Disclaimer: ignore null assignments, I'm using them to keep the code relatively short and they are just enough to see what compiler wants to tell us.

**Let's start with a hierarchy of classes:**

class Animal { }

class Mammal : Animal { }

class Dog : Mammal { }

**Now define some interfaces, to illustrate what in and out generic modifiers actually do:**

interface IInvariant<T>

{

T Get(); // ok, an invariant type can be both put into and returned

void Set(T t); // ok, an invariant type can be both put into and returned

}

interface IContravariant<in T>

{

//T Get(); // compilation error, cannot return a contravariant type

void Set(T t); // ok, a contravariant type can only be \*\*put into\*\* our class (hence "in")

}

interface ICovariant<out T>

{

T Get(); // ok, a covariant type can only be \*\*returned\*\* from our class (hence "out")

//void Set(T t); // compilation error, cannot put a covariant type into our class

}

Ok, so why bother using interfaces with in and out modifiers if they **restrict** us? Let's see:

**Invariance**

Lets start with invariance (no in, no out modifiers)

Invariance experiment

Consider IInvariant<Mammal>

* IInvariant<Mammal>.Get() - returns a Mammal
* IInvariant<Mammal>.Set(Mammal) - accepts a Mammal

What if we try: IInvariant<Mammal> invariantMammal = (IInvariant<Animal>)null?

* Whoever calls IInvariant<Mammal>.Get() expects a Mammal, but IInvariant<Animal>.Get() - returns an Animal. Not every Animal is a Mammal so it's **incompatible**.
* Whoever calls IInvariant<Mammal>.Set(Mammal) expects that a Mammal can be passed. Since IInvariant<Animal>.Set(Animal) accepts **any** Animal (including Mammal), it's **compatible**
* **CONCLUSION**: such assignment is **incompatible**

And what if we try: IInvariant<Mammal> invariantMammal = (IInvariant<Dog>)null?

* Whoever calls IInvariant<Mammal>.Get() expects a Mammal, IInvariant<Dog>.Get() - returns a **Dog**, every Dog is a Mammal, so it's **compatible**.
* Whoever calls IInvariant<Mammal>.Set(Mammal) expects that a Mammal can be passed. Since IInvariant<Dog>.Set(Dog) accepts **only** Dogs (and not every Mammal as a Dog), it's **incompatible**.
* **CONCLUSION**: such assignment is **incompatible**

Let's check if we're right

IInvariant<Animal> invariantAnimal1 = (IInvariant<Animal>)null; // ok

IInvariant<Animal> invariantAnimal2 = (IInvariant<Mammal>)null; // compilation error

IInvariant<Animal> invariantAnimal3 = (IInvariant<Dog>)null; // compilation error

IInvariant<Mammal> invariantMammal1 = (IInvariant<Animal>)null; // compilation error

IInvariant<Mammal> invariantMammal2 = (IInvariant<Mammal>)null; // ok

IInvariant<Mammal> invariantMammal3 = (IInvariant<Dog>)null; // compilation error

IInvariant<Dog> invariantDog1 = (IInvariant<Animal>)null; // compilation error

IInvariant<Dog> invariantDog2 = (IInvariant<Mammal>)null; // compilation error

IInvariant<Dog> invariantDog3 = (IInvariant<Dog>)null; // ok

**THIS ONE IS IMPORTANT:** It's worth noticing that depending on whether the generic type parameter is higher or lower in class hierarchy, the generic types themselves are **incompatible for different reasons**.

Ok, so let's find out how could we exploit it.

**Covariance (out)**

You have covariance when you use out generic modifier (see above)

If our type looks like: ICovariant<Mammal>, it declares 2 things:

* Some of my methods return a Mammal (hence out generic modifier) - this is boring
* None of my methods accept a Mammal - this is interesting though, because this is the actual **restriction** imposed by the **out generic modifier**

How can we benefit from out modifier restrictions? Look back at the results of the "Invariance experiment" above. Now try to see what happens when make the same experiment for covariance?

Covariance experiment

What if we try: ICovariant<Mammal> covariantMammal = (ICovariant<Animal>)null?

* Whoever calls ICovariant<Mammal>.Get() expects a Mammal, but ICovariant<Animal>.Get() - returns an Animal. Not every Animal is a Mammal so it's **incompatible**.
* ~~ICovariant.Set(Mammal)~~ - this is no longer an issue thanks to the out modifier restrictions!
* **CONCLUSION** such assignment is **incompatible**

And what if we try: ICovariant<Mammal> covariantMammal = (ICovariant<Dog>)null?

* Whoever calls ICovariant<Mammal>.Get() expects a Mammal, ICovariant<Dog>.Get() - returns a **Dog**, every Dog is a Mammal, so it's **compatible**.
* ~~ICovariant.Set(Mammal)~~ - this is no longer an issue thanks to the out modifier restrictions!
* **CONCLUSION** such assignment is **COMPATIBLE**

Let's confirm it with the code:

ICovariant<Animal> covariantAnimal1 = (ICovariant<Animal>)null; // ok

ICovariant<Animal> covariantAnimal2 = (ICovariant<Mammal>)null; // ok!!!

ICovariant<Animal> covariantAnimal3 = (ICovariant<Dog>)null; // ok!!!

ICovariant<Mammal> covariantMammal1 = (ICovariant<Animal>)null; // compilation error

ICovariant<Mammal> covariantMammal2 = (ICovariant<Mammal>)null; // ok

ICovariant<Mammal> covariantMammal3 = (ICovariant<Dog>)null; // ok!!!

ICovariant<Dog> covariantDog1 = (ICovariant<Animal>)null; // compilation error

ICovariant<Dog> covariantDog2 = (ICovariant<Mammal>)null; // compilation error

ICovariant<Dog> covariantDog3 = (ICovariant<Dog>)null; // ok

**Contravariance (in)**

You have contravariance when you use in generic modifier (see above)

If our type looks like: IContravariant<Mammal>, it declares 2 things:

* Some of my methods accept a Mammal (hence in generic modifier) - this is boring
* None of my methods return a Mammal - this is interesting though, because this is the actual **restriction** imposed by the **in generic modifier**

Contravariance experiment

What if we try: IContravariant<Mammal> contravariantMammal = (IContravariant<Animal>)null?

* ~~IContravariant<Mammal>.Get()~~ - this is no longer an issue thanks to the in modifier restrictions!
* Whoever calls IContravariant<Mammal>.Set(Mammal) expects that a Mammal can be passed. Since IContravariant<Animal>.Set(Animal) accepts **any** Animal (including Mammal), it's **compatible**
* **CONCLUSION**: such assignment is **COMPATIBLE**

And what if we try: IContravariant<Mammal> contravariantMammal = (IContravariant<Dog>)null?

* ~~IContravariant<Mammal>.Get()~~ - this is no longer an issue thanks to the in modifier restrictions!
* Whoever calls IContravariant<Mammal>.Set(Mammal) expects that a Mammal can be passed. Since IContravariant<Dog>.Set(Dog) accepts **only** Dogs (and not every Mammal as a Dog), it's **incompatible**.
* **CONCLUSION**: such assignment is **incompatible**

Let's confirm it with the code:

IContravariant<Animal> contravariantAnimal1 = (IContravariant<Animal>)null; // ok

IContravariant<Animal> contravariantAnimal2 = (IContravariant<Mammal>)null; // compilation error

IContravariant<Animal> contravariantAnimal3 = (IContravariant<Dog>)null; // compilation error

IContravariant<Mammal> contravariantMammal1 = (IContravariant<Animal>)null; // ok!!!

IContravariant<Mammal> contravariantMammal2 = (IContravariant<Mammal>)null; // ok

IContravariant<Mammal> contravariantMammal3 = (IContravariant<Dog>)null; // compilation error

IContravariant<Dog> contravariantDog1 = (IContravariant<Animal>)null; // ok!!!

IContravariant<Dog> contravariantDog2 = (IContravariant<Mammal>)null; // ok!!!

IContravariant<Dog> contravariantDog3 = (IContravariant<Dog>)null; // ok

BTW, this feels a bit counterintuitive, doesn't it?

// obvious

Animal animal = (Dog)null; // ok

Dog dog = (Animal)null; // compilation error, not every Animal is a Dog

// but this looks like the other way around

IContravariant<Animal> contravariantAnimal = (IContravariant<Dog>) null; // compilation error

IContravariant<Dog> contravariantDog = (IContravariant<Animal>) null; // ok

**Why not both?**

So can we use both in and out generic modifiers? - obviously **not**.

Why? Look back at what restrictions do in and out modifiers impose. If we wanted to make our generic type parameter both covariant and contravariant, we would basically say:

* None of the methods of our interface returns T
* None of the methods of our interface accepts T

Which would essentially make our generic interface **non-generic**.

**How to remember it?**

You can use my tricks :)

1. "covariant" is shorter than "contravaraint" and this **opposite** to the lengths of their modifiers ("out" and "in" respectively)
2. **contra**varaint is a little **counter**intuitive (see the example above)