Visual Basic and C# Breaking Changes from Visual Studio 2005 to Visual Studio 2008

November 2007

For the latest information, please see [www.microsoft.com/vstudio](http://www.microsoft.com/vstudio)

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarked, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2007 Microsoft Corporation. All rights reserved.

Microsoft, MS-DOS, MS, Windows, Windows NT, MSDN, Active Directory, BizTalk, SQL Server, SharePoint, Outlook, PowerPoint, FrontPage, Visual Basic, Visual C++, Visual J++, Visual InterDev, Visual SourceSafe, Visual C#, Visual J#, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Visual Basic and C# Breaking Changes from Visual Studio 2005 to Visual Studio 2008

Breaking changes are changes that could make existing code behave differently when compiled under the latest compilers shipped with Visual Studio 2008. While every effort is made to avoid breaking changes, sometimes such changes are necessary, either for security reasons or to fix issues with clearly erroneous code generation. In virtually all cases where we make a breaking change, we believe the change will impact very few users.

# Visual Basic Changes

## Overload resolution change with generic and ParamArray parameters

The goal of overload resolution is to try to pick the method that most closely matches the types of the parameters passed to the method.  In the example below, Foo is overloaded across an inheritance hierarchy, and is called with the parameters (Integer, Short).

Using VS2005’s overload resolution rules, type inference fails for the overload in the derived class (since Y cannot be both an Integer and a Short; exact matches are required).  As a result the base class overload is used, and T is inferred as Integer and Short widens into the ParamArray x.  VS2008 uses a new algorithm that in general picks the same overloads, except for this specific case.  In VS2008, the “dominant type” is Integer (since Short widens to Integer), and thus type parameter Y is inferred to be Integer.  Overload resolution rules stipulate that direct matches (in this case (Integer, Integer)) are better matches than those found requiring ParamArray parameters.  As a result in VS2008, the version of Foo in the derived class is called.

Module Module1

    Class C1

        Sub Foo(Of T)(ByVal arg1 As T, ByVal ParamArray x() As Integer)

            Console.WriteLine("In VS2005 we call Foo in the \*base\* class")

        End Sub

    End Class

    Class C2 : Inherits C1

        Overloads Sub Foo(Of Y)(ByVal arg1 As Y, ByVal x As Y)

            Console.WriteLine("In VS2008 we call Foo in the \*derived\* class")

        End Sub

    End Class

    Sub Main()

        Dim a As New C2

        Dim b As Integer

        Dim c As Short

        a.Foo(b, c)

    End Sub

End Module

### To resolve this issue

The workaround is to cast a to type C1 explicitly before calling Foo, which will result in calling the overload on the base class.  Alternatively parameter c could be passed in as an array which would also cause the base class overload to be picked.

## Overload resolution change between generic and non-generic classes

In VS2005 there was a bug that (in certain rare instances) caused overload resolution to behave differently depending on whether or not the class was generic.  In the example below both C1 and C2 are exactly identical except that C2 has a generic parameter defined on it.  The call to x.Foo is ambiguous since the parameter passed in could legally bind to either overload of Foo.  The call to y.Foo likewise should have generated the same error in VS2005, but it did not and so the method bound to the unconstrained overload of Foo (i.e. the first one).

This issue is now fixed in VS2008 so both calls now generate compile errors due to the ambiguity.

Class C1

    Sub Foo(Of T)(ByVal x As T)

    End Sub

    Sub Foo(Of T As Structure)(ByVal x As Nullable(Of T))

    End Sub

End Class

Class C2(Of U)

    Sub Foo(Of T)(ByVal x As T)

    End Sub

    Sub Foo(Of T As Structure)(ByVal x As Nullable(Of T))

    End Sub

End Class

Module M1

    Sub Main()

        Dim x As New C1

        Dim y As New C2(Of String)

        x.Foo(New Nullable(Of Integer)) 'Error in both VS2005 and VS2008

        y.Foo(New Nullable(Of Integer)) 'In VS2008 this is now an error

    End Sub

End Module

### To resolve this issue

The workaround is to either change the overloads so they’re no longer ambiguous, or specify the type arguments explicitly:

x.Foo(Of Integer?)(New Nullable(Of Integer))

## T widening to T? is now treated as a predefined conversion

In VS2005 a user-defined conversion could be created to allow T to widen to T? (Nullable(Of T)).  With the new nullable support in VS2008 this support is now intrinsic, and thus this conversion already exists as a predefined conversion in the language.

Overload resolution considers both predefined and user-defined conversions, so if both exist there is a potential ambiguity.  This means that code that contains a user-defined widening conversion from T to T? is now potentially ambiguous.  In the example below, the call to y.Bar works in VS2005, but in VS2008 the new rules result in an ambiguity.

Class Foo

    Public Shared Widening Operator CType(ByVal x As Nullable(Of Short)) As Foo

        Return New Foo()

    End Operator

End Class

Class Foo2

    Public Shared Widening Operator CType(ByVal x As Integer) As Foo2

        Return New Foo2()

    End Operator

End Class

Class C1

    Sub Bar(ByVal x As Foo2)

    End Sub

    Sub Bar(ByVal x As Foo)

    End Sub

End Class

Module M1

    Sub Main()

        Dim y As New C1

        Dim x As Short = 2

        y.Bar(x) 'Compiles in VS2005, now an error in VS2008

    End Sub

End Module

### To resolve this issue

The workaround is to use an explicit cast to either Foo or Foo2, or remove the redundant user-defined conversion and instead allow overload resolution to use the predefined one.

# Visual C# Changes

## Definite assignment rules changed for if statements

Definite assignment rules require that local variables be assigned a value before they are used. The compiler does this by finding an assignment in the method before the use of the variable that it can guarantee will always be run. When assignment statements are included inside **if** blocks, code flow analysis is performed to determine whether the **if** condition will always be **true** or always be **false**. If the compiler determines that the **if** statement will always branch in one direction, it can be assumed that the assignments in that branch will always be run.

There are some **if** statement conditions that were treated as being always **true** or always **false** in earlier versions of the compiler that are no longer treated as such. If you have any one of the following **if** conditions in your code, assignments inside those **if** blocks will no longer be considered to always run, and you may receive error CS0165.

class Program

{

 public static int Main()

 {

 int i1, i2, i3, i4, i5, i6;

 if (null as object == null)

 i1 = 1;

 if (!(null is object))

 i2 = 1;

 int j3 = 0;

 if ((0 == j3 \* 0) && (0 == 0 \* j3))

 i3 = 1;

 int j4 = 0;

 if ((0 == (j4 & 0)) && (0 == (0 & j4)))

 i4 = 1;

 int? j5 = 1;

 if (j5 == j5)

 i5 = 1;

 if (null == (int?)(null ?? null))

 i6 = 1;

 System.Console.WriteLine("{0}{1}{2}{3}{4}{5}",

 i1, i2, i3, i4, i5, i6);

 return 1;

 }

}

### To resolve this issue

Remove the surrounding **if** statement or move the assignment outside of the **if** statement.

## Inline assignment on the right side of a null coalescing operator no longer causes the variable to be considered definitely assigned

In the previous version of the compiler, an inline assignment to a variable on the right side of a null coalescing operator always causes the variable to be considered definitely assigned. This is true even if the operator’s left side may be non-**null**. The variable could then be accessed, even though it might not always be set. This now causes compiler error CS0165.

public class MyClass

{

 public int member;

}

public class Program

{

 static void Method(MyClass c) { }

 static void Main()

 {

 MyClass mc1;

 MyClass mc2 = new MyClass();

 Bar(mc2 ?? (mc1 = new MyClass()));

 System.Console.WriteLine(mc1.member);

 }

}

### To resolve this issue

Move the assignment out of the null coalescing expression to a location where it will always be executed.

## Inline assignment now occurs for assignment of null to nullable variable on the left side of the null coalescing operator

In the previous version of the compiler, an inline assignment to null of a nullable variable on the left side of the null coalescing operator (**??**) did not actually occur. The nullable variable instead retained its original value. The null assignment side effect is now correctly evaluated in this case.

class Program

{

 static void Main(string[] args)

 {

 int? b;

 b = 123;

 System.Console.WriteLine((b = null) ?? 17);

 System.Console.WriteLine(b == null);

 }

}

### To resolve this issue

Remove the assignment, and replace the null coalescing expression with its right operand.

## Ambiguous method invocations with anonymous method parameters are not allowed

Method invocations on overloaded methods must be resolved by the compiler to determine which specific overload to invoke. When an invocation’s parameter type is partially inferred, the specific overload to invoke can become ambiguous. This causes a compiler error.

In the case of an anonymous method being passed as a delegate parameter, the anonymous method’s delegate type is partially inferred. This can lead to ambiguity when the compiler is selecting the correct overload. In the previous version of the compiler, this ambiguity does not cause a compiler error. Method invocations that are ambiguous because of inferred anonymous method parameter types now cause error CS0121.

class Program

{

 delegate int D(int x);

 delegate T D<T>(T x);

 static int F(D d1) { return 0; }

 static int F<T>(D<T> d1t) { return 1; }

 static void Main()

 {

 int i = F(delegate(int x) { return 2; });

 System.Console.WriteLine(i);

 }

}

### To resolve this issue

Assign the anonymous method to an explicitly typed delegate variable in a previous statement, or add a cast to a specific delegate type in the method invocation.

## All constant expressions equal to 0 are now implicitly convertible to enum types

A literal 0 is implicitly convertible to any **enum** type. In earlier versions of the compiler, there were also some constant expressions which evaluate to 0 that could implicitly convert to any **enum** type, but the rule deciding which of these expressions were convertible was unclear. Now, all constant expressions that are equal to 0 can be implicitly converted to any **enum** type.

This could cause some changes in the behavior of existing code, such as method overload resolution that relies on the lack of this implicit conversion. The following code compiles successfully on earlier compilers, resolving the Method invocation on the **short** value only to the **int** overload. Now, this invocation is ambiguous, as the **short** value is also implicitly convertible to **MyEnum**.

enum MyEnum { }

class Program

{

 static void Method(MyEnum e)

 {

 System.Console.WriteLine("Method(MyEnum e)");

 }

 static void Method(int i)

 {

 System.Console.WriteLine("Method(int i)");

 }

 const short Zero = 0;

 static void Main(string[] args)

 {

 Method(Program.Zero);

 }

}

### To resolve this issue

Modify any existing code that relies on expressions evaluating to literal zero not being convertible to an **enum** type.

## Assignment of an out-of-range decimal literal to an integral type is not allowed

Earlier versions of the compiler allowed the assignment of decimal literals to variables of integral types, even if the decimal value was out of the range of the destination type. This now causes compiler error CS0031 in both checked and unchecked contexts.

public class Program

{

 public static void Main()

 {

 byte b = (byte)256M;

 }

}

### To resolve this issue

Remove the assignment or change the type of the variable.

## Accessing elements of an unsafe struct’s fixed array field before the struct pointer is assigned is not allowed

Definite assignment rules for unsafe pointers require that the pointer be set before dereferencing the pointer. In earlier versions of the compiler, the pointer was not required to be set before you could use the **->** operator to access an element of a fixed buffer field on the dereferenced struct. This now causes compiler error CS0165.

unsafe class Program

{

 static void Main()

 {

 S\* ps;

 ps->i[0]++;

 }

}

unsafe struct S

{

 public fixed int i[10];

}

### To resolve this issue

Assign a value to the struct pointer before you try to dereference the pointer and access its members.

## Arrays of unsafe pointers to reference types are not allowed

Unsafe pointers to reference types are not allowed, and they cause compiler errors. In earlier versions of the compiler, there are cases in which an array of unsafe pointers to reference types does not generate a compiler error. These cases now generate compiler error CS0208.

unsafe class Stuff

{

 static Stuff\*[] x = { };

 static void Method1(Stuff\*[] arr)

 {

 }

 static Stuff\*[] Method2()

 {

 return x;

 }

}

### To resolve this issue

Change the reference type to a struct, or use managed references instead of pointers.

## Switch condition cannot be void

Previously, the compiler would allow a **void** method invocation to be used as a **switch** condition. This now causes compiler error CS0151.

class Program

{

 static void Main()

 {

 switch (M())

 {

 default:

 break;

 }

 }

 static void M()

 {

 }

}

### To resolve this issue

Eliminate the **switch** statement or replace the **switch** condition.

## Anonymous method parameters cannot have static types

Static types cannot be used as the types of method parameters because they cannot be instantiated. The previous version of the compiler allows static types as parameter types within delegates and anonymous method declarations. If you pass **null** as the parameter, such delegates can be invoked. Trying to declare a delegate that accepts a parameter of a static type now causes error CS0721.

public static class StaticClass { }

delegate int D(StaticClass f);

public class Program

{

 public static void Main()

 {

 D d = delegate(StaticClass f) { return 1; };

 System.Console.WriteLine(d(null));

 }

}

### To resolve this issue

Remove the parameter that has the static type.

## Duplicate type forwarder definitions are not allowed

Previously, the compiler allowed duplicate **TypeForwardedTo** attributes to be defined for a given type. This now causes compiler error CS0739.

// Source.cs (in SourceAssembly, which references TargetAssembly)

[assembly: System.Runtime.CompilerServices.TypeForwardedTo(typeof(Target))]

[assembly: System.Runtime.CompilerServices.TypeForwardedTo(typeof(Target))]

public class Program

{

 public static void Main()

 {

 Target = new Target();

 System.Console.WriteLine(t.getValue());

 }

}

// Target.cs (in TargetAssembly)

public class Target

{

 public int getValue()

 {

 return 0;

 }

}

### To resolve this issue

Remove the duplicate type forwarder definition.

## /pdb option not allowed when you are not using /debug

The **/debug** option causes the compiler to generate a .pdb file that contains debugging information with the same location and file name as the .exe or .dll file being generated. You can use the **/pdb** option to specify an alternative path and/or file name for the .pdb file.

Previously, you could use the **/pdb** option without the **/debug** option. This now causes error CS2036.

csc /pdb:DebugInfo Program.cs

### To resolve this issue

Remove the **/pdb** compiler option, or specify the **/debug** compiler option.

## Ambiguous interface implementation of methods from generic base classes now generates a warning

When a type with a generic base class implements an interface, there can be ambiguity in selecting between overloads of the relevant base class method if the types chosen for type parameters cause collisions. The resolution of this ambiguity by the runtime is undefined and implementation-dependent. In earlier versions of the compiler, this code compiles without issue. This situation now generates compiler warning CS1956.

class Base<T>

{

 public void Method(T x)

 {

 System.Console.WriteLine("Method(T x)");

 }

 public void Method(int x)

 {

 System.Console.WriteLine("Method(int x)");

 }

}

interface IFace

{

 void Method(int x);

}

class Derived : Base<int>, IFace

{

 static void Main()

 {

 IFace x = new Derived();

 int y = 3;

 x.Method(y);

 }

}

### To resolve this issue

Remove one of the ambiguous methods from the base class, or change the type parameter that is used so that no overload conflicts occur.

## Accessing members of a struct’s reference field before that field is assigned generates a warning

Definite assignment rules for structs require that either the struct be set to an existing instance of its type or that each of its members be assigned to before it is referenced. In earlier versions of the compiler, for structs that contain a field of a reference type, assignment of an instance to this field is not required before allowing that field’s members to be assigned values. This now causes compiler warning CS1060.

public class Class

{

 public int i;

}

public struct Struct

{

 public Class c;

}

public class Program

{

 static void Main()

 {

 Struct s;

 s.c.i = 0;

 }

}

### To resolve this issue

Explicitly set the reference field to an instance of its type before the assignment to that field’s members.

## Marking an override method as obsolete generates a warning

Methods can be marked with the **Obsolete** attribute to cause either errors or warnings at compile-time if the methods are invoked. When you place this attribute on virtual methods, the attribute must be placed on the base method. If the **Obsolete** attribute is placed on an override, it will not trigger compiler errors or warnings on invocation. Earlier versions of the compiler allowed code that placed the **Obsolete** attribute on an override method, even though the attribute did not take effect when placed there. This now causes compiler warning CS0809.

public class BaseClass

{

 protected string dataString;

 public virtual string DataString

 {

 get { return dataString; }

 set { dataString = value; }

 }

}

class DerivedClass : BaseClass

{

 [System.ObsoleteAttribute("Obsolete", false)]

 public override string DataString

 {

 get { return dataString; }

 set { dataString = value.ToUpper(); }

 }

}

class Program

{

 static void Main()

 {

 new DerivedClass().DataString = "abc123";

 }

}

### To resolve this issue

Remove the **Obsolete** attribute from the override method, or move the attribute to a base virtual method where it will have an effect.