|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Object-Oriented Programming, Part IV |

|  |
| --- |
| By Per Laursen  25-01-2019 |

[Introduction 2](#_Toc535422728)

[Indexers 3](#_Toc535422729)

[Making a Class Enumerable 6](#_Toc535422730)

[Operator Overloading 10](#_Toc535422731)

[Defining arithmetic operator overloads 11](#_Toc535422732)

[Defining comparison operator overloads 14](#_Toc535422733)

[Extension Methods 18](#_Toc535422734)

[Anonymous Types 18](#_Toc535422735)

[Reflection 18](#_Toc535422736)

[Exercises 19](#_Toc535422737)

[OOP.4.1 19](#_Toc535422738)

[OOP.4.2 20](#_Toc535422739)

[OOP.4.3 21](#_Toc535422740)

[OOP.4.4 22](#_Toc535422741)

# Introduction

TBD

# Indexers

Suppose we are developing a game, where we need to model game participants which are controlled by the game itself – such entities are ofted referred to as **non-player-characters** or just **NPC**s. We have therefore defined a class **NPC**:

**public class NPC**

**{**

**public enum NPCStateTypes**

**{**

**hungry, rested, aggressive, fear, gullible**

**}**

**private Dictionary<NPCStateTypes, int> State;**

**public NPC()**

**{**

**State = new Dictionary<NPCStateTypes, int>();**

**SetDefaultValues();**

**}**

**public void SetStateValue(NPCStateTypes stateType, int value)**

**{**

**State[stateType] = value;**

**}**

**public int GetStateValue(NPCStateTypes stateType)**

**{**

**return State[stateType];**

**}**

**private void SetDefaultValues() { … }**

**}**

Fairly straightforward; we have defined that an **NPC** has a “state”, which amounts to a set of values each describing a certain aspect of the **NPC**, like e.g. the “level” of hun­ger, aggressiveness, etc.. Once created, we can retrieve and update these values by using the methods **GetStateValue** and **SetStateValue**, respectively. All this is per­fectly fine, but the developers using the **NPC** class feel that the **Get**…/**Set**… syntax is a bit cumber­some, and would like a more succinct way of working with these value.

One way of fulfilling this request could be to introduce **indexers** into the **NPC** class. An indexer can be perceived as a way of interacting with values in an object, which is syntactically closer to how you interact with a “raw” collection class object like a **List** or **Dictionary**. For **List** and **Dictionary**, we use the []-oriented syntax, like:

**List<string> names = new List<string>();**

**…**

**Console.WriteLine(names[3]);**

**Dictionary<string, int> ages = new Dictionary<string, int>();**

**…**

**Console.WriteLine(ages["Allan"]);**

If the developers wish to be able to use a syntax like that when interacting with **NPC** objects, couldn’t we just make the **State** property public, such that the develop­ers can interact directly with the dictionary object? It’s possible, but that would be in direct opposition to good practices about encapsulation. What if we decide to use a diffe­rent internal state representation? What we want is to:

* Maintain encapsulation of internal representation
* Offer []-oriented syntax to **NPC** clients.

Indexers can do exactly that, They do not add new functionality to the class, but offer a more convenient (depending on taste, of course…) syntax. For the **NPC** class, we want to be able to write code like this:

**NPC npcA = new NPC();**

**npcA[NPC.NPCStateTypes.aggressive] = 12;**

**npcA[NPC.NPCStateTypes.hungry] = 3;**

**Console.WriteLine(npcA[NPC.NPCStateTypes.aggressive]);**

In the two lines following object creation, we use the []-syntax in a **set**-like manner, i.e. if the left-hand side was a property, we would be invoking the **set**-part of that pro­perty. In the last line, it would be the **get**-part of the property.

The implementation of an indexer reflects these considerations. Below is an imple­mentation of an indexer for the **NPC** class:

**public int this[NPCStateTypes stateType]**

**{**

**get { return State[stateType]; }**

**set { State[stateType] = value; }**

**}**

The syntax may look slightly cryptic, but if you compare it to an ordinary property, the dif­ference is actually not that great. Suppose we had a single property for each of the **NPC** state values, like this:

**public int Hungry**

**{**

**get { return State[NPCStateTypes.hungry]; }**

**set { State[NPCStateTypes.hungry] = value; }**

**}**

The indexer defined above and the **Hungy** property are both called on **NPC** objects:

**npcA.Hungry = 3;**

**npcA[NPC.NPCStateTypes.hungry] = 3;**

The indexer is thus nothing more than a nameless, parameterised property, which makes it a bit more convenient to interact with an object. Still, we do retain the pos­si­bility of e.g. performing validation of input values. An updated version of the index­er could like like this (having defined a method **ValueIsValid** in the class as well):

**public int this[NPCStateTypes stateType]**

**{**

**get{ return State[stateType]; }**

**set**

**{**

**if (ValueIsValid(value))**

**{**

**State[stateType] = value;**

**}**

**else**

**{**

**throw new ArgumentException("...");**

**}**

**}**

**}**

For completeness, it should be mentioned that you can define as many indexers as you wish in a single class, as long as the parameter to the indexer has a different type in each indexer definition. If it somehow makes sense to define an indexer based on strings in the **NPC** class, you can just go ahead and add it:

**public int this[string stateDescription]**

**{**

**get { … }**

**set { … }**

**}**

# Making a Class Enumerable

With the addition of an indexer to the **NPC** class, you can now retrieve all the state values in an **NPC** object with a **foreach**-loop, which is outlined below:

**foreach (var state in listOfStateTypes)**

**{**

**Console.WriteLine(npcA[state]);**

**}**

Fairly convenient, but there are a couple of hitches here… First of all, where do we get this list of state types from? Since we wish to query the **NPC** object for the value of all existing state types, that list must contain all these types. So, we can just create such a list:

**List<NPC.NPCStateTypes> listOfStateTypes = new List<NPC.NPCStateTypes>**

**{**

**NPC.NPCStateTypes.aggressive,**

**NPC.NPCStateTypes.hungry,**

**NPC.NPCStateTypes.rested,**

**NPC.NPCStateTypes.fear,**

**NPC.NPCStateTypes.gullible**

**};**

**foreach (var state in listOfStateTypes)**

**{**

**Console.WriteLine(npcA[state]);**

**}**

It will work, but it is a bit fragile to changes. What if we at some point add new states types? Will we remember to update this list? An improved version can be made by using the fact that you can enumerate all member of an **enum** definition, with a bit of grey magic:

**List<NPC.NPCStateTypes> stateTypes =**

**Enum.GetValues(typeof(NPC.NPCStateTypes))**

**.Cast<NPC.NPCStateTypes>()**

**.ToList();**

**foreach (var state in listOfStateTypes)**

**{**

**Console.WriteLine(npcA[state]);**

**}**

Now we will always loop through all existing state types.. but the developers are not really comfortable with this approach. They would much rather be able to do some­thing like this:

**foreach (var stateTypeEntry in npcA)**

**{**

**Console.WriteLine(stateTypeEntry);**

**}**

Instead of iterating over a list of state types, and querying the **NPC** object for each state type, they wish to iterate over the **NPC** object itself! This iteration should then produce a sort of “entry”, which could e.g. be the state type itself, plus its current value in the object. Can we make that happen?

This is indeed possible, but requires that we turn **NPC** objects into “enumerable” objects. The **foreach**-loop can iterate over any object, as long a one requirement is fulfilled: the object must implement the **IEnumerable<T>** interface, **T** being the type of the values “returned” when iterating over the object. The work “returned” has to be used carefully here. Consider this somewhat useless **foreach**-loop:

**foreach (var state in listOfStateTypes)**

**{**

**}**

This doesn’t “return” anything, right? The “return” we are talking about here is the set of values which the **foreach**-loop will extract from the object following the **in** keyword. In this case, that object has the type **List<NPCStateTypes>**, and the values extracted by the foreach-loop has type **NPCStateTypes**. What decides this is the fact that the **List<T>** class implements the interface **IEnumerable<T>**. So, given a **List** containing items of type **T**, the **foreach**-loop will extract values of type **T**. Now we are handing the **foreach**-loop an **NPC** object, and we want it to extract a set of *statetype-plus-current-value* values from it. This translates to requiring that **NPC** implements the interface **IEnumerable**<*statetype-plus-current-value*>.

Let’s first focus on the type of this *statetype-plus-current-value* thing. If we go with the idea that the returned values should be a “pair” consisting of a state type and its current value, we have a couple of options:

* Define a new class specifically for that purpose.
* Use the **Tuple**[[1]](#footnote-1) class.

We decide to go with the **Tuple** stategy, and define the type of the returned values to be of type **Tuple<NPCStateTypes, int>**. The consequence of this is that the **NPC** class must implement the interface **IEnumerable< Tuple<NPCStateTypes, int>>**. If we add this to the **NPC** class definition, **ReSharper** can generate a skeleton version of the needed code:

**public IEnumerator<Tuple<NPCStateTypes, int>> GetEnumerator()**

**{**

**yield break;**

**}**

**IEnumerator IEnumerable.GetEnumerator()**

**{**

**return GetEnumerator();**

**}**

The last of the two methods is generated for backwards compatibility reasons, and we will not focus further on this method. The first method is however interesting. What is the significance of the **yield** keyword? In order to understand this, we need a bit of background on how the **foreach**-loop interacts with the object it iterates over.

In each iteration, the **foreach**-loop will ask the object for the next value, i.e. the next value to assign to the loop variable. This question is asked by calling **GetEnumerator** on the object. However, instead of returning the entire set of values when called, **GetEnumerator** will only return the next of these values. Furthermore, when it gets called again, it will not start over and return the “first” value again. In a sense, **Get­Enu­merator** is asked for the next value, gets suspended, is asked for the next value, gets suspended, and so on until **GetEnumerator** does not return any more values. The **yield return** keyword expresses this special mode of execution. The actual imple­mentation can then be done in several ways. One way could simply be to return (one by one) a set of fixed values:

**public IEnumerator<Tuple<NPCStateTypes, int>> GetEnumerator()**

**{**

**yield return new Tuple<NPCStateTypes, int>(..., ...);**

**yield return new Tuple<NPCStateTypes, int>(..., ...);**

**yield return new Tuple<NPCStateTypes, int>(..., ...);**

**}**

Another – more robust – way is to simply use the source for the values in the imple­mentation, in our case the **State** dictionary:

**public IEnumerator<Tuple<NPCStateTypes, int>> GetEnumerator()**

**{**

**foreach (var keyValuePair in State)**

**{**

**NPCStateTypes stateType = keyValuePair.Key;**

**int val = keyValuePair.Value;**

**yield return new Tuple<NPCStateTypes, int>(stateType, val);**

**}**

**}**

**GetEnumerator** will now return exactly one value corresponding to each entry in the **State** dictionary, which is precisely want we want. All that remains is the logic for tur­ning each entry in **State** (which is a dictionary key-value pair) into a **Tuple** with the same values.

With this definition added to the **NPC** class, we can now iterate directly over an NPC objects, like this:

**foreach (var entry in npcA)**

**{**

**Console.WriteLine($"{entry.Item1} -> {entry.Item2}");**

**}**

The idea of making an object “enumerable” does of course only make sense if the object contains (or can generate) data of a collection-type nature. If a **Person** class only conatins simple fields like name, address, etc.., it hardly makes sense to let it implement **IEnumerable**. What if the **Person** class also contains a list of telephone numbers? Should you then rush ahead and make it enumerable? Only if it makes sense in some context, i.e. makes life easier for clients of the **Person** class. Don’t make a class enumerable just because you can do it, since it may end up being more confusing – and even error-prone – than not including such functionality.

# Operator Overloading

One of the first topics we discussed in the early stages of our journey through C# was the concept of **arithmetics**, i.e. that we can perform arithmetic operations such as addition, multiplication etc. on variables of a numeric type. Code like this should be trivial to understand by now:

**int a = 17;**

**int b = 25;**

**int c = a + b;**

Let’s now look at the last statement in more detail. The right-hand side is an expres­sion, which consists of two operands (the variables **a** and **b**) and an operator (the **+**). The **+** operator is a binary operator, since it requires two operands to do its job. In con­trast, the **++** operator is a unary operator, since it only applies to a single ope­rand, like this:

**a++;**

The meaning of an operator, i.e. the algorithm it follows to produce the result of applying it to the operand(s), is often fairly obvious; we expect the result of applying the **+** operator to two numeric operands simply to be the sum of the two values. We can also apply the **+** operator to two operands of type **string**:

**string firstName = "Per ";**

**string lastName = "Laursen";**

**string fullname = firstName + lastName;**

The result will here be that **fullName** is set equal to the string *“Per Laursen”*. This is something we have been accustomed to as well, but this is maybe not so obvious a definition as for numeric operands. Now, suppose we have defined a simple class **Car**, and try the following:

**Car carA = new Car("AB 123", 50000);**

**Car carB = new Car("CD 345", 80000);**

**Car carC = carA + carB; // NB: Not valid code**

As suggested by the comment, this code will not compile! The compiler reports that it *“…cannot apply operator ‘+’ to operands of type Car and Car”*. This makes perfect sense. What should the result be of “adding” two **Car** objects…? That is probably hard to define in a meaningful way, so we cannot use the ‘+’ here.

We can, however, imagine classes that have a nature where the idea of “adding” two objects of that class will make sense. Suppose we have defined a simple class **Time** for representing a period of time (the .NET class library does have built-in classes for this purpose, but we’ll ignore those for the sake of the example…):

**public class Time**

**{**

**public int Hours { get; private set; }**

**public int Minutes { get; private set; }**

**public Time(int hours, int minutes)**

**{**

**Hours = hours;**

**Minutes = minutes;**

**}**

**public override string ToString()**

**{**

**return $"The time is {Hours}:{Minutes}";**

**}**

**}**

With that definition, we could write up code like this:

**Time timeA = new Time(2, 25);**

**Time timeB = new Time(1, 45);**

**Time timeC = timeA + timeB; // NB: Not valid code (yet...)**

Just as before, this does not work as-is. But in this case, it could actually make sense to be able to “add” two **Time** object. The logic for adding two periods of time is fairly straightforward; in the case above, the expected result would be a **Time** object with **Hours** = 4 and **Minutes** = 10. The question is now; can we make it possible to add two **Time** object? Yes, by use of **Operator Overloading**.

## Defining arithmetic operator overloads

First of all, it may seem strange to denote what we are about to do as “overloading”. The usual definition of “overloading” is to define several methods with the same name, but with different argument lists, like:

**int Add(int a, int b);**

**double Add(double a, double b);**

But if e.g. the **Time** class does not contain any definition of the **+** operator to begin with, are we then “overloading”? The reason for use of the term is probably that by defining the **+** operator in the **Time** class, we are creating yet another “overload” of the **+** operator in general; it can now be applied to operands of one more type.

So, we will now implement an “overloading” of the **+** operator in the **Time** class. The general syntax for overloading a binary operator looks like this:

**public static TReturn operator *operatorGoesHere*(TA opA, TB opB)**

**{**

**…**

**}**

Notice that the overload is **static**, i.e. you do not invoke an operator on an object it­self, but rather apply it to one or more objects. So, this operator takes two operands; one of type **TA** and one of type **TB**, and returns a result of type **TResult**. This may look a bit cryptic, so let’s see a concrete example: the outline of overloading the **+** opera­tor in the **Time** class:

**public static Time operator +(Time tA, Time tB)**

**{**

**…**

**}**

This is a very typical structure for an operator overload definition: both operands have the same type, and the result has the same type as the operands. This is as we would expect it for e.g. adding two integers: an integer plus an integer is an integer. We can now implement the logic for the operator:

**public static Time operator +(Time tA, Time tB)**

**{**

**int totalMin = (tA.Hours \* 60 + tA.Minutes) +**

**(tB.Hours \* 60 + tB.Minutes);**

**return new Time(totalMin / 60, totalMin % 60);**

**}**

The details of the logic are not as such interesting (feel free to verify it ☺); the point is that with this definition in place, the below line of code becomes valid:

**Time timeC = timeA + timeB;**

A maybe not-so-obvious consequence is that if we create some additional **Time** objects, we can also write (valid) code like this:

**Time timeSum = timeA + timeB + timeC + timeD;**

We can thus use the **+** operator for **Time** objects in the same way as we would use it for numerical types. So far, so good…

Could we imagine other operators that would make sense to apply to **Time** object(s)? Subtrac­tion (i.e. the **–** operator) could also make sense, and the implementation of it would be very simi­lar to the implementation of the **+** operator. What about multipli­ca­tion, i.e. the **\*** operator? It does not seem very meaningful to multiply two **Time** objects, but it could make semse to multiply a **Time** object with e.g. an integer value. This is also fairly easy to implement:

**public static Time operator \*(int val, Time t)**

**{**

**int totalMin = (t.Hours \* 60 + t.Minutes) \* val;**

**return new Time(totalMin / 60, totalMin % 60);**

**}**

With this definition in place, let’s try out this code:

**Time timeMult3A = 3 \* timeA; // OK**

**Time timeMultA3 = timeA \* 3; // Error**

First line of code works fine, but not the second one…? The problem lies (of course) in the definition of the **\*** operator, but in a rather subtle way. Since the order of the parameters to the **\*** operator is (**int**, **Time**), the **\*** operator can then only be applied in expressions where the operands are placed in that specific order… This is some­what limiting, so we will often define the “mirrored” vesion as well:

**public static Time operator \*( Time t, int val)**

**{**

**return val \* t;**

**}**

One point to note here is that we have not just copied the logic from the first defini­tion, but instead implemented the second definition by invoking the first definition. In this way, we have only defined the logic for the **\*** operator in a single place (we like DRY code ☺), while offering the client of the **Time** class freedom of choice w.r.t. the ordering of the operands.

Overloading binary operators is thus fairly simple, once you have figured out the appropriate logic for the operators. Overloading unary operators like **++** is also possible, and could also make sense for the **Time** class. Here is an overload of the **++** operator for the **Time** class:

**// ++ will add one minute to the time**

**public static Time operator ++(Time t)**

**{**

**return (t + new Time(0,1));**

**}**

We can now write:

**timeA++; // OK**

**++timeA; // Also OK**

Again, just as we are used to being able to do for numeric types. Note that for unary operators, you must comply to a definition like the above; the parameter type and the return type must be identical.

## Defining comparison operator overloads

Another useful feature for the **Time** class could be the ability to compare **Time** objects to each other. The first consideration here should again be to consider if it makes sense to compare two **Time** objects. Suppose we wish to be able to decide if one **Time** object is “larger” than another. A natural definition of the “size” of a **Time** object could simply be the length of the time period it represents. First, we augment the **Time** class itself with a **Length** property:

**public int Length**

**{**

**get { return Hours \* 60 + Minutes; }**

**}**

We can then define an overload of the **>=** (larger-than-or-equal-to) operator like this:

**public static bool operator >=(Time tA, Time tB)**

**{**

**return (tA.Length >= tB.Length);**

**}**

Quite simple, but the compiler complains… More specifically, it reports that *“…the operator >= requires a matching operator <= to be defined.”.* So, no >= without <=. We therefore add a corresponding definition of **<=** to the **Time** class:

**public static bool operator <=(Time tA, Time tB)**

**{**

**return (tA.Length <= tB.Length);**

**}**

Now the compiler is satisfied, and we can compare **Time** objects as we see fit:

**bool aVSb = timeA <= timeB;**

Note that if we also wish to be able to compare using **>** (larger-than), we must imple­ment this operator (and **<**) as well.

The final operator to consider is the equality operator **==**. It would probably also be convenient to be able to decide if two **Time** objects are “equal”. This is, however, a bit more tricky. Consider the below code, which is valid even without having defined the **==** operator in **Time**:

**Time tX = new Time(3, 45);**

**Time tY = new Time(3, 45);**

**Console.WriteLine($"tX == tY is {tX == tY}");**

What is the result…? The result is **false**, the reason being that if we do not override the **==** operator, it will test whether or not the two objects are identical, which is dif­fe­rent that being equal. What are we actually comparing in the expression? We are comparing if the objects referred to by the variables **tX** and **tY** are identical, i.e. that the two variables refer to the same object! This is not the case here… Consider this alternative code:

**Time tX = new Time(3, 45);**

**Time tY = tX;**

**Console.WriteLine($"tX == tY is {tX == tY}");**

The result will now be **true**. If we want to change this behavior, we must override the **==** operator (which in turn also requires us to override the **!=** operator) in the **Time** class. A first attempt could look like this:

**public static bool operator ==(Time tA, Time tB)**

**{**

**return (tA.Length == tB.Length);**

**}**

**public static bool operator !=(Time tA, Time tB)**

**{**

**return !(tA == tB);**

**}**

The compiler complains a bit about the risk that **tA** or **tB** could be **null**, but we’ll ig­nore that for now. If we run the examples from above, the result will now be **true** in both cases. All seems to be fine… Consider, however, the below code:

**Time tX = new Time(3, 45);**

**Time tY = new Time(3, 45);**

**Console.WriteLine($"tX == tY is {tX == tY}");**

**Console.WriteLine($"tX Equals tY is {tX.Equals(tY)}")**

First of all: where does the method **Equals** come from? It is one of those methods which is defined up in the *mother-of-all-classes*, the **Object** class. **Equals** compares two object references to each other, and will only return **true** if they refer to the same object, i.e. just as **==** worked originally. So, if we run the code above, compari­son by **==** returns **true**, while comparison by **Equals** returns **false**. So what? Is that a problem? Yes, this is per definition a problem, or at the very least consider bad prac­tice[[2]](#footnote-2). The standard approach to overloading equality operators is therefore:

1. First override **Equals** appropriately
2. As a consequence of overriding **Equals**, we must also override the method **GetHashCode** from **Object**.
3. Now override **==** and **!=** by using **Equals**.

Step 2 is also in the category of good programming practices[[3]](#footnote-3). Our task has now become a bit more complex. Let’s start with overriding **Equals**. Again, there is a standardised way of doing this, which we will follow here as well:

**public override bool Equals(object other)**

**{**

**if (other is null) return false;**

**if (other.GetType() != GetType()) return false;**

**Time tOther = (Time)other;**

**return tOther.Length == Length;**

**}**

In short; if **other** is **null** or does not refer to a **Time** object, return **false**. Otherwise, use our specific definition for equality. As mentioned, we also need to override the **GetHashCode** method. This is more or less a topic on its own; we will just go with an implementation that **ReSharper** can generate for us:

**public override int GetHashCode()**

**{**

**Unchecked { return (Hours \* 397) ^ Minutes; }**

**}**

With this in place, we can now rewrite our overload of the **==** operator:

**public static bool operator ==(Time tA, Time tB)**

**{**

**return tA?.Equals(tB) ?? (tB is null);**

**}**

We have four cases which need to be covered:

|  |  |  |
| --- | --- | --- |
| **tA** | **tB** | **returns** |
| **null** | **null** | **true** (right-hand side of **??**) |
| **null** | **not null** | **false** (right-hand side of **??**) |
| **not null** | **null** | **false** (first check in **Equals**) |
| **not null** | **not null** | what **Equals** returns |

With this final definition in place, you should have a reasonable overview of how to overload various operators in a class definition. The pressing question is now: should you override certain operators in your class definition? The best answer is the some­what vague: if it makes sense. Does it make sense to e.g. define if one **Car** object is “larger” than another? Probably not… Does it makes to be able to compare two **Car** objects for equality? It probably does… Use operator overloading in a mindful man­ner, and always consider if the operator you are about to overload really makes any sense to anybody else than yourself.

# Extension Methods

TBD

# Anonymous Types

TBD

# Reflection

TBD

# Exercises

|  |  |
| --- | --- |
| **Exercise** | OOP.4.1 |
| **Solution** | **ImprovedCatalog** |
| **Purpose** | See the index operator and enumeration interface in action. |
| **Description** | The project contains several interfaces and classes related to the concept of a **catalog**. A catalog is simply a container for a collection of values of one specific type. A catalog will typically support essential CRUD operations, and possibly other general-purpose methods and properties.  The folder **CatalogBaseClasses** contains interface definitions and imple­menta­tions of several variants of catalogs. These classes and interfaces are then used in the classes in the **Model** folder, and in the **Run** method in the **Tester** class. |
| **Steps** | 1. Study the interfaces and class definitions in the **CatalogBaseClasses** folder, and be sure you understand the role of each interface and class.(**NB**: implementation of two of the classes is not complete yet – see below). 2. Study the **DomainModel** class. Note that the type of the two properties referring to catalog objects is set to **IAll<…>**. 3. Go to the **Tester** class. In the **Run** method, we are initially only calling the test method **TestIterateOverAll**. Take a look at **TestIterateOverAll**, and make sure you understand what it tests. 4. In the **Run** method, now uncomment the section consisting of calls of **TestRead** (also take a look at the **TestRead** method itself). As it stands, this code cannot be compiled. Why not…? 5. In the **DomainModel** class, change the type of the two properties to **Catalog<…,…>**, and see what effect this has. 6. Perform steps 4 and 5 again for the next code section. In this case, you need to change the type of the properties to **IndexableCatalog<…,…>**. However, running the test will reveal that the index operator does not work properly yet. Open **IndexableCatalog** and implement it correctly. 7. Perform steps 4 and 5 again for the last code section. In this case, you need to change the type of the properties to **EnumerableCatalog<…,…>**. Running the test will reveal that the enumeration interface has not been implemented correctly yet. Open **EnumerableCatalog** and implement it correctly. 8. In all the steps above, we changed the type of the two properties in **Domain­Model** to a class type. Why did we not use the interface types instead? Try it, and see what happens... Can (or should) we “fix” this? |

|  |  |
| --- | --- |
| **Exercise** | OOP.4.2 |
| **Solution** | **VectorUtilitiesV10** |
| **Purpose** | Define and use overloaded operators. |
| **Description** | The project contains a single class **Vector** (plus the class **Tester** for running tests), which is a very basic implementation of a 2-dimensional vector (see the code). |
| **Steps** | 1. From the outset, the **Vector** class only contains the most fundamental vector functionaity. Run the application, and see how a couple of sample vectors are created and printed. 2. The first addition to the **Vector** class will be to define addition and subtraction for vectors. Addition and subtraction for vectors is simply defined as adding or subtracting the individual coordinates. Adding two vectors (2, 3) and (6, 1) will thus produce the resulting vector (2 + 6, 3 + 1) = (8, 4). Likewise, subtraction will produce the resulting vector (2 - 6, 3 - 1) = (-4, 2). Implement this functionality in the **Vector** class, by overloading the operators + and -. When implemented, you can test the functionality by uncommenting the corresponding section in the **Run** method in the **Tester** class (Expected output is: vAB = (8, 5), vCD = (-3,11), vSum = (5, 16), vDiff = (3, -1)). 3. The next addition will be to define the scalar or “dot” product of two vectors. The dot product is calculated by first multiplying each pair of coordinates, and then sum up the values. The dot product of the two vectors (2, 3) and (6, 1) will thus be 2\*6 + 3\*1 = 12 + 3 = 15. Implement this functionality in the **Vector** class, by overloading the \* operator. When implemented, you can test the functionality by uncommenting the corresponding section in the **Run** method in the **Tester** class (Expected output is: vA \* vB = 16, vC \* vD = 30). 4. Next, we want to be able to decide if one vector is “larger” than another. We de­fine the size of a vector simply as its length. Given this definition, now over­load the operators >= and <= accordingly. When implemented, you can test the func­tionality by uncommenting the corresponding section in the **Run** method in the **Tester** class (Expected output is: vA >= vB = False, vA <= vB = True, vC <= vSum = True, vA + vB + vC + vD >= vSum – vDiff = False). 5. As the final step, try to uncomment the last section of code in **Run**. It contains a small test of equality. Before running the test, consider what you would expect the result to be. Now run the test. Did it produce what you expected? What would it take to get the second case to produce True (Hint: you can also overload the == and != operators)? |

|  |  |
| --- | --- |
| **Exercise** | OOP.4.3 |
| **Solution** | **VectorUtilitiesV20** |
| **Purpose** | Define and use extension methods. |
| **Description** | The project contains a single class **Vector** (plus the class **Tester** for running tests), which from the outset contains several operator overloads, plus overrides of **Equals** and **GetHashCode**.  We now wish to be able to calculate the sum of a collection of vectors. The sum of a collection of vectors is simply found by adding up all the vectors in the given collection. In the example: vA + vB + Vc + vD = (5, 16). |
| **Steps** | 1. Write up code for calculating the sum, using a nice old-fashioned loop of your own choice. Apply it to the **vectors** collection, and store the result in the variable **vSumLoop**. 2. Write up code for calculating the sum, but this time by using LINQ. Feel free to use Select, Aggregate or whatever LINQ function(s) you prefer. Apply it to the **vectors** collection, and store the result in the variable **vSumLINQ**. 3. It would be very neat if we could solve 2) just by calling **Sum()** on **vectors** directly. That is however not possible, since the LINQ method **Sum** (which is an extension method) is not defined for the **Vector** type. Therefore, you should now write your own **Sum** extension method! Add a static class **VectorExtensions** to the pro­ject, and implement a **Vector**-compatible **Sum** method in that class. Then apply it to the vectors collection, and store the result in the variable **vSumExt**. 4. Finally, we need to address a problem in the **Vector** class. Take a closer look at the **Equals** overload. The compiler is not very happy about the way we check for coordinate equality (why not…?). We decide to fix this by defining that two **double** values are equal if they are “close enough”, i.e. that the numeric diffe­rence is less than some caller-supplied tolerance. Instead of implementing this directly in the **Vector** class, we want this functionality to be generally available for **double** values. Therefore, you should add yet another static class **Double­Extensions** to the project, and here implement an extension method as outlined. Once implemented, use it in the **Equals** overload in **Vector**. If you have time, write up a small test for the method as well. |

|  |  |
| --- | --- |
| **Exercise** | OOP.4.4 |
| **Solution** | **RpgV1** |
| **Purpose** | Implement RPG game logic in a somewhat dubious way, using index­ers, the **IEnumerable** interface and operator overloading. |
| **Description** | The project models a small aspect of a role-playing game, with specific focus on handling of continuous spell damage. In this game world, one or more Players can engage in combat with an Opponent. Part of such a combat is the ability of a Player to deal various types of “spell damage”, which is characteried as dealing a certain amount of “damage per second” (DPS) to the Opponent. Each Player can deal several types of spell damage simultaneously, and several Players can deal damage to the same Opponent simultaneously.  In order to model this logic, two classes **Player** and **Opponent** are defined in the project. Since the classes have a lot in common, an interface **IParticipant** and a base class **ParticipantBase** have also been created.  A central concept for participants is the **SpellVector**. A **SpellVector** is essentially a dictionary, with a spell type as key (available spell types are defined in the *enum* **SpellType**), and a numeric value as the corresponding value. Both Players and Opponents contain a **SpellVector**, but its content is interpreted differently (see the comments in the source code for further details).  Initially, we can thus create rudimentary **Player** and **Opponent** objects, and print out some information about these objects. We now wish to implement function­ality which can calculate the effective DPS dealt by a group of players to a single opponent. This can be broken down to two tasks:   1. Implement a way to perceive a group of players as a single “aggregated” player, which will then deal DPS to an opponent. 2. Calculate the effective DPS dealt by the aggregate player to the opponent.   This can be implemented in several ways; one way is to utilise indexers, operator overloading and the **IEnumerable** interface. **NB**: This may not be the most logical way of implementing this functionality, but we will try it out here. |
| **Steps** | 1. As usual, take a good look at the project in its initial state. There is quite a bit of information to digest. Also read the comments, since a lot of useful information can be found there. 2. First, we wish to implement indexing functionality. This should be done in the **ParticipantBase** class. The idea is that you can index a participant (be it **Player** or **Opponent**) using a spell type as key. In the imple­mentation of the indexer, the key should thus be used as a key for the **SpellType** dictionary. For a given key, the returned value should thus be the value found by lookup in the **SpellVector** dictionary. The only small caveat occurs if the dictionary does not contain an entry for the given key. In that case, the **get**-part of the indexer should return the value of the **NotPresentValue** property. 3. Once you have implemented the index­er, go to the **Run** method in **Tester**, and uncomment all the sections beneath the **Player** and **Opponent** object creation (i.e. where values are filled into each object by use of the indexer). You should also open the **PrintParticipant** method, and uncomment the first **foreach**-loop. Run the aplication, and (hopefully) see how the **Player** and **Opponent** objects are now filled with data about spell damage. 4. Next, we implement the concept of “adding” two **Player** objects, by implemen­ting the **+** operator for **Player** objects. This should be done in the **Player** class. The – slightly convoluted – logic for adding two **Player** objects is as follows:    1. The resulting **Player** object will have **Name** set to *“SumPlayer”*, and **Level** set to 1.    2. The **SpellVector** in the resulting **Player** object will – for each spell type – be set to the sum of the effective DPS for each **Player** for that spell type. That is, the DPS value for each of the two players should be multiplied with the **Modifier** value, before the values are added. **NB**: This is easiest to implement using the just-defined indexer! 5. Once you have implemented the **+** operator, open the *“Adding and printing Play­ers”* section in the **Run** method, and uncomment the code. Run the aplication, and (hopefully) see how two (or more) **Player** objects can now be added. Make sure to check that the DPS values for the “aggregated” player looks reasonable. 6. Next, we wish to implement a way of retrieving the modified values in the **Spell­Vector**, i.e. the original values multipled by the **Modifier** property. This should be done by implementing the **IEnumerable** interface for both classes, i.e. it should be implemented in **ParticipantBase**. The idea is that the enumerator will – for each spell type – return a **Tuple** object (if you don’t know what a **Tuple** is in C#, now is the time to find out…) with the spell type and the corresponding effective value. The needed steps are therefore to let the **ParticipantBase** class inherit from **IEnumerable<Tuple<SpellType, double>>** as well, and then implement the enume­rator itself in **ParticipantBase**. 7. Once you have implemented the enumerator, go to the **PrintParticipant** method, and uncomment the second **foreach**-loop. Run the aplication; it should now print out both the original and the modified values in **SpellVector** for all participants. 8. Finally, we must implement the logic for calculating the effective DPS dealt by a player (which may be an aggregate of several players) against an opponent. The logic is as follows: For each spell type, the effective DPS dealt to the **Opponent** is the product of the effective DPS dealt by the **Player**, multiplied by the effective spell vunerability of the **Opponent**. The sum of all these spell-specific DPS values is then the total effective DPS dealt. **An example**: For a specific spell type, the **Player** deals 100 DPS, and has a modifier value of 1.4. The effective DPS is thus 100 x 1.4 = 140 DPS. The **Opponent** has – for this specific spell type – a vunerabi­lity of 60 %, and a modifier value of 0.65. The effective vunerability is thus 60 % x 0.65 = 39 %. The effective DPS dealt for this spell type is thus 140 DPS x 39 % = 54,6 DPS. The total effective DPS is then the sum of all spell-specific DPS values. This definition is actually very similar to the definition of scalar or “dot” product for vectors in general. Therefore, it has been decided to implement this logic by overloading the **\*** operator in the **Player** class. However, since we are calculating a “pro­duct” of a **Player** and an **Opponent**, the two arguments to the operator definition are not both of type **Player**; one is of type **Player** and the other of type **Opponent**. Given these specifications, now implement the **\*** operator in the **Player** class. Again, the index operator defined earlier will come in handy… 9. Once you have implemented the operator, go to the **Run** method and uncom­ment the two sections named *“Calculate and print DPS…”*. They test a setup with two and five players against an opponent, respectively. Run the application, and see if your calculation works properly (expected result for the two scenarios are **81,72 DPS** (two-player scenario), and **252,53 DPS** (five-player scenario). 10. Now that you made it so far, you can reflect a bit on the implementation. Does this seem like an appropriate way to implement this logic? If not, try to come up with ideas for an alternative implementation. If you are up for the challenge, try to re-implement the logic without using any of the constructions we have used here. |

1. https://docs.microsoft.com/en-us/dotnet/csharp/tuples [↑](#footnote-ref-1)
2. https://docs.microsoft.com/en-us/visualstudio/code-quality/ca2224-override-equals-on-overloading-operator-equals?view=vs-2017 [↑](#footnote-ref-2)
3. https://docs.microsoft.com/en-us/visualstudio/code-quality/ca2218-override-gethashcode-on-overriding-equals?view=vs-2017 [↑](#footnote-ref-3)