**Covariance and Contravariance (C#)**

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.

C#Copy

// 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.

C#Copy

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#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/variance-in-delegates) and [Using Variance in Delegates (C#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/using-variance-in-delegates).

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

C#Copy

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#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/variance-in-generic-interfaces) and [Variance in Delegates (C#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/variance-in-delegates).

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

C#Copy

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.