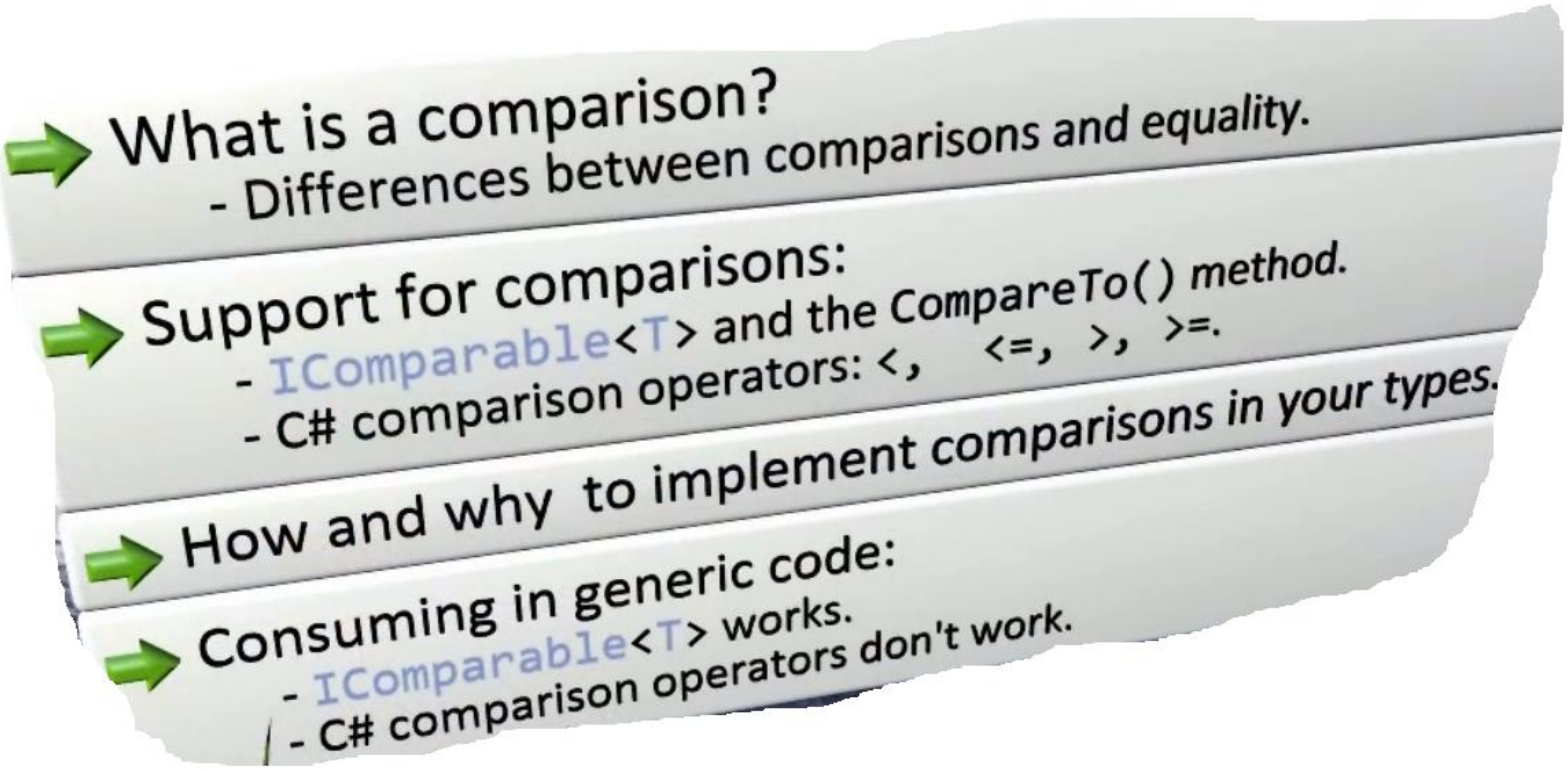
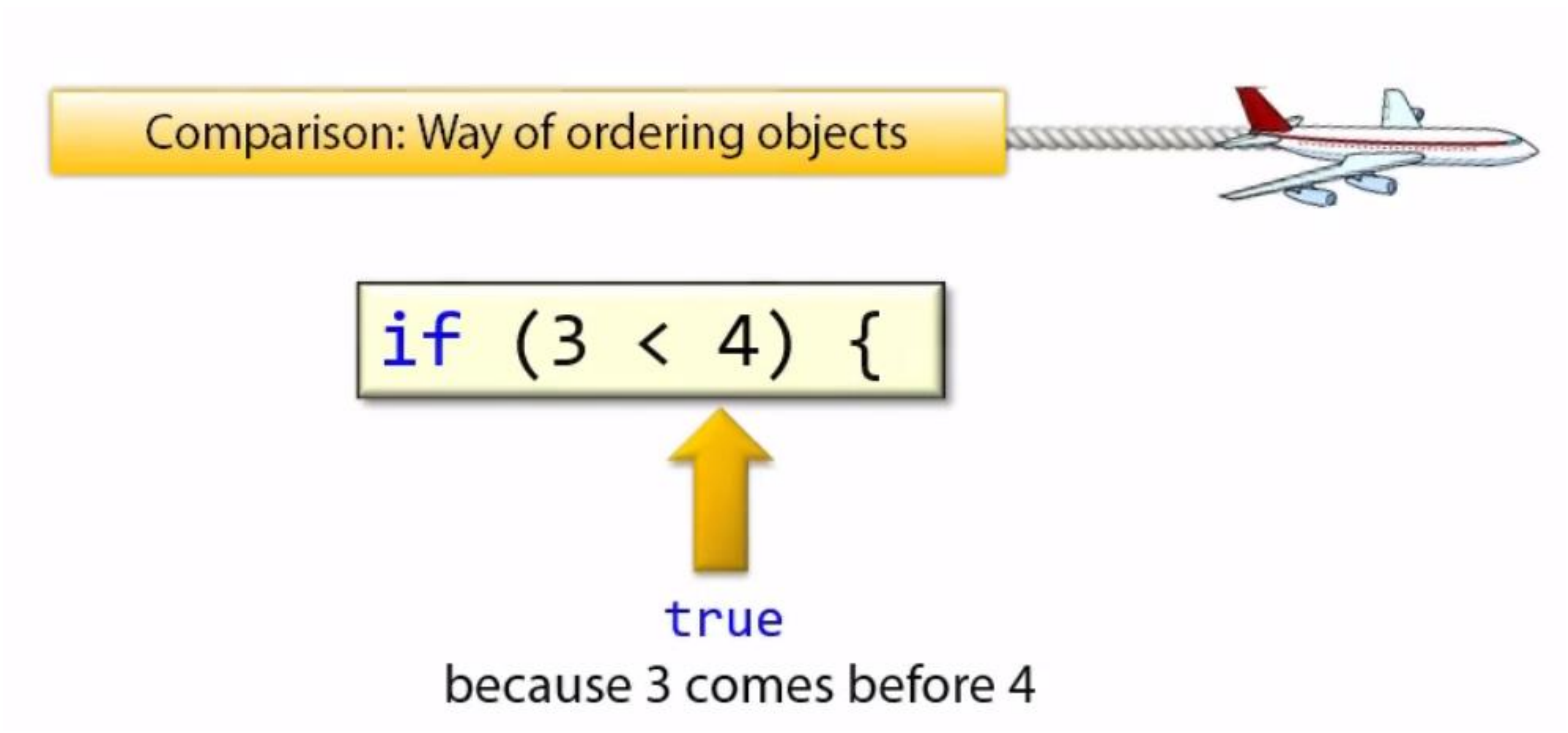


- What is a comparison?
  - Differences between comparisons and *equality*.
- Support for comparisons:
  - `IComparable<T>` and the `CompareTo()` *method*.
  - C# comparison operators: `<,  <=,  >,  >=`.
- How and why to implement *comparisons in your types.*
- Consuming in generic code:
  - `IComparable<T>` works.
  - C# comparison operators don't work.

What is a Comparison



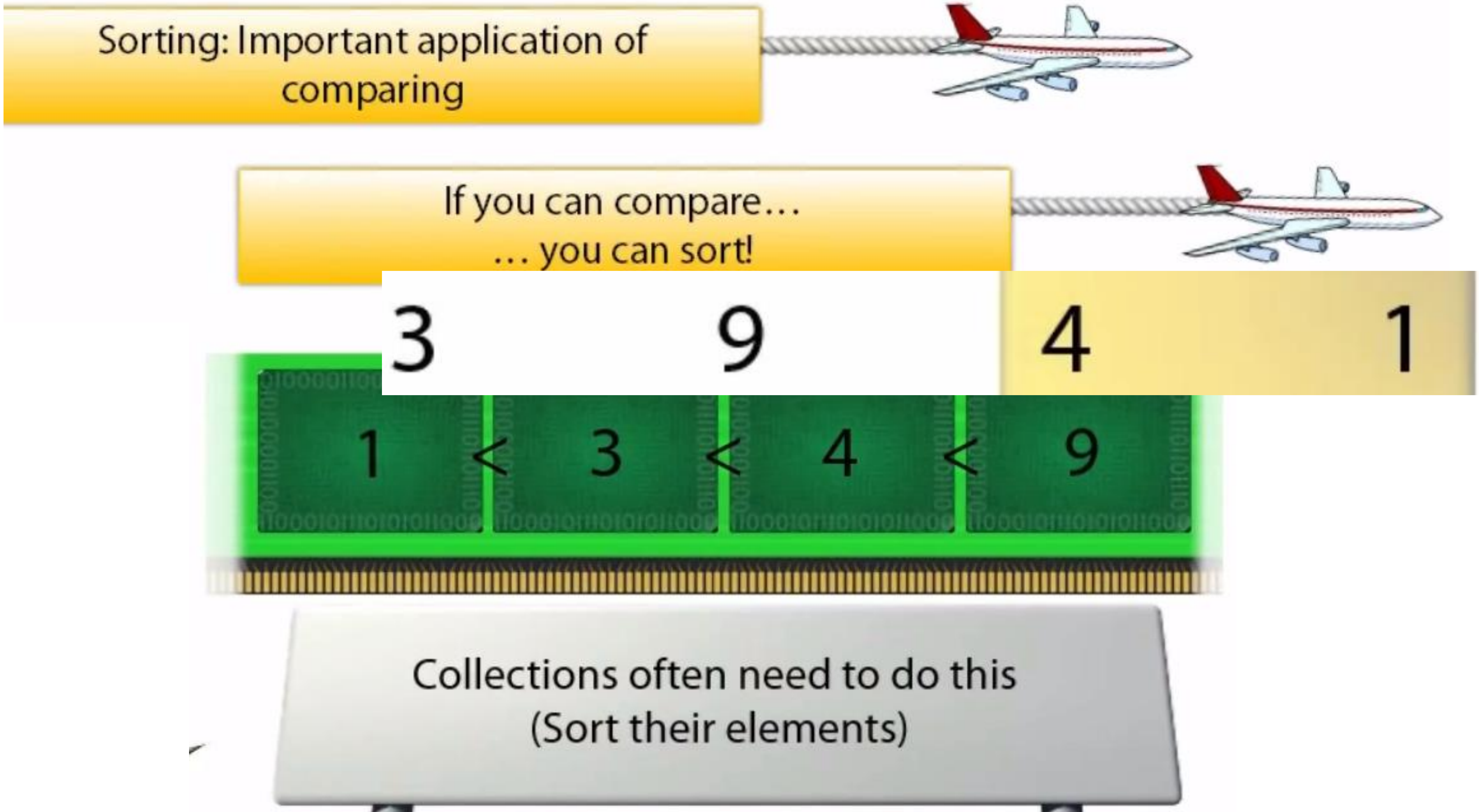
Comparison: Way of ordering objects

```
if (3 < 4) {
```

```
true
```

because 3 comes before 4

Comparing and Sorting


Sorting: Important application of comparing
If you can compare…
… you can sort!
3
9
4
1
1 < 3 < 4 < 9
Collections often need to do this
(Sort their elements)

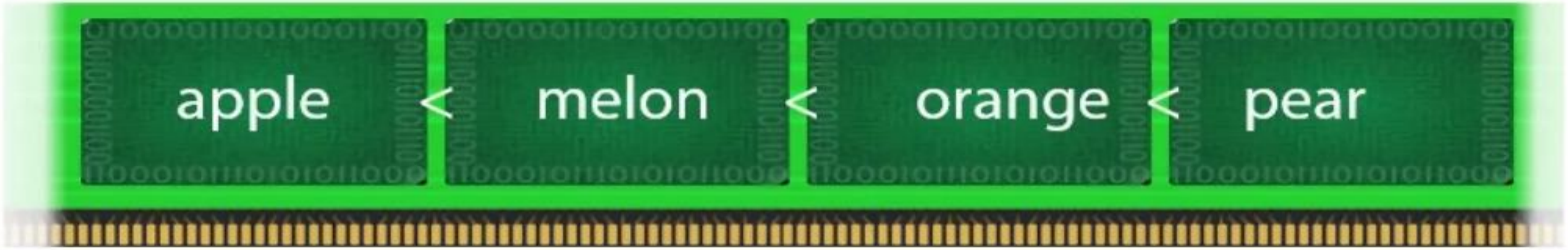
Comparisons and Equality

If the comparison says…	Then equality says..
x = y	Equal
x > y	Not equal
x < y	Not equal

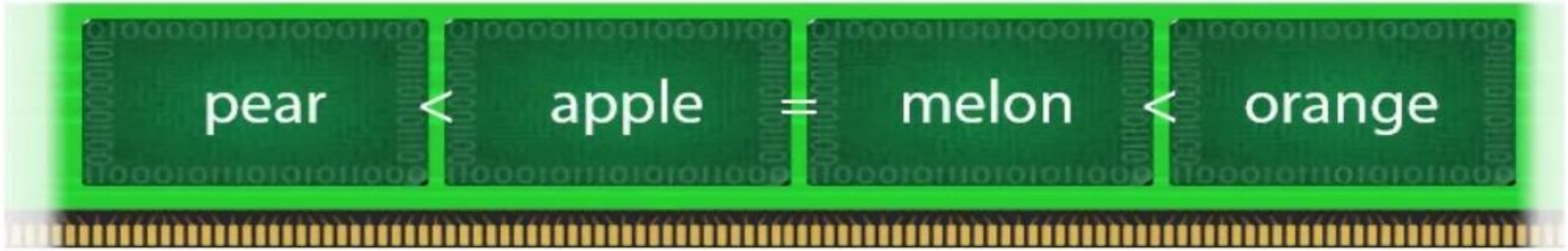
‘Natural’ and ‘Plugged-In’ Comparisons

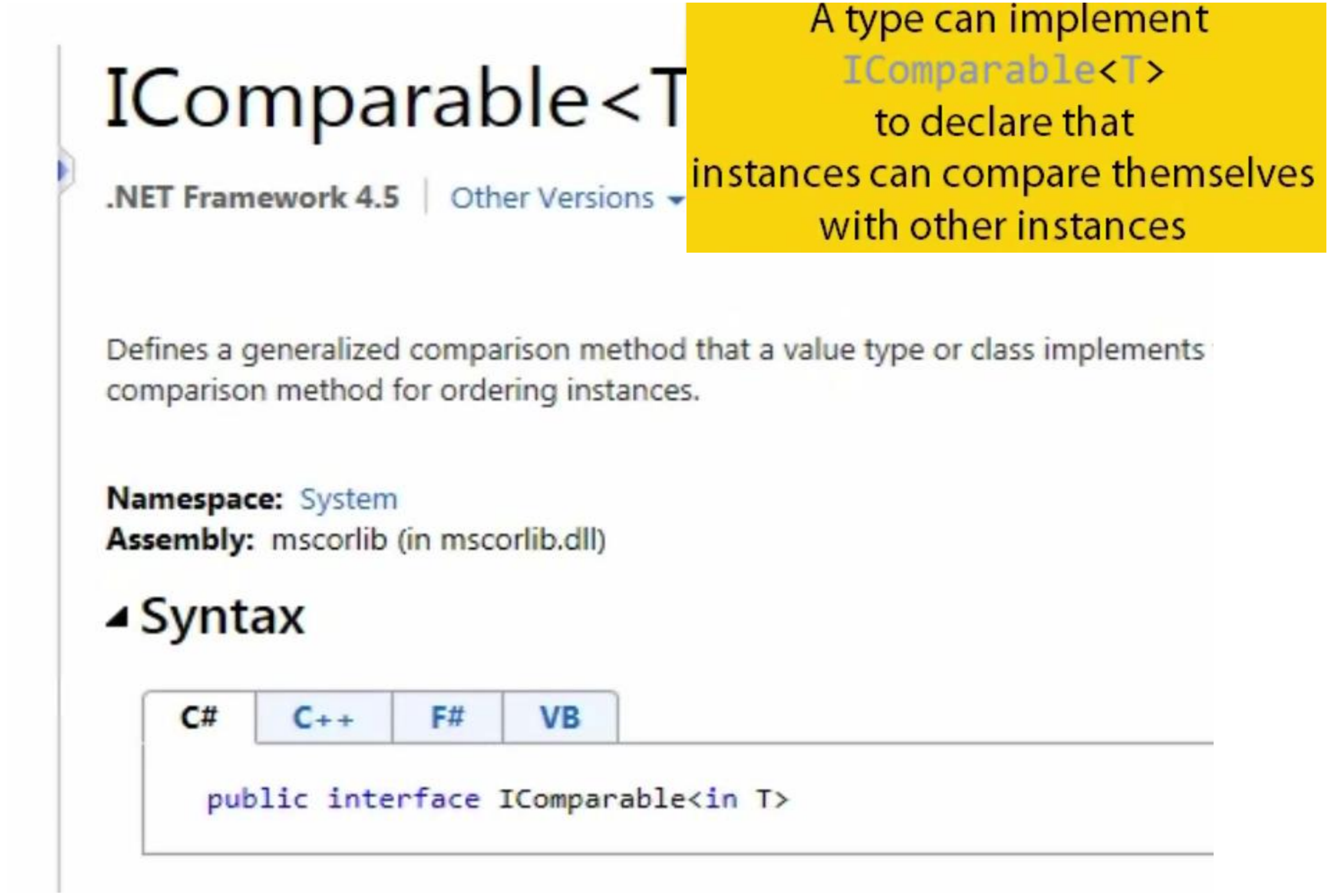
Eg. For strings…

Natural comparison: Alphabetical



A 'plugged-in' comparison: Compare by string length





IComparable<T

.NET Framework 4.5 | Other Versions

A type can implement
IComparable<T>
to declare that
instances can compare themselves
with other instances

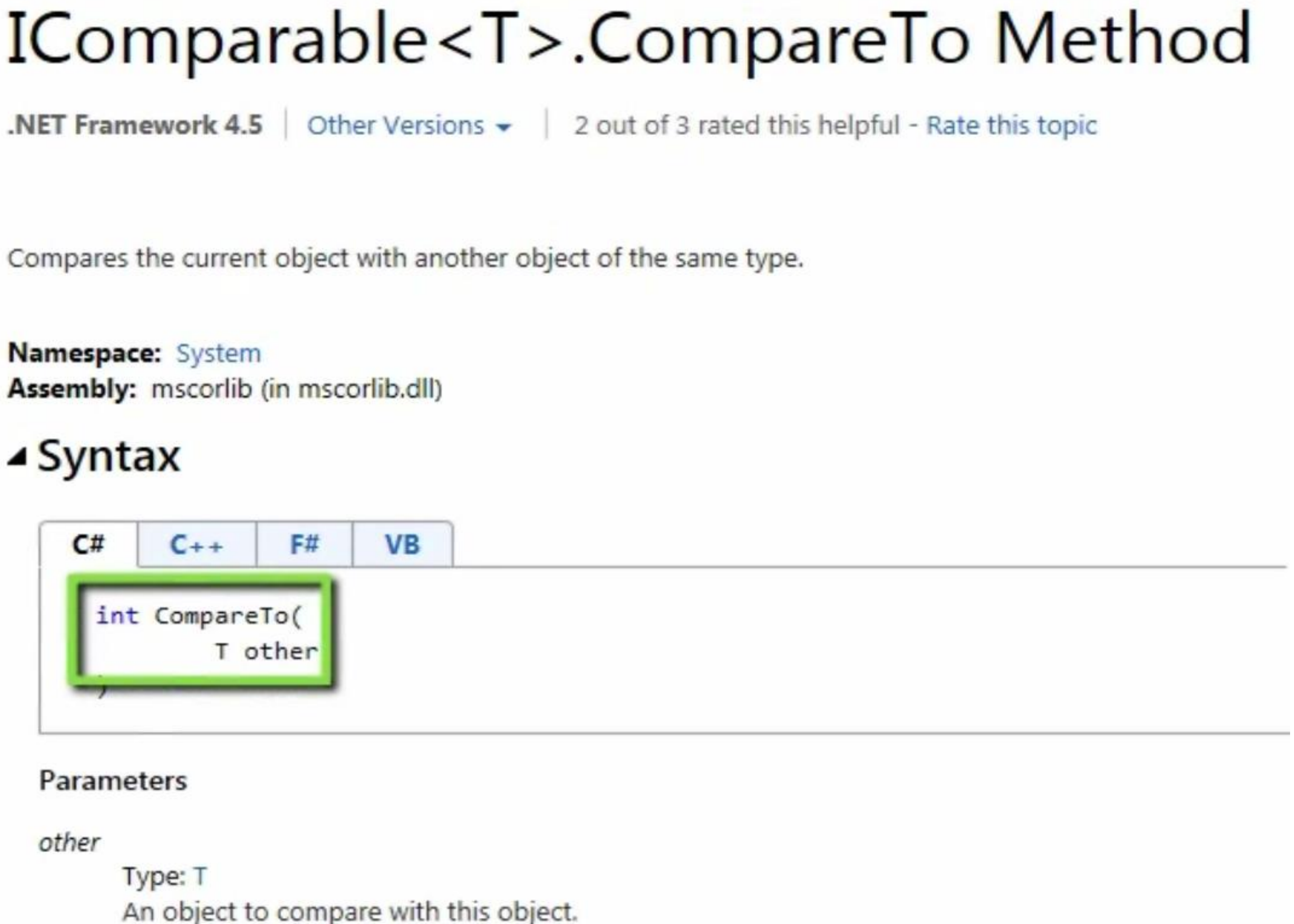
Defines a generalized comparison method that a value type or class implements
comparison method for ordering instances.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

C# | C++ | F# | VB

```
public interface IComparable<in T>
```



IComparable<T>.CompareTo Method

.NET Framework 4.5 | Other Versions | 2 out of 3 rated this helpful - Rate this topic

Compares the current object with another object of the same type.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

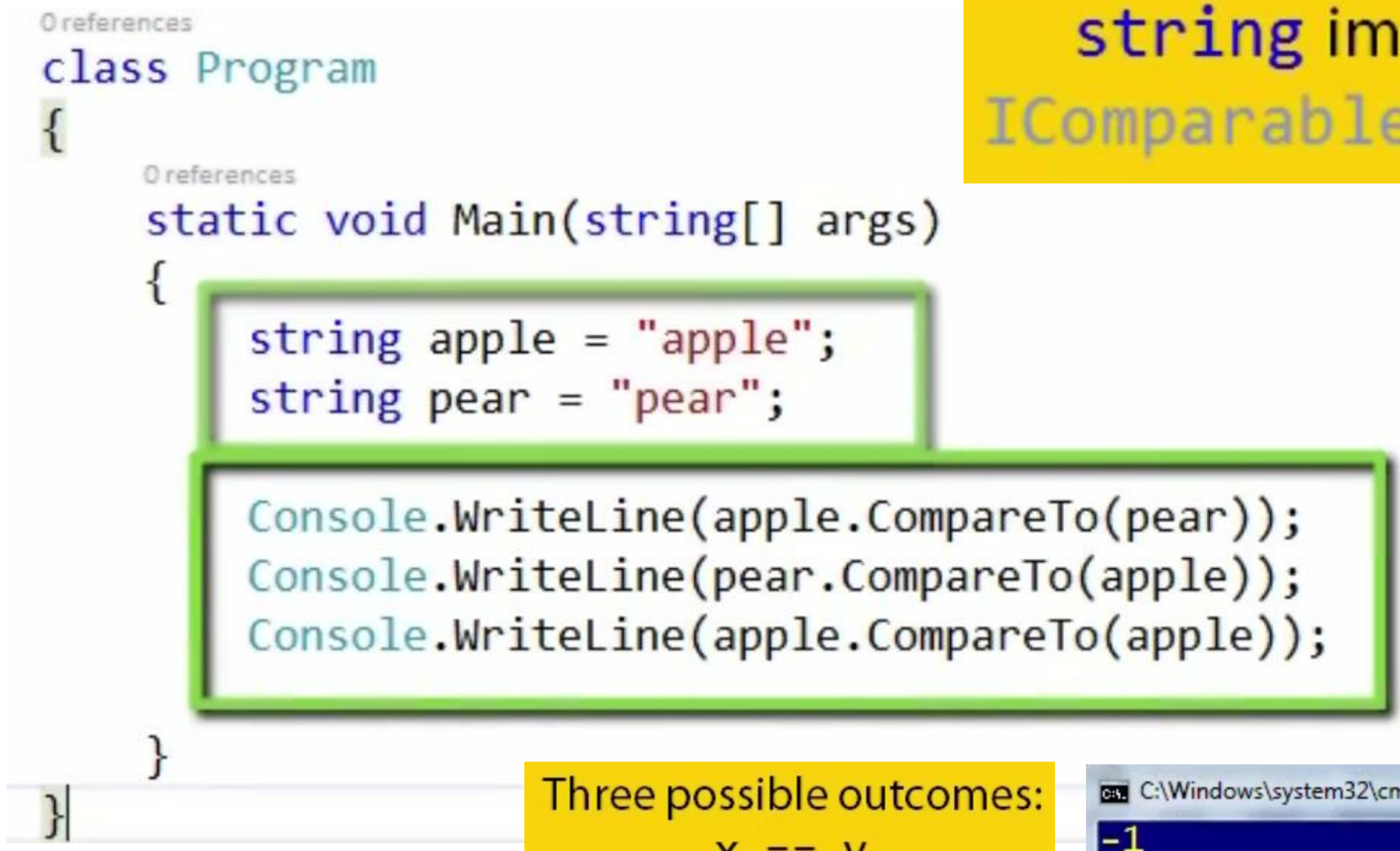
```
int CompareTo(
	T other
)
```

Parameters

other

Type: T
An object to compare with this object.

string implements IComparable<string >



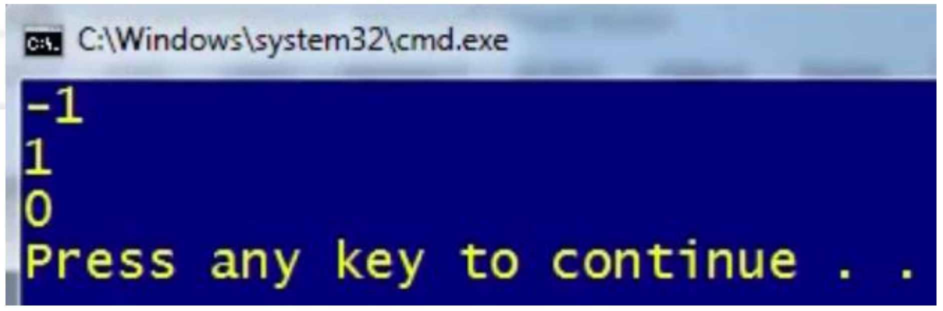
```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        string apple = "apple";
        string pear = "pear";

        Console.WriteLine(apple.CompareTo(pear));
        Console.WriteLine(pear.CompareTo(apple));
        Console.WriteLine(apple.CompareTo(apple));

    }
}
```

Three possible outcomes:

x == y

x < y

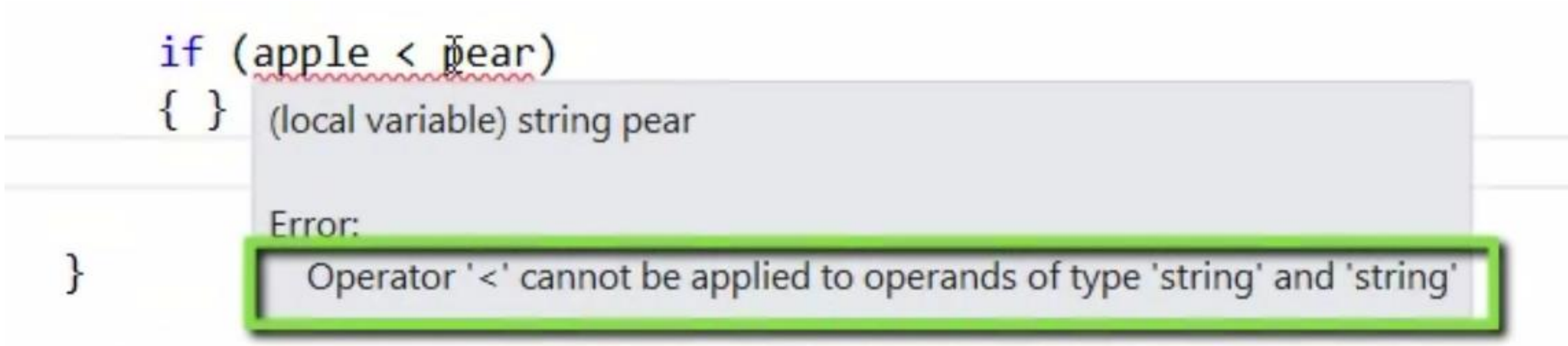
x > y



C:\Windows\system32\cmd.exe

```
-1
1
0
Press any key to continue . .
```

Strings do not use the Compare operators



```
if (apple < pear)
{ }
}
```

(local variable) string pear

Error:

Operator '<' cannot be applied to operands of type 'string' and 'string'

<, >, <=, >=
Are not supported for
string

string
also implements its own
methods for comparisons

object Doesn't Support Comparisons

Equality	Comparisons
`object.Equals()` (and other methods)	✗ No support in `System.object`

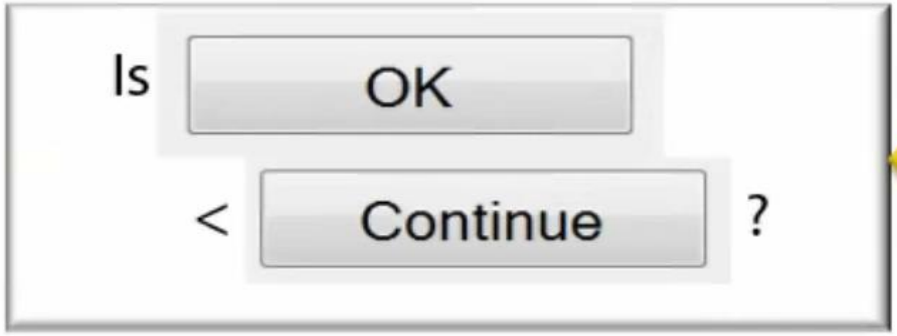
Equality makes sense for **all** types

Comparisons **don't** make sense for many types

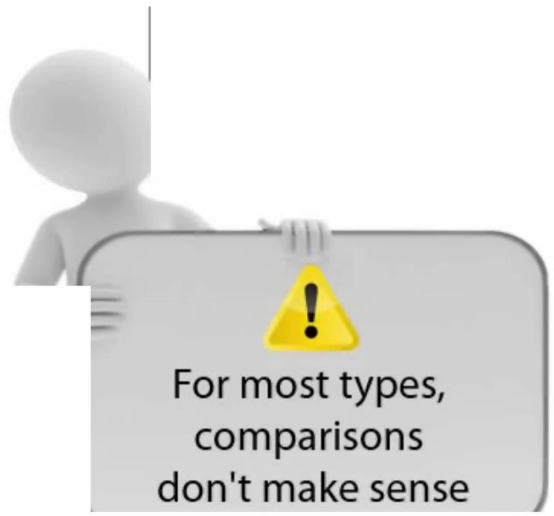
Comparisons

Is 3 < 4 ?

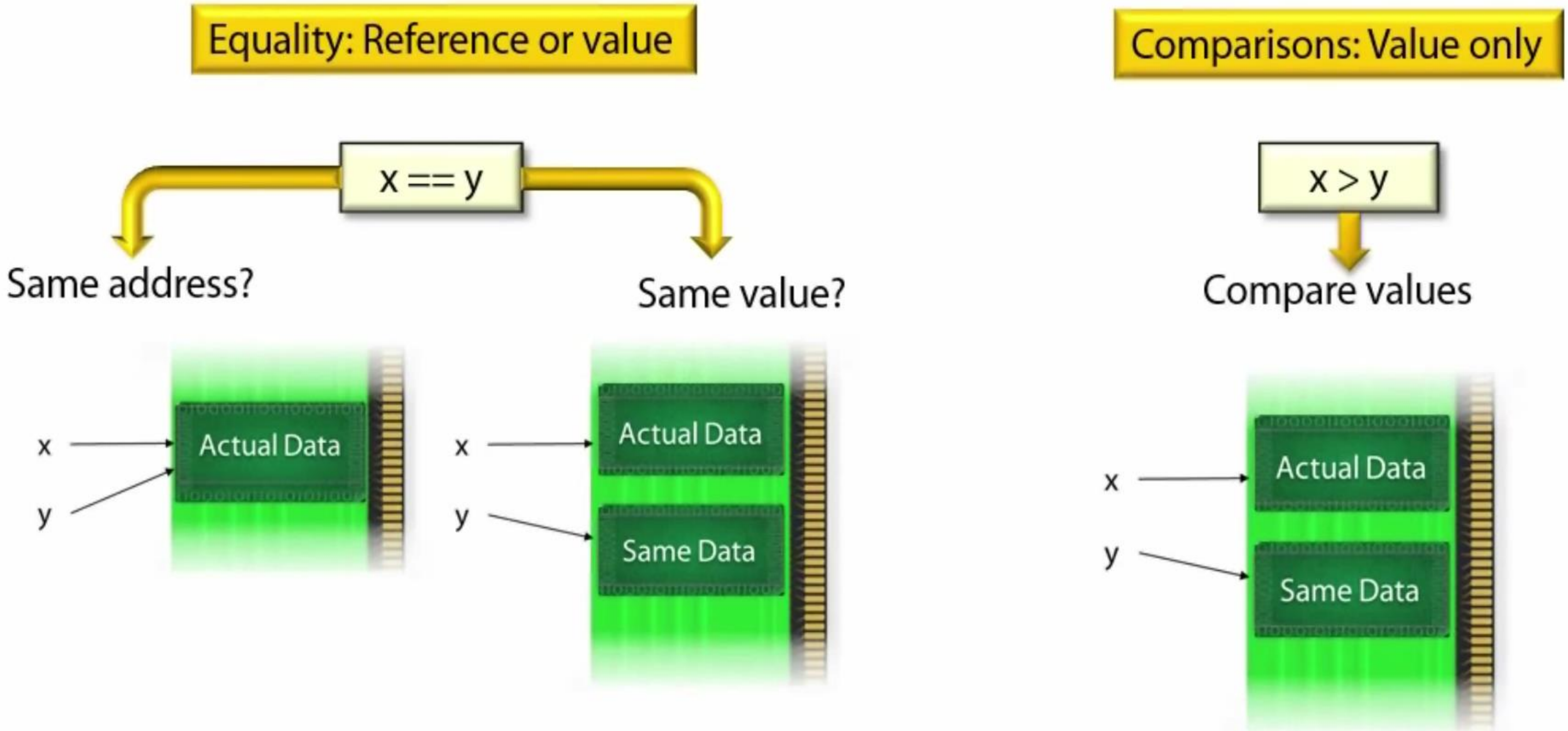
This question makes sense ✓


Is OK < Continue ?

This question is nonsense


For most types, comparisons don't make sense

Comparisons are Value Only



Equality and Comparison Operators

==, != operators

Work out of the box
for all
primitive and reference types

>, <, >=, <= operators

Work out of the box
only for
primitive types.

Implementing Comparisons

More often than not…

… Don't! …

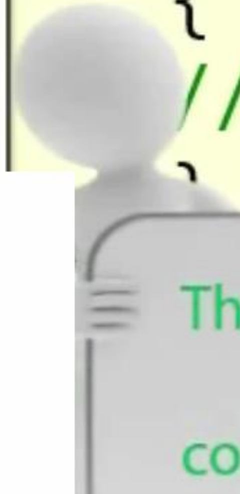
Comparing Foods....

Compare by name?

Compare by food group?

Compare by calories?

```
public class Food : IComparable<Food>
{
   public int CompareTo(Food other)
   {
   // what do you put here?
   }
```



These are all good comparisons

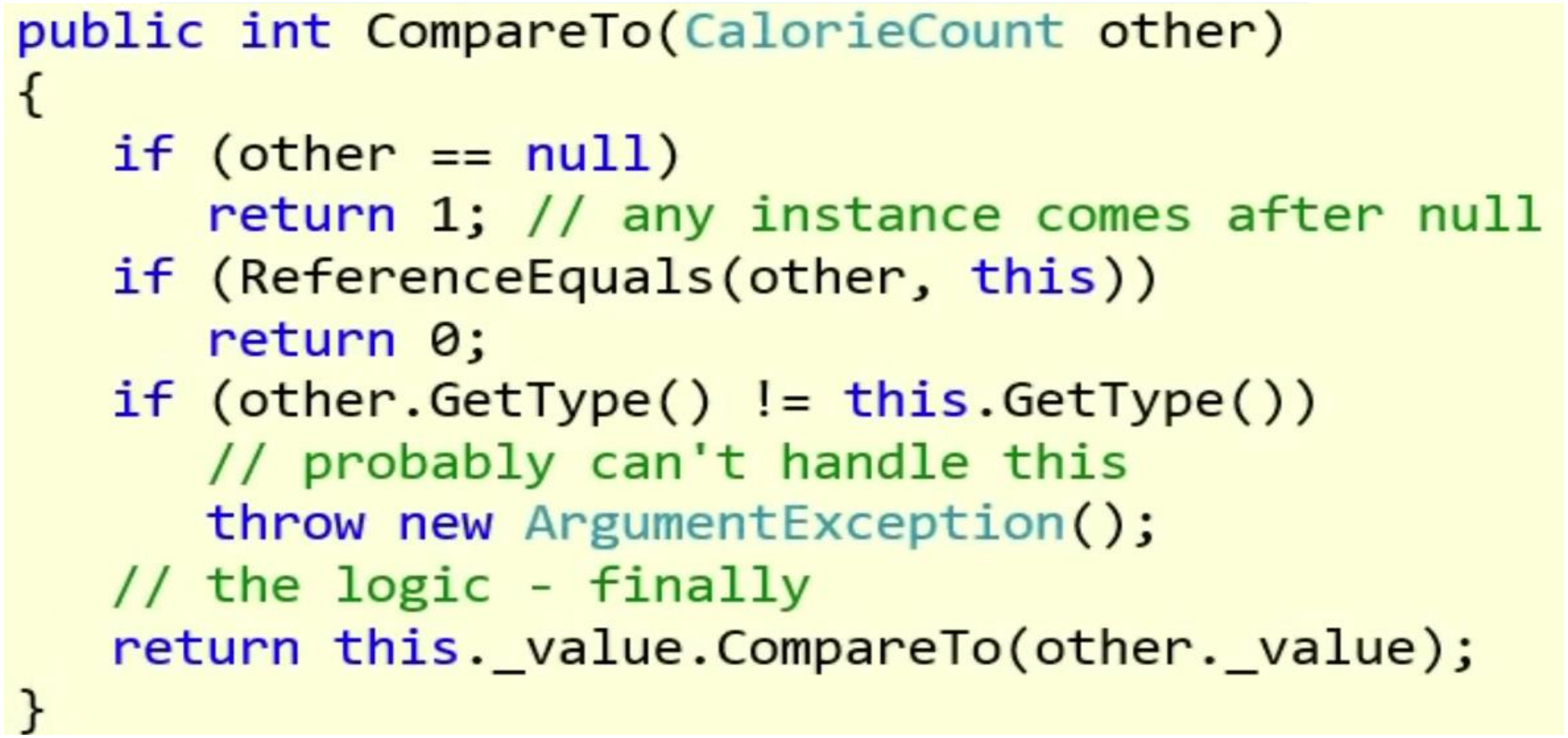
None are natural comparisons

IComparable<T> is not appropriate for Food

But writing a comparer for Food is fine

Better Example of Using IComparable

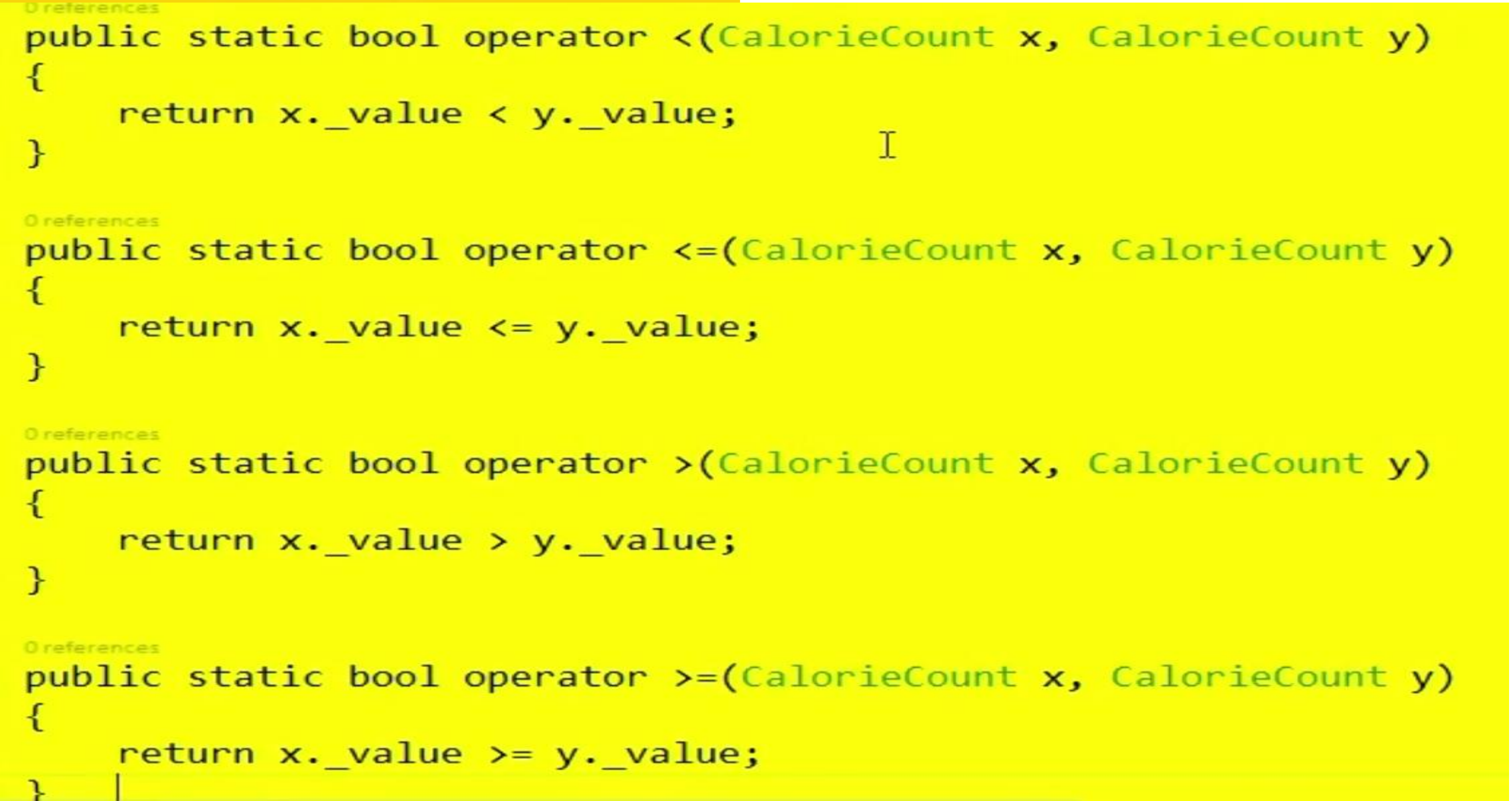
```
public class CalorieCount : IComparable<CalorieCount>
{
    public float Value { get; set; }
```



```
public int CompareTo(CalorieCount other)
{
    if (other == null)
        return 1; // any instance comes after null
    if (ReferenceEquals(other, this))
        return 0;
    if (other.GetType() != this.GetType())
        // probably can't handle this
        throw new ArgumentException();
    // the logic - finally
    return this._value.CompareTo(other._value);
}
```

4 operators to implement:
<, >, <=, >=

You **must** implement all four operators individually



```
0 references
public static bool operator <(CalorieCount x, CalorieCount y)
{
    return x._value < y._value;
}

0 references
public static bool operator <=(CalorieCount x, CalorieCount y)
{
    return x._value <= y._value;
}

0 references
public static bool operator >(CalorieCount x, CalorieCount y)
{
    return x._value > y._value;
}

0 references
public static bool operator >=(CalorieCount x, CalorieCount y)
{
    return x._value >= y._value;
}
```

Extra Issues for Reference Types

If CalorieCount is an unsealed class…

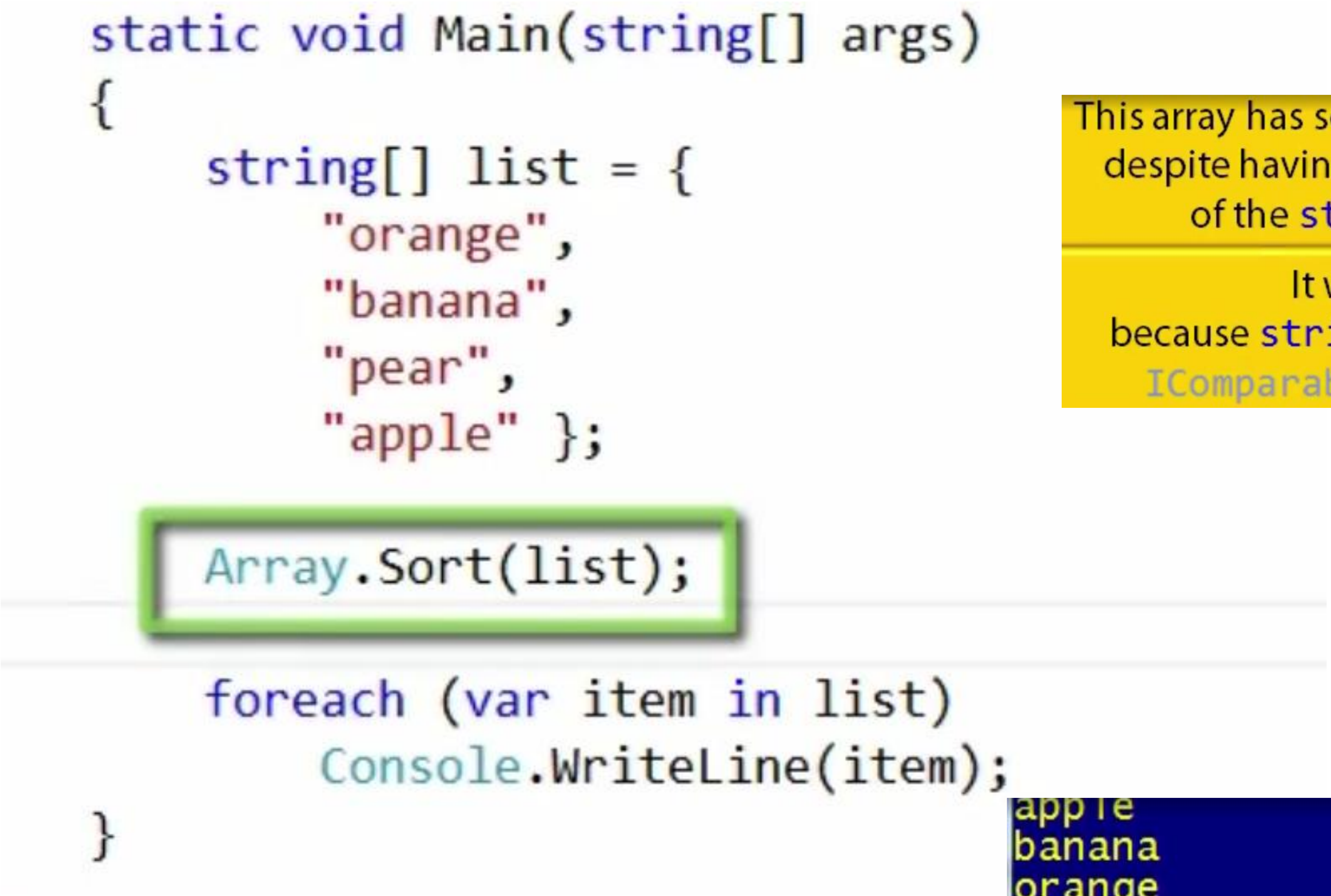
… this happens…

```
public int CompareTo(CalorieCount other)
{
```

If this is a derived type instance
– good luck!

I suggest:
Avoid implementing comparisons on nonsealed classes

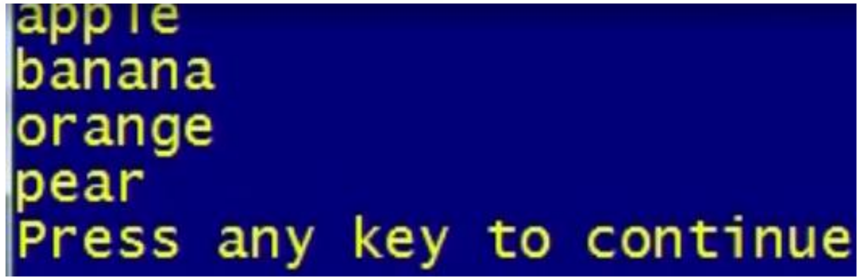
Comparers and Equality Comparers



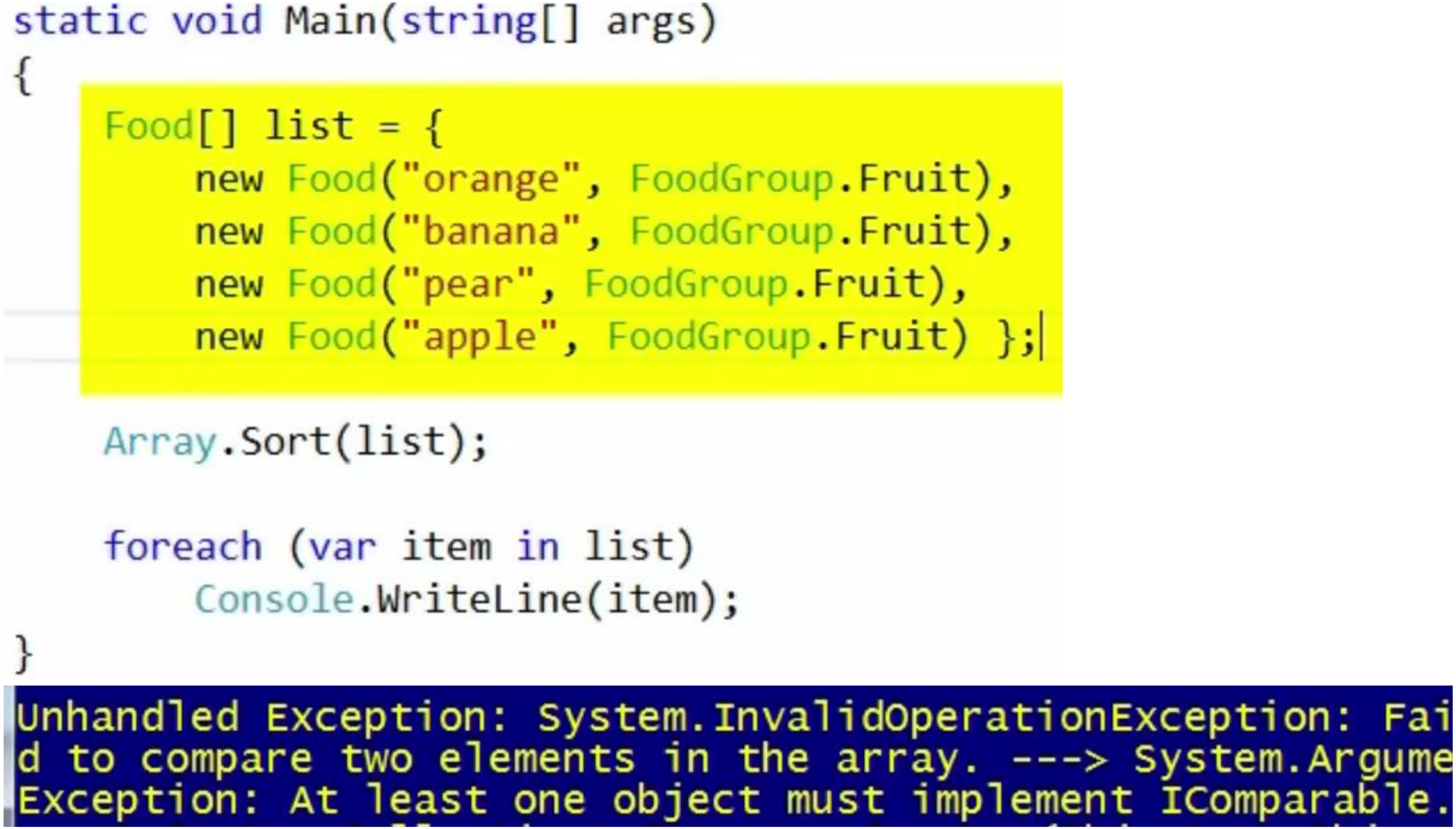
```
static void Main(string[] args)
{
    string[] list = {
        "orange",
        "banana",
        "pear",
        "apple" };

    Array.Sort(list);

    foreach (var item in list)
        Console.WriteLine(item);
}
```

This array has sorted the strings – despite having no knowledge of the `string` type!

It works because `string` implements `IComparable<string>`



```
apple
banana
orange
pear
Press any key to continue
```

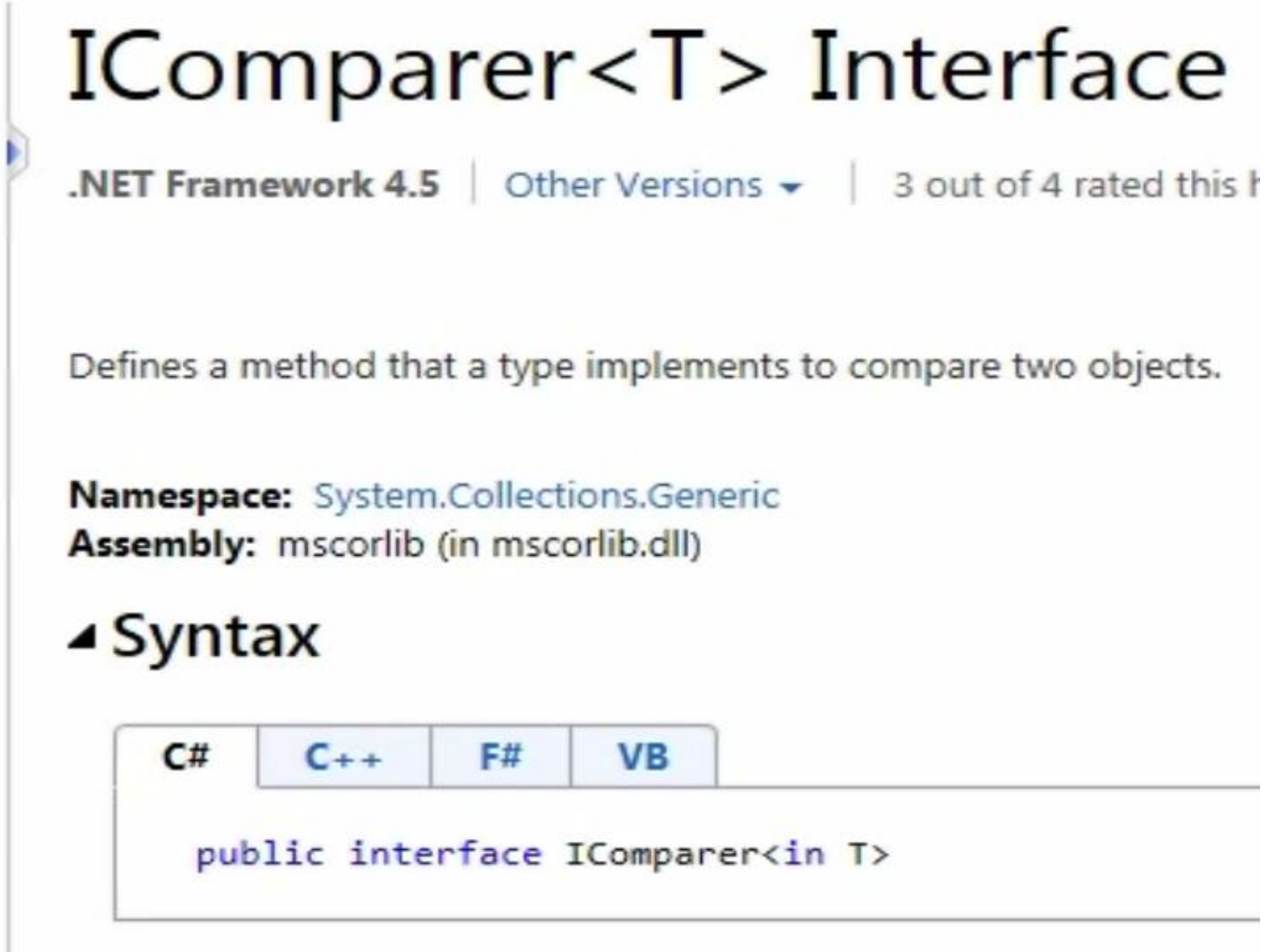


```
static void Main(string[] args)
{
    Food[] list = {
        new Food("orange", FoodGroup.Fruit),
        new Food("banana", FoodGroup.Fruit),
        new Food("pear", FoodGroup.Fruit),
        new Food("apple", FoodGroup.Fruit) };

    Array.Sort(list);

    foreach (var item in list)
        Console.WriteLine(item);
}
```

```
Unhandled Exception: System.InvalidOperationException: Fai
d to compare two elements in the array. ---> System.Argume
Exception: At least one object must implement IComparable.
```

Comparers

A **comparer** knows how to compare other objects (according to some criteria)!


IComparer<T> Interface

.NET Framework 4.5 | Other Versions | 3 out of 4 rated this h

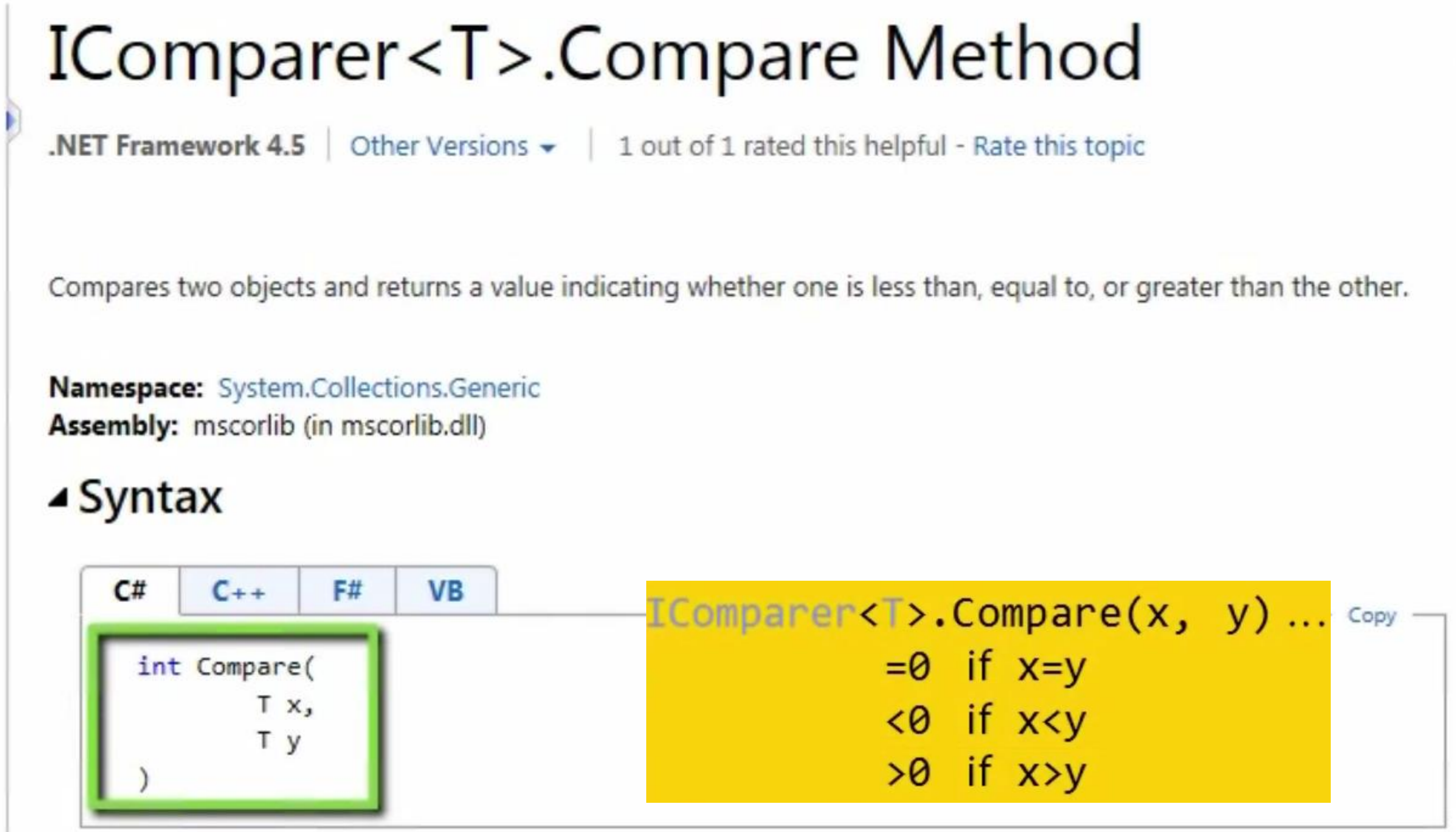
Defines a method that a type implements to compare two objects.

Namespace: System.Collections.Generic
Assembly: mscorlib (in mscorlib.dll)

Syntax

C# | C++ | F# | VB

```
public interface IComparer<in T>
```



IComparer<T>.Compare Method

.NET Framework 4.5 | Other Versions | 1 out of 1 rated this helpful - Rate this topic

Compares two objects and returns a value indicating whether one is less than, equal to, or greater than the other.

Namespace: System.Collections.Generic
Assembly: mscorlib (in mscorlib.dll)

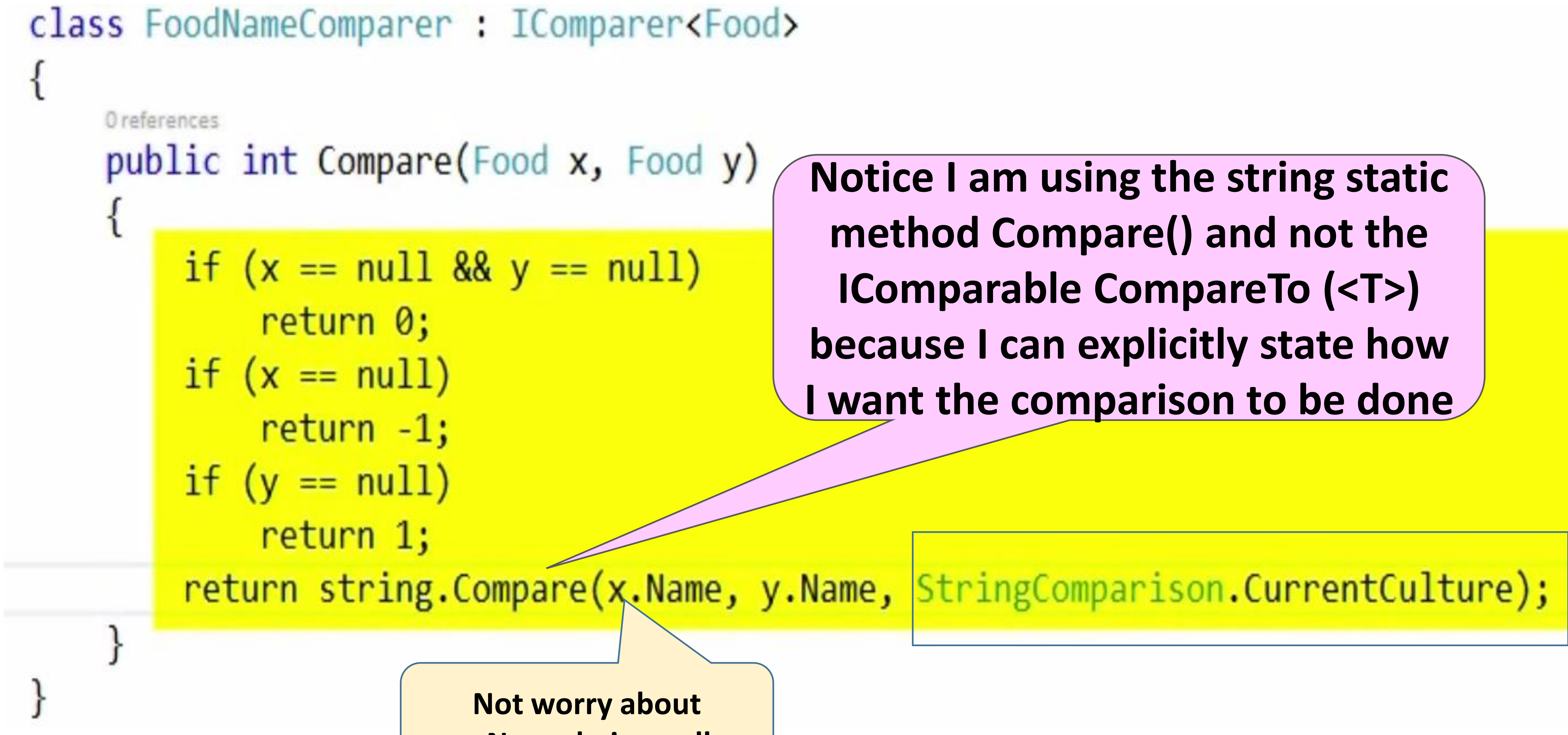
Syntax

C# | C++ | F# | VB | Copy

```
int Compare(
        T x,
        T y
)
```

```
IComparer<T>.Compare(x, y) ...
           =0  if  x=y
           <0  if  x<y
           >0  if  x>y
```

Comparer Demo

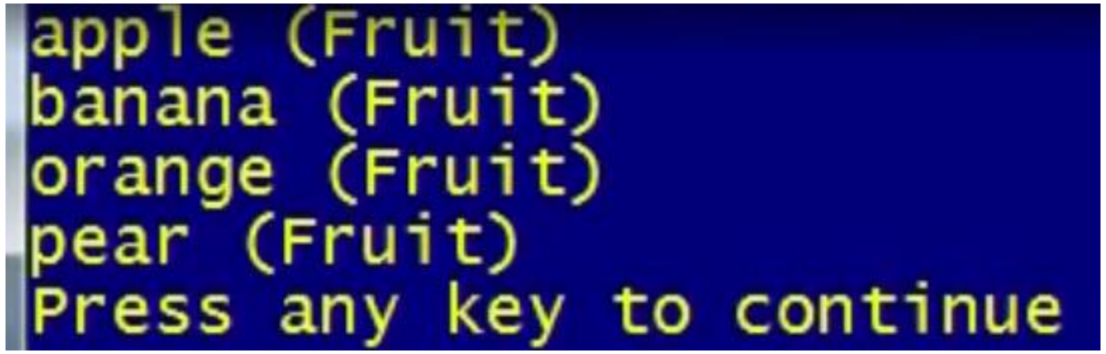


```
class FoodNameComparer : IComparer<Food>
{
    0 references
    public int Compare(Food x, Food y)
    {
        if (x == null && y == null)
            return 0;
        if (x == null)
            return -1;
        if (y == null)
            return 1;
        return string.Compare(x.Name, y.Name, StringComparison.CurrentCulture);
    }
}
```

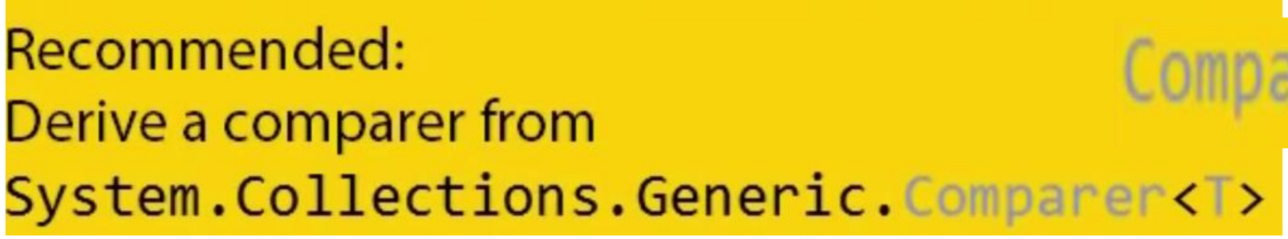
Notice I am using the string static method Compare() and not the IComparable CompareTo (<T>) because I can explicitly state how I want the comparison to be done

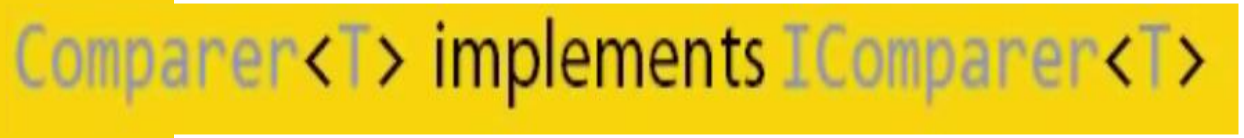
Not worry about x.Name being null

```
static void Main(string[] args)
{
    Food[] list = {
        new Food("orange", FoodGroup.Fruit),
        new Food("banana", FoodGroup.Fruit),
        new Food("pear", FoodGroup.Fruit),
        new Food("apple", FoodGroup.Fruit) };

    Array.Sort(list, new FoodNameComparer());

    foreach (var item in list)
        Console.WriteLine(item);
}
```



```
apple (Fruit)
banana (Fruit)
orange (Fruit)
pear (Fruit)
Press any key to continue
```


Recommended:
Derive a comparer from
`System.Collections.Generic.Comparer<T>`

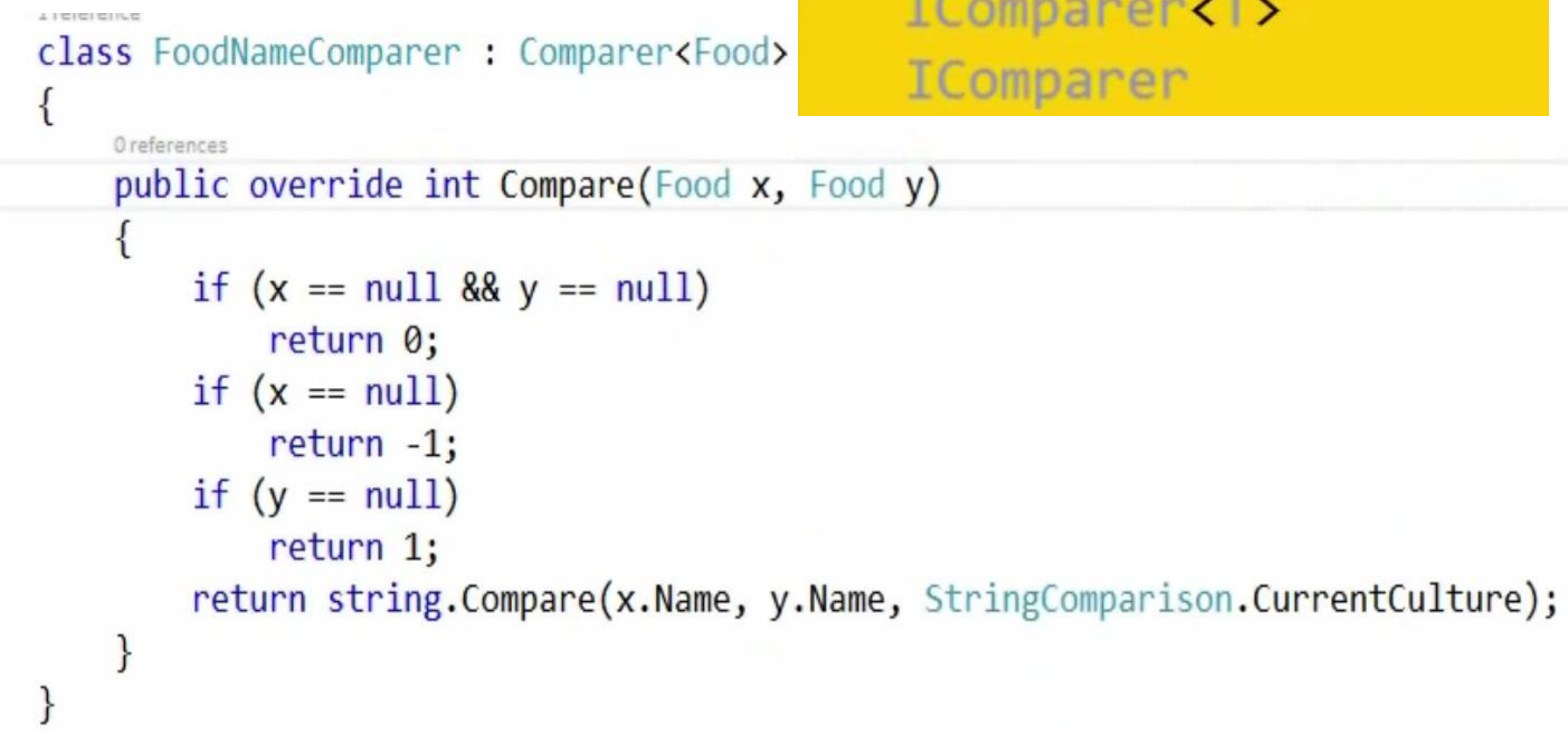

Comparer<T> implements IComparer<T>

Why change from IComparer<T> to Comparer<T> ?

There are additional interfaces you must implement for Comparison eg the legacy IComparer,Compare(Object x, Object y) which is NOT strongly typed

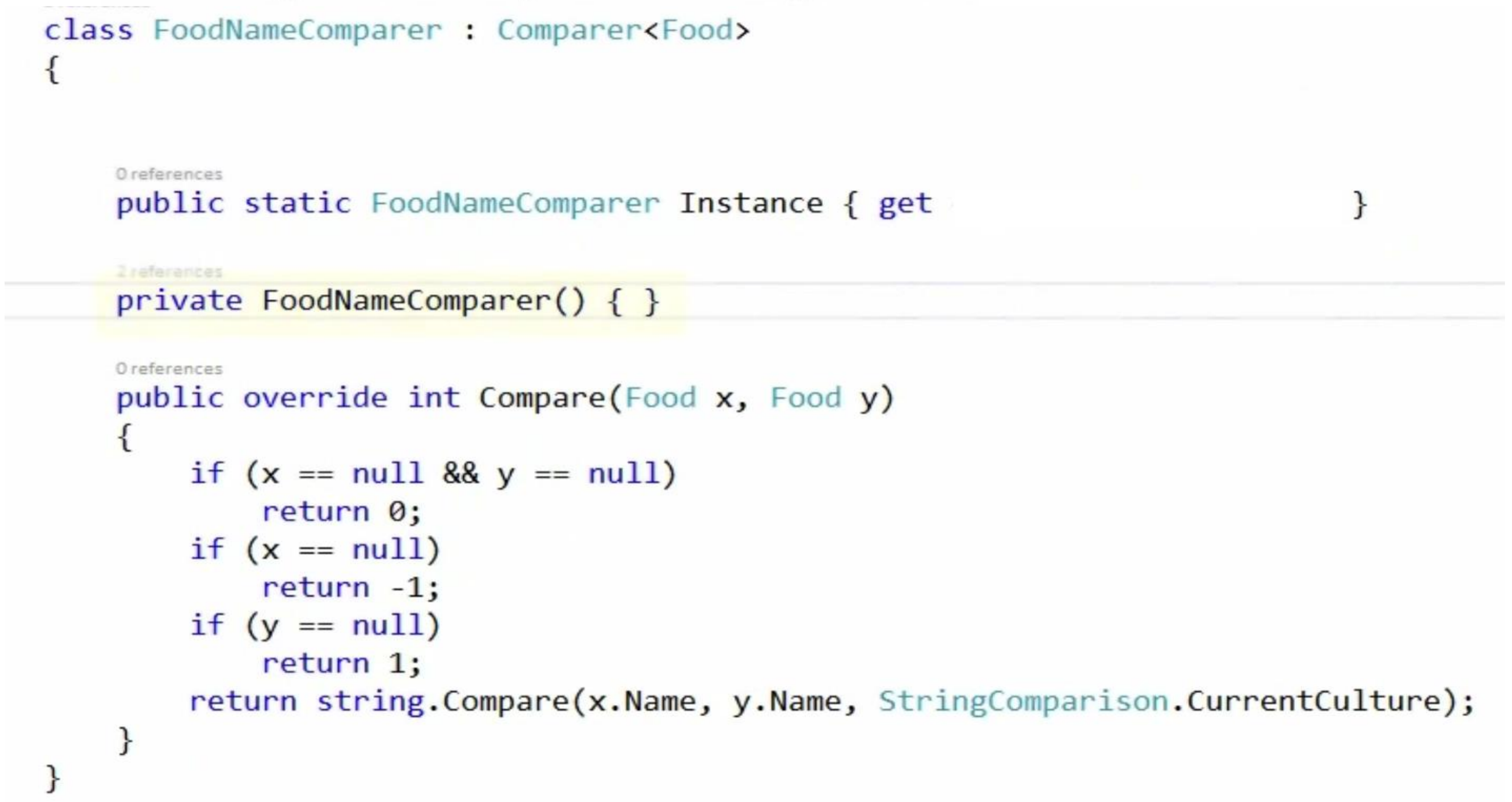
Why use it? - for backword compatibility. Code that cannot use the Generic version can still use the non Genertic interface.

Comparer<T> implements:
    IComparer<T>
    IComparer



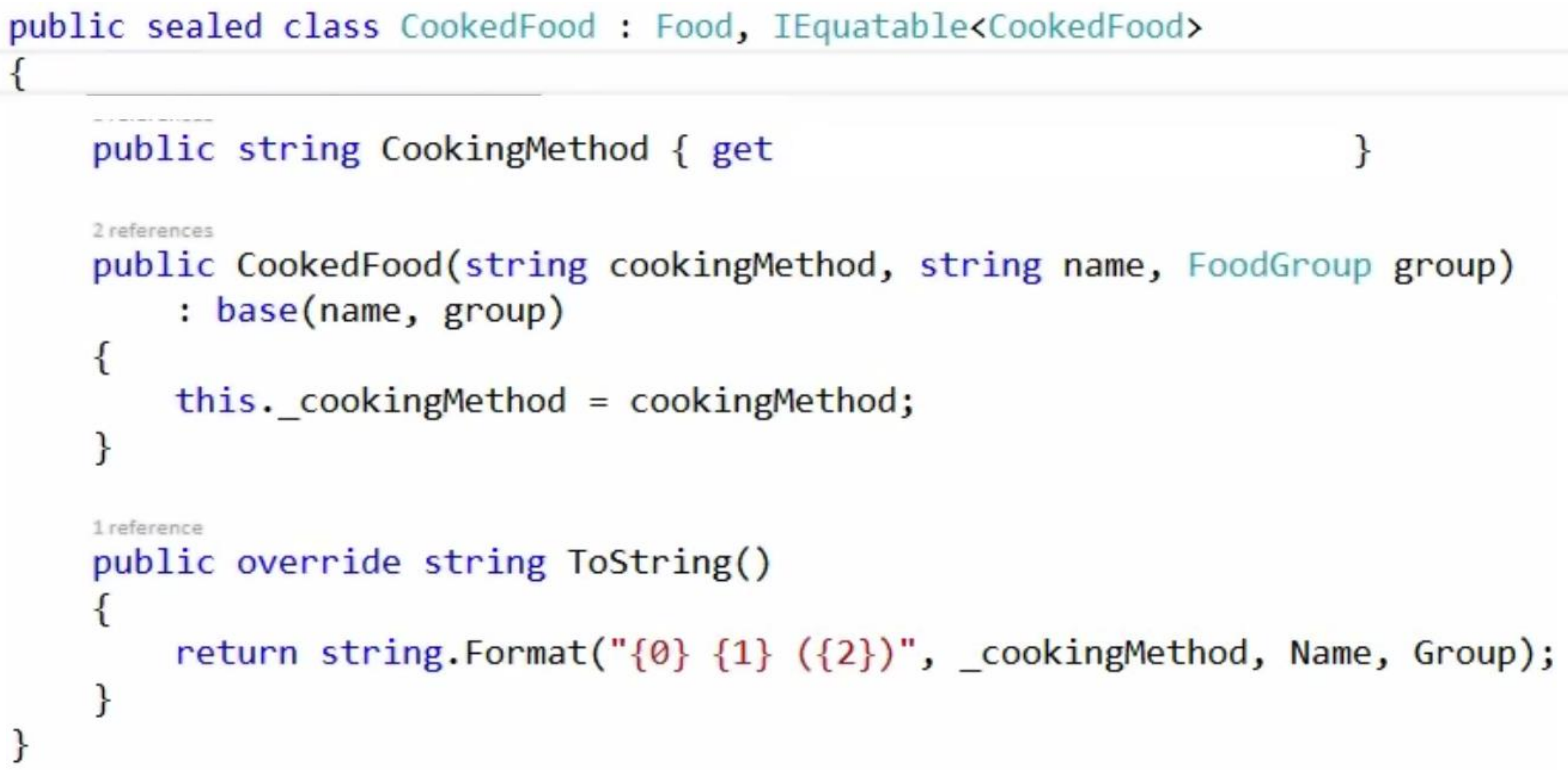
```
1 reference
class FoodNameComparer : Comparer<Food>
{
    0 references
    public override int Compare(Food x, Food y)
    {
        if (x == null && y == null)
            return 0;
        if (x == null)
            return -1;
        if (y == null)
            return 1;
        return string.Compare(x.Name, y.Name, StringComparison.CurrentCulture);
    }
}
```

Making the Comparer a Singleton

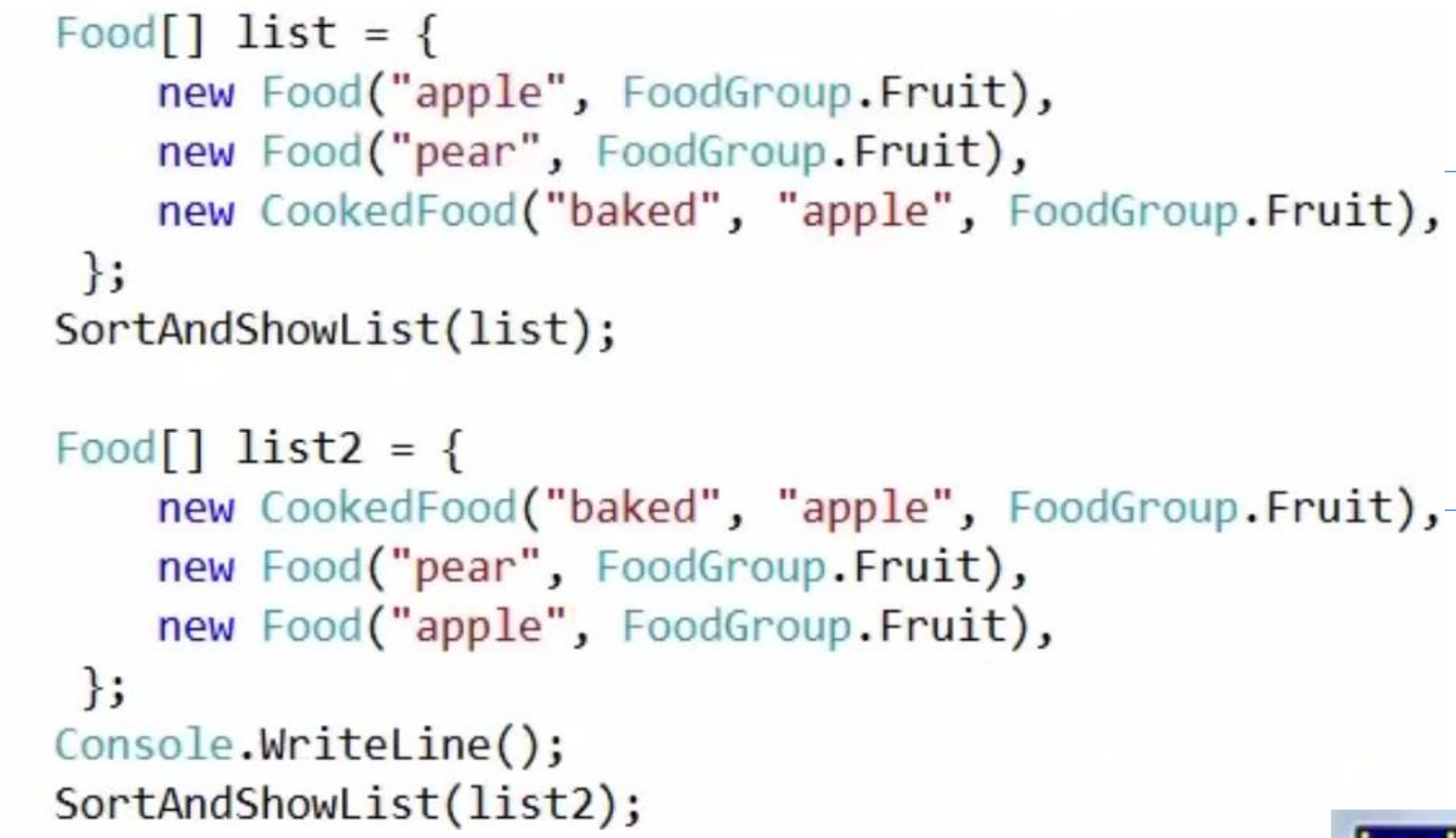


```
class FoodNameComparer : Comparer<Food>
{


    0 references
    public static FoodNameComparer Instance { get                    }

    2 references
    private FoodNameComparer() { }

    0 references
    public override int Compare(Food x, Food y)
    {
        if (x == null && y == null)
            return 0;
        if (x == null)
            return -1;
        if (y == null)
            return 1;
        return string.Compare(x.Name, y.Name, StringComparison.CurrentCulture);
    }
}
```

What about sub classes ?

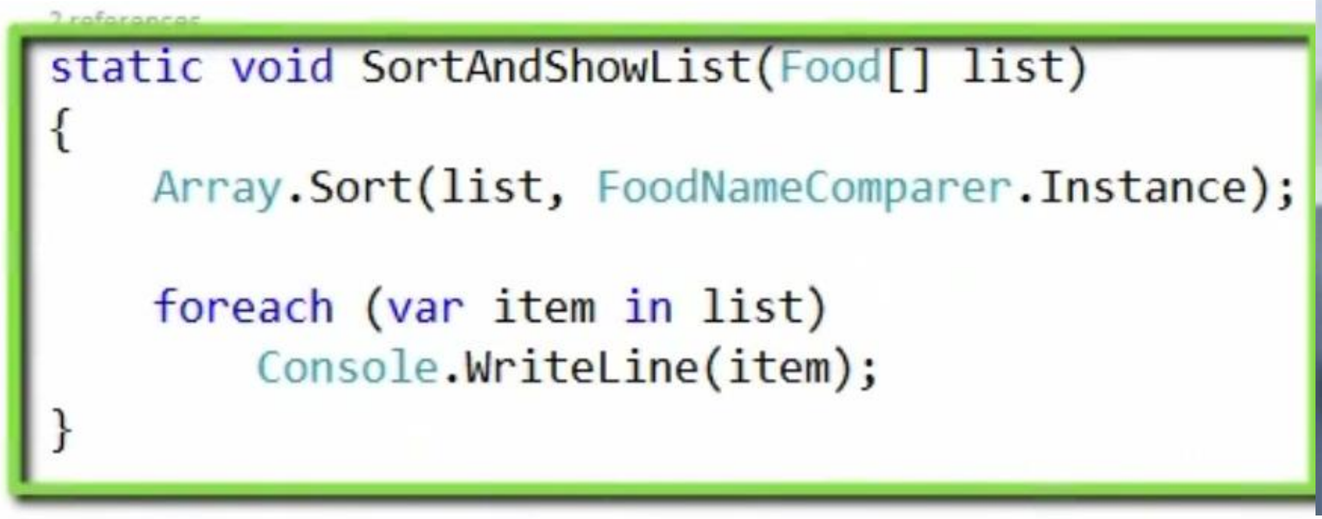


```
public sealed class CookedFood : Food, IEquatable<CookedFood>
{
    public string CookingMethod { get                              }

    2 references
    public CookedFood(string cookingMethod, string name, FoodGroup group)
        : base(name, group)
    {
        this._cookingMethod = cookingMethod;
    }

    1 reference
    public override string ToString()
    {
        return string.Format("{0} {1} ({2})", _cookingMethod, Name, Group);
    }
}
```

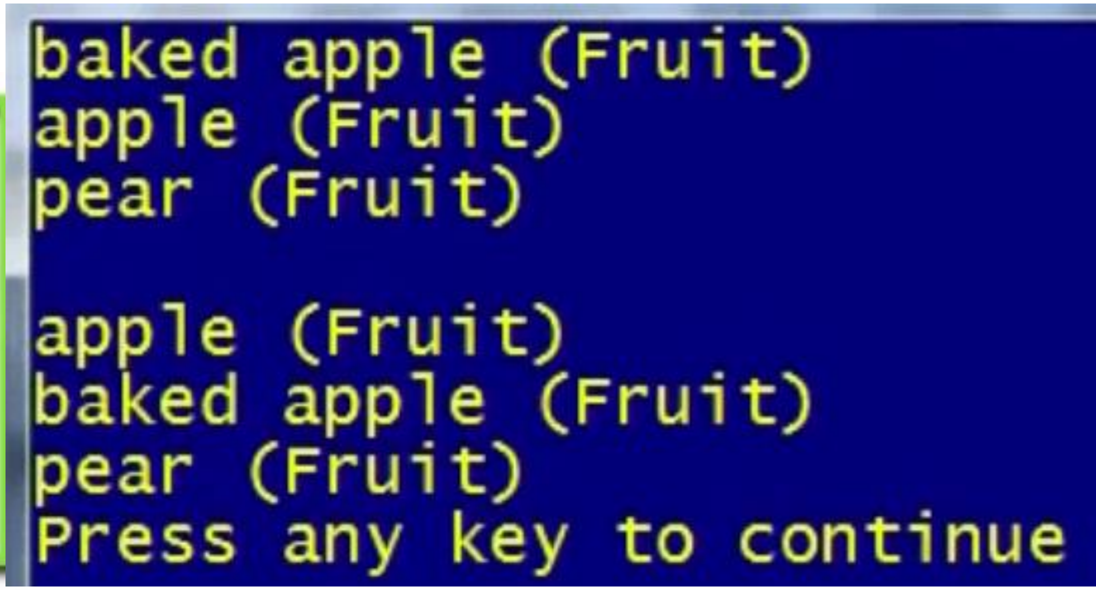


```
Food[] list = {
    new Food("apple", FoodGroup.Fruit),
    new Food("pear", FoodGroup.Fruit),
    new CookedFood("baked", "apple", FoodGroup.Fruit),
 };
SortAndShowList(list);

Food[] list2 = {
    new CookedFood("baked", "apple", FoodGroup.Fruit),
    new Food("pear", FoodGroup.Fruit),
    new Food("apple", FoodGroup.Fruit),
 };
Console.WriteLine();
SortAndShowList(list2);
```

Same lists but different order – should still give u the same results

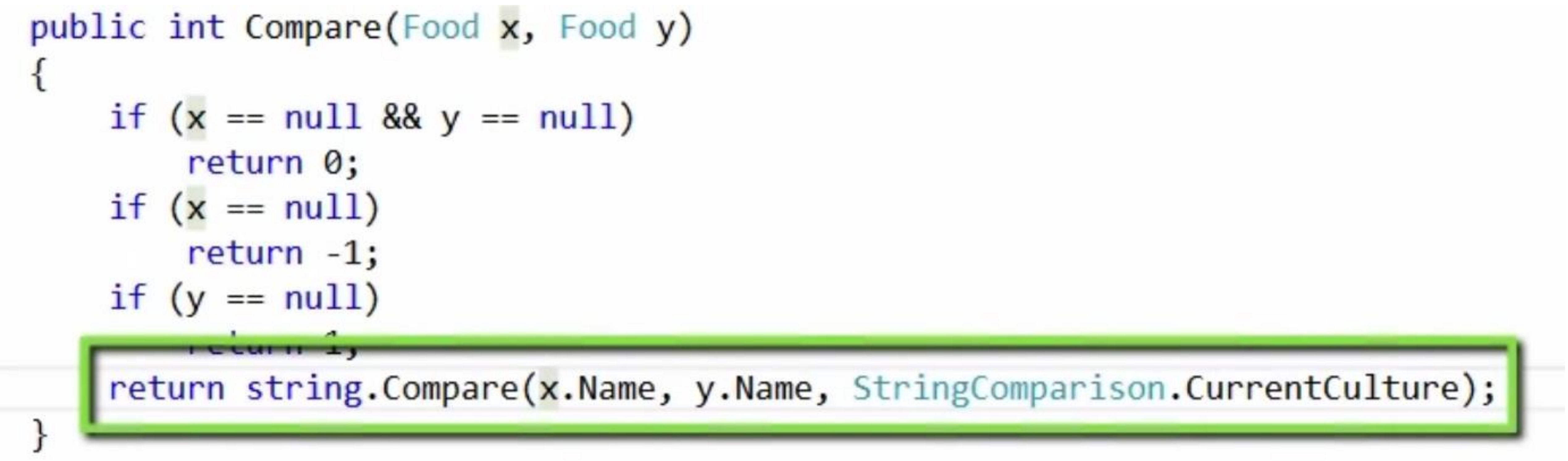


```
2 references
static void SortAndShowList(Food[] list)
{
    Array.Sort(list, FoodNameComparer.Instance);

    foreach (var item in list)
        Console.WriteLine(item);
}
```



```
baked apple (Fruit)
apple (Fruit)
pear (Fruit)

apple (Fruit)
baked apple (Fruit)
pear (Fruit)
Press any key to continue
```

What Happened?



```
public int Compare(Food x, Food y)
{
    if (x == null && y == null)
        return 0;
    if (x == null)
        return -1;
    if (y == null)
        return 1;
    return string.Compare(x.Name, y.Name, StringComparison.CurrentCulture);
}
```

This comparer considers that

```
new Food("apple", FoodGroup.Fruit) ==
    new CookedFood("baked", "apple", FoodGroup.Fruit)
```

- because they both have the same name

Sort order of equal items is indeterminate

Would this be a problem??

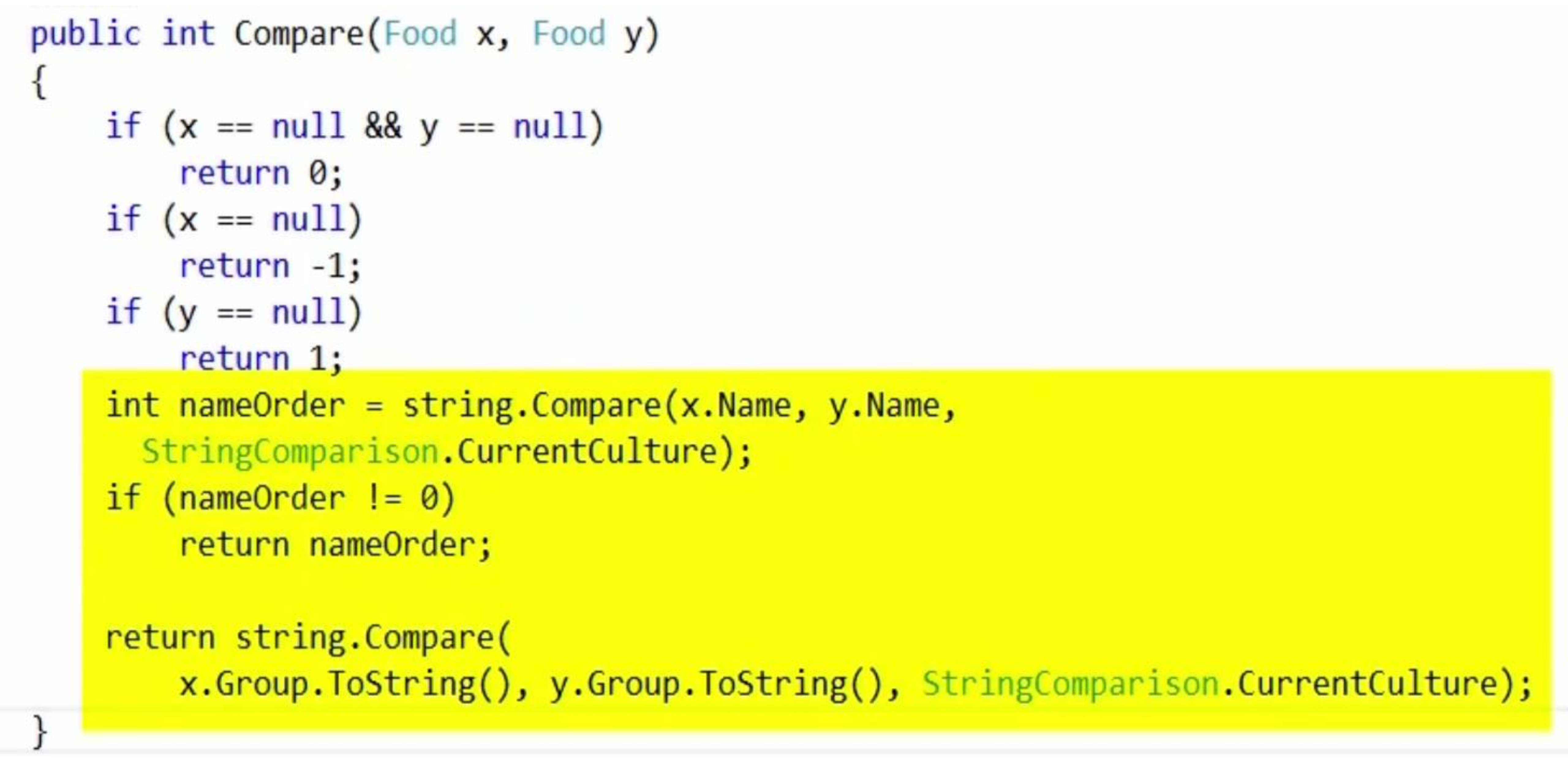
Indeterminate sort order:

- Might confuse end users
- Might break other code

When designing a comparer:

- Think about 'equal' instances
- Consider checking all fields

FIX – Compare all relevant fields



```
public int Compare(Food x, Food y)
{
    if (x == null && y == null)
        return 0;
    if (x == null)
        return -1;
    if (y == null)
        return 1;
    int nameOrder = string.Compare(x.Name, y.Name,
      StringComparison.CurrentCulture);
    if (nameOrder != 0)
        return nameOrder;

    return string.Compare(
        x.Group.ToString(), y.Group.ToString(), StringComparison.CurrentCulture);
}
```

But does this solve the issue when using sub classes – What about the sub class additional Fields ??

A Comparer(of T)
needs knowledge of every type derived from T
to work properly

The real lesson here is...

Writing a comparer for non-sealed classes is problematic:

Consider sealing classes
if you might need to compare them

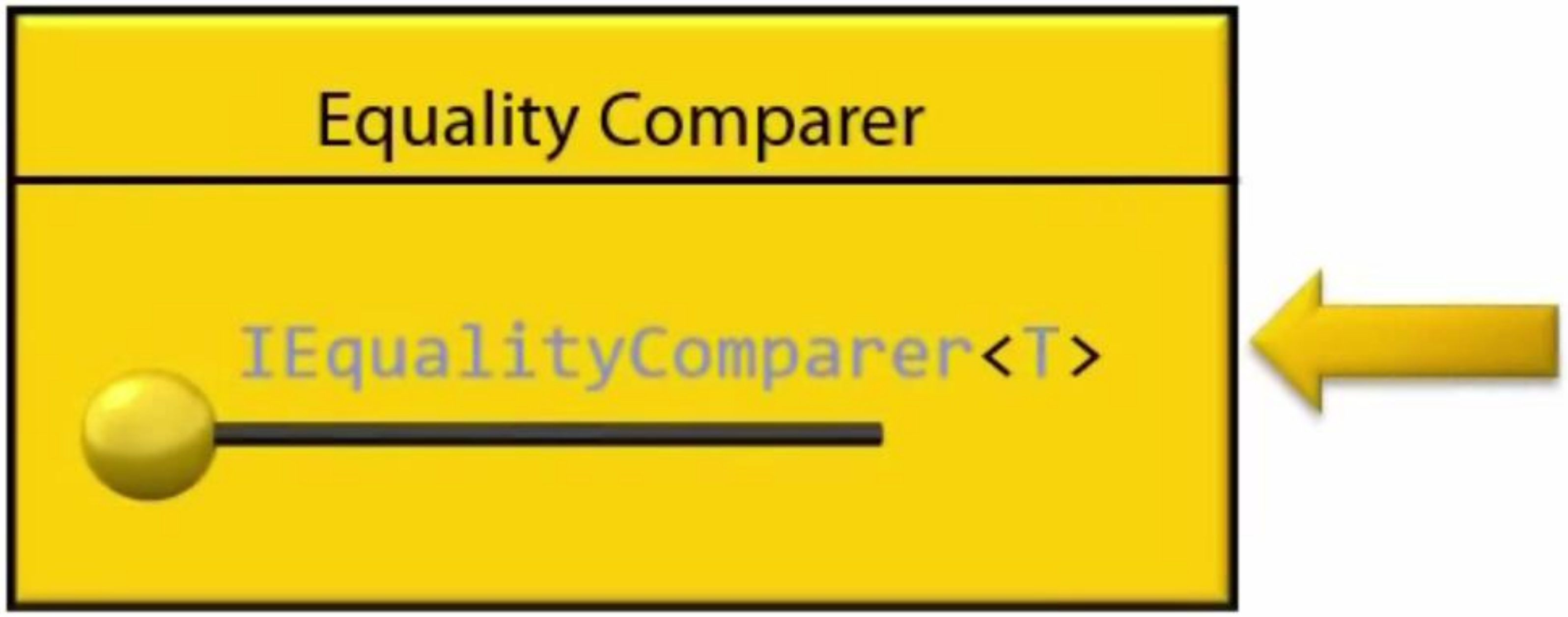
There are no good solutions for derived types – look at your business needs to determine a viable solution – Just be aware of this problem!!

IComparer<T> vs IComparable<T>

IComparable<T>	IComparer<T>
Implemented by T (eg. `class CalorieCount : IComparable<CalorieCount>`)	Implemented by a custom comparer (eg. `class FoodNameComparer : IComparer<Food>`)
Suited to 'natural' comparisons	Allows 'plugging in' alternative comparisons
	Means you can have as many comparers as you want for T

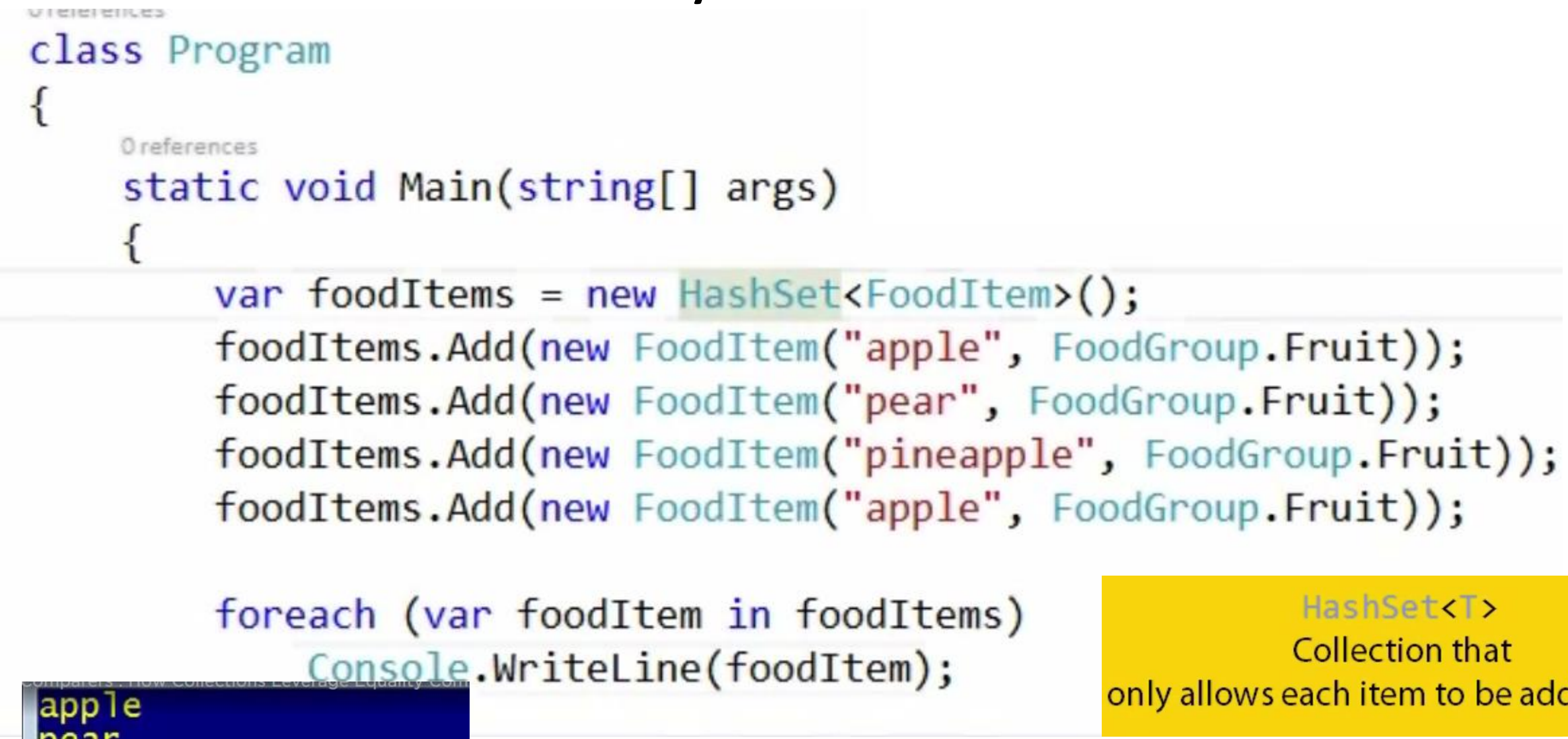
IComparer<T>:
Great example of single responsibility principle

Equality Comparers

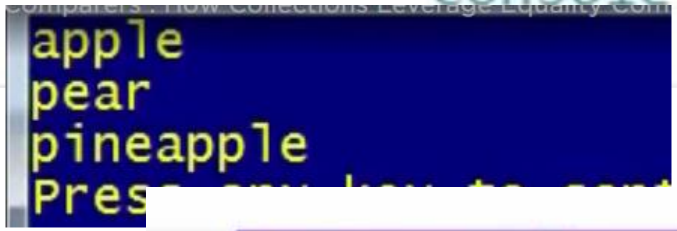


Like `IComparer<T>`
but for equality comparisons

DEMO – Why we need it – Part 1

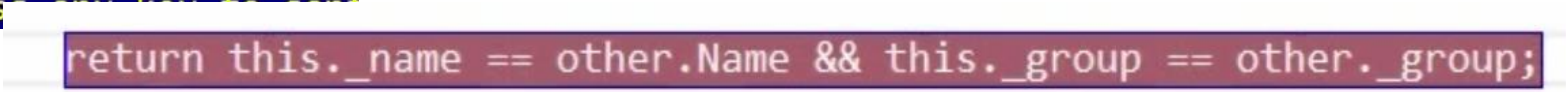


```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        var foodItems = new HashSet<FoodItem>();
        foodItems.Add(new FoodItem("apple", FoodGroup.Fruit));
        foodItems.Add(new FoodItem("pear", FoodGroup.Fruit));
        foodItems.Add(new FoodItem("pineapple", FoodGroup.Fruit));
        foodItems.Add(new FoodItem("apple", FoodGroup.Fruit));

        foreach (var foodItem in foodItems)
            Console.WriteLine(foodItem);
```



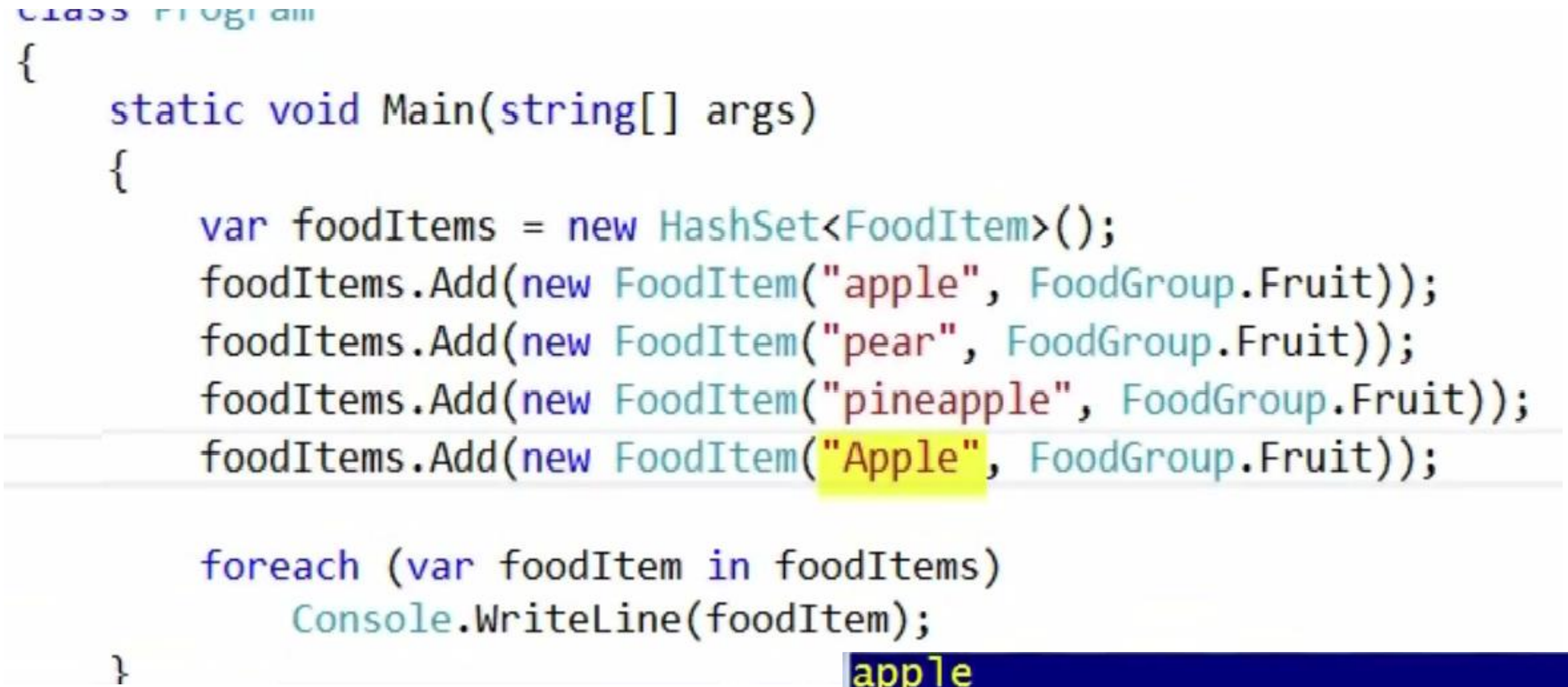
```
apple
pear
pineapple
Pres
```

HashSet<T>
Collection that
only allows each item to be added once

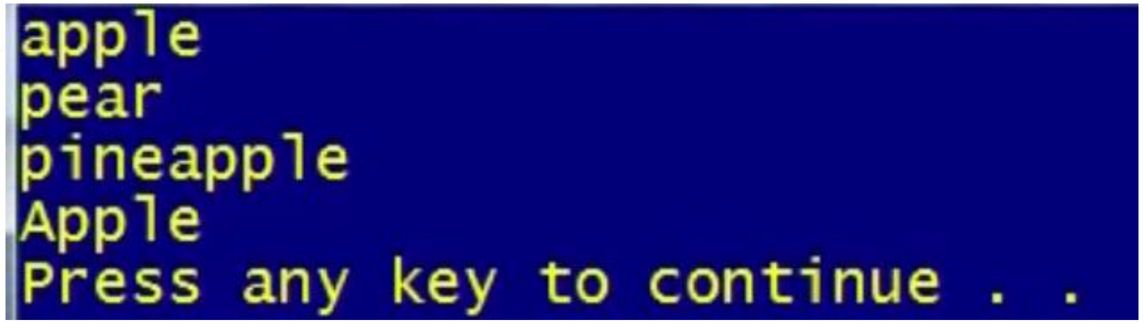


```
return this._name == other.Name && this._group == other._group;
```

DEMO – Why we need it – Part 2



```
class Program
{
    static void Main(string[] args)
    {
        var foodItems = new HashSet<FoodItem>();
        foodItems.Add(new FoodItem("apple", FoodGroup.Fruit));
        foodItems.Add(new FoodItem("pear", FoodGroup.Fruit));
        foodItems.Add(new FoodItem("pineapple", FoodGroup.Fruit));
        foodItems.Add(new FoodItem("Apple", FoodGroup.Fruit));

        foreach (var foodItem in foodItems)
            Console.WriteLine(foodItem);
    }
```



```
apple
pear
pineapple
Apple
Press any key to continue . . .
```

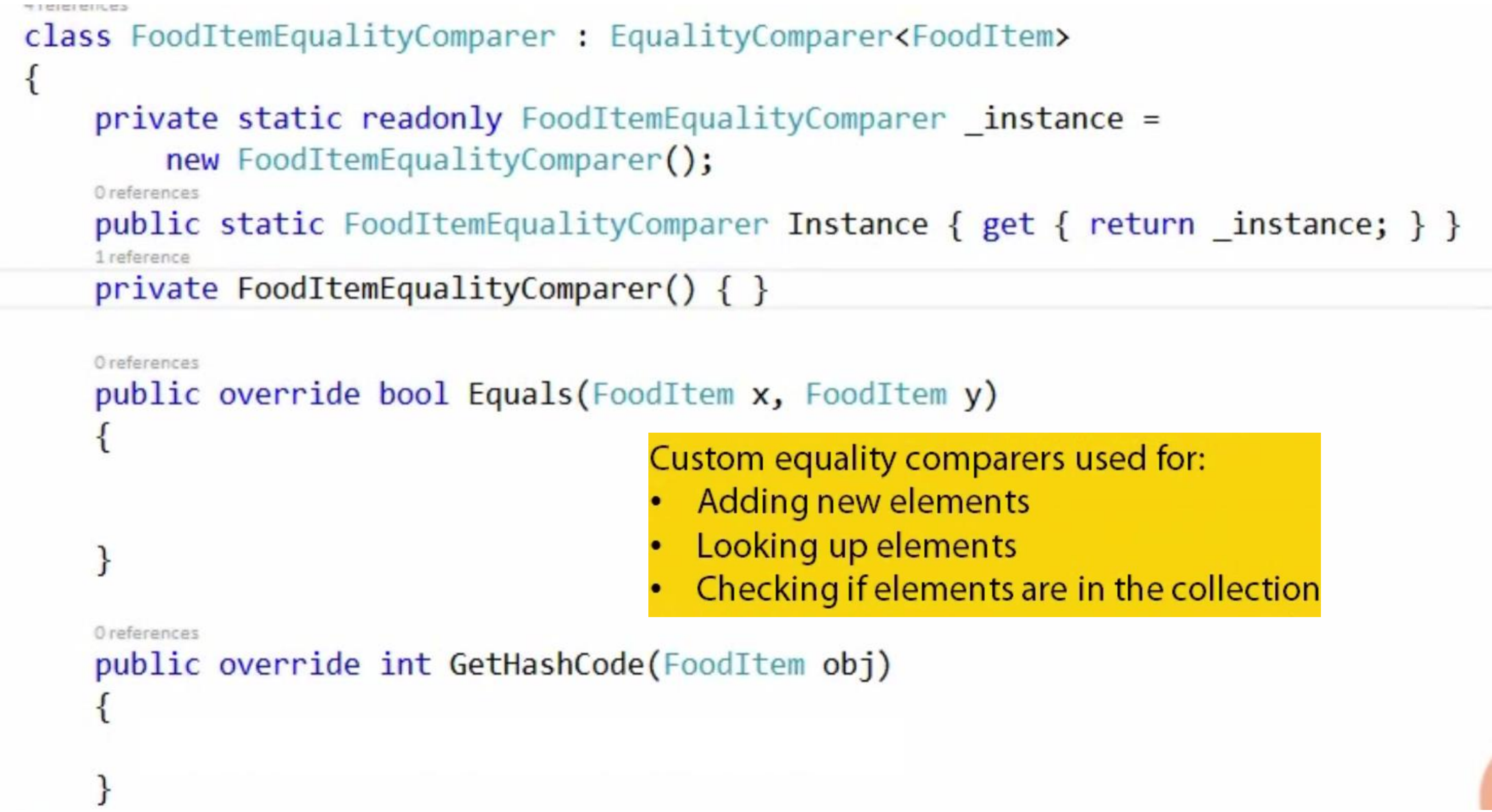
Implementing an Equality Comparer

We need an equality comparer
to modify FoodItem equality behaviour
just for this collection (ie. not everywhere)

When you replace an equality implementation…
… You **always** need to supply
a hash code implementation to match

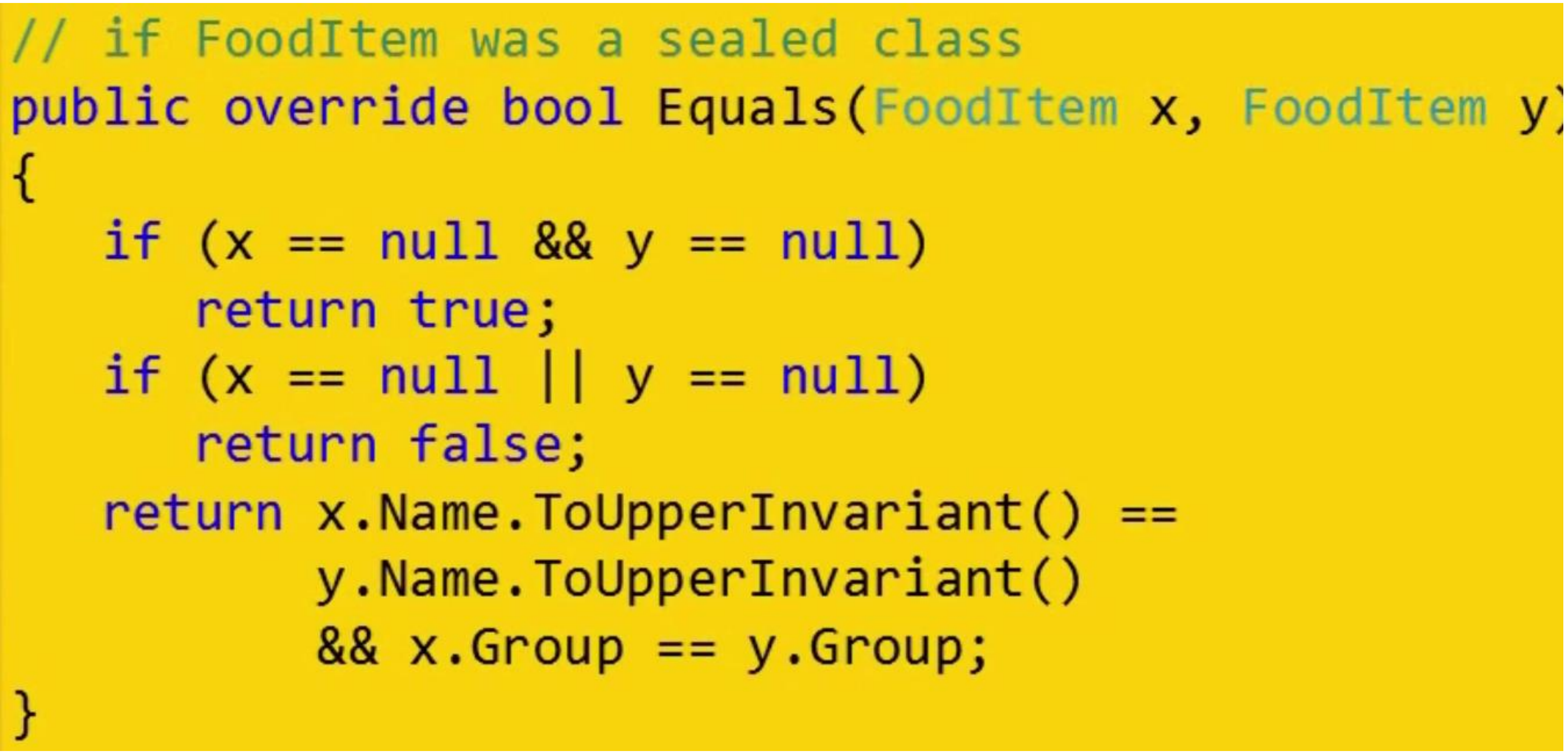
```
EqualityComparer<T>:
  IEqualityComparer<T>, IEqualityComparer
```

- Implements these by using `override Equals()` and `override GetHashCode()`

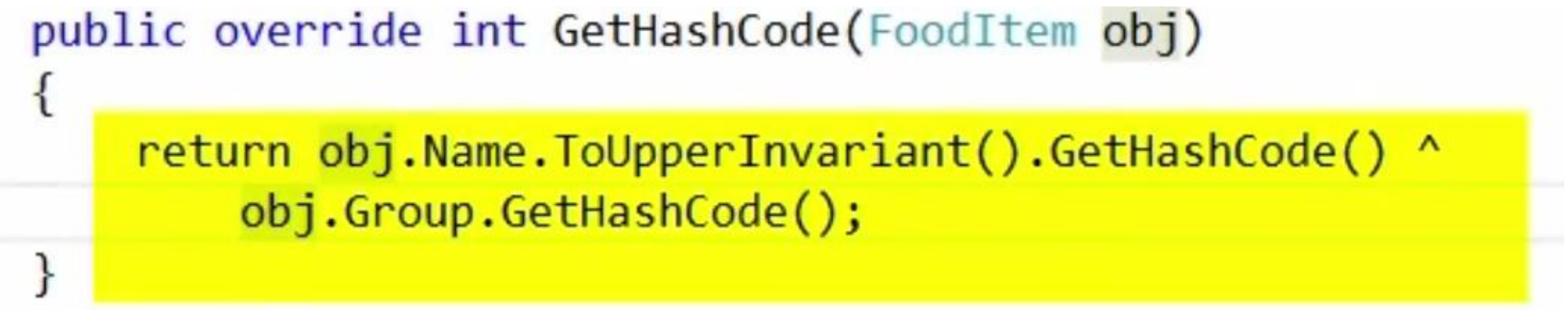


```
class FoodItemEqualityComparer : EqualityComparer<FoodItem>
{
    private static readonly FoodItemEqualityComparer _instance =
        new FoodItemEqualityComparer();
    0 references
    public static FoodItemEqualityComparer Instance { get { return _instance; } }
    1 reference
    private FoodItemEqualityComparer() { }

    0 references
    public override bool Equals(FoodItem x, FoodItem y)
    {


    }

    0 references
    public override int GetHashCode(FoodItem obj)
    {

    }
```

Custom equality comparers used for:
- Adding new elements
- Looking up elements
- Checking if elements are in the collection

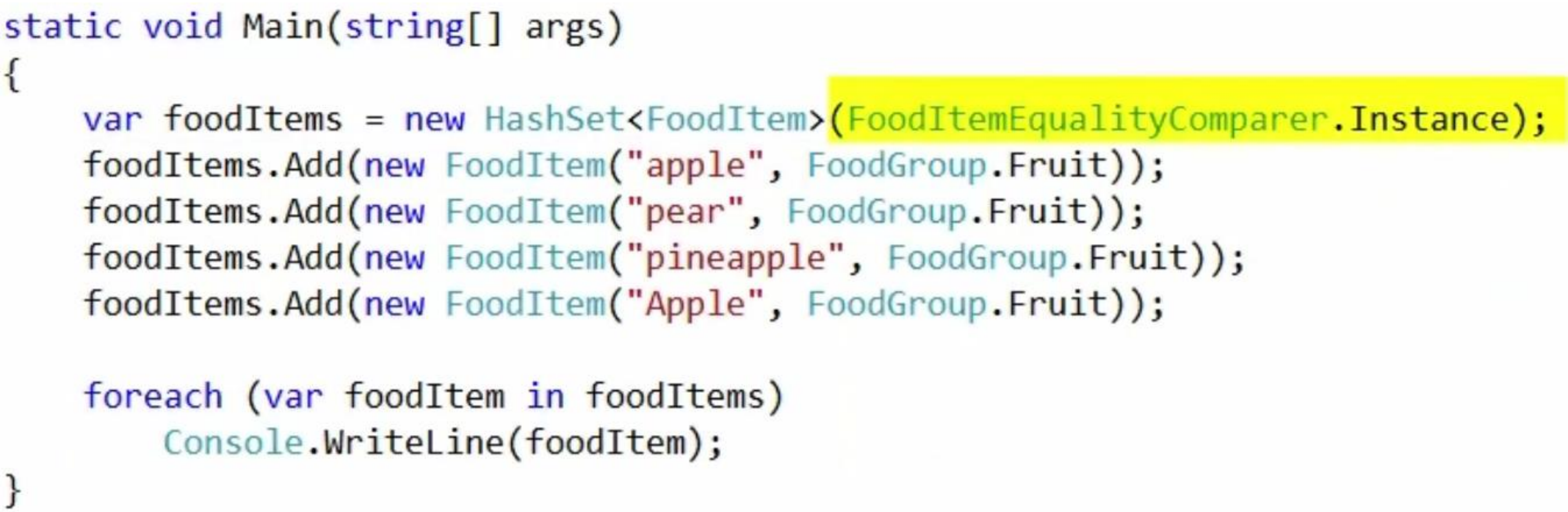


```
// if FoodItem was a sealed class
public override bool Equals(FoodItem x, FoodItem y)
{
   if (x == null && y == null)
      return true;
   if (x == null || y == null)
      return false;
   return x.Name.ToUpperInvariant() ==
          y.Name.ToUpperInvariant()
          && x.Group == y.Group;
}
```

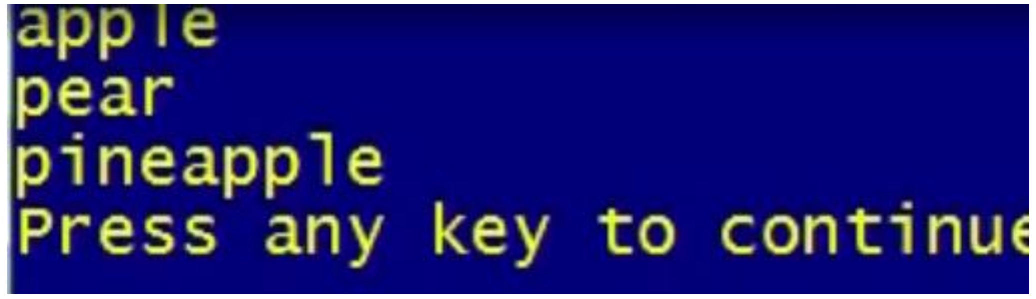


```
public override int GetHashCode(FoodItem obj)
{
   return obj.Name.ToUpperInvariant().GetHashCode() ^
       obj.Group.GetHashCode();
}
```

DEMO – Adding FoodItem instances to HashSet



```
static void Main(string[] args)
{
    var foodItems = new HashSet<FoodItem>(FoodItemEqualityComparer.Instance);
    foodItems.Add(new FoodItem("apple", FoodGroup.Fruit));
    foodItems.Add(new FoodItem("pear", FoodGroup.Fruit));
    foodItems.Add(new FoodItem("pineapple", FoodGroup.Fruit));
    foodItems.Add(new FoodItem("Apple", FoodGroup.Fruit));

    foreach (var foodItem in foodItems)
        Console.WriteLine(foodItem);
}
```



```
apple
pear
pineapple
Press any key to continue
```

Next Week…
Sorting and Searching Algorithms