


File Format APIs

CREATE CONVERT PRINT
MODIFY COMBINE

Files in your applications!

Try for FREE

.NET, Java, Cloud



HOME | ABOUT | CODEBETTER CI | COMMUNITY | EDITORS | 

Sponsored By Aspose - File Format APIs for .NET

Aspose are the market leader of .NET APIs for file business formats – natively work with DOCX, XLSX, PPT, PDF, MSG, MPP, images formats and many more!

4 major principles of Object-Oriented Programming

Posted by [raymondlewallen](#) on [July 19, 2005](#)

For you new programmers or programmers new to OOP, this article will briefly explain the 4 major principles that make a language object-oriented: Encapsulation, Data Abstraction, Polymorphism and Inheritance. *All examples will be in VB.Net, because in my opinion its easier for a new OOP programmer to read and understand at first. Certainly don't think I'm saying you should use one .Net based language over another, as they all are based on the CLR/CLS and all end up as the same assembly language when compiled. Its your preference that determines what language you use. Of course, there are other OOP languages out there, such as Ruby, a pure OOP language, and hybrid languages such as Python, C++ and Java to mention a few.*

Encapsulation

What is encapsulation? Well, in a nutshell, encapsulation is the hiding of data implementation by restricting access to accessors and mutators. First, lets define accessors and mutators:

Accessor

An accessor is a method that is used to ask an object about itself. In OOP, these are usually in the form of properties, which have, under normal conditions, a *get* method, which is an accessor method. However, accessor methods are not restricted to properties and can be any public method that gives information about the state of the object.

```
Public Class Person
    ' We use Private here to hide the implementation of the objects
    ' fullName, which is used for the internal implementation of Person.
    Private _fullName As String = "Raymond Lewallen"

    ' This property acts as an accessor. To the caller, it hides the
    ' implementation of fullName and where it is set and what is
    ' setting its value. It only returns the fullname state of the
    ' Person object, and nothing more. From another class, calling
    ' Person.FullName() will return "Raymond Lewallen".
    ' There are other things, such as we need to instantiate the
    ' Person class first, but thats a different discussion.
    Public ReadOnly Property FullName() As String
        Get
            Return _fullName
        End Get
    End Property
End Class
```



Trouble Redi

Experi
seaml
scalabi

Learn n



ScaleOut Soft

Archives

December 2008
September 2008
August 2008
July 2008
January 2008
December 2007
November 2007
October 2007
September 2007
July 2007
June 2007
May 2007
April 2007
March 2007
February 2007
January 2007
December 2006
September 2006
August 2006
July 2006
June 2006
May 2006
April 2006
March 2006
February 2006
January 2006
December 2005
November 2005
October 2005
September 2005

Mutator Mutators are public methods that are used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. Mutators are commonly another portion of the property discussed above, except this time its the *set* method that lets the caller modify the member data behind the scenes.

```
Public Class Person
    ' We use Private here to hide the implementation of the objects
    ' fullName, which is used for the internal implementation of Person.
    Private _fullName As String = "Raymond Lewallen"

    ' This property now acts as an accessor and mutator. We still
    ' have hidden the implementation of fullName.
    Public Property FullName() As String
        Get
            Return _fullName
        End Get
        Set(ByVal value As String)
            _fullName = value
        End Set
    End Property
End Class
```

August 2005
 July 2005
 June 2005
 May 2005
 April 2005
 March 2005
 February 2005
 January 2005
 December 2004



Ok, now lets look at a different example that contains an accessor and a mutator:

```
Public Class Person
    Private _fullName As String = "Raymond Lewallen"

    ' Here is another example of an accessor method,
    ' except this time we use a function.
    Public Function GetFullName() As String
        Return _fullName
    End Function

    ' Here is another example of a mutator method,
    ' except this time we use a subroutine.
    Public Sub SetFullName(ByVal newName As String)
        _fullName = newName
    End Sub
End Class
```

So, the use of mutators and accessors provides many advantages. By hiding the implementation of our Person class, we can make changes to the Person class without the worry that we are going to break other code that is using and calling the Person class for information. If we wanted, we could change the fullName from a String to an array of single characters (FYI, this is what a string object actually is behind the scenes) but they callers would never have to know because we would still return them a single FullName string, but behind the scenes we are dealing with a character array instead of a string object. Its transparent to the rest of the program. This type of data protection

and implementation protection is called *Encapsulation*. Think of accessors and mutators as the pieces that surround the data that forms the class.

Abstraction

Data abstraction is the simplest of principles to understand. Data abstraction and encapsulation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details. Abstraction denotes a model, a view, or some other focused representation for an actual item. Its the development of a software object to represent an object we can find in the real world. Encapsulation hides the details of that implementation.

Abstraction

is used to manage complexity. Software developers use abstraction to decompose complex systems into smaller components. As development progresss, programmers know the functionality they can expect from as yet undeveloped subsystems. Thus, programmers are not burdened by considering the waysin which the implementation of later subsystesm will affect the design of earlier development.

The best

definition of abstraction I've ever read is: "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer." — G. Booch, Object-Oriented Design With Applications, Benjamin/Cummings, Menlo Park, California, 1991.

Lets look at this code for a person

object. What are some things that a person can do? Those things must be represented here in our software model of a person. Things such as how tall the person is, and the age of the person; we need to be able to see those. We need the ability for the person to do things, such as run. We need to be able to ask the person if they can read.

```
Public Class Person

    Private _height As Int16
    Public Property Height() As Int16
        Get
            Return _height
        End Get
        Set(ByVal Value As Int16)
            _height = Value
        End Set
    End Property

    Private _weight As Int16
    Public Property Weight() As Int16
        Get
            Return _weight
        End Get
        Set(ByVal Value As Int16)
            _weight = Value
        End Set
    End Property
```

```

Private _age As Int16
Public Property Age() As Int16
    Get
        Return _age
    End Get
    Set(ByVal Value As Int16)
        _age = Value
    End Set
End Property

Public Sub Sit()
    ' Code that makes the person sit
End Sub

Public Sub Run()
    ' Code that makes the person run
End Sub

Public Sub Cry()
    ' Code that make the person cry
End Sub

Public Function CanRead() As Boolean
    ' Code that determines if the person can read
    ' and returns a true or false
End Function

End Class

```

So, there we have started to create a software model of a person object; we have created an abstract type of what a person object is to us outside of the software world. The abstract person is defined by the operations that can be performed on it, and the information we can get from it and give to it. What does the abstracted person object look like to the software world that doesn't have access to its inner workings? It looks like this:



You can't really see what the code is that makes the person run. This is encapsulation that we discussed.

So, in short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing.

Inheritance

Now let's discuss inheritance. Objects can relate to each other with either a "has a", "uses a" or an "is a" relationship. "Is a" is the inheritance way of object relationship. The example of this that has always stuck with me over the years is a library (I think I may have read it in something Grady Booch wrote). So, take a library, for example. A library lends more than just books, it also lends magazines, audiocassettes and microfilm. On some level, all of these items can be treated the same: All four types represent assets of the library that can be loaned out to people. However, even though the 4 types can be viewed as the same, they are not identical. A book has an ISBN and a magazine does not.

And audiocassette has a play length and microfilm cannot be checked out overnight.

Each of these library's assets should be represented by its own class definition. Without inheritance though, each class must independently implement the characteristics that are common to all loanable assets. All assets are either checked out or available for checkout. All assets have a title, a date of acquisition and a replacement cost. Rather than duplicate functionality, inheritance allows you to inherit functionality from another class, called a *superclass* or *base class*.

Let us look at loanable assets base class. This will be used as the base for assets classes such as book and audiocassette:

```
Public Class LibraryAsset

    Private _title As String
    Public Property Title() As String
        Get
            Return _title
        End Get
        Set(ByVal Value As String)
            _title = Value
        End Set
    End Property

    Private _checkedOut As Boolean
    Public Property CheckedOut() As Boolean
        Get
            Return _checkedOut
        End Get
        Set(ByVal Value As Boolean)
            _checkedOut = Value
        End Set
    End Property

    Private _dateOfAcquisition As DateTime
    Public Property DateOfAcquisition() As DateTime
        Get
            Return _dateOfAcquisition
        End Get
        Set(ByVal Value As DateTime)
            _dateOfAcquisition = Value
        End Set
    End Property

    Private _replacementCost As Double
    Public Property ReplacementCost() As Double
        Get
            Return _replacementCost
        End Get
        Set(ByVal Value As Double)
            _replacementCost = Value
        End Set
    End Property

End Class
```

End Class

This LibraryAsset is a superclass, or base class, that maintains only the data and methods that are common to all loanable assets. Book, magazine, audiocassette and microfilm will all be *subclasses* or *derived classes* or the LibraryAsset class, and so they inherit these characteristics. The inheritance relationship is called the “is a” relationship. A book “is a” LibraryAsset, as are the other 3 assets.

Let’s look at book and audiocassette classes that inherit from our LibraryAsset class:

```
Public Class Book

    Inherits LibraryAsset

    Private _author As String
    Public Property Author() As String
        Get
            Return _author
        End Get
        Set(ByVal Value As String)
            _author = Value
        End Set
    End Property

    Private _isbn As String
    Public Property Isbn() As String
        Get
            Return _isbn
        End Get
        Set(ByVal Value As String)
            _isbn = Value
        End Set
    End Property

End Class

Public Class AudioCassette

    Inherits LibraryAsset

    Private _playLength As Int16
    Public Property PlayLength() As Int16
        Get
            Return _playLength
        End Get
        Set(ByVal Value As Int16)
            _playLength = Value
        End Set
    End Property

End Class
```

Now, let's create an instance of the book class so we can record a new book into the library inventory:

```
Dim myBook As Book = New Book
myBook.Author = "Sahil Malik"
myBook.CheckedOut = False
myBook.DateOfAcquisition = #2/15/2005#
myBook.Isbn = "0-316-63945-8"
myBook.ReplacementCost = 59.99
myBook.Title = "The Best Ado.Net Book You'll Ever Buy"
```

You see, when we create a new book, we have all the properties of the LibraryAsset class available to us as well, because we inherited the class. Methods can be inherited as well. Let's add a few methods to our LibraryAsset class:

```
Public Class LibraryAsset

    ' Pretend the properties listed above are right here

    Public Sub CheckOut()
        If Not _checkedOut Then _checkedOut = True
    End Sub

    Public Sub CheckIn()
        If _checkedOut Then _checkedOut = False
    End Sub

End Class
```

Now, our "myBook" we created above automatically inherited these methods, and we didn't even have to touch the Book class in order for it to happen. The book and audiocassette classes above automatically inherited the abilities to be checked out and checked in. In our "myBook" above, now we can check the book out by calling "myBook.CheckOut()". Simple! One of the most powerful features of inheritance is the ability to extend components without any knowledge of the way in which a class was implemented.

Declaration options, such as Public and Private, dictate which members of a superclass can be inherited. For more information on this, see the Declaration Option section of [Eric's post](#).

Polymorphism

Polymorphism means *one name, many forms*. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality. Many VB6ers are familiar with interface polymorphism. I'm only going to discuss polymorphism from the point of view of inheritance because this is the part that is new to many people. Because of this, it can be difficult to fully grasp the full potential of polymorphism until you get some practice with it and see exactly what happens under different scenarios. We're only going to talk about polymorphism, like the other topics, at the basic level.

There are 2 basic types of polymorphism. Overriding, also called run-time polymorphism, and overloading, which is referred to as compile-time polymorphism. This difference is, for method

overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled. Which method will be used for method overriding is determined at runtime based on the dynamic type of an object.

Let's look at some code:

' Base class for library assets

Public MustInherit Class LibraryAsset

' Default fine per day for overdue items

Private Const _finePerDay As Double = 1.25

' Due date for an item that has been checked out

Private _dueDate As DateTime

Public Property DueDate() As DateTime

Get

Return _dueDate

End Get

Set(ByVal Value As DateTime)

_dueDate = Value

End Set

End Property

' Calculates the default fine amount for an overdue item

Public Overridable Function CalculateFineTotal() As Double

Dim daysOverdue As Int32 = CalculateDaysOverdue()

If daysOverdue > 0 Then

Return daysOverdue * _finePerDay

Else

Return 0.0

End If

End Function

' Calculates how many days overdue for an item being returned

Protected Function CalculateDaysOverdue() As Int32

Return DateDiff(DateInterval.Day, _dueDate, DateTime.Now())

End Function

End Class

' Magazine class that inherits LibraryAsset

Public NotInheritable Class Magazine

Inherits LibraryAsset

End Class

' Book class that inherits LibraryAsset

Public NotInheritable Class Book

Inherits LibraryAsset

' This is morphing the CalculateFineTotal() function of the base class.

' This function overrides the base class function, and any call

' to CalculateFineTotal from any instantiated Book class will

' use this function, not the base class function.

' This type of polymorphism is called overriding.

Public Overrides Function CalculateFineTotal() As Double


```

Dim daysOverdue As Int32 = CalculateDaysOverdue()
If daysOverdue > 0 Then
    Return daysOverdue * 0.75
Else
    Return 0.0
End If
End Function

End Class

' AudioCassette class that inherits LibraryAsset
Public NotInheritable Class AudioCassette
    Inherits LibraryAsset

    ' This is morphing the CalculateFineTotal() function of the base class.
    ' This is morphing the CalculateFineTotal(double) function of the
    ' audiocassette class.
    ' This function overrides the base class function, and any call
    ' to CalculateFineTotal() from any instantiated AudioCassette
    ' Class will use this function, not the base class function.
    ' This type of polymorphism is called overloading and overriding.
    Public Overloads Overrides Function CalculateFineTotal() As Double
        Dim daysOverdue As Int32 = CalculateDaysOverdue()
        If daysOverdue > 0 Then
            Return daysOverdue * 0.25
        Else
            Return 0.0
        End If
    End Function

    ' This is morphing the CalculateFineTotal() function of the
    ' audiocassette class.
    ' This type of polymorphism is called overloading.
    Public Overloads Function CalculateFineTotal(ByVal finePerDay As Double) As Double
        Dim daysOverdue As Int32 = CalculateDaysOverdue()
        If daysOverdue > 0 AndAlso finePerDay > 0.0 Then
            Return daysOverdue * finePerDay
        Else
            Return 0.0
        End If
    End Function
End Class

```

You see our library asset class. Pay attention to the overridable function `CalculateFineTotal()`. In `LibraryAsset`, we have defined the default functionality for this method that any derived classes can use. Any class derived from `LibraryAsset` can use this default behavior and calculate fines based on the default implementation of \$1.25 per day late. This is true for our `Magazine` class. We didn't override the function so when late fees are calculated for late magazine returns, it will use the default implementation.

Now look at the book class. We have overridden the `CalculateFineTotal` to use a different value when determining late fees. The overrides keyword in VB tells the caller that any method call will use the virtual method found in `Book`, not the default

implementation found in LibraryAsset. We have implemented runtime polymorphism – method overriding.

Lets move on to AudioCassette. Here we have the same method overriding we found in the book class. Fines are calculated based on \$0.25 per day. Notice we've added something extra. We've added the Overloads keyword to our function and to a new function with the same name, except the new function now accepts a parameter. Now the caller can call either method, and depending on whether or not a parameter is passed, that determines which method will be executed. Notice we do not include the overrides keyword in the 2nd function with a parameter. This is because no method exists in LibraryAsset with that same signature (accepting a parameter of type double). You can only override methods with the same signature in a base class.

Now lets look at some code that creates all these library items and checks them in and calculates our fines based on returning them 3 days late:

Public Class Demo

Public Sub Go()

 ' Set the due date to be three days ago

 Dim dueDate As DateTime = DateAdd(DateInterval.Day, -3, Now())

 ReturnMagazine(dueDate)

 ReturnBook(dueDate)

 ReturnAudioCassette(dueDate)

End Sub

Public Sub ReturnMagazine(ByVal dueDate As DateTime)

 Dim myMagazine As LibraryAsset = New Magazine

 myMagazine.DueDate = dueDate

 Dim amountDue As Double = myMagazine.CalculateFineTotal()

 Console.WriteLine("Magazine: {0}", amountDue.ToString())

End Sub

Public Sub ReturnBook(ByVal dueDate As DateTime)

 Dim myBook As LibraryAsset = New Book

 myBook.DueDate = dueDate

 Dim amountDue As Double = myBook.CalculateFineTotal()

 Console.WriteLine("Book: {0}", amountDue.ToString())

End Sub

Public Sub ReturnAudioCassette(ByVal dueDate As DateTime)

 Dim myAudioCassette As AudioCassette = New AudioCassette

 myAudioCassette.DueDate = dueDate

 Dim amountDue As Double

 amountDue = myAudioCassette.CalculateFineTotal()

 Console.WriteLine("AudioCassette1: {0}", amountDue.ToString())

 amountDue = myAudioCassette.CalculateFineTotal(3.0)

 Console.WriteLine("AudioCassette2: {0}", amountDue.ToString())

End Sub

End Class

The output will look like the following:

```
Magazine: 3.75
Book: 2.25
AudioCassette1: 0.75
AudioCassette2: 9
```

You can see how all of our output was different, based on the method that was executed. We created a new Magazine, which is a type of LibraryAsset. That is why the instantiation says “myMagazine As LibraryAsset”. However, since we actually want a magazine, we create a “New Magazine”. Same thing with book. For Book, its a little bit more tricky. Since we created a Book of the type LibraryAsset, this is where the polymorphism comes into play. Book overrides the CalculateFineTotal of LibraryAsset. Audiocassette is a little bit different. It actually extends the implementation of LibraryAsset by including an overloaded function for CalculateFineTotal(). If we weren’t going to use the function that took a parameter, we would create it the same way we created the Book and Magazine classes. But in order to use the overloaded function, we have to create a new AudioCassette of the type AudioCassette, because LibraryAsset doesn’t support the overloaded function.

Only the Magazine used the default method found in the base class. Book and AudioCassette used their own implementations of the method. Also, at compile time, the decision was made which method would be used when we calculate amountDue for the AudioCassette class. The first call used the 1st method in AudioCassette without parameters. The 2nd call used the 2nd method with a parameter.

This entry was posted in [OOP](#). Bookmark the [permalink](#). Follow any comments here with the [RSS feed for this post](#).

[← Introduction to Refactoring](#)

[Extreme Programming Workshop Right Around the Corner! →](#)

27 Responses to 4 major principles of Object-Oriented Programming



Online Games says:

May 19, 2010 at 4:24 pm

Revising for my exam tomorrow and found this really helpful for the major OOD principles, Thanks.



Aeden Jameson says:

April 7, 2010 at 5:02 pm

I don’t quite understand how these four items are principles. What is the principle of Inheritance? Always use inheritance. What is the principle of Encapsulation? Wouldn’t a principle of OO provide guidance on how the concepts of OO are used.

For example, the principles: encapsulate variation, tell don’t ask, favor composition over inheritance.



dd says:

February 16, 2010 at 1:44 pm

I don't think mutators are a major principle of OO. In fact, from OO's point of view they violate encapsulation. Mutators are a principle of component based programming and a compromise w/OO principles.



jrroch says:

February 2, 2010 at 3:09 pm

Great article. Loved the polymorphism section, brought a greater understanding of it for me. Thank you.



nkubiser@yahoo.fr says:

April 2, 2009 at 9:17 am

how i can used the different codes in java



nkubiser@yahoo.fr says:

April 2, 2009 at 9:16 am

how we can make the the full program in java



TrickDaddy says:

April 10, 2008 at 2:20 pm

Raymond, for your piece of mind, look back at what Fransisco said and I believe you'll see that he didn't rip on your examples as being "bad", he said you lacked showing us some Bad Examples to compare to good ones. That is a great learning tool (seeing why something is bad and how to correct it). So I wouldn't take that as a bash on what you did do here.



Aphrodite says:

March 1, 2008 at 2:08 pm

im a bit confused..isnt Encapsulation the ability to hold similat information together (not information hiding) and abstraction is the ability to hide data which is achieved via encapsulation ?



Sundar says:

December 2, 2007 at 6:04 am

EXCELLENT WORK, FOR THOSE WHO ARE NEW TO THE COMPUTER FIELD, AND WISHES TO LEARN THE LANGUAGE.



Dev says:

December 1, 2007 at 4:32 pm

Hi Raymond,

It is really an excellent article that I have ever seen online. Basically the way explained takes to the bottom of understanding thats what made me understand how they can be implemented.

Could you help me with an online source paid or free or any DVD course that I could buy for a complete sample application (with out the front end I can take care of the front end) that takes a small database that has few tables in it and then creating the .NET objects using these 4 OOP prinicples based on the database we have in place.

I will look forward to hear from you.

Thank you in advance,

Dev

contact info:

movvap@yahoo.com

313 598 1446



Imran Abdulla says:

November 15, 2007 at 10:37 pm

Excellent just enough for refreshing/ brushing up on OOP concepts. I really liked the examples especially on polymorphism



Mullaiselvan says:

August 20, 2007 at 11:33 am

this artical is good. but the example ? . thanks for a good explanation



RAZEL says:

February 15, 2007 at 12:37 pm

HAY



RAZEL says:

February 15, 2007 at 12:34 pm

HAY



lewis says:

January 19, 2007 at 11:17 am

Raymond, u out did yourself on this one, leave the haters alone, why don't they post something paramount to programmers like this. in the years i have been programming this is the best article on OO i have come across including detailed examples!

lewomaniac@yahoo.com

Thanks Dude.



DIYguy says:

September 30, 2006 at 9:35 pm

Many thanks for taking time to explane basics so simply, using a library.

I have struggled to write an accounts package in Access, now Im going further and want a good grasp of this OOP way of thinking.

diyguys@hotmail.com



Bipul says:

July 16, 2006 at 8:25 pm

I have nothing to say about the above 4 OOP concepts, Its really very nice. In my 4 years exp. in .net I have never learnt this way. I think this is very usefull for every programmer(s) life and may be the easyest way to understand OOP concepts wth ur examples.

Thanks again.

Bipul
(mbip2000@yahoo.com)



[rlewallen](#) says:

July 7, 2006 at 9:50 pm

Yudhi, I've never read that book. I call them principles. They call them basics. Either way, its important to understand what makes up the high level foundation of OOP.



Yudhi Widyatama says:

July 2, 2006 at 1:22 am

Err.. I'm a bit confused about which is which. O'Reilly's Head First Design Patterns said that the things you wrote were named OO Basics. Under the OO Principles there are entries "Encapsulate what varies", "Favor composition over inheritance", "Program to interfaces, not implementation". So, which is which?



[MADHU](#) says:

June 16, 2006 at 4:16 pm

CAN ADD SOME SELF TESTS OR SO TO TEST THE KNOWLEDGE GAINED. WILL BE STILL MORE HELPFUL.



[MADHU](#) says:

June 16, 2006 at 4:10 pm

nice site. but will be more helpfull if it goes on related deep and deep thro links if needed.



[Xanax](#) says:

May 15, 2006 at 7:01 pm

Good site. But we must distinguish when a thing exists potentially and when it does not; for it is not at any and every time.



[rlewallen](#) says:

April 15, 2006 at 3:36 pm

Francisco, would you care to elaborate why I have provided bad design examples? If you're going to criticize what I publish, you should at least explain why my examples are bad, otherwise your comment is meaningless and without merit.



Francisco says:

April 14, 2006 at 8:51 pm

With all my respect and bad english... this article lacks of really bad design examples.



[Raymond Lewallen](#) says:

August 24, 2005 at 9:33 pm

Sahil, if anybody buy's your book because they saw it on this post, you owe me money 😊

**sahilmalik** says:

August 24, 2005 at 7:40 pm

I really like the inheritance section. Nice nice !!

**vasu vasili** says:

July 29, 2005 at 4:23 pm

Good article Ray it was good brush up the basics one more time,especially liked the polymorphism section.

I don't know who maintains code better but I have a small suggestion: When u click the link post comment it takes the user to top of the page. I found a wonder full control by Steve Stucher

sstchur:SmartScroller. Its really good control for maintaining the page position for the user.

Vasu

Leave a Reply

You must be [logged in](#) to post a comment.

[Home](#)[About](#)[CodeBetter CI](#)[Community](#)[Editors](#)[Search Results](#)

Friends of CodeBetter.Com

[Red-Gate Tools For SQL and .NET](#)[Telerik .NET Tools](#)[JetBrains - ReSharper](#)[Beyond Compare](#)[NDepend](#)[Ruby In Steel](#)[SlickEdit](#)[SmartInspect .NET Logging](#)[NGEDIT: ViEmu and Codekana](#)[DevExpress](#)[NHibernate Profiler](#)[Balsamiq Mockups](#)[Scrumy](#)[Umbraco](#)[NServiceBus](#)[RavenDb](#)[Web Sequence Diagrams](#)[Ducksboard](#)

CodeBetter.Com © '18

Stuff you need to Code Better!

Proudly powered by [WordPress](#).