Contents

Covariance and Contravariance (C#)

5/4/2018 • 3 minutes to read • Edit Online

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see Variance in Delegates (C#) and Using Variance in Delegates (C#).

The following code example shows covariance and contravariance support for method groups.

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

In .NET Framework 4 or newer C# supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see Variance in Generic Interfaces (C#) and Variance in Delegates (C#).

The following code example shows implicit reference conversion for generic interfaces.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. C# enables you to create your own variant interfaces and delegates. For more information, see Creating Variant Generic Interfaces (C#) and Variance in Delegates (C#).

Related Topics

TITLE	DESCRIPTION
Variance in Generic Interfaces (C#)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.
Creating Variant Generic Interfaces (C#)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (C#)	Shows how covariance and contravariance support in the IEnumerable<T> and IComparable<T> interfaces can help you reuse code.
Variance in Delegates (C#)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in the .NET Framework.
Using Variance in Delegates (C#)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (C#)	Shows how covariance and contravariance support in the `Func` and `Action` delegates can help you reuse code.

Variance in Generic Interfaces (C#)

1/23/2019 • 2 minutes to read • Edit Online

.NET Framework 4 introduced variance support for several existing generic interfaces. Variance support enables implicit conversion of classes that implement these interfaces. The following interfaces are now variant:

- IEnumerable<T> (T is covariant)
- IEnumerator<T> (T is covariant)
- IQueryable<T> (T is covariant)
- IGrouping<TKey,TElement> (`TKey` and `TElement` are covariant)
- IComparer<T> (T is contravariant)
- IEqualityComparer<T> (T is contravariant)
- IComparable<T> (T is contravariant)

Covariance permits a method to have a more derived return type than that defined by the generic type parameter of the interface. To illustrate the covariance feature, consider these generic interfaces: `IEnumerable<Object>` and `IEnumerable<String>`. The `IEnumerable<String>` interface does not inherit the `IEnumerable<Object>` interface. However, the `String` type does inherit the `Object` type, and in some cases you may want to assign objects of these interfaces to each other. This is shown in the following code example.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

In earlier versions of the .NET Framework, this code causes a compilation error in C# with `Option Strict On`. But now you can use `strings` instead of `objects`, as shown in the previous example, because the IEnumerable<T> interface is covariant.

Contravariance permits a method to have argument types that are less derived than that specified by the generic parameter of the interface. To illustrate contravariance, assume that you have created a `BaseComparer` class to compare instances of the `BaseClass` class. The `BaseComparer` class implements the `IEqualityComparer<BaseClass>` interface. Because the IEqualityComparer<T> interface is now contravariant, you can use `BaseComparer` to compare instances of classes that inherit the `BaseClass` class. This is shown in the following code example.

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

For more examples, see Using Variance in Interfaces for Generic Collections (C#).

Variance in generic interfaces is supported for reference types only. Value types do not support variance. For example, `IEnumerable<int>` cannot be implicitly converted to `IEnumerable<object>`, because integers are represented by a value type.

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler errror,
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

It is also important to remember that classes that implement variant interfaces are still invariant. For example, although List<T> implements the covariant interface IEnumerable<T>, you cannot implicitly convert `List<Object>` to `List<String>`. This is illustrated in the following code example.

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

See also

- Using Variance in Interfaces for Generic Collections (C#)
- Creating Variant Generic Interfaces (C#)
- Generic Interfaces
- Variance in Delegates (C#)

Creating Variant Generic Interfaces (C#)

1/23/2019 • 4 minutes to read • Edit Online

You can declare generic type parameters in interfaces as covariant or contravariant. *Covariance* allows interface methods to have more derived return types than that defined by the generic type parameters. *Contravariance* allows interface methods to have argument types that are less derived than that specified by the generic parameters. A generic interface that has covariant or contravariant generic type parameters is called *variant*.

NOTE

.NET Framework 4 introduced variance support for several existing generic interfaces. For the list of the variant interfaces in the .NET Framework, see Variance in Generic Interfaces (C#).

Declaring Variant Generic Interfaces

You can declare variant generic interfaces by using the `in` and `out` keywords for generic type parameters.

IMPORTANT

`ref`, `in`, and `out` parameters in C# cannot be variant. Value types also do not support variance.

You can declare a generic type parameter covariant by using the `out` keyword. The covariant type must satisfy the following conditions:

- The type is used only as a return type of interface methods and not used as a type of method arguments. This is illustrated in the following example, in which the type `R` is declared covariant.

  ```
  interface ICovariant<out R>
  {
      R GetSomething();
      // The following statement generates a compiler error.
      // void SetSometing(R sampleArg);

  }
  ```

  There is one exception to this rule. If you have a contravariant generic delegate as a method parameter, you can use the type as a generic type parameter for the delegate. This is illustrated by the type `R` in the following example. For more information, see Variance in Delegates (C#) and Using Variance for Func and Action Generic Delegates (C#).

  ```
  interface ICovariant<out R>
  {
      void DoSomething(Action<R> callback);
  }
  ```

- The type is not used as a generic constraint for the interface methods. This is illustrated in the following code.

```
interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic contstraints.
    // void DoSomething<T>() where T : R;
}
```

You can declare a generic type parameter contravariant by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a return type of interface methods. The contravariant type can also be used for generic constraints. The following code shows how to declare a contravariant interface and use a generic constraint for one of its methods.

```
interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

It is also possible to support both covariance and contravariance in the same interface, but for different type parameters, as shown in the following code example.

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSometings(A sampleArg);
}
```

Implementing Variant Generic Interfaces

You implement variant generic interfaces in classes by using the same syntax that is used for invariant interfaces. The following code example shows how to implement a covariant interface in a generic class.

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Classes that implement variant interfaces are invariant. For example, consider the following code.

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Extending Variant Generic Interfaces

When you extend a variant generic interface, you have to use the `in` and `out` keywords to explicitly specify whether the derived interface supports variance. The compiler does not infer the variance from the interface that is being extended. For example, consider the following interfaces.

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

In the `IInvariant<T>` interface, the generic type parameter `T` is invariant, whereas in `IExtCovariant<out T>` the type parameter is covariant, although both interfaces extend the same interface. The same rule is applied to contravariant generic type parameters.

You can create an interface that extends both the interface where the generic type parameter `T` is covariant and the interface where it is contravariant if in the extending interface the generic type parameter `T` is invariant. This is illustrated in the following code example.

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

However, if a generic type parameter `T` is declared covariant in one interface, you cannot declare it contravariant in the extending interface, or vice versa. This is illustrated in the following code example.

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Avoiding Ambiguity

When you implement variant generic interfaces, variance can sometimes lead to ambiguity. This should be avoided.

For example, if you explicitly implement the same variant generic interface with different generic type parameters in one class, it can create ambiguity. The compiler does not produce an error in this case, but it is not specified which interface implementation will be chosen at runtime. This could lead to subtle bugs in your code. Consider the following code example.

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

In this example, it is unspecified how the `pets.GetEnumerator` method chooses between `Cat` and `Dog`. This could cause problems in your code.

See also

- Variance in Generic Interfaces (C#)
- Using Variance for Func and Action Generic Delegates (C#)

Using Variance in Interfaces for Generic Collections (C#)

1/23/2019 • 2 minutes to read • Edit Online

A covariant interface allows its methods to return more derived types than those specified in the interface. A contravariant interface allows its methods to accept parameters of less derived types than those specified in the interface.

In .NET Framework 4, several existing interfaces became covariant and contravariant. These include IEnumerable<T> and IComparable<T>. This enables you to reuse methods that operate with generic collections of base types for collections of derived types.

For a list of variant interfaces in the .NET Framework, see Variance in Generic Interfaces (C#).

Converting Generic Collections

The following example illustrates the benefits of covariance support in the IEnumerable<T> interface. The `PrintFullName` method accepts a collection of the `IEnumerable<Person>` type as a parameter. However, you can reuse it for a collection of the `IEnumerable<Employee>` type because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }

    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Comparing Generic Collections

The following example illustrates the benefits of contravariance support in the IComparer<T> interface. The `PersonComparer` class implements the `IComparer<Person>` interface. However, you can reuse this class to compare a sequence of objects of the `Employee` type because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName = "Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

See also

- Variance in Generic Interfaces (C#)

Variance in Delegates (C#)

1/23/2019 • 5 minutes to read • Edit Online

.NET Framework 3.5 introduced variance support for matching method signatures with delegate types in all delegates in C#. This means that you can assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. This includes both generic and non-generic delegates.

For example, consider the following code, which has two classes and two delegates: generic and non-generic.

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

When you create delegates of the `SampleDelegate` or `SampleGenericDelegate<A, R>` types, you can assign any one of the following methods to those delegates.

```
// Matching signature.
public static First ASecondRFirst(Second first)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

The following code example illustrates the implicit conversion between the method signature and the delegate type.

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

For more examples, see Using Variance in Delegates (C#) and Using Variance for Func and Action Generic Delegates (C#).

Variance in Generic Type Parameters

In .NET Framework 4 or later you can enable implicit conversion between delegates, so that generic delegates that have different types specified by generic type parameters can be assigned to each other, if the types are inherited from each other as required by variance.

To enable implicit conversion, you must explicitly declare generic parameters in a delegate as covariant or contravariant by using the `in` or `out` keyword.

The following code example shows how you can create a delegate that has a covariant generic type parameter.

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

If you use only variance support to match method signatures with delegate types and do not use the `in` and `out` keywords, you may find that sometimes you can instantiate delegates with identical lambda expressions or methods, but you cannot assign one delegate to another.

In the following code example, `SampleGenericDelegate<String>` cannot be explicitly converted to `SampleGenericDelegate<Object>`, although `String` inherits `Object`. You can fix this problem by marking the generic parameter `T` with the `out` keyword.

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Generic Delegates That Have Variant Type Parameters in the .NET Framework

.NET Framework 4 introduced variance support for generic type parameters in several existing generic delegates:

- `Action` delegates from the System namespace, for example, Action<T> and Action<T1,T2>
- `Func` delegates from the System namespace, for example, Func<TResult> and Func<T,TResult>
- The Predicate<T> delegate

- The Comparison<T> delegate
- The Converter<TInput,TOutput> delegate

For more information and examples, see Using Variance for Func and Action Generic Delegates (C#).

Declaring Variant Type Parameters in Generic Delegates

If a generic delegate has covariant or contravariant generic type parameters, it can be referred to as a *variant generic delegate*.

You can declare a generic type parameter covariant in a generic delegate by using the `out` keyword. The covariant type can be used only as a method return type and not as a type of method arguments. The following code example shows how to declare a covariant generic delegate.

```
public delegate R DCovariant<out R>();
```

You can declare a generic type parameter contravariant in a generic delegate by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a method return type. The following code example shows how to declare a contravariant generic delegate.

```
public delegate void DContravariant<in A>(A a);
```

IMPORTANT

`ref`, `in`, and `out` parameters in C# can't be marked as variant.

It is also possible to support both variance and covariance in the same delegate, but for different type parameters. This is shown in the following example.

```
public delegate R DVariant<in A, out R>(A a);
```

Instantiating and Invoking Variant Generic Delegates

You can instantiate and invoke variant delegates just as you instantiate and invoke invariant delegates. In the following example, the delegate is instantiated by a lambda expression.

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combining Variant Generic Delegates

You should not combine variant delegates. The Combine method does not support variant delegate conversion and expects delegates to be of exactly the same type. This can lead to a run-time exception when you combine delegates either by using the Combine method or by using the `+` operator, as shown in the following code example.

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Variance in Generic Type Parameters for Value and Reference Types

Variance for generic type parameters is supported for reference types only. For example, `DVariant<int>` can't be implicitly converted to `DVariant<Object>` or `DVariant<long>`, because integer is a value type.

The following example demonstrates that variance in generic type parameters is not supported for value types.

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

See also

- Generics
- Using Variance for Func and Action Generic Delegates (C#)
- How to: Combine Delegates (Multicast Delegates)

Using Variance in Delegates (C#)

1/23/2019 • 2 minutes to read • Edit Online

When you assign a method to a delegate, *covariance* and *contravariance* provide flexibility for matching a delegate type with a method signature. Covariance permits a method to have return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Example 1: Covariance

Description

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by `DogsHandler` is of type `Dogs`, which derives from the `Mammals` type that is defined in the delegate.

Code

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
    }
}
```

Example 2: Contravariance

Description

This example demonstrates how delegates can be used with methods that have parameters of a type that are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. For example, you can create an event handler that accepts an `EventArgs` input parameter and use it with a `Button.MouseClick` event that sends a `MouseEventArgs` type as a parameter, and also with a `TextBox.KeyDown` event that sends a `KeyEventArgs` parameter.

Code

```
// Event handler that accepts a parameter of the EventArgs type.
private void MultiHandler(object sender, System.EventArgs e)
{
    label1.Text = System.DateTime.Now.ToString();
}

public Form1()
{
    InitializeComponent();

    // You can use a method that has an EventArgs parameter,
    // although the event expects the KeyEventArgs parameter.
    this.button1.KeyDown += this.MultiHandler;

    // You can use the same method
    // for an event that expects the MouseEventArgs parameter.
    this.button1.MouseClick += this.MultiHandler;

}
```

See also

- Variance in Delegates (C#)
- Using Variance for Func and Action Generic Delegates (C#)

Using Variance for Func and Action Generic Delegates (C#)

1/23/2019 • 2 minutes to read • Edit Online

These examples demonstrate how to use covariance and contravariance in the `Func` and `Action` generic delegates to enable reuse of methods and provide more flexibility in your code.

For more information about covariance and contravariance, see Variance in Delegates (C#).

Using Delegates with Covariant Type Parameters

The following example illustrates the benefits of covariance support in the generic `Func` delegates. The `FindByTitle` method takes a parameter of the `String` type and returns an object of the `Employee` type. However, you can assign this method to the `Func<String, Person>` delegate because `Employee` inherits `Person`.

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Using Delegates with Contravariant Type Parameters

The following example illustrates the benefits of contravariance support in the generic `Action` delegates. The `AddToContacts` method takes a parameter of the `Person` type. However, you can assign this method to the `Action<Employee>` delegate because `Employee` inherits `Person`.

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

See also

- Covariance and Contravariance (C#)
- Generics