

Forum:[Hack Forums](#)

Thread: [Extend your type with operators](#)

Barret - November 12, 2017, 12:21 pm

A little tutorial that might expand your knowledge and save some time, when you create a type and you can in addition create operator functionality to it.

For example, lets say we make our own version class that has two fields, major and minor numbers as integers.

```
Code Public Class Version
    Public Property Major As Integer
    Public Property Minor As Integer
    Sub New(major As Integer, minor As Integer)
        Me.Major = major
        Me.Minor = minor
    End Sub
End Class
```

Say you have two versions to compare, we would normally do

```
Code If (v1.Major = v2.Major And v1.Minor = v2.Minor) Then
    '...code...
End If
```

But you can simplify this by defining the operators and methods for it, as follows

```
Code If (v1 = v2) Then
    '...code...
End If
```

For this to work we need to define the custom comparison extensions, as follows

```
Code Public Class Version
    Public Property Major As Integer
    Public Property Minor As Integer
    Sub New(major As Integer, minor As Integer)
        Me.Major = major
        Me.Minor = minor
    End Sub
    Public Shared Operator +(a As Version, value As Integer) As Version
        Return New Version(a.Major + value, a.Minor + value)
    End Operator
    Public Shared Operator -(a As Version, value As Integer) As Version
        Return New Version(a.Major - value, a.Minor - value)
    End Operator
    Public Shared Operator +(a As Version, b As Version) As Version
        Return New Version(a.Major + b.Major, a.Minor + b.Minor)
    End Operator
    Public Shared Operator -(a As Version, b As Version) As Version
        Return New Version(a.Major - b.Major, a.Minor - b.Minor)
    End Operator
```

```

Public Shared Operator <>(a As Version, b As Version) As Boolean
    Return a.Major <> b.Major AndAlso a.Minor <> b.Minor
End Operator
Public Shared Operator =(a As Version, b As Version) As Boolean
    Return a.Major = b.Major AndAlso a.Minor = b.Minor
End Operator
Public Shared Operator >(a As Version, b As Version) As Boolean
    Return a.Major > b.Major Or a.Major = b.Major AndAlso a.Minor > b.Minor
End Operator
Public Shared Operator <(a As Version, b As Version) As Boolean
    Return a.Major < b.Major Or a.Major = b.Major AndAlso a.Minor < b.Minor
End Operator
Public Shared Operator >=(a As Version, b As Version) As Boolean
    Return a.Major >= b.Major AndAlso a.Minor >= b.Minor
End Operator
Public Shared Operator <=(a As Version, b As Version) As Boolean
    Return a.Major >= b.Major AndAlso a.Minor >= b.Minor
End Operator
End Class

```

Now this type supports the following formats:

Code version += 1

version1 += version2

version3 = version1 - version2

Remark:

Operator methods in these context come in pairs, one cannot exist without the other by design., + and - , > and < and so on...

Usage:

These extensions often come in handy with vector, version, point(f) and size(f) classes.

msdn entry:

<https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/procedures/how-to-define-an-operator>

+Haze - November 12, 2017, 1:56 pm

Operator overloading is one of the many things that make .NET sexy

0xDEAD10CC - November 12, 2017, 2:01 pm

Adding vectors makes sense, and is a great example for overloading operators, but I can't see what it would mean to add/subtract versions in the real world. Version comparisons are most common. Another acceptable implementation that would make sense for overloaded operators is for Matrix operations (multiplication, etc...). Operators are alright in most mathematical situations, but sometimes discouraged if they have ambiguous meaning. It is encouraged for value types to overload equality operators because there is no default, and your version class (like the .NET standard one), could be a struct with immutable properties.

You should indicate some of the guidelines with overloading a few of those operators too:

<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/equality-operators>

Equals() should have the same semantics for example. There's also things you shouldn't do like

overload equality operators on mutable reference types for instance.

edit: Decompiled (could have gone to referencesource instead...):

```
Code using System.Globalization;
using System.Runtime.InteropServices;
using System.Text;
```

```
namespace System
{
    public sealed class Version : ICloneable, IComparable, IComparable<Version>,
IEquatable<Version>
    {
        private static readonly char[] SeparatorsArray = new char[1] {'.'};

        private int _Build = -1;
        private int _Revision = -1;
        private int _Major;
        private int _Minor;
        private const int ZERO_CHAR_VALUE = 48;

        public Version(int major, int minor, int build, int revision)
        {
            if (major < 0)
                throw new ArgumentOutOfRangeException(nameof(major),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            if (minor < 0)
                throw new ArgumentOutOfRangeException(nameof(minor),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            if (build < 0)
                throw new ArgumentOutOfRangeException(nameof(build),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            if (revision < 0)
                throw new ArgumentOutOfRangeException(nameof(revision),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            this._Major = major;
            this._Minor = minor;
            this._Build = build;
            this._Revision = revision;
        }

        public Version(int major, int minor, int build)
        {
            if (major < 0)
                throw new ArgumentOutOfRangeException(nameof(major),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            if (minor < 0)
                throw new ArgumentOutOfRangeException(nameof(minor),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            if (build < 0)
                throw new ArgumentOutOfRangeException(nameof(build),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
            this._Major = major;
            this._Minor = minor;
```

```

        this._Build = build;
    }

    public Version(int major, int minor)
    {
        if (major < 0)
            throw new ArgumentOutOfRangeException(nameof(major),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
        if (minor < 0)
            throw new ArgumentOutOfRangeException(nameof(minor),
Environment.GetResourceString("ArgumentOutOfRange_Version"));
        this._Major = major;
        this._Minor = minor;
    }

    public Version(string version)
    {
        Version version1 = Version.Parse(version);
        this._Major = version1.Major;
        this._Minor = version1.Minor;
        this._Build = version1.Build;
        this._Revision = version1.Revision;
    }

    public Version()
    {
        this._Major = 0;
        this._Minor = 0;
    }

    public int Major { { return this._Major; } }
    public int Minor { { return this._Minor; } }
    public int Build { { return this._Build; } }
    public int Revision { { return this._Revision; } }
    public short MajorRevision { { return (short)(this._Revision >> 16); } }
    public short MinorRevision { { return (short)(this._Revision & (int) ushort.MaxValue); } }

    public object Clone()
    {
        return (object) new Version()
        {
            _Major = this._Major,
            _Minor = this._Minor,
            _Build = this._Build,
            _Revision = this._Revision
        };
    }

    public int CompareTo(object version)
    {
        if (version == null)
            return 1;
        Version version1 = version as Version;

```

```

    if (version1 == (Version) null)
        throw new ArgumentException(Environment.GetResourceString("Arg_MustBeVersion"));
    if (this._Major != version1._Major)
        return this._Major > version1._Major ? 1 : -1;
    if (this._Minor != version1._Minor)
        return this._Minor > version1._Minor ? 1 : -1;
    if (this._Build != version1._Build)
        return this._Build > version1._Build ? 1 : -1;
    if (this._Revision == version1._Revision)
        return 0;
    return this._Revision > version1._Revision ? 1 : -1;
}

public int CompareTo(Version value)
{
    if (value == (Version) null)
        return 1;
    if (this._Major != value._Major)
        return this._Major > value._Major ? 1 : -1;
    if (this._Minor != value._Minor)
        return this._Minor > value._Minor ? 1 : -1;
    if (this._Build != value._Build)
        return this._Build > value._Build ? 1 : -1;
    if (this._Revision == value._Revision)
        return 0;
    return this._Revision > value._Revision ? 1 : -1;
}

public override bool Equals(object obj)
{
    Version version = obj as Version;
    return !(version == (Version) null) && this._Major == version._Major && (this._Minor ==
version._Minor && this._Build == version._Build) && this._Revision == version._Revision;
}

public bool Equals(Version obj)
{
    return !(obj == (Version) null) && this._Major == obj._Major && (this._Minor == obj._Minor
&& this._Build == obj._Build) && this._Revision == obj._Revision;
}

public override int GetHashCode()
{
    return 0 | (this._Major & 15) << 28 | (this._Minor & (int) byte.MaxValue) << 20 | (this._Build &
(int) byte.MaxValue) << 12 | this._Revision & 4095;
}

public override string ToString()
{
    if (this._Build == -1)
        return this.ToString(2);
    if (this._Revision == -1)
        return this.ToString(3);
}

```

```

        return this.ToString(4);
    }

    public string ToString(int fieldCount)
    {
        switch (fieldCount)
        {
            case 0:
                return string.Empty;
            case 1:
                return this._Major.ToString();
            case 2:
                StringBuilder sb1 = StringBuilderCache.Acquire(16);
                Version.AppendPositiveNumber(this._Major, sb1);
                sb1.Append('.');
                Version.AppendPositiveNumber(this._Minor, sb1);
                return StringBuilderCache.GetStringAndRelease(sb1);
            default:
                if (this._Build == -1)
                    throw new
ArgumentException(Environment.GetResourceString("ArgumentOutOfRange_Bounds_Lower_Uppe
r", (object) "0", (object) "2"), nameof(fieldCount));
                if (fieldCount == 3)
                {
                    StringBuilder sb2 = StringBuilderCache.Acquire(16);
                    Version.AppendPositiveNumber(this._Major, sb2);
                    sb2.Append('.');
                    Version.AppendPositiveNumber(this._Minor, sb2);
                    sb2.Append('.');
                    Version.AppendPositiveNumber(this._Build, sb2);
                    return StringBuilderCache.GetStringAndRelease(sb2);
                }
                if (this._Revision == -1)
                    throw new
ArgumentException(Environment.GetResourceString("ArgumentOutOfRange_Bounds_Lower_Uppe
r", (object) "0", (object) "3"), nameof(fieldCount));
                if (fieldCount == 4)
                {
                    StringBuilder sb2 = StringBuilderCache.Acquire(16);
                    Version.AppendPositiveNumber(this._Major, sb2);
                    sb2.Append('.');
                    Version.AppendPositiveNumber(this._Minor, sb2);
                    sb2.Append('.');
                    Version.AppendPositiveNumber(this._Build, sb2);
                    sb2.Append('.');
                    Version.AppendPositiveNumber(this._Revision, sb2);
                    return StringBuilderCache.GetStringAndRelease(sb2);
                }
                throw new
ArgumentException(Environment.GetResourceString("ArgumentOutOfRange_Bounds_Lower_Uppe
r", (object) "0", (object) "4"), nameof(fieldCount));
        }
    }
}

```

```

private static void AppendPositiveNumber(int num, StringBuilder sb)
{
    int length = sb.Length;
    do
    {
        int num1 = num % 10;
        num /= 10;
        sb.Insert(length, (char)(48 + num1));
    }
    while (num > 0);
}

public static Version Parse(string input)
{
    if (input == null)
        throw new ArgumentNullException(nameof(input));
    Version.VersionResult result = new Version.VersionResult();
    result.Init(nameof(input), true);
    if (!Version.TryParseVersion(input, ref result))
        throw result.GetVersionParseException();
    return result.m_parsedVersion;
}

public static bool TryParse(string input, out Version result)
{
    Version.VersionResult result1 = new Version.VersionResult();
    result1.Init(nameof(input), false);
    bool version = Version.TryParseVersion(input, ref result1);
    result = result1.m_parsedVersion;
    return version;
}

private static bool TryParseVersion(string version, ref Version.VersionResult result)
{
    if (version == null)
    {
        result.SetFailure(Version.ParseFailureKind.ArgumentNullException);
        return false;
    }
    string[] strArray = version.Split(Version.SeparatorsArray);
    int length = strArray.Length;
    if (length < 2 || length > 4)
    {
        result.SetFailure(Version.ParseFailureKind.ArgumentException);
        return false;
    }
    int parsedComponent1;
    int parsedComponent2;
    if (!Version.TryParseComponent(strArray[0], nameof(version), ref result, out
parsedComponent1) || !Version.TryParseComponent(strArray[1], nameof(version), ref result, out
parsedComponent2))
        return false;
}

```

```

int num = length - 2;
if (num > 0)
{
    int parsedComponent3;
    if (!Version.TryParseComponent(strArray[2], "build", ref result, out parsedComponent3))
        return false;
    if (num - 1 > 0)
    {
        int parsedComponent4;
        if (!Version.TryParseComponent(strArray[3], "revision", ref result, out
parsedComponent4))
            return false;
        result.m_parsedVersion = new Version(parsedComponent1, parsedComponent2,
parsedComponent3, parsedComponent4);
    }
    else
        result.m_parsedVersion = new Version(parsedComponent1, parsedComponent2,
parsedComponent3);
}
else
    result.m_parsedVersion = new Version(parsedComponent1, parsedComponent2);
return true;
}

```

```

private static bool TryParseComponent(string component, string componentName, ref
Version.VersionResult result, out int parsedComponent)
{
    if (!int.TryParse(component, NumberStyles.Integer, (IFormatProvider)
CultureInfo.InvariantCulture, out parsedComponent))
    {
        result.SetFailure(Version.ParseFailureKind.FormatException, component);
        return false;
    }
    if (parsedComponent >= 0)
        return true;
    result.SetFailure(Version.ParseFailureKind.ArgumentOutOfRangeException,
componentName);
    return false;
}

```

```

public static bool operator ==(Version v1, Version v2)
{
    if ((object) v1 == null)
        return (object) v2 == null;
    return v1.Equals(v2);
}

```

```

public static bool operator !=(Version v1, Version v2)
{
    return !(v1 == v2);
}

```

```

public static bool operator <(Version v1, Version v2)

```

```

{
    if ((object) v1 == null)
        throw new ArgumentNullException(nameof(v1));
    return v1.CompareTo(v2) < 0;
}

public static bool operator <=(Version v1, Version v2)
{
    if ((object) v1 == null)
        throw new ArgumentNullException(nameof(v1));
    return v1.CompareTo(v2) <= 0;
}

public static bool operator >(Version v1, Version v2)
{
    return v2 < v1;
}

public static bool operator >=(Version v1, Version v2)
{
    return v2 <= v1;
}

internal enum ParseFailureKind
{
    ArgumentNullException,
    ArgumentException,
    ArgumentOutOfRangeException,
    FormatException,
}

internal struct VersionResult
{
    internal Version m_parsedVersion;
    internal Version.ParseFailureKind m_failure;
    internal string m_exceptionArgument;
    internal string m_argumentName;
    internal bool m_canThrow;

    internal void Init(string argumentName, bool canThrow)
    {
        this.m_canThrow = canThrow;
        this.m_argumentName = argumentName;
    }

    internal void SetFailure(Version.ParseFailureKind failure)
    {
        this.SetFailure(failure, string.Empty);
    }

    internal void SetFailure(Version.ParseFailureKind failure, string argument)
    {
        this.m_failure = failure;
    }
}

```

Here's an example on how the .NET Version class overloads the operators and relevant methods and opposing operators for appropriate semantics. (Sorry C#.)

(Nov 12, 2017 03:01 PM)0xDEAD10CC Wrote: [CITE](#) Adding vectors makes sense, and is a great example for overloading operators, but I can't see what it would mean to add/subtract versions in the real world. Version comparisons are most common. Another acceptable implementation that would make sense for overloaded operators is for Matrix operations (multiplication, etc...). Operators are alright in most mathematical situations, but sometimes discouraged if they have ambiguous meaning. It is encouraged for value types to overload equality operators because there is no default, and your version class (like the .NET standard one), could be a struct with immutable properties.

You should indicate some of the guidelines with overloading a few of those operators too:
<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/equality-operators>*sigh*.... you make me tired, im not gonna put effort into that.

DBots - November 13, 2017, 1:53 pm

Never knew you could implement this, interesting to see.

Very often I find myself comparing properties from same type objects and this would make it look cleaner.

Thank for the knowledge.

Ninja - November 13, 2017, 2:38 pm

Wait. People did not know they could overload operators? Or am I missing something?

0xDEAD10CC - November 13, 2017, 3:27 pm

Arithmetic and binary operator overloading should be pretty basic, more people probably don't know that they can define conversion operators though. (

<https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/procedures/how-to-define-a-conversion-operator>)

Sn0wX - November 15, 2017, 10:37 am

(Nov 12, 2017 03:01 PM)0xDEAD10CC Wrote: [CITE](#) edit: Decompiled (could have gone to referencesource instead...):

Code edited

```
}

internal Exception GetVersionParseException()
{
    switch (this.m_failure)
    {
        case Version.ParseFailureKind.ArgumentNullException:
            return (Exception) new ArgumentNullException(this.m_argumentName);
        case Version.ParseFailureKind.ArgumentException:
            return (Exception) new
ArgumentException(Environment.GetResourceString("Arg_VersionString"));
        case Version.ParseFailureKind.ArgumentOutOfRangeException:
            return (Exception) new ArgumentOutOfRangeException(this.m_exceptionArgument,
Environment.GetResourceString("ArgumentOutOfRange_Version"));
        case Version.ParseFailureKind.FormatException:
            try
            {
                int.Parse(this.m_exceptionArgument, (IFormatProvider)
CultureInfo.InvariantCulture);
            }
            catch (FormatException ex)
            {
                return (Exception) ex;
            }
            catch (OverflowException ex)
            {
                return (Exception) ex;
            }
            return (Exception) new
```

```
FormatException(Environment.GetResourceString("Format_InvalidString"));
        default:
            return (Exception) new
ArgumentException(Environment.GetResourceString("Arg_VersionString"));
    }
}
}
}
```

Here's an example on how the .NET Version class overloads the operators and relevant methods and opposing operators for appropriate semantics. (Sorry C#.)

Jesus learn how to handle exceptions. Why you even throwing an exception here..... You are abusing the hell out of it when can be validated. It's used UNAVOIDABLE BUT NOT UNEXPECTED.

It's laugh worthy as you are throwing it up the stack yet it's not needed to help this question.

0xDEAD10CC - November 15, 2017, 11:13 am

(Nov 15, 2017 11:37 AM)Sn0wX Wrote: [\[CITE\]](#) Jesus learn how to handle exceptions. Why you even throwing an exception here..... You are abusing the hell out of it when can be validated. It's used UNAVOIDABLE BUT NOT UNEXPECTED.

It's laugh worthy as you are throwing it up the stack yet it's not needed to help this question.
 ? You attempt to mock me as clueless as you are, without even knowing that it's the .NET framework code and not mine... And throwing meaningful exceptions is part of proper framework design, are you a fool? This is the decompiled .NET source for System.Version... It's proper design to throw meaningful exceptions when necessary.

It's laugh worthy that you even try to mock me without even having a full understanding of what's going on in this thread. Come back when you understand proper framework design, and next time read the thread you moron. I posted that decompiled code to show how System.Version overloads operators.

Refernece: <http://referencesource.microsoft.com/#mscorlib/system/version.cs,0c15515dfe051d35>

Make yourself look like a bigger fool and tell the .NET team that they are wrong. :)

+Haze - November 15, 2017, 1:38 pm

somebody got rekt on this thread and i'm not going to say who it is

Ninja - November 15, 2017, 1:48 pm

(Nov 15, 2017 02:38 PM)+Haze Wrote: [\[CITE\]](#) somebody got rekt on this thread and i'm not going to say who it is

It's already obvious who it is. No need to kick a dead horse.

+Haze - November 15, 2017, 2:13 pm

(Nov 15, 2017 02:48 PM)Ninja Wrote: [\[CITE\]](#) It's already obvious who it is. No need to kick a dead horse.

I guess you didn't get the jo... Nvm

Ninja - November 15, 2017, 2:33 pm

(Nov 15, 2017 03:13 PM)+Haze Wrote: [\[CITE\]](#) I guess you didn't get the jo... Nvm

I got the joke.