default `DataPortal_XYZ` methods implemented by the CSLA .NET base classes.

This last point is important for backward compatibility. Though it is preferable to implement the `DataPortal_XYZ` methods to accept a strongly typed parameter, older code may still be overriding the default methods from the base classes, and accepting a parameter of type `Object`. This second call to `FindMethod()` handles that common case, and ensures that existing business object code will continue to function as expected.

If both those attempts fail then a direct reflection call is made to try and find any matching method by name. This part of the code is unchanged from version 2.0.

GetCreateMethod

The `GetCreateMethod()` method is responsible for locating and returning the appropriate `DataPortal_Create()` method, based on any parameters passed to `DataPortal.Create()`. It is a straightforward method, because it is able to leverage the work done by the two `FindMethod()` overloads.

The only bit of complexity comes into play because I opted not to change the `IDataPortalProxy` and `IDataPortalServer` interfaces. Both of these interfaces require that *some* parameter be passed to the create method. Since `Nothing` is a valid option, I needed some other value that could act as a placeholder for "no parameter".

The valid parameter options for the create call are `Nothing` or a criteria object. A criteria object can be either a nested class within a business class, or a class that inherits from `CriteriaBase`. Either way, the criteria object must be a reference type. This means that any value type, such as an `Integer`, can't be passed to the `DataPortal.Create()` call.

Since `Integer` can't be passed as a valid parameter, I can use it as a placeholder to represent the "no parameter" option:

```
Public Function GetCreateMethod( _
  ByVal objectType As Type, ByVal criteria As Object) As MethodInfo

  Dim method As MethodInfo
  If TypeOf criteria Is Integer Then
    ' an "Integer" criteria is a special flag indicating
    ' that criteria is empty and should not be used
    method = MethodCaller.GetMethod(objectType, "DataPortal_Create")

  Else
    method = MethodCaller.GetMethod(objectType, "DataPortal_Create", criteria)
  End If
  Return method

End Function
```

This code first checks to see if an `Integer` was passed as a parameter, which would indicate that no parameter was actually passed to the `DataPortal.Create()` call. In that case, `GetMethod()` is called with no parameter array, indicating that there are no parameters for the `DataPortal_Create()` method.

Otherwise `GetMethod()` is called, passing the supplied criteria object (or `Nothing`) as a parameter for `DataPortal_Create()`.

GetFetchMethod

The `GetFetchMethod()` method follows the same scheme as `GetCreateMethod()`:

```
Public Function GetFetchMethod( _
  ByVal objectType As Type, ByVal criteria As Object) As MethodInfo

  Dim method As MethodInfo
  If TypeOf criteria Is Integer Then
    ' an "Integer" criteria is a special flag indicating
    ' that criteria is empty and should not be used
    method = MethodCaller.GetMethod(objectType, "DataPortal_Fetch")

  Else
    method = MethodCaller.GetMethod(objectType, "DataPortal_Fetch", criteria)
  End If
  Return method

End Function
```

Again, the appropriate parameters, if any, are passed to the `GetMethod()` call based on the type of the criteria parameter. The `Integer` type is a special placeholder indicating the “no parameter” case.

Changes to Client\DataPortal

The client-side `DataPortal` class has been altered to make use of the new `GetCreateMethod()` and `GetFetchMethod()` functionality in the `MethodCaller` class. A constant value is used to indicate the special “no parameter” placeholder passed as a faux criteria value to the `DataPortal.Create()` and `DataPortal.Fetch()` methods:

```
Private Const EmptyCriteria As Integer = 1
```

The overload of `Create()` that accepts no parameters now uses this constant:

```
Public Function Create(Of T)() As T
  Return DirectCast(Create(GetType(T), EmptyCriteria), T)
End Function
```

Of course this method just delegates to another `Create()` method, which has been altered to use the new `GetCreateMethod()` functionality:

```
Private Function Create( _
  ByVal objectType As Type, ByVal criteria As Object) As Object

  Dim result As Server.DataPortalResult

  Dim method As MethodInfo = MethodCaller.GetCreateMethod(objectType, criteria)

  Dim proxy As DataPortalClient.IDataPortalProxy
  proxy = GetDataPortalProxy(RunLocal(method))
```

```
    Dim dpContext As New Server.DataPortalContext( _
      GetPrincipal, proxy.IsServerRemote)

    OnDataPortalInvoke(New DataPortalEventArgs(dpContext))

    Try
      result = proxy.Create(objectType, criteria, dpContext)

    Catch ex As Server.DataPortalException
      result = ex.Result
      If proxy.IsServerRemote Then
        ApplicationContext.SetGlobalContext(result.GlobalContext)
      End If
      Throw New DataPortalException( _
        String.Format("DataPortal.Create {0} ({1})", _
          My.Resources.Failed, ex.InnerException.InnerException), _
        ex.InnerException, result.ReturnObject)
    End Try

    If proxy.IsServerRemote Then
      ApplicationContext.SetGlobalContext(result.GlobalContext)
    End If

    OnDataPortalInvokeComplete(New DataPortalEventArgs(dpContext))

    Return result.ReturnObject

  End Function
```

I've also highlighted a change to the exception handling. In case of an exception, this code throws a `DataPortalException`, which now includes the message text of the original exception as part of the `DataPortalException` object's message. This simplifies the typical debugging scenario, because the original exception message is immediately visible to the developer.

The same basic changes have been applied to the `Fetch()` methods as well. There's a new `Fetch()` method overload that accepts no parameter:

```
  Public Function Fetch(Of T)() As T

    Return DirectCast(Fetch(GetType(T), EmptyCriteria), T)

  End Function
```

It uses the `EmptyCriteria` constant value, just like the `Create()` equivalent, and delegates the call to another `Fetch()` overload:

```
  Private Function Fetch( _
    ByVal objectType As Type, ByVal criteria As Object) As Object

    Dim result As Server.DataPortalResult

    Dim method As MethodInfo = MethodCaller.GetFetchMethod(objectType, criteria)

    Dim proxy As DataPortalClient.IDataPortalProxy
    proxy = GetDataPortalProxy(RunLocal(method))

    Dim dpContext As New Server.DataPortalContext( _
      GetPrincipal, proxy.IsServerRemote)

    OnDataPortalInvoke(New DataPortalEventArgs(dpContext))

    Try
      result = proxy.Fetch(objectType, criteria, dpContext)
```

```
    Catch ex As Server.DataPortalException
      result = ex.Result
      If proxy.IsServerRemote Then
        ApplicationContext.SetGlobalContext(result.GlobalContext)
      End If
      Throw New DataPortalException( _
        String.Format("DataPortal.Fetch {0} ({1})", _
          My.Resources.Failed, ex.InnerException.InnerException), _
        ex.InnerException, result.ReturnObject)
    End Try

    If proxy.IsServerRemote Then
      ApplicationContext.SetGlobalContext(result.GlobalContext)
    End If

    OnDataPortalInvokeComplete(New DataPortalEventArgs(dpContext))

    Return result.ReturnObject

  End Function
```

This `Fetch()` method has been changed to use the new `GetFetchMethod()` functionality, and to throw a more detailed `DataPortalException` object.

Changes to Server\DataPortal

The server-side `DataPortal` class has also been changed to use the new `GetCreateMethod()` and `GetFetchMethod()` functionality. In the `Create()` method the call looks like this:

```
      Dim method As MethodInfo = MethodCaller.GetCreateMethod(objectType, criteria)
```

And in the `Fetch()` method the call looks like this:

```
      Dim method As MethodInfo = MethodCaller.GetFetchMethod(objectType, criteria)
```

The server-side `DataPortal` class doesn't invoke the `DataPortal_XYZ` methods, but it does need access to the `MethodInfo` object so it can determine whether to route the method call through Enterprise Services or a `TransactionScope` based on the `Transactional` attribute applied to the method.

Ultimately the method call is relayed to `SimpleDataPortal`, which does invoke the business object method.

Changes to Server\SimpleDataPortal

The `SimpleDataPortal` class contains the code that directly interacts with the business object. In the case of `Create()` and `Fetch()` operations, `SimpleDataPortal` creates an instance of the business object before calling the appropriate `DataPortal_XYZ` method on the object. This means both the `Create()` and `Fetch()` methods require updates. Here's the `Create()` method:

```
    Public Function Create( _
      ByVal objectType As System.Type, _
      ByVal criteria As Object, _
      ByVal context As Server.DataPortalContext) As Server.DataPortalResult _
      Implements Server.IDataPortalServer.Create

      Dim obj As Object = Nothing

      Try
```

```
        ' create an instance of the business object
        obj = Activator.CreateInstance(objectType, True)

        ' tell the business object we're about to make a DataPortal_xyz call
        MethodCaller.CallMethodIfImplemented( _
          obj, "DataPortal_OnDataPortalInvoke", _
          New DataPortalEventArgs(context))

        ' tell the business object to fetch its data
        Dim method As MethodInfo = MethodCaller.GetCreateMethod(objectType, criteria)
        If TypeOf criteria Is Integer Then
          ' an "Integer" criteria is a special flag indicating
          ' that criteria is empty and should not be used
          MethodCaller.CallMethod(obj, method)

        Else
          MethodCaller.CallMethod(obj, method, criteria)
        End If

        ' mark the object as new
        MethodCaller.CallMethodIfImplemented(obj, "MarkNew")

        ' tell the business object the DataPortal_xyz call is complete
        MethodCaller.CallMethodIfImplemented( _
          obj, "DataPortal_OnDataPortalInvokeComplete", _
          New DataPortalEventArgs(context))

        ' return the populated business object as a result
        Return New DataPortalResult(obj)

      Catch ex As Exception
        Try
          ' tell the business object there was an exception
          MethodCaller.CallMethodIfImplemented( _
            obj, "DataPortal_OnDataPortalException", _
            New DataPortalEventArgs(context), ex)
        Catch
          ' ignore exceptions from the exception handler
        End Try
        Throw New DataPortalException("DataPortal.Create " & _
          My.Resources.FailedOnServer, ex, New DataPortalResult(obj))
      End Try

    End Function
```

Notice that not only is `GetCreateMethod()` used to get the appropriate method to invoke, but the code also checks to see if the criteria parameter is of type `Integer`. If it is an `Integer`, that indicates that no *actual* parameter was passed to the `Create()` call, and so `CallMethod()` is invoked without passing any parameters:

```
          MethodCaller.CallMethod(obj, method)
```

On the other hand, if the `criteria` parameter is of any other type, then it is either `Nothing` or a valid criteria object, and so the value is passed as a parameter to `CallMethod()`:

```
          MethodCaller.CallMethod(obj, method, criteria)
```

The end result is that the method calling semantics for `Create()` and `Fetch()` are the same, and they also conform to the list shown earlier in Table 11.

Using the Enhancements

The data portal enhancements bring the calling semantics of the `DataPortal.Create()` and `DataPortal.Fetch()` methods into line with common .NET usage. Table 12 is a complete list illustrating what `DataPortal_XYZ` method is called based on how each `DataPortal` method is called in your factory methods.

Client-side	Server-side
`DataPortal.Create(Of Person)()`	`DataPortal_Create()`
`DataPortal.Create(Of Person)(Nothing)`	`DataPortal_Create( _` `  ByVal criteria As Object)`

`DataPortal.Create(Of Person) _`
`  (New Criteria())`
 	
Or
 	
`DataPortal.Create(New Criteria())`
	`DataPortal_Create( _`
	`  ByVal criteria As Criteria)`
	
	Falls back to
	
	`DataPortal_Create( _`
	`  ByVal criteria As Object)`

`DataPortal.Fetch(Of Person)()`	`DataPortal_Fetch()`
`DataPortal.Fetch(Of Person)(Nothing)`	`DataPortal_Fetch( _` `  ByVal criteria As Object)`

`DataPortal.Fetch(Of Person) _`
`  (New Criteria())`
 	
Or
 	
`DataPortal.Create(New Criteria())`
	`DataPortal_Fetch( _`
	`  ByVal criteria As Criteria)`
	
	Falls back to
	
	`DataPortal_Fetch( _`
	`  ByVal criteria As Object)`

`DataPortal.Update(Of Person)()`
 	
Or
 	
`DataPortal.Update()`
	`DataPortal_Insert()`
	
	Or
	
	`DataPortal_Update()`
	
	Or
	
	`DataPortal_DeleteSelf()`

Client-side	Server-side

`DataPortal.Delete(Of Person)()`
 	
Or
 	
`DataPortal.Delete()`
	`DataPortal_Delete( _`
	`  ByVal criteria As Criteria)`
	
	Falls back to
	
	`DataPortal_Delete( _`
	`  ByVal criteria As Object)`

`DataPortal.Execute(Of Person)()`
 	
Or
 	
`DataPortal.Execute()`
	`Sub DataPortal_Execute()`

Table 12. Data portal method calling cross-reference

The basic usage of the data portal doesn't change, and there is little on most business object implementations due to the changes made in version 2.1. However, if you were calling `DataPortal.Create()` with no parameters, then your code will be impacted by these changes.

More importantly, you should now feel comfortable calling either `DataPortal.Create()` or `DataPortal.Fetch()` with various parameter types based on the information in Table 12.

SmartDate

The `SmartDate` class has a number of enhancements designed to provide better ease of use for the type. These enhancements include:

- A `Shared` method allowing you to set the default format string for all new `SmartDate` values
- An enumerated value to more clearly indicate whether an empty `SmartDate` is the largest or smallest possible date
- A new overload of `ToString()` to better match the functionality of the `DateTime` type

While none of these enhancements are major changes, they increase the usability of the `SmartDate` type and bring it more in line with the functionality provided by the `DateTime` type.

Framework Changes

The enhancements to `SmartDate` involve adding a new type to CSLA .NET:

- `EmptyValue`

And they involve changes to one class:

- `SmartDate`

Implementing the Changes

I'll explain the new `EmptyValue` enumerated type first, and then walk through the changes to `SmartDate` itself.

EmptyValue Type

The `EmptyValue` enumerated type is nested within the `SmartDate` class itself. The reason for this is that `EmptyValue` is designed only for use by `SmartDate`, and making it a nested type helps keep the main `Csla` namespace organized.

The enumerated type itself is not complex:

```
Public Enum EmptyValue
  MinDate
  MaxDate
End Enum
```

When a `SmartDate` value is created, it can treat an empty value as either the largest or smallest possible date (for comparison purposes). In CSLA .NET 2.0, this was indicated using a Boolean value. This resulted in code that was difficult to read. These enumerated values offer a more readable alternative. For example:

```
Dim sm As New SmartDate(EmptyValue.MinDate)
```

This clearly creates a new `SmartDate` value, where an empty value is the smallest possible date. The older approach is still supported for backward compatibility:

```
Dim sm As New SmartDate(True)
```

Obviously, the intent of this code is far less clear, though the result is the same.

Changes to SmartDate

The `SmartDate` class itself is changed to make use of the new `EmptyValue` type, and for the addition of the new `Shared` property to control the default format string and new `ToString()` overload.

Supporting EmptyValue

The `SmartDate` type used to use a `Boolean` value to determine whether an empty value was the smallest or largest possible date. For clarity within `SmartDate` itself, the code now uses the `EmptyValue` type instead, so the instance field is now of this type:

```
  Private mEmptyValue As EmptyValue
```

There are also new constructors that accept this type. For example:

```
  Public Sub New(ByVal emptyValue As EmptyValue)
    mEmptyValue = emptyValue
    SetEmptyDate(mEmptyValue)
  End Sub
```

Perhaps more importantly, the existing constructors that accept `Boolean` values have been altered to translate those values into an `EmptyValue` type. This is done using a helper method:

```
  Private Shared Function GetEmptyValue(ByVal emptyIsMin As Boolean) As EmptyValue
    If emptyIsMin Then
      Return EmptyValue.MinDate
    Else
      Return EmptyValue.MaxDate
    End If
  End Function
```

The older constructors are retained for backward compatibility, allowing existing CSLA .NET business object code to upgrade seamlessly to version 2.1 in this regard. However, those constructors now use the `GetEmptyValue()` helper method to translate their parameter value. For example:

```
  Public Sub New(ByVal value As Date, ByVal emptyIsMin As Boolean)
    mEmptyValue = GetEmptyValue(emptyIsMin)
    Me.Date = value
  End Sub
```

Throughout the code, anywhere the old `Boolean` field was used, the new `EmptyType` field is used in its place. There are several places where the behavior of a method is controlled by this field, for instance:

```
  Public ReadOnly Property IsEmpty() As Boolean
    Get
      If mEmptyValue = EmptyValue.MinDate Then
```

```
        Return Me.Date.Equals(Date.MinValue)
      Else
        Return Me.Date.Equals(Date.MaxValue)
      End If
    End Get
  End Property
```

While the functionality remains the same, this code is more clear and easier to read than in version 2.0. These changes are mechanical, and so I'm not going to go through each case. You can look at the `SmartDate` code to see how the `EmptyValue` type has been used consistently to replace the previous `Boolean` field.

Default Format String

In version 2.0, `SmartDate` had a hard-coded default format string of `d`, which is the short date format. While you could change this value for each `SmartDate` you created, there was no way to globally change the default. In version 2.1, you can now change the default format string using a `Shared` method: `SetDefaultFormatString()`.

The default format string value is stored in a `Shared` field:

```
  Private Shared mDefaultFormat As String
```

In the `Shared` constructor its value is set to `d`, the same default value as was used in CSLA .NET version 2.0:

```
  Shared Sub New()
    mDefaultFormat = "d"
  End Sub
```

This preserves backward compatibility with previous version of CSLA .NET. However, there's now a way for a developer to globally change the default format string that will be used by `SmartDate` values:

```
  Public Shared Sub SetDefaultFormatString(ByVal formatString As String)
    mDefaultFormat = formatString
  End Sub
```

Finally, the existing `FormatString()` property has been enhanced to use the default value:

```
  Public Property FormatString() As String
    Get
      If mFormat Is Nothing Then
        mFormat = mDefaultFormat
      End If
      Return mFormat
    End Get
    Set(ByVal value As String)
      mFormat = value
    End Set
  End Property
```

Any request for the format string first checks to see if the value for this particular `SmartDate` has been set. If not, then `mFormat` will be `Nothing` and the default value is used. Otherwise, the existing value of `mFormat` is used.

This approach means that you can still change the format string for individual `SmartDate` values. Otherwise you'll get the default of `d`, or whatever you've set using the `SetDefaultFormatString()` method.

Overload of ToString()

`SmartDate` has been enhanced with a new overload of `ToString()` to provide more consistency with the `Date` and `DateTime` types. The new overload allows you to specify a format string to use in converting the value to text:

```
Public Overloads Function ToString(ByVal format As String) As String

  Return DateToString(Me.Date, format, mEmptyValue)

End Function
```

This overload ignores the `FormatString` property value and uses the value provided as a parameter instead.

While the enhancements to `SmartDate` are relatively minor in terms of changes to CSLA .NET, they provide substantial benefits to developers using the type.

Using the Enhancements

The enhancements to `SmartDate` provide more clarity to your code, and offer more control over how `SmartDate` values are translated into text.

Using the SmartDate Enhancements

I'll walk through each new feature in turn.

Using the New Constructors

You can now create `SmartDate` values using the `EmptyValue` type. Table 13 shows the possible constructors and their results.

Constructor	Result
`sm = New SmartDate()`	Creates an empty `SmartDate` value where an empty value is the smallest possible date.
`sm = New SmartDate( _` `  SmartDate.EmptyValue.MinDate)`	Creates an empty `SmartDate` value where an empty value is the smallest possible date.
`sm = New SmartDate(Today)`	Creates a `SmartDate` value with today's date, where an empty value is the smallest possible date.

`sm = New SmartDate( _`
`  Today, _`
`  SmartDate.EmptyValue.MinDate)`
	Creates a `SmartDate` value with today's date, where an empty value is the smallest possible date.

`sm = New SmartDate("1/1/2007")`	Creates a `SmartDate` value for January 1, 2007, where an empty value is the smallest possible date.

`sm = New SmartDate( _`
`  "1/1/2007", _`
`  SmartDate.EmptyValue.MinDate)`
	Creates a `SmartDate` value for January 1, 2007, where an empty value is the smallest possible date.

Table 13. Using the new SmartDate constructors

The older `Boolean` constructors continue to function, but the new constructors provide better clarity for your code.

Using the Default Format String

The default format string for a `SmartDate` is `d`, which is the short date format. You can change the format string for individual `SmartDate` values using the `FormatString` property:

```
Dim sm As SmartDate
sm.FormatString = "D"
```

What is new in version 2.1, is that you can now specify a different default format string so you don't need to change the value on all individual `SmartDate` fields. To do this, you use the new `SetDefaultFormatString()` method:

```
SmartDate.SetDefaultFormatString("D")
```

Any `SmartDate` values created after this point will use this new format string value.

> **Note:** The default format string is stored as a `Shared` field, which means it exists at the AppDomain level. In an ASP.NET environment, this means that the default format string is shared by all users of your virtual root.

Typically, you'll set the default format string as your application starts up, so all `SmartDate` values have the same default.

Using the ToString() Overload

The final new feature of the `SmartDate` type is a new `ToString()` overload that allows you to control the format string used by that particular method call. To use this, pass a format string to the `ToString()` method:

```
Dim sm As New SmartDate(Today)

Dim output As String = sm.ToString("D")
```

The value of the `SmartDate` value's `FormatString` property is ignored in this case, and the format string you pass as a parameter is used instead.

CslaDataSource

The `CslaDataSource` web data control has been changed in a couple different ways since CSLA .NET version 2.0. First, it has been enhanced in an effort to allow the control to reload your business assemblies as they change during development, so you can refresh the schema information without having to exit and reload Visual Studio. Second, the control can now indicate that you are supporting the paging and sorting features of a collection through your code, so ASP.NET data binding acts properly.

In version 2.0, the `CslaDataSource` control used simple reflection to get schema information about the shape of your business objects. This schema information is returned to ASP.NET data binding so the Visual Studio designers can properly display rich content in grid and list controls.

Unfortunately, once an assembly has been loaded into memory, it can't be unloaded from that AppDomain. Visual Studio only has one AppDomain for the web page designers, so after your business assembly was loaded the first time to get the schema data, it couldn't be reloaded as your assembly changed.

To address this, `CslaDataSource` now loads your assembly into a temporary AppDomain, gets the schema information, then discards that temporary AppDomain entirely. Using this technique it is possible to refresh the schema information about your objects as your business assembly changes; without having to reload Visual Studio itself.

CSLA .NET now includes the `IReportTotalRowCount` interface, as discussed earlier in this book, so your collection classes can be implemented to support the concept of paging. While your collection may only load a subset of the total data available, you can report the total *possible* number of rows of data through `IReportTotalRowCount`.

The `CslaDataSource` control has been enhanced to understand and use the `IReportTotalRowCount` interface, and to expose a property so you can indicate whether your underlying collection class will be implementing this interface. This property value is then returned to the Web Forms designer to indicate whether your data source supports paging.

Similarly, a property has been added to `CslaDataSource` so you can indicate to the Web Forms designer whether your underlying collection will support sorting. Remember that ASP.NET data binding doesn't fully automate the sorting process, so by setting this property to `True` you also agree to write some extra code to trigger the sorting itself. More importantly, however, you are agreeing that your underlying collection supports sorting; either because it is of type `SortedBindingList`, or because you can re-fetch the collection using different sort criteria as required.

Framework Changes

The changes to `CslaDataSource` can be grouped into three functional areas:

- Supporting dynamic schema refresh
- Supporting paging
- Supporting sorting

Let's discuss each functional area.

Implementing Dynamic Schema Refresh

Dynamically refreshing the schema, or shape, of the data sources is particularly difficult when those data sources are business objects from a business assembly. The reason is twofold: the business assembly must be loaded into a temporary AppDomain so it can be later unloaded, and there's no publicly available API in .NET you can use to *find* the business assembly that you should load.

Implementing dynamic schema refresh required changing the following files:

- `CslaDesignerDataSourceView`
- `ObjectViewSchema`

And adding one class:

- `TypeLoader`

As you'll see, the `TypeLoader` class does the majority of the work. It is a relatively complex class, because it includes `Shared` methods that execute in the main Visual Studio AppDomain, and instance methods that execute in the temporary AppDomain created to load the business assembly.

The important thing to remember in all this, is that the only code that can safely interact with your business assembly is contained in the instance methods of `TypeLoader`. Absolutely no code in the rest of `CslaDataSource` or its related classes can directly interact with your business assembly without loading that assembly into the Visual Studio AppDomain and thus always running against an old version of your assembly.

Changes to CslaDesignerDataSourceView

The `CslaDesignerDataSourceView` class has been altered to make use of the `TypeLoader` class to get schema information from the business assembly and type. For example, the `CanDelete()` method needs to determine whether your business object supports deletion, which requires reflecting against your business type. That reflection can only be done in `TypeLoader` in the temporary AppDomain, so `TypeLoader` is used to find this information:

```
    Public Overrides ReadOnly Property CanDelete() As Boolean
      Get
        Return TypeLoader.CanDelete( _
          mOwner.DataSourceControl.TypeAssemblyName, mOwner.DataSourceControl.TypeName)
      End Get
    End Property
```

As you can see, `TypeLoader` now has a `Shared` method called `CanDelete()`. The `CanDelete()` method safely abstracts the process of finding the temporary shadow directory, by creating a temporary AppDomain, doing the reflection and returning the result.

The same change is applied to the `CanInsert()` and `CanUpdate()` methods in `CslaDesignerDataSourceView`.

Changes to ObjectViewSchema

The changes to the `ObjectViewSchema` class are similar, but more drastic, than those in `CslaDesignerDataSourceView`. The `ObjectViewSchema` class must implement a `GetFields()` method that returns the schema information about the data source, or business object. That

method must now delegate all its work to `TypeLoader` so the process can occur safely in a temporary AppDomain:

```
    Public Function GetFields() As _
      System.Web.UI.Design.IDataSourceFieldSchema() _
      Implements System.Web.UI.Design.IDataSourceViewSchema.GetFields

      Return TypeLoader.GetFields(mTypeAssemblyName, mTypeName)

    End Function
```

The result is that `ObjectViewSchema` now does virtually no work at all. Instead, all the work is handled by `TypeLoader`.

TypeLoader Class

The `TypeLoader` class is somewhat complex. It is responsible for abstracting the process of safely retrieving schema information from a business type in a business assembly.

To do this, it includes code to perform the following functions:

- Locate the shadow directory containing the current version of the business assembly
- Create a temporary AppDomain
- Return the results from that AppDomain back to the main Visual Studio AppDomain where the `CslaDataSource` control is running

What makes this complex, is that the `TypeLoader` class contains some code designed to run in the Visual Studio AppDomain, and some designed to run in the temporary AppDomain. The following are `Shared` methods designed to run in the Visual Studio AppDomain:

- `GetFields()`
- `CanDelete()`
- `CanInsert()`
- `CanUpdate()`

These methods are invoked by `CslaDesignerDataSourceView` and `ObjectViewSchema` to retrieve schema data as required. These Shared methods create a temporary AppDomain and delegate the work to instance methods that are running in that other AppDomain. Those instance methods are:

- `GetFields()`
- `CanDelete()`
- `CanInsert()`
- `CanUpdate()`

Additionally, `TypeLoader` includes a set of helper methods used to find the shadow directory and create the temporary AppDomain:

- `GetTypeLoader()`
- `GetTemporaryAppDomain()`
- `GetOriginalPath()`
- `GetType()`
- `GetCodeBase()`

In general, the sequence of any call to get schema information follows the same sequence:

1. Get the original path to the business assembly (old shadow directory)
2. Create a temporary AppDomain that contains a `TypeLoader` instance
3. Delegate the call to the `TypeLoader` instance

Then, in the temporary AppDomain:

1. Get the current shadow directory path
2. Load the business assembly from the current shadow directory
3. Reflect against the business assembly to get the metadata
4. Return the results

The `Shared` methods called to initiate this process all follow a similar structure to trigger this process in each case.

Implementing the Shared Methods

The `Shared` methods called by `CslaDesignerDataSourceView` and `ObjectViewSchema` follow the same structure. For example, here's the `GetFields()` method:

```
Public Shared Function GetFields( _
  ByVal assemblyName As String, ByVal typeName As String) _
  As IDataSourceFieldSchema()

  Dim result As List(Of ObjectFieldInfo) = New List(Of ObjectFieldInfo)()

  Dim originalPath As String = GetOriginalPath(assemblyName, typeName)

  Dim tempDomain As AppDomain = GetTemporaryAppDomain()
  Try
    result = _
      GetTypeLoader(tempDomain).GetFields(originalPath, assemblyName, typeName)
  Finally
    AppDomain.Unload(tempDomain)
  End Try
  Return result.ToArray()

End Function
```

You can see how the `GetOriginalPath()` method is called to get the path to the old, original shadow directory used by Visual Studio for this project. The result from that method is later passed into the `GetFields()` instance method running in the temporary AppDomain.

The `GetTemporaryAppDomain()` method is called to create the temporary AppDomain itself, while the `GetTypeLoader()` method creates an instance of `TypeLoader` in that temporary AppDomain.

In the end, the temporary AppDomain is unloaded and the result returned to the calling code.

The other three `Shared` methods follow this exact structure, delegating to the appropriate instance methods of the `TypeLoader` object in the temporary AppDomain.

Finding the Current Shadow Directory

The hardest issue to resolve is finding the directory path to the business assembly. Though you and I see the business assembly in the web project's `\bin` directory, the assembly can't be loaded from that location. This is because Windows itself won't release a file lock on any DLL it loads into memory. To avoid this, both ASP.NET and Visual Studio use a technology called *shadow copies*, where they copy the DLL to a temporary directory and load it from there.

When building web projects in Visual Studio, every time you build your solution a new temporary shadow directory is created, containing shadow copies of all the assemblies referenced by your web project. Visual Studio, and any controls, load the assemblies from that directory, until the next time the solution is built, at which point another temporary directory is used.

Unfortunately, there's no API in .NET that allows you to determine the path to the current shadow directory being used by Visual Studio. Without knowing that directory path, there's no way to safely load the business assembly into a temporary AppDomain.

To find the current shadow directory path, I make some assumptions about how these shadow directories are named and used by Visual Studio. It *is possible* to get the path to `Csla.dll`, and any other referenced assembly, in one of the shadow directories, because Visual Studio loads `Csla.dll` when it sites the `CslaDataSource` control on the web forms designer surface. Using that shadow directory path, I can find all the shadow directories for the current project.

The trick behind this is that Visual Studio can't unload `Csla.dll` once it has loaded the first instance of `CslaDataSource` onto a web form designer. This means that my `CslaDataSource` code, running in Visual Studio's designer environment, always comes from that first shadow directory, even if subsequent shadow directories have been created when the solution was rebuilt. Any attempt to directly load other assemblies always causes those assemblies to load from this same shadow directory from where `Csla.dll` was originally loaded.

The shadow directory path for `Csla.dll` follows this structure:

```
file:///c:/dir1/dir2/dir3/Csla.dll
```

It turns out that part of this path is consistent for all shadow directories created for the project:

```
file:///c:/dir1/dir2/
```

Only that last directory name changes each time the project is built, and I use this to resolve the issue. The `TypeLoader` class was added in version 2.0.1, and it includes a method to locate the most recently created shadow directory for a project, given the path to an assembly in any one of the shadow directories as a parameter:

```
Private Shared Function GetCodeBase(ByVal cslaPath As String) As String

  If cslaPath.StartsWith("file:///") Then
    cslaPath = cslaPath.Substring(8)
    cslaPath = cslaPath.Replace("/", "\")
  End If
  Dim count As Integer = 0
  Dim [end] As Integer = 1
  For pos As Integer = cslaPath.Length - 1 To 1 Step -1
    If cslaPath.Substring(pos, 1) = "\" Then
      count += 1
      If count = 2 Then
        [end] = pos
        Exit For
      End If
    End If
  Next pos
  Dim codeBase As String = cslaPath.Substring(0, [end])

  Dim baseDir As DirectoryInfo = New DirectoryInfo(codeBase)
  Dim result As DirectoryInfo = Nothing
  Dim maxDate As DateTime = DateTime.MinValue
  For Each dir As DirectoryInfo In baseDir.GetDirectories()
    If dir.LastWriteTime > maxDate Then
      maxDate = dir.LastWriteTime
      result = dir
    End If
  Next dir

  If Not result Is Nothing Then
    Return result.FullName & "\"
  Else
    Return Nothing
  End If

End Function
```

This method parses the path to pull out the consistent part of the shadow directory path. It then uses a `DirectoryInfo()` object to get a list of all the shadow directories for the project and it scans that list to find the one that was most recently altered or created. That path points to the most recent, and thus current, shadow directory for the project.

Of course the key piece of information that makes this all work is a path to one of the shadow directories for the project. That is determined in the Visual Studio AppDomain using the `GetOriginalPath()` method:

```
Private Shared Function GetOriginalPath( _
  ByVal assemblyName As String, ByVal typeName As String) As String

  Dim asm As System.Reflection.Assembly = _
    System.Reflection.Assembly.Load(assemblyName)
  Return asm.CodeBase

End Function
```

All this method does is load the business assembly into the Visual Studio AppDomain and then ask for the path (`CodeBase`) to the assembly. Though this is likely an old instance of the business assembly, from an older shadow directory, it doesn't matter because *this* instance of

the assembly isn't used to get the schema information. The only metadata retrieved from this instance is the path to the old shadow directory so it can be used to find the latest shadow directory when `GetCodeBase()` is called.

Creating the Temporary AppDomain

Creating an AppDomain is not complex, and the work is handled by the `GetTemporaryAppDomain()` method:

```
Private Shared Function GetTemporaryAppDomain() As AppDomain

  Dim fulltrust As System.Security.NamedPermissionSet = _
    New System.Security.NamedPermissionSet("FullTrust")
  Dim tempDomain As AppDomain = _
    AppDomain.CreateDomain("__CslaDataSource__temp", _
    AppDomain.CurrentDomain.Evidence, _
    AppDomain.CurrentDomain.SetupInformation, fulltrust, _
    New System.Security.Policy.StrongName() {})
  Return tempDomain

End Function
```

Since this temporary AppDomain will be making use of dynamic assembly loading and reflection, it requires `FullTrust` from code access security (CAS). The `fullTrust` field contains the `NamedPermissionSet` corresponding to `FullTrust` security, and that field is passed to the `CreateDomain()` method to indicate that the new AppDomain should get `FullTrust`.

The `CreateDomain()` method accepts other parameters, including a unique name for the temporary AppDomain, security evidence (copied from the current AppDomain) and setup information (copied from the current AppDomain). The result is an empty AppDomain that contains nothing more than the basic .NET system types.

The `GetTypeLoader()` method is then used to load an instance of `TypeLoader` into this new AppDomain:

```
Private Shared Function GetTypeLoader(ByVal tempDomain As AppDomain) As TypeLoader

  ' load the TypeLoader object in the temp AppDomain
  Dim thisAssembly As System.Reflection.Assembly = _
    System.Reflection.Assembly.GetExecutingAssembly()
  Dim loader As TypeLoader = _
    CType(tempDomain.CreateInstanceFromAndUnwrap( _
      thisAssembly.CodeBase, GetType(TypeLoader).FullName), TypeLoader)
  Return loader

End Function
```

The `AppDomain` object is passed as a parameter to the method so its `CreateInstanceFromAndUnwrap()` method can be called to create an instance of `TypeLoader` in the temporary AppDomain. The "`AndUnwrap`" part of this process is required because the object is created in another AppDomain, and what is returned is a generic proxy object. The "`AndUnwrap`" unwraps that proxy object to get a specific proxy object for the `TypeLoader` object itself.

Implementing the Instance Methods

The instance methods implemented in `TypeLoader` take care of reflecting against the business assembly to get the required metadata. First though, they need to load the business assembly from the most recent shadow directory. All the instance methods follow the same basic structure. Here's `GetFields()` for example:

```
Public Function GetFields( _
  ByVal originalPath As String, _
  ByVal assemblyName As String, _
  ByVal typeName As String) As List(Of ObjectFieldInfo)

  Dim result As List(Of ObjectFieldInfo) = New List(Of ObjectFieldInfo)()

  Dim t As Type = TypeLoader.GetType(originalPath, assemblyName, typeName)
  If GetType(IEnumerable).IsAssignableFrom(t) Then
    ' this is a list so get the item type
    t = Utilities.GetChildItemType(t)
  End If
  Dim props As PropertyDescriptorCollection = TypeDescriptor.GetProperties(t)
  For Each item As PropertyDescriptor In props
    If item.IsBrowsable Then
      result.Add(New ObjectFieldInfo(item))
    End If
  Next item

  Return result

End Function
```

Remember that this code is all running in the temporary AppDomain, so any assemblies or types loaded by this code will be unloaded when the temporary AppDomain is discarded.

This method uses the `Shared GetType()` method to get a `Type` object for your business assembly, using the `originalPath` parameter to locate the latest shadow directory as I discussed earlier. The `GetType()` method itself looks like this:

```
Private Overloads Shared Function [GetType]( _
  ByVal originalPath As String, _
  ByVal assemblyName As String, ByVal typeName As String) As Type

  Dim assemblyPath As String = GetCodeBase(originalPath)

  Dim asm As System.Reflection.Assembly = _
    System.Reflection.Assembly.LoadFrom(assemblyPath & assemblyName & ".dll")
  Dim result As Type = asm.GetType(typeName, True, True)
  Return result

End Function
```

The `GetCodeBase()` method I discussed earlier is used to get the latest shadow directory path. That path is then used to build a path to the business assembly and the `Assembly.LoadFrom()` method is used to dynamically load the assembly from that location into the temporary AppDomain. This `Type` object is then returned to the calling method.

Back in the `GetFields()` method, the `Type` object is interrogated using reflection to retrieve the requested metadata. This process is no different from what was done in CSLA .NET 2.0, except now the code is running in a temporary AppDomain.

Implementing Paging

Web Forms data binding, and specifically the `GridView` control, supports the concept of paging. If the total number of rows of data is very large, you may choose to only retrieve a subset of the data at any given time: a single page of data. However, when you do this you must still provide data binding with the number of *total* rows of data available.

Earlier in the book I discussed the `IReportTotalRowCount` interface and how you can implement this in your business collections to support paging. The `CslaDataSource` control has been adapted to use this interface as well, providing you with the tools you need to implement paged data in your web pages.

These changes impacted the following classes:

- `SelectObjectArgs`
- `CslaDataSource`
- `CslaDataSourceView`
- `CslaDesignerDataSourceView`

There are three aspects to the paging support:

1. `CslaDataSource` uses the `IReportTotalRowCount` interface to return the total number of rows of data to data binding on request
2. The `SelectObjectArgs` parameter passed to the `SelectObject` event now includes information about the start index and number of rows of data to retrieve
3. A `TypeSupportsPaging` property has been added to the `CslaDataSource` control so you can control whether data binding thinks you support paging or not

These changes combine to allow the web page and business collection developers to support the concept of paging.

Changes to SelectObjectArgs

The `SelectObjectArgs` object is created by `CslaDataSourceView` and is provided to the UI developer as an argument to the `SelectObject` event raised by any `CslaDataSource` control. The `SelectObjectArgs` parameter allows the UI developer to return the requested business object through the `BusinessObject` property, and with these changes it now also provides the UI developer with extra information about the data to be retrieved.

The new information is provided through the properties listed in Table 14.

Property	Description
`StartRowIndex`	The 0-based index of the first row of to be retrieved in this request.
`MaximumRows`	The maximum number of rows of data to be retrieved as part of this request. This value corresponds to the page size requested by the UI control (such as a `GridView`).
`RetrieveTotalRowCount`	A Boolean value indicating whether data binding requires that the total row count be returned through the `IReportTotalRowCount` interface.

Table 14. New paging properties of SelectObjectArgs

These are all simple read-only properties provided to the UI developer. For example, here's the `StartRowIndex` property implementation:

```
Private mStartRowIndex As Integer

Public ReadOnly Property StartRowIndex() As Integer
  Get
    Return mStartRowIndex
  End Get
End Property
```

These values are set in the constructor, which now accepts an ASP.NET `DataSourceSelectArguments` object as a parameter:

```
Public Sub New(ByVal args As System.Web.UI.DataSourceSelectArguments)

  mStartRowIndex = args.StartRowIndex
  mMaximumRows = args.MaximumRows
  mRetrieveTotalRowCount = args.RetrieveTotalRowCount

  mSortExpression = args.SortExpression
  If Not String.IsNullOrEmpty(mSortExpression) Then
    If Len(mSortExpression) >= 5 AndAlso Right(mSortExpression, 5) = " DESC" Then
      mSortProperty = Left(mSortExpression, mSortExpression.Length - 5)
      mSortDirection = ListSortDirection.Descending

    Else
      mSortProperty = args.SortExpression
      mSortDirection = ListSortDirection.Ascending
    End If
  End If

End Sub
```

I've highlighted the lines of code pertaining to paging. The other lines of code relate to sorting, and I'll discuss them later in this book.

The `SelectObjectArgs` class is also now marked as `Serializable`:

```
<Serializable()> _
Public Class SelectObjectArgs
  Inherits EventArgs
```

As you'll see, this simplifies the use of paging and sorting by allowing a business developer to simply include the `SelectObjectArgs` object as part of the collection's `Criteria` object, making the values it contains available to `DataPortal_Fetch()`.

Changes to CslaDataSource

The `CslaDataSource` control itself doesn't do much work. It is primarily a "traffic cop" that routes calls to sub-objects like `CslaDataSourceView` to do the work. Following this idea, the `TypeSupportsPaging` property simply delegates the call:

```
Public Property TypeSupportsPaging() As Boolean
  Get
    Return CType(Me.GetView("Default"), CslaDataSourceView).TypeSupportsPaging
  End Get
  Set(ByVal value As Boolean)
    CType(Me.GetView("Default"), CslaDataSourceView).TypeSupportsPaging = value
  End Set
End Property
```

The `CslaDataSourceView` class is responsible for maintaining the actual value in this case.

Changes to CslaDataSourceView

Most of the changes occur in the `CslaDataSourceView` class, as it is this object that does the bulk of the work for data binding. This class declares a field and property for the `TypeSupportsPaging` property:

```
Private mTypeSupportsPaging As Boolean

Public Property TypeSupportsPaging() As Boolean
  Get
    Return mTypeSupportsPaging
  End Get
  Set(ByVal value As Boolean)
    mTypeSupportsPaging = value
  End Set
End Property
```

More importantly, `CslaDataSourceView` implements the `ExecuteSelect()` method, where retrieval of data is handled. The actual retrieval of data is delegated to the UI developer's code in the page through the `SelectObject` event. However, the results of that event are handled by `ExecuteSelect()`, and it is here that the `IReportTotalRowCount` interface becomes important.

Remember that in the `SelectObject` event handler, the UI developer is responsible for creating an appropriate business object for data binding to use. If you are implementing paging, this business object will be a special collection object that inherits from `BusinessListBase` or `ReadOnlyList` base *and* which implements `IReportTotalRowCount`.

Here's the complete code for `ExecuteSelect()`, with the lines dealing with `IReportTotalRowCount` highlighted:

```
Protected Overrides Function ExecuteSelect( _
  ByVal arguments As System.Web.UI.DataSourceSelectArguments) As _
  System.Collections.IEnumerable

  ' get the object from the page
  Dim args As New SelectObjectArgs(arguments)
```

```
      mOwner.OnSelectObject(args)
      Dim result As Object = args.BusinessObject

      If arguments.RetrieveTotalRowCount Then
        Dim rowCount As Integer
        If result Is Nothing Then
          rowCount = 0

        ElseIf TypeOf result Is Csla.Core.IReportTotalRowCount Then
          rowCount = CType(result, Csla.Core.IReportTotalRowCount).TotalRowCount

        ElseIf TypeOf result Is IList Then
          rowCount = CType(result, IList).Count

        ElseIf TypeOf result Is IEnumerable Then
          Dim temp As IEnumerable = CType(result, IEnumerable)
          Dim count As Integer = 0
          For Each item As Object In temp
            count += 1
          Next
          rowCount = count

        Else
          rowCount = 1
        End If
        arguments.TotalRowCount = rowCount
      End If

      ' if the result isn't IEnumerable then
      ' wrap it in a collection
      If Not TypeOf result Is IEnumerable Then
        Dim list As New ArrayList
        If result IsNot Nothing Then
          list.Add(result)
        End If
        result = list
      End If

      ' now return the object as a result
      Return CType(result, IEnumerable)

    End Function
```

Notice how the `SelectObjectArgs` object is now created by passing the `DataSourceSelectArguments` parameter into the constructor. This `DataSourceSelectArguments` parameter contains a variety of data collected by data binding and provided to a data source control. With the changes to `SelectObjectAgs`, some of this information is now provided to the UI developer as well, so they can act on it in the `SelectObject` event handler.

The total page count code is only triggered if the select request from data binding included a request for the total row count. Such a request is common, and typically occurs when the UI control is any sort of grid or list control. In that case, the code checks to see if the business object returned from the `SelectObject` event implements the `IReportTotalRowCount` interface, and if it does then the total row count is retrieved via that interface.

If the business object doesn't implement the interface; then the normal approach is taken, where the collection's `Count` property is returned.

Changes to CslaDesignerDataSourceView

The `TypeSupportsPaging` property in `CslaDataSource` exists specifically to allow the UI developer to control whether the `CanPage()` method returns `True` or `False` in the

CslaDesignerDataSourceView object. This CanPage() method is used by the Web Forms page designer so the designer knows how to render the control and its options pages at design time.

The CanPage() method looks like this:

```
    Public Overrides ReadOnly Property CanPage() As Boolean
      Get
        Return mOwner.DataSourceControl.TypeSupportsPaging
      End Get
    End Property
```

Notice how the call delegates to the CslaDataSource control's TypeSupportsPaging property value.

Implementing Sorting

The CslaDataSource control itself doesn't support sorting of data. However, you might choose to support sorting in your UI code or your collection class. You can do this by using SortedBindingList, or by reloading the collection from the database, allowing your database to do the sorting for you.

Either way, if you do support sorting in your UI or collection, you need some way to tell the Web Forms designer that you are supporting the concept so the designer can render the control and its options pages properly at design time. Additionally, you'll need to know the name of the column on which to sort, and whether the sort should be ascending or descending.

Providing Sort Information to the SelectObject Event Handler

The SelectObject event handler receives a SelectObjectArgs parameter, which contains details about the data requested by data binding. This information can be used by the UI or business collection developer to sort the data as requested. Table 15 lists the new properties of SelectObjectArgs that provide information about sorting:

Property	Description
SortExpression	The SortExpression property provided by data binding. In many cases, this is just a property name, but it can be a comma separated list of property names if the UI developer handles the Sorting property of a UI control.
SortProperty	The name of the property on which to sort. This value is only valid if a single property is used for sorting (which is the default behavior).
SortDirection	A value indicating whether to sort in ascending or descending order. This value is only valid if a single property is used for sorting (which is the default behavior).

Table 15. New sorting properties of SelectObjectArg

You can either use the raw value in `SortExpression` directly, or use the simpler pre-processed `SortProperty` and `SortDirection` properties.

`SortProperty` and `SortDirection` are only valid for the default behavior of sorting by a single column or property. If the UI developer handles the `Sorting` property of a UI control, they can manually set `SortExpression` to more complex values such a list of column names. In that case, the `SortProperty` and `SortDirection` may not return useful values.

Implementing the CanSort property

Data binding determines whether a data source supports paging through the `CanSort()` method of `CslaDesignerDataSourceView`. Like the `TypeSupportsPaging` property on `CslaDataSource` and `CslaDataSourceView`, there is also a `TypeSupportsSorting` property. The implementations are identical, so I won't review that code. Here's the `CanSort()` method implementation:

```
    Public Overrides ReadOnly Property CanSort() As Boolean
      Get
        Return mOwner.DataSourceControl.TypeSupportsSorting
      End Get
    End Property
```

Like the `CanPage()` method, notice how this method delegates the call to the `CslaDataSource` control itself. This allows the UI developer to set `TypeSupportsSorting` to control the `CanSort()` result.

Using the Enhancements

The enhancements to `CslaDataSource` can be grouped into three functional areas:

- Supporting dynamic schema refresh
- Supporting paging
- Supporting sorting

The dynamic schema refresh enhancements exist to support the `Refresh` link you can use in the Visual Studio designer to refresh the schema on a data control, grid control or list control. As such, I won't discuss their use in this book – you can just click those links to see the results.

The paging and sorting support however, do require some discussion, because in each case you must take extra steps as you code your collections and pages in order to utilize this functionality.

Using Paging

The paging support provided by the `CslaDataSource` control merely opens the door so you can implement the paging yourself. Implementing paging requires that you design your business collection to support paging, and then you can use `CslaDataSource` to tell data binding that your collection supports the concept.

The following is a list of the high level steps required to implement paging:

1. Implement `IReportTotalRowCount` in your collection
2. Accept the `SelectObjectArgs` parameter in your `Shared` factory method
3. Include the `SelectObjectArgs` value as a field in your `Criteria` object
4. Use the starting row index and page size values from `SelectObjectArgs` in your `DataPortal_Fetch()` method to load the collection with only the specified page of data
5. In your `DataPortal_Fetch()` method, load the total number of rows of data available
6. Set the `TypeSupportsPaging` property of your `CslaDataSource` control to `True`
7. Enable paging in the `GridView` (or other paging-enabled UI) control

I'll walk through the basic structure of a paged collection and using a `GridView` control to support paging.

Implementing a Paged Collection

A paged collection is much like a normal business collection in many ways, but it is certainly unique in other ways. A paged collection can inherit from either `BusinessListBase` or `ReadOnlyListBase`, but it must also implement the `IReportTotalRowCount` interface. And of course it will only load pages (subsets) of the total data rather than loading all the data available, so its `Criteria` class and `DataPortal_Fetch()` implementations will need to take care of those details.

The basic structure of a paged collection is this:

```
<Serializable()> _
Public Class PersonList
  Inherits ReadOnlyListBase(Of PersonList, Person)

  Implements Core.IReportTotalRowCount

#Region " Business Methods "

  Private mTotalRowCount As Integer

  Private ReadOnly Property TotalRowCount() As Integer _
      Implements Csla.Core.IReportTotalRowCount.TotalRowCount
    Get
      Return mTotalRowCount
    End Get
  End Property

#End Region

#Region " Factory Methods "

  Public Shared Function GetPage( _
    ByVal selectArgs As Csla.Web.SelectObjectArgs) As PersonList

    Return DataPortal.Fetch(Of PersonList)(New Criteria(selectArgs))

  End Function

  Private Sub New()
    ' require use of factory methods
  End Sub
```

```
#End Region

#Region " Data Access "

  <Serializable()> _
  Private Class Criteria

    Private _args As Csla.Web.SelectObjectArgs

    Public ReadOnly Property SelectArgs() As Csla.Web.SelectObjectArgs
      Get
        Return _args
      End Get
    End Property

    Public Sub New(ByVal args As Csla.Web.SelectObjectArgs)
      _args = args
    End Sub

  End Class

  Private Overloads Sub DataPortal_Fetch(ByVal criteria As Criteria)

    ' load total row count
    If criteria.SelectArgs.RetrieveTotalRowCount Then
      mTotalRowCount = 42
    End If

    ' load page of data
    IsReadOnly = False
    Dim startValue As Integer = criteria.SelectArgs.StartRowIndex
    Dim endValue As Integer = _
      criteria.SelectArgs.StartRowIndex + 1 + criteria.SelectArgs.MaximumRows
    If endValue > 42 Then
      endValue = 42
    End If
    For index As Integer = criteria.SelectArgs.StartRowIndex + 1 To endValue
      Add(Person.GetPerson(index))
    Next
    IsReadOnly = True

  End Sub

#End Region

End Class
```

The `DataPortal_Fetch()` method in this example is obviously artificial, but illustrates the idea that this method must both set the count for the total number of available rows, and load the collection with the requested page of data based on the starting row index and page size information passed in through the `Criteria` object.

The `Shared` factory method accepts the `SelectedObjectArgs` value:

```
  Public Shared Function GetPage( _
    ByVal selectArgs As Csla.Web.SelectObjectArgs) As PersonList

    Return DataPortal.Fetch(Of PersonList)(New Criteria(selectArgs))

  End Function
```

This object is then used to populate the `Criteria` object so the values it contains are available to `DataPortal_Fetch()`.

Using Paging in a GridView

The `SelectObjectArgs` parameter provided to the UI developer by `CslaDataSource` contains properties indicating the starting row index and page size. These values come from the UI control, such as `GridView`. Data binding automatically gets the values from the control and provides them to `CslaDataSource`, so no special work is required by the UI developer to make paging work.

In the `SelectObject` event handler, the UI code merely takes the `SelectObjectArgs` parameter value and provides it to the `Shared` factory method:

```
Protected Sub CslaDataSource1_SelectObject( _
  ByVal sender As Object, ByVal e As Csla.Web.SelectObjectArgs) _
  Handles CslaDataSource1.SelectObject

  e.BusinessObject = _
    CslaDSTestLibrary.PersonList.GetPage(e)

End Sub
```

You can control the initial page index and the page size by setting the `PageIndex` and `PageSize` properties of the `GridView` control at design time or runtime.

You must also specify that the `GridView` control should use paging as shown in Figure 1.

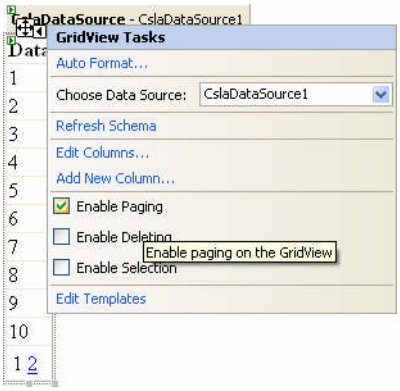

Figure 1. Enabling paging in the GridView control

For this option to work properly, remember that you need to set the `CslaDataSource` control's `TypeSupportsPaging` property to `True` as well.

You should now understand how to create a collection that implements `IReportTotalRowCount` to support paging, and how to use a `GridView` control's properties to determine the page number and page size of the data to be retrieved on each request.

Using Sorting

The sorting support provided by the `CslaDataSource` control allows you to implement sorting of a collection for data binding. The sorting support provided by `CslaDataSource` is very flexible and open-ended, which means there are several ways you can choose to implement sorting, including the following options:

- Sort in the UI using `SortedBindingList`
- Sort the data in the object's factory method
- Sort the data in the database, then load the collection with pre-sorted data

The first two options can not be easily combined with paging, while the third option can be combined with paging relatively easily (assuming your data lends itself to paging in the first place).

Sorting in the UI

Sorting in the UI can only be done in the case that your collection is not paged. The assumption is that the `SelectObject` event handler in the UI is able to retrieve the full collection of data, and it can then apply a sort before returning the list to data binding.

The implementation is not complex, and is entirely contained within the `SelectObject` event handler in your web page. The process follows these steps:

1. Retrieve the full list of data
2. See if `SortExpression` is specified, and if not return the unsorted list
3. If `SortExpression` is specified, sort the list and return the sorted result

For example:

```
Protected Sub CslaDataSource1_SelectObject( _
  ByVal sender As Object, ByVal e As Csla.Web.SelectObjectArgs) _
  Handles CslaDataSource1.SelectObject

  ' get unsorted list
  Dim list As PersonList = PersonList.GetList

  ' do sort
  If String.IsNullOrEmpty(e.SortExpression) Then
    ' return unsorted result
    e.BusinessObject = list

  Else
    Dim sorted As New SortedBindingList(Of Person)(list)
    sorted.ApplySort(e.SortProperty, e.SortDirection)

    ' return sorted result
    e.BusinessObject = sorted
  End If

End Sub
```

The unsorted data is retrieved by calling a normal `Shared` factory method.

Once the data has been retrieved, `e.SortExpression` is used to determine whether a sort was requested. If this value is `Nothing` or an empty `String` then no sort was requested so the unsorted list is returned to data binding.

On the other hand, if a sort was specified then a `SortedBindingList` is created to provide a sorted view of the original collection. Notice that the `e.SortProperty` and `e.SortDirection` are then used to apply the sort to the `SortedBindingList` object.

> **Note:** If the UI developer handles the `Sorting` event of their UI control and alters the `SortExpression` to be a more complex comma separated list of column names, the `SortProperty` and `SortDirection` properties will not contain valid information. In that case, the `SortExpression` value must be parsed and used directly.

Because the `ApplySort()` method already accepts property name and sort direction parameters, the values from the `SelectObjectArgs` parameter can be passed directly to `ApplySort()`.

Sorting in the Database

In some cases, it may be more efficient to sort the data in the database as the collection is loaded in `DataPortal_Fetch()` rather than sorting the data on the web server in the `SelectObject` event handler. Additionally, sorting in the database can allow you to implement both sorting and paging of the data.

To implement sorting in the database, the values in the `SelectObjectArgs` parameter must be passed through to the `DataPortal_Fetch()` method. The technique used to do this is the same as I discussed earlier for implementing paging:

1. The `Shared` factory method accepts a `SelectObjectArgs` parameter
2. The `Criteria` object includes a `SelectObjectArgs` field
3. The code in `DataPortal_Fetch()` uses the values in the `SelectObjectArgs` object to determine if, and how sorting should occur

The basic structure of a collection class is similar to that for paging, as I discussed earlier. Rather than repeat the complete code, I'll highlight the key parts. I'll start with the `Shared` factory method:

```
Public Shared Function GetList( _
  ByVal selectArgs As Csla.Web.SelectObjectArgs) As PersonList

  Return DataPortal.Fetch(Of PersonList)(New Criteria(selectArgs))

End Function
```

The `SelectObjectArgs` value provided to the UI developer in the `SelectObject` event handler is accepted as a parameter in this factory method, and is then passed to the collection's `Criteria` object, which looks like this:

```
<Serializable()> _
Private Class Criteria

  Private mArgs As Csla.Web.SelectObjectArgs

  Public ReadOnly Property SelectArgs() As Csla.Web.SelectObjectArgs
    Get
      Return mArgs
    End Get
  End Property

  Public Sub New(ByVal args As Csla.Web.SelectObjectArgs)
    mArgs = args
  End Sub
```

```
End Class
```

This object is passed through the data portal to the collection's `DataPortal_Fetch()` method, where the values from the `SelectObjectArgs` object can be used to control the appropriate sorting behavior:

```
Private Overloads Sub DataPortal_Fetch(ByVal criteria As Criteria)

  Using cn As New SqlConnection
    cn.Open()
    Using cm As SqlCommand = cn.CreateCommand
      If String.IsNullOrEmpty(criteria.SelectArgs.SortExpression) Then
        cm.CommandText = _
          "SELECT data FROM Person"

      Else
        cm.CommandText = _
          "SELECT data FROM Person ORDER BY " & criteria.SelectArgs.SortExpression
      End If
      cm.CommandType = CommandType.Text

      ' execute command and
      ' load collection with data

    End Using
  End Using

End Sub
```

To keep things simple, I've cut out the code that loads the collection with data, because the important part of the code is the use of the `SortExpression` property to control the sorting process. Notice that if `SortExpression` is `Nothing` or an empty `String`, that no sort is applied to the `SELECT` query. Otherwise, the text of `SortExpression` is used to build the `ORDER BY` clause.

Obviously, you might use other SQL techniques or stored procedures to do the sorting. I'm keeping the code here intentionally simple to illustrate the concept.

In this case, the UI code doesn't have to create a `SortedBindingList`, because the collection will be sorted as it comes from the database. The `SelectObject` event handler becomes simpler:

```
Protected Sub CslaDataSource1_SelectObject( _
  ByVal sender As Object, ByVal e As Csla.Web.SelectObjectArgs) _
  Handles CslaDataSource1.SelectObject

  ' get sorted list
  Dim list As PersonList = PersonList.GetList(e)

End Sub
```

All that is required of the UI code is to pass the `SelectObjectArgs` parameter value to the factory method. The collection and database take care of the rest of the work.

The paging implementation I discussed earlier, and this sorting implementation, can be merged together. The database will first sort the data, and then only return the appropriate rows of data to populate a specific page. The exact technique used to do this in the database is different for each database vendor and version, but the changes to `CslaDataSource` now make it possible for you to implement these paging and sorting features.

Miscellaneous Changes

CSLA .NET version 2.1 includes a number of other enhancements and bug fixes that don't fit within the broader thematic areas discussed earlier in this book. Table 16 lists the miscellaneous changes to the framework.

Change	Description
`SortedBindingList` implements `ICancelAddNew`	This change provides better parity with `BindingList(Of T)`.
`BusinessListBase.IsDirty`	`IsDirty` now only considers items in `DeletedList` if they are not new. New objects in `DeletedList` do not cause the collection to be dirty.
`BusinessBase.Delete()`	`Delete()` is now `Overridable`, so you can override the method to prevent the accidental use of deferred deletion.
`Initialize()` method	The CSLA .NET base classes now invoke an `Intialize()` method as they are being created or deserialized. This method is designed to allow C# code generators to re-hook event delegates, and is typically unnecessary for VB thanks to the `WithEvents`/`Handles` feature of the language.

Table 16. List of miscellaneous changes in CSLA .NET 2.1

I'll walk through each of these changes to the framework, and then discuss how you can use each of them.

Framework Changes

As CSLA .NET continues to evolve, some changes are narrowly focused on solving or addressing a very specific need. As such, each change is largely independent of any other changes to the framework, so let's discuss each in turn.

Implementing ICancelAddNew in SortedBindingList

Microsoft .NET 2.0 introduced the new `ICancelAddNew` interface. This interface is designed to make the undo operation simpler for the case that the add operation for an item in a collection is cancelled through data binding. In that scenario, the newly added item must be removed from the collection.

This removal was complex in Microsoft .NET 1.x, because it was the *child object* in the collection that was notified of the cancel operation. That child object then had to contact the collection in which it was contained and request that the collection remove the child. This was handled through the `IEditableObject` interface. CSLA .NET implements this interface in the `BusinessBase` class.

The ICancelAddNew interface simplifies this process by allowing data binding to communicate directly to the collection, so the collection itself can simply remove the now-cancelled new child object.

Because SortedBindingList doesn't inherit from BindingList(Of T), it must directly implement this new interface.

Changes to SortedBindingList

Implementing the ICancelAddNew interface means implementing two new methods that data binding can use to indicate that a newly added item should be kept or removed from the collection: CancelNew() and EndNew().

The CancelNew() method is called when the new child object should be discarded and removed from the collection. The EndNew() method is called if the user doesn't cancel the new item, and the new child should be permanently kept in the collection.

Implementing the ICancelAddNew interface requires that these methods be added to SortedBindingList:

```
#Region " ICancelAddNew "

  Public Sub CancelNew(ByVal itemIndex As Integer) _
    Implements System.ComponentModel.ICancelAddNew.CancelNew

    Dim can As ICancelAddNew = TryCast(mList, ICancelAddNew)
    If can IsNot Nothing Then
      can.CancelNew(itemIndex)

    Else
      mList.RemoveAt(itemIndex)
    End If

  End Sub

  Public Sub EndNew(ByVal itemIndex As Integer) _
    Implements System.ComponentModel.ICancelAddNew.EndNew

    Dim can As ICancelAddNew = TryCast(mList, ICancelAddNew)
    If can IsNot Nothing Then
      can.EndNew(itemIndex)
    End If

  End Sub

#End Region
```

Like much of the code in SortedBindingList, these methods simply delegate the work to the original list. Remember that SortedBindingList is merely a sorted view over an existing list object, and if that list object implements ICancelAddNew then the process is simply delegated.

However, if the original list does not implement ICancelAddNew, then SortedBindingList must do the work itself. This only impacts the CancelNew() method, where the child object is removed directly if the original list doesn't implement ICancelAddNew:

```
    Else
      mList.RemoveAt(itemIndex)
    End If
```

This change brings `SortedBindingList` more in line with `BindingList(Of T)`, making the collection more consistent and easier to use with data binding.

Changing BusinessListBase.IsDirty

The `IsDirty` property in the `BusinessListBase` class returns `True` if any of the child objects contained in the collection have been changed. In version 2.0, it returned `True` if any items had been removed from the list, but that turns out not to be entirely accurate.

The problem is that a *newly added* item can be removed from the list, which effectively could return the list to its original state; in which case the list shouldn't be considered to be changed and `IsDirty` should return `False`.

Changes to BusinessListBase

In version 2.1, the `IsDirty` property has been changed to only count removed child objects if they are *not new*:

```
Public ReadOnly Property IsDirty() As Boolean
  Get
    ' any non-new deletions make us dirty
    For Each item As C In DeletedList
      If Not item.IsNew Then
        Return True
      End If
    Next

    ' run through all the child objects
    ' and if any are dirty then the
    ' collection is dirty
    For Each Child As C In Me
      If Child.IsDirty Then Return True
    Next
    Return False
  End Get
End Property
```

This change firms up the rules around when a list is marked as having been changed.

Changing BusinessBase.Delete

Editable root objects can support two mechanisms for deletion: immediate and deferred.

To implement immediate deletion, the business object developer must create a `Shared` factory method that calls `DataPortal.Delete()`. They must also implement the corresponding `DataPortal_Delete()` method to remove the object's data from the database.

Deferred deletion is enabled by default, though the business object developer does need to implement the `DataPortal_DeleteSelf()` method to remove the object's data from the database. However, the `BusinessBase` class in the `Csla.Core` namespace implements a `Public` method called `Delete()` that allows any other code to mark your editable root object for deletion. When that object's `Save()` method is called, the object's `DataPortal_DeleteSelf()` method is called to delete the object's data.

But what if you *don't* want to support deferred deletion? While you could throw an exception from `DataPortal_DeleteSelf()`, or just not implement that method and allow CSLA .NET to throw an exception on your behalf, that could mean a round-trip to the application server for no reason.

Changes to BusinessBase

In version 2.1, the `Delete()` method is now marked as `Overridable`, so you can override the method and throw an exception immediately to indicate that deferred deletion isn't supported by your object:

```
    Public Overridable Sub Delete() Implements IEditableBusinessObject.Delete

      If Me.IsChild Then
        Throw New NotSupportedException(My.Resources.ChildDeleteException)
      End If

      MarkDeleted()

    End Sub
```

This change provides a more elegant way for you to disable the deferred deletion behavior in your objects.

Implementing the Initialize Methods

Code generation is a powerful tool in any developer's toolkit. Many people have created code generators, or templates for existing code generators, to build their CSLA .NET business objects.

The most common way to build such templates is to use an inheritance-based scheme where the code generator creates a base class with most of the object's code, and the human developer creates a subclass that customizes the generated code if needed.

For example, if you have a `Customer` business object, the code generator would create a class named `CustomerBase`, and the developer would subclass that class to create a `Customer` class. This `Customer` class only contains overrides of existing propeties and methods, and then only if the generated code is somehow inadequate for this specific object.

Microsoft .NET 2.0 includes the new concept of partial classes, which provides an alternative to this inheritance-based approach. With partial classes, the code generator creates a `Customer` class, and the developer also manually creates a `Customer` class. The compiler merges these two classes into one as the project is compiled.

The challenge with partial classes is that there's no way to override any method or property implemented in the code generated part of the class. The user-created code can only add to the class, it can't alter anything.

The partial class concept was invented to simplify the creation of forms in Windows Forms, and in that environment the user-created code can simply respond to a wealth of events raised by the `Form` base class. The same is true for Web Forms with the `Page` base class and `DataTable` objects with the `DataTable` class.

But the CSLA .NET base classes, such as `BusinessBase`, raise very few events. And normal business objects don't raise all that many events either. So it is relatively difficult to use partial classes with code generation.

Nonetheless, partial class code generation templates have been created for CSLA .NET objects. In most cases, the generated code raises many `Private` events that can be handled by the user-created code, thus solving the problem. And with VB this is pretty smooth, because

VB has the concept of `WithEvents` and the `Handles` clause, so no explicit hookup of the events is required.

But with C# the event solution is more complex. This is because events require explicit hookup, connecting the event to the method that will handle the event. The C# compiler doesn't help in this regard.

The result is that the events need to be explicitly hooked up when the object is first created. This issue is not unique to CSLA .NET objects. Windows Forms follows a standard pattern of calling an initialization method when any form object is created, to allow the hookup of events among other tasks.

I followed this same pattern in version 2.1 of CSLA .NET, by adding an `Initialize()` method to all of the CSLA .NET base classes, including:

- `BusinessBase`
- `BusinessListBase`
- `ReadOnlyBase`
- `ReadOnlyListBase`
- `NameValueListBase`
- `CommandBase`

In each class, the `Initialize()` method is invoked as the object is created. Develoeprs who need to hook events can put that code in the `Initialize()` method, knowing that this method will run when the object is created.

The result is that the following sequence of methods are invoked as a CSLA .NET object is created:

- `Initialize()`
- `New()`
- Methods to initialize validation/authorization rules

This sequence seems odd. How can `Initialize()` run before the constructor? Remember that constructors are run in order, from the deepest class in your inheritance hierarchy out to your actual class. What happens here is that the `Initialize()` method is called from the constructor in the base class, and so it is called *before* your constructor gets to run.

The constructor is normally part of the generated code, but the `Initialize()` method can be implemented in the user-created part of the class, allowing the developer to explicitly hook events to methods as they are implemented.

Changes to Base Classes

The changes to the CSLA .NET base classes are simple and consistent. In each base class an `Intialize()` method is declared:

```
Protected Overridable Sub Initialize()
  ' allows a generated class to set up events to be
  ' handled by a partial class containing user code
End Sub
```

The method is `Overridable`, so a business developer can override it, but it doesn't *require* overriding. This way a business developer can ignore the `Inialize()` method entirely, or override it to hookup events as needed.

Then, in the base class constructor, the `Initialize()` method is invoked. Here's the code from `BusinessBase`, for example:

```
    Protected Sub New()

      Initialize()
      AddInstanceBusinessRules()
      If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
        SyncLock Me.GetType
          If Not Validation.SharedValidationRules.RulesExistFor(Me.GetType) Then
            AddBusinessRules()
          End If
        End SyncLock
      End If
      AddInstanceAuthorizationRules()
      If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
        SyncLock Me.GetType
          If Not Csla.Security.SharedAuthorizationRules.RulesExistFor(Me.GetType) Then
            AddAuthorizationRules()
          End If
        End SyncLock
      End If

    End Sub
```

This ensures that the `Initialize()` method is invoked immediately before the constructor in the business subclass.

> **Note:** Technically, `Initialize()` is invoked *during* the call to the constructor, but it is always invoked before the constructor code in a subclass gets to run, so you can consider that `Initialize()` runs before the constructor in any meaningful sense.

This pattern is repeated in all the CSLA .NET base classes. The result is that it is now easier to implement code generators or templates that use the partial class concepts in Microsoft .NET 2.0.

Using the Enhancements

Two of the enhancements I just discussed are so low-level that they aren't intended for direct use. The implementation of `ICancelAddNew` in the `SortedBindingList` class is used by data binding and the benefit is automatic to anyone using data binding against a `SortedBindingList`. Similarly, the refinement of the `IsDirty` method in `BusinessListBase` is automatic for anyone using the `BusinessListBase` class.

However, the changes to the `Delete()` method in `BusinessBase` and the new `Initialize()` method in all base classes are intended for use by business developers and code generation authors.

Overriding BusinessBase.Delete

If you are creating an editable root object, you'll be inheritng from `BusinessBase(Of T)` to create your object. If you don't want to support deferred deletion of your object, you should consider overriding the `Delete()` method like this:

```
Public Overrides Sub Delete()

  Throw New NotSupportedException("Deferred deletion not supported")

End Sub
```

This will give any developer calling the `Delete()` method immediate and clear feedback that the feature they are trying to use isn't supported.

Using the Initialize Methods

The `Initialize()` method is designed for use with code generation, where the partial class technology is used. In that case, the code generator will typically generate much of the code for your business class, and the business developer will write any user-created code in another file, using the same class name.

Defining a PropertyChangingEventArgs Class

The most common case for using events in partial classes is to allow the user code to respond as a property is changed. To make this work smoothly, you'll typically want a custom `EventArgs` class that contains the proposed property value:

```
Public Class PropertyChangingEventArgs(Of T)
  Inherits EventArgs

  Private mProposedValue As T
  Public Property ProposedValue() As T
    Get
      Return mProposedValue
    End Get
    Set(ByVal value As T)
      mProposedValue = value
    End Set
  End Property

  Public Sub New(ByVal proposedValue As T)
    mProposedValue = proposedValue
  End Sub

End Class
```

This can be used to declare an event for each property, indicating that the property is changing. That event can be raised by the generated code, allowing the user code to respond to the event.

Generated Business Class Example

When using partial classes, the code generator will create a partial class with the majority of the code in the business class. This includes the properties for the object, along with the constructor and data access code. For example, the code generator might create the following:

```
<Serializable()> _
Partial Public Class Customer
  Inherits BusinessBase(Of Customer)

  Private Event IdChanging As EventHandler(Of PropertyChangingEventArgs(Of Integer))

  Private mId As Integer

  Public Property Id() As Integer
    <System.Runtime.CompilerServices.MethodImpl( _
      Runtime.CompilerServices.MethodImplOptions.NoInlining)> _
    Get
      CanReadProperty(True)
      Return mId
    End Get
    <System.Runtime.CompilerServices.MethodImpl( _
      Runtime.CompilerServices.MethodImplOptions.NoInlining)> _
    Set(ByVal value As Integer)
      CanWriteProperty(True)
      If Not mId.Equals(value) Then
        Dim tmp As New PropertyChangingEventArgs(Of Integer)(value)
        RaiseEvent IdChanging(Me, tmp)
        mId = tmp.ProposedValue
        PropertyHasChanged()
      End If
    End Set
  End Property

  Protected Overrides Function GetIdValue() As Object
    Return mId
  End Function

  Private Sub New()
    ' require use of factory methods
  End Sub

  ' factory methods and data access methods go here

End Class
```

The highlighted lines of code indicate the parts of the generated code that matter to the current discussion.

Notice that the constructor is declared in the generated code. This is required, because a good code generator will call `MarkAsNew()` for child objects, and may take other steps in the constructor as well.

Near the top of the class, a private `IdChanging` event is declared. Because this is a `Private` event, there aren't any issues with serialization. This means the simple event declaration syntax can be used, rather than the more complex block structure that must be used for non-`Private` events.

The `IdChanging` event is raised before the property value is actually changed, providing the user code with the knowledge that the property is changing, and access to the proposed value. It is important to realize that the `ProposedValue` property is read-write, so the user code can change the value. This value is then used to set the property.

User Code Business Class Example

While the code generator creates the majority of the business code, the developer can extend that generated code by creating another partial class containing user code. If the generated code raises a set of `Private` events, the user code can handle those events to respond

appropriately. In the case that a property is changing, the user code might alter the proposed value:

```
Public Class Customer

  Private Sub Customer_IdChanging( _
    ByVal sender As Object, ByVal e As PropertyChangingEventArgs(Of Integer)) _
    Handles Me.IdChanging

    ' make sure the id value doesn't exceed 1000
    If e.ProposedValue > 1000 Then
      e.ProposedValue = e.ProposedValue - 1000
    End If

  End Sub

End Class
```

Here the `ProposedValue` property is checked to ensure it doesn't exceed the value `1000`. Remember that this code is *part of* the same class as the generated code, so you have full access to the `mId` field, and any other fields, properties and methods defined in the generated code. The `PropertyChangingEventArgs` object gives you access to the proposed value for the property, which means your event handler has access to all the information you should need to work with the property as it is changing.

If you prefer not to use the `Handles` clause, you can use the `Initialize()` method to accomplish the same result:

```
Public Class Customer

  Protected Overrides Sub Initialize()

    AddHandler Me.IdChanging, AddressOf Customer_IdChanging

  End Sub

  Private Sub Customer_IdChanging( _
    ByVal sender As Object, ByVal e As PropertyChangingEventArgs(Of Integer))

    ' make sure the id value doesn't exceed 1000
    If e.ProposedValue > 1000 Then
      e.ProposedValue = e.ProposedValue - 1000
    End If

  End Sub

End Class
```

In this case, the event is explicitly hooked up to the method that handles the event. This technique is required in C#, but the `Handles` clause is a simpler solution in VB.

At this point, you should understand how the `Initialize()` method can be used to perform any object initialization, most commonly event hookups, as the object is created when using partial classes and code generation.

CSLA .NET version 2.1 is an evolutionary step forward from version 2.0. The primary focus is on performance and memory consumption around validation and authorization rules. However, a number of other enhancements have been made that support some important scenarios that many people encounter when using CSLA .NET to build applications.

Index