<https://msdn.microsoft.com/fr-fr/library/ee207183.aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-1>

En C# et Visual Basic, la covariance et la contravariance permettent la conversion implicite des références pour des types de tableau, les types délégués et les arguments de type générique.La covariance conserve la compatibilité d'affectation et la contravariance l'inverse.

Le code suivant illustre la différence entre la compatibilité d'affectation, la covariance et la 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;

La covariance des tableaux permet la conversion implicite d'un tableau d'un type plus dérivé en tableau d'un type moins dérivé.Mais ces opérations ne sont pas sécurisées, comme l'illustre l'exemple de code suivant.

object[] array = new String[10];

// The following statement produces a run-time exception.

// array[0] = 10;

La prise en charge de la covariance et de la contravariance des groupes de méthode permet d'établir une correspondance entre les signatures de méthode et les types délégués.Cela vous permet d'assigner aux délégués pas uniquement les méthodes ayant des signatures correspondantes, mais également des méthodes qui retournent des types plus dérivés (covariance) ou qui acceptent des paramètres ayant des types moins dérivés (contravariance) que ceux spécifiés par le type délégué.

L'exemple de code suivant indique la prise en charge de la covariance et de la contravariance des groupes de méthode.

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;

}

Dans .NET Framework 4 et Visual Studio 2010, C# et Visual Basic prennent tous deux en charge la covariance et la contravariance dans les interfaces et les délégués génériques et autorisent la conversion implicite des paramètres de type générique.

L'exemple de code suivant décrit une conversion de référence implicite pour les interfaces génériques.

IEnumerable<String> strings = new List<String>();

IEnumerable<Object> objects = strings;

Une interface ou un délégué générique est appelé variant si ses paramètres génériques sont déclarés covariant ou contravariant.C# et Visual Basic vous permettent de créer vos propres interfaces et délégués variants.Pour plus d'informations, consultez [Création](https://msdn.microsoft.com/fr-fr/library/dd997386.aspx)

----------------------------------------------------------------------------------------------------------------------

<http://stackoverflow.com/questions/2662369/covariance-and-contravariance-real-world-example>

Let's say you have a class Person and a class that derives from it, Teacher. You have some operations that take an IEnumerable<Person> as the argument. In your School class you have a method that returns an IEnumerable<Teacher>. Covariance allows you to directly use that result for the methods that take an IEnumerable<Person>.

public class Person { }

public class Teacher : Person { }

public class MailingList

{

public void Add( IEnumerable<Person> people ) { ... }

}

public class School

{

public IEnumerable<Teacher> GetTeachers() { ... }

}

...

var teachers = school.GetTeachers();

var mailingList = new MailingList();

mailingList.Add( teachers );

----------------------------------------------------------------------------------------------------------------------

<http://stackoverflow.com/questions/3445631/still-confused-about-covariance-and-contravariance-in-out>

Both covariance and contravariance in C# 4.0 refer to the ability of using a derived class instead of base class. The in/out keywords are compiler hints to indicate whether or not the type parameters will be used for input and output.

**Covariance**

Covariance in C# 4.0 is aided by out keyword and it means that a generic type using a derived class of the out type parameter is OK. Hence

IEnumerable<Fruit> fruit = new List<Apple>();

Since Apple is a Fruit, List<Apple> can be safely used as IEnumerable<Fruit>

**Contravariance**

Contravariance is the in keyword and it denotes input types, usually in delegates. The principle is the same, it means that the delegate can accept more derived class.

public delegate void Func<in T>(T param);

This means that if we have a Func<Fruit>, it can be converted to Func<Apple>.

Func<Fruit> fruitFunc = (fruit)=>{};

Func<Apple> appleFunc = fruitFunc;

**Why are they called co/contravariance if they are basically the same thing?**

Because even though the principle is the same, safe casting from derived to base, when used on the input types, we can safely cast a less derived type (Func<Fruit>) to a more derived type (Func<Apple>), which makes sense, since any function that takes Fruit, can also take Apple.

----------------------------------------------------------------------------------------------------------------------

<https://sebastiencourtois.wordpress.com/2010/04/14/nouveauts-c-net-4-introduction-la-covariance-contravariance/>

**EDIT (08/05/2010) : La Covariance/Contravariance n’est pas disponible pour Silverlight 4**

* **Définitions**

La covariance et la contravariance dixit Wikipedia :

|  |
| --- |
| Within the [type system](http://en.wikipedia.org/wiki/Type_system) of a [programming language](http://en.wikipedia.org/wiki/Programming_language), a typing rule or a type conversion operator is:   * **covariant** if it preserves the [ordering, ≤, of types](http://en.wikipedia.org/wiki/Subtype), which orders types from more specific to more generic; * **contravariant** if it reverses this ordering, which orders types from more generic to more specific; * **invariant** if neither of these apply. |

Bon, je suis d’accord qu’avec cette définition, on est pas plus avancé. On voit que cela parle de conversion de type et de généricité mais ça reste très abstrait.

Nous allons donc voir des exemples puis revenir sur ces définitions.

* **Exemples sur la covariance**

Nous prendrons une architecture simple pour ces exemples :

public abstract class Person { }

public class Employe : Person { }

public class Customer : Person { }

public class Manager : Employe { }

Nous allons maintenant jouer avec des tableaux de ces objets :

Person[] MaSociete = new Manager[10]; // .NET 3.5 / 4 : OK

Employe[] MaSociete2 = new Manager[10]; // .NET 3.5 / 4 : OK

Rien de transcendant là dessus. On a un tableau de Manager. Un manager étant une personne (ça dépend dans quel société …:)), il n’y a pas de problème à mettre les données d’un tableau de manager dans un tableau de personnes. De même pour mettre nos données manager dans le tableau d’employés (Manager étant un employé).

Faison la même chose avec des listes

Manager[] MaSociete3 = new Manager[10];

IEnumerable<Manager> im = MaSociete3.Where(p => p.ToString() == "Manager"); // .NET 3.5 / 4 : OK

IEnumerable<Manager> im2 = new List<Manager>(); // .NET 3.5 / 4 : OK

Le requête LINQ renvoie un IEnumerable<Manager> que je peux donc mettre dans im. De même pour une liste de manager car List<T> implémente IEnumerable<T>.

IEnumerable<Person> ip = im; // .NET 3.5 : Erreur de compilation / .NET 4 : OK

Sur cette ligne, on obtient une erreur de compilation en .NET 3.5 (Cannot Convert IEnumerable<Manager> to IEnumerable<Person>). Pourtant cela devrait marcher comme un tableau (un manager est une personne donc un tableau de personne peut contenir des données venant d’un tableau de manager). Cela n’est pas possible car, en .NET 3.5, IEnumerable<T> n’est pas covariant (à l’inverse de Array qui lui est covariant). C’est pour corriger ce manque que Microsoft a autorisé la covariance des interfaces et des délégués en .NET 4.

En .NET 4, je peux ainsi écrire :

IEnumerable<Person> ipl = new List<Manager>(); // .NET 3.5 : Erreur de compilation / .NET 4 : OK

Func<string> funcString = () => { return "Bonjour le monde"; };

Func<object> funcObject = funcString; // .NET 3.5 : Erreur de compilation / .NET 4 : OK

Pour résumer, on peut dire que **la covariance permet de caster un type générique A<T> dans un type générique A<K> si T hérite directement ou indirectement de K et si le type T est un paramètre de sortie** (nous reviendrons sur cette notion plus loin).

Remarque : La covariance (ainsi que la contravariance) fonctionne **uniquement avec des interfaces et des délégués**. Les arguments génériques doivent être **des types références**. Il n’est pas possible de faire les lignes suivantes :

List<Person> lo = new List<Manager>(); // .NET 3.5 / 4 : Erreur de compilation

IEnumerable<ValueType> lo = new List<int>(); // .NET 3.5 / 4 : Erreur de compilation

* **Exemples sur la contravariance**

Dans certains cas, on aimerait que l’inverse soit possible.

Action<object> actObject = (object o) => { };

Action<string> actString = actObject; // .NET 3.5 : Erreur de compilation / .NET 4 : OK

Dans cet exemple, je souhaite pouvoir caster mon délégués avec une valeur de retour de type string ou un de ses parents. En effet,lorsque je récupère une donnée, je peux la caster dans une variable d’un type dont elle hérite. C’est le principe de la contravariance introduite dans .NET 4.

**La contravariance est la possiblité de caster un type générique A<T> dans un type générique A<K> si T est parent direct ou indirect de K et si T est un paramètre d’entrée** (nous reviendrons sur cette notion plus loin).

* **Les interfaces / Délégués déja covariant et contravariant**

Voici la liste des interfaces/délégués ayant été mis à jour au sein du framework .net 4 pour tenir compte de cette nouvelle possibilité.

* [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(VS.100).aspx) (covariant)
* [IEnumerator<T>](http://msdn.microsoft.com/en-us/library/78dfe2yb(VS.100).aspx) (covariant)
* [IQueryable<T>](http://msdn.microsoft.com/en-us/library/bb351562(VS.100).aspx) (covariant)
* [IGrouping<TKey, TElement>](http://msdn.microsoft.com/en-us/library/bb344977(VS.100).aspx) (covariant)
* [IComparer<T>](http://msdn.microsoft.com/en-us/library/8ehhxeaf(VS.100).aspx) (contravariant)
* [IEqualityComparer<T>](http://msdn.microsoft.com/en-us/library/ms132151(VS.100).aspx) (contravariant)
* [IComparable<T>](http://msdn.microsoft.com/en-us/library/4d7sx9hd(VS.100).aspx) (contravariant)
* [Action<T>](http://msdn.microsoft.com/en-us/library/018hxwa8(VS.100).aspx) et ses frères et soeurs (contravariants)
* [Func<TResult>](http://msdn.microsoft.com/en-us/library/bb534960(VS.100).aspx) et [Func<T, TResult>](http://msdn.microsoft.com/en-us/library/bb549151(VS.100).aspx) (*TResult* est covariant; *T,T1,T2 sont contravariants*)
* [Predicate<T>](http://msdn.microsoft.com/en-us/library/bfcke1bz(VS.100).aspx) (contravariant)
* [Comparison<T>](http://msdn.microsoft.com/en-us/library/tfakywbh(VS.100).aspx) (contravariant)
* [Converter<TInput, TOutput>](http://msdn.microsoft.com/en-us/library/kt456a2y(VS.100).aspx) (*TInput* est contravariant; *TOutput* est covariant.)
* **Exemple pratique : La création d’une interface covariant et contravariant**

Prenons l’exemple d’une classe réalisant de la transformations de données.

public interface IDataTransformer<T, K>

{

K Transform(T data);

}

public class DataTransformer<T, K> : IDataTransformer<T, K>

{

public K Transform(T data)

{

return default(K);

}

}

Nous avons donc une interface IDataTransformer comprenant deux paramètres génériques T et K et une méthode qui va transformer une variable de type T en type K. Nous allons créer un type implémentant cette interface puis l’utiliser.

IDataTransformer<Manager, string> dataTransformer = new DataTransformer<Manager, string>(); // .NET 3.5 / 4 : OK

IDataTransformer<Person, string> dataTransformer2 = new DataTransformer<Manager, string>(); // .NET 3.5 / 4 : Erreur de compilation

IDataTransformer<Manager, object> dataTransformer3 = new DataTransformer<Manager, string>(); // .NET 3.5 / 4 : Erreur de compilation

Si le premier cas fonctionne logiquement, les deux cas suivants sont moins logique. Dans le cas de dataTransfomer2, si on crée un DataTransformer prenant un Manager en entrée, on devrait aussi pouvoir prendre une personne (rappelons le, les managers sont des personnes).Pourtant le compilateur n’est pas de cette avis car il n’arrive pas à convertir DataTransformer<Manager,string> en IDataTransformer<Person,string>. De même pour dataTransformer3. Si on récupère en sortie une chaine de caractère, on pourrait très bien récupérer un object car string dérive de object.

Voila les limites de .NET 3.5 et voyons comment faire pour corriger cela en .NET 4. Pour cela nous allons créer une nouvelle interface et une nouvelle classe Variant.

public interface IDataTransformerVariant<in T,out K>

{

K Transform(T data);

}

public class DataTransformerVariant<T, K> : IDataTransformerVariant<T, K>

{

public K Transform(T data)

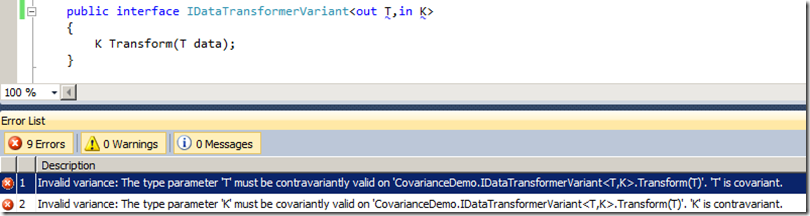
{

return default(K);

}

}

La différence se situe uniquement dans la déclaration de l’interface IDataTransformerVariant. On a rajouté un mot clé in devant le type d’entrée (T) et un out devant le type de sortie (K). En procédant ainsi, on active la covariance et la contravariance et on oblige le paramètre générique T a toujours être en paramètres de méthodes et le paramètre générique K a toujours être en valeur retour d’une méthode. Ne pas suivre cette règle entraine l’erreur de compilation suivante :



----------------------------------------------------------------------------------------------------------------------

<http://www.codeproject.com/Articles/72467/C-4-0-Covariance-And-Contravariance-In-Generics>

[C#](http://csharp.net/) 4.0 (and [.NET](http://www.microsoft.com/net/) 4.0) introduced covariance and contravariance to generic interfaces and delegates. But what is this variance thing?

According to [Wikipedia](http://wikipedia.org/), in multilinear algebra and tensor analysis, **covariance** and **contravariance** describe how the quantitative description of certain geometrical or physical entities changes when passing from one coordinate system to another.[(\*)](http://en.wikipedia.org/wiki/Covariance_and_contravariance_of_vectors)

But what does this have to do with **C#** or **.NET**?

In type theory, a the type **T** is greater (**>**) than type **S** if **S** is a subtype (derives from) **T**, which means that there is a quantitative description for types in a type hierarchy.

So, how does **covariance** and **contravariance** apply to **C#** (and **.NET**) generic types?

In **C#** (and **.NET**), variance is a relation between a generic type definition and a particular generic type parameter.

Given two types **Base** and **Derived**, such that:

* There is a reference (or identity) conversion between **Base** and **Derived**
* **Base ≥** **Derived**

A generic type definition **Generic<T>** is:

* **covariant in T** if the ordering of the constructed types follows the ordering of the generic type parameters: **Generic<Base> ≥ Generic<Derived>**.
* **contravariant in T** if the ordering of the constructed types is reversed from the ordering of the generic type parameters: **Generic<Base> ≤ Generic<Derived>**.
* **invariant in T** if neither of the above apply.

If this definition is applied to arrays, we can see that arrays have always been covariant in relation to the type of the elements because this is valid code:

object[] objectArray = new string[] { "string 1", "string 2" };

objectArray[0] = "string 3";

objectArray[1] = new object();

However, when we try to run this code, the second assignment will throw an [ArrayTypeMismatchException](http://msdn.microsoft.com/library/system.arraytypemismatchexception.aspx). Although the compiler was fooled into thinking this was valid code because an **object** is being assigned to an element of an **array of** **object**, at run time, there is always a type check to guarantee that the runtime type of the definition of the elements of the array is greater or equal to the instance being assigned to the element. In the above example, because the runtime type of the array is **array of string**, the first assignment of array elements is valid because **string ≥** **string** and the second is invalid because **string ≤** **object**.

This leads to the conclusion that, although arrays have always been covariant in relation to the type of the elements, they are not safely covariant – code that compiles is not guaranteed to run without errors.

In **C#**, variance is enforced in the declaration of the type and not determined by the usage of each the generic type parameter.

**Covariance** in relation to a particular generic type parameter is enforced, is using the [out generic modifier](http://msdn.microsoft.com/library/dd469487(VS.100).aspx):

public interface IEnumerable<out T>

{

IEnumerator<T> GetEnumerator();

}

public interface IEnumerator<out T>

{

T Current { get; }

bool MoveNext();

}

Notice the convenient use the pre-existing **out** keyword. Besides the benefit of not having to remember a new hypothetic **covariant** keyword, **out** is easier to remember because it defines that the generic type parameter can only appear in output positions — read-only properties and method return values.

In a similar way, the way **contravariance** is enforced in relation a particular generic type parameter, is using the [in generic modifier](http://msdn.microsoft.com/library/dd469484(v=VS.100).aspx):

Hide Copy Code

public interface IComparer<in T>

{

int Compare(T x, T y);

}

Once again, the use of the pre-existing **in** keyword makes it easier to remember that the generic type parameter can only be used in input positions — write-only properties and method non **ref** and non **out** parameters.

A generic type parameter that is not marked **covariant** (**out**) or **contravariant** (**in**) is **invariant**.

Because **covariance** and **contravariance** applies to the relation between a generic type definition and a particular generic type parameter, a generic type definition can be both **covariant**, **contravariant** and **invariant** depending on the generic type parameter.

public delegate TResult Func<in T, out TResult>(T arg);

In the above delegate definition, **Func<T, TResult>** is **contravariant in T** and **convariant in TResult**.

All the types in the [.NET Framework](http://cli.gs/cligs/Microsoft-NET) where variance could be applied to its generic type parameters have been modified to take advantage of this new feature.

In summary, the rules for variance in **C#** (and **.NET**) are:

* Variance in relation to generic type parameters is restricted to generic interface and generic delegate type definitions.
* A generic interface or generic delegate type definition can be **covariant**, **contravariant** or **invariant** in relation to different generic type parameters.
* Variance applies only to reference types: a **IEnumerable<int>** is not an **IEnumerable<object>**.
* Variance does not apply to delegate combination. That is, given two delegates of types **Action<Derived>** and **Action<Base>**, you cannot combine the second delegate with the first although the result would be type safe. Variance allows the second delegate to be assigned to a variable of type **Action<Derived>**, but delegates can combine only if their types match exactly.