Covariance and Contravariance in .NET 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

1- Covariance (out) itself or child

Create IReader<T>Interface With one parameter less method Read Which will return T

```
public interface IReader<out TEntity>
    {
        TEntity Read();
    }
```

IReader<T> is Covariant, this means that :

1. All members either don't deal with T or have it in the return type, not the input parameters
2. In a call, IReader<T> could be replaced by any IReader<TAnotherEntity> given that TAnotherEntity is a child -directly or indirectly- of T

After That We Create A TestReader Method Take IReader<B> param and return Nothing

```
public static void TestReader(IReader<B> param)
{
    var b = param.Read();
    b.F1();
    b.F2();
}
```

Create 3 Instances of IReader Interface

```
IReader<A> readerA = new Reader<A>();
IReader<B> readerB = new Reader<B>();
IReader<C> readerC = new Reader<C>();
```

And We will Call TestReader 3 Times

```
// TestReader(readerB) is ok because TestReader is already expecting IReader<B>
TestReader(readerB);

// TestReader(readerC) is ok because C is a child of B
TestReader(readerC);

// TestReader(readerA) is NOT ok because A is a not a child of B
TestReader(readerA); // Compilation Error
```

But Why we can call TestReader With Tybe b,C But can not By Type A.

What if the compiler allows calling TestReader with a param of type IReader<A>, This means that:

1. param.Read() would return an instance of class A, not B
2. the var b would actually be of type A, not B
3. This would lead to the b.F2() line in the code above to fail as the var b doesn't have F2()

What if the compiler allows calling TestReader with a param of type IReader<C>, This means that:

1. param.Read() would return an instance of class C, not B
2. the var b would actually be of type C, not B
3. This would lead to the b.F2() line in the code above to work fine as the var b would have F2()

2- Contravariance (in) itself or parent

Create IWriter<T> Interface With One Method Write Take 1 Argument typeof(IWriter<T>) and return nothing (void)

```
public interface IWriter<in TEntity>
    {
        void Write(TEntity entity);
    }
```

IWriter<T> is Contravariant, this means that :

1. All members either don't deal with T or have it in the input parameters, not in the return type
2. In a call, IWriter<T> could be replaced by any IWriter<TAnotherEntity> given that TAnotherEntity is a parent -directly or indirectly- of T

After That We Create TestWriter Method Take IWriter<B> param and return Nothing

```
static void TestWriter(IWriter<B> param)
{
    var b = new B();
    param.Write(b);
}
```

Create 3 Instances of IWriter Interface

```
IWriter<A> writerA = new Writer<A>();
IWriter<B> writerB = new Writer<B>();
IWriter<C> writerC = new Writer<C>();
```

And we call TestWriter3 Times

```
TestWriter(writerB);

TestWriter(writerA);  //is ok because A is a parent of B

TestWriter(writerC) //is NOT ok because C is a not a parent of B , Compilation Error
```

But Why we can call TestWriter With Tybe A,b But can not By Type C.

- **What if the compiler allows calling TestWriter with a param of type IWriter<A>, This means that:**
  1. param.Write() line in the code above would be expecting to receive a parameter of type A, not B
  2. So, calling param.Write() while passing in a parameter of type A or B would both work

- What if the compiler allows calling TestWriter with a param of type IWriter<C>, This means that:
  1. param.Write() line in the code above would be expecting to receive a parameter of type C, not B
  2. So, calling param.Write() while passing in a parameter of type B would not work

---

3- Invariance

Create IReaderWriter<T> Interface With Two Methods:

1. parameter less method Read Which will return T
2. Write Take 1 Argument typeof(IReaderWriter<T>) and return nothing (void)

```
    public interface IReaderWriter<TEntity>
    {
        TEntity Read();
        void Write(TEntity entity);
    }
```

IReaderWriter<T> is Invariant, this means that:

1. Some members have Tin the input parameters and others have T in the return type
2. In a call, IReaderWriter<T> could not be replaced by any IReaderWriter<TAnotherEntity>

After That We Create TestReaderWriter Method Take IReaderWriter<B> param and return Nothing.

```
static void TestReaderWriter(IReaderWriter<B> param)
{
    var b = param.Read();
    b.F1();
    b.F2();

    param.Write(b);
}
```

Create 3 Instances of IReaderWriter Interface :

```
IReaderWriter<A> readerWriterA = new ReaderWriter<A>();
IReaderWriter<B> readerWriterB = new ReaderWriter<B>();
IReaderWriter<C> readerWriterC = new ReaderWriter<C>();
```

And we call TestWriter3 Times

```
TestReaderWriter(readerWriterB); //is ok because TestReaderWriter is already expecting IReaderWriter<B>

TestReaderWriter(readerWriterA); // is NOT ok because IReaderWriter<B> can not be replaced by IReaderWriter<A>,Compilation Error

TestReaderWriter(readerWriterC); // is NOT ok because IReaderWriter<B> can not be replaced by IReaderWriter<C>,Compilation Error
```

But Why we can call TestReaderWriterWith Tybe B But can not By Type C,A.

- What if the compiler allows calling TestReaderWriter with a param of type IReaderWriter<A>, This means that:
  - param.Read() would return an instance of class A, not B So, the var b would actually be of type A, not B This would lead to the b.F2() line in the code above to fail as the var b doesn't have F2()
  - param.Write() line in the code above would be expecting to receive a parameter of type A, not B So, calling param.Write() while passing in a parameter of type A or B would both work
- What if the compiler allows calling TestReaderWriter with a param of type IReaderWriter<C>, This means that:
  - param.Read() would return an instance of class C, not B So, the var b would actually be of type C, not B This would lead to the b.F2() line in the code above to work fine as the var b would have F2()
  - param.Write() line in the code above would be expecting to receive a parameter of type C, not B So, calling param.Write() while passing in a parameter of type B would not work