BME AAIT

Language tools 2

Software engineering

Zoltán Benedek

2024.02.22.

Content

The var keyword

For **local** variables, instead of specifying the type, you can use the **var** keyword if the variable is initialized. Example:

```
var list = new Complex(10, 20);
```

In the example, the type of the `list` variable is `Complex`, and the compiler infers its type from the right-hand side of the "=". This technique can only be used for local variables (not for member variables, function parameters).

Generic types

Generic types have already been introduced in C++ and Java (as class and function templates in C++). The following is a brief overview of the benefits of their use and how they can be used in .NET/C#.

Generic types are used when you don't want to predefine the type of data a class/function works with. An alternative, yet somewhat flawed approach is to treat the data as `objects`. We will first look at this simple solution and then move on to the use of generic types.

In .NET, all classes/types are derived from the `object` base class (even simple types like int). Building on this, you can write generic classes of any type (e.g., `ArrayList` container), functions, etc. Let's look at a minimalist (but complete) stack storage implementation:

```
public class Stack
{
    readonly int size;
    int current = 0;
    object[] items;

    public Stack(int size) {
        this.size = size;
        // Reserving space for references
        items = new object[size];
    }

    public void Push(object object ) {
        ...
        items[current++] = item;
    }

    public object Pop() {
        ...
        return items[current--];
    }
}
Stack objectStack = new Stack();
```

```
objectStack.Push(1);
objectStack.Push("sss"); // we can build
this code ☹
int x = (int)objectStack.Pop(); // and may
later encounter hard to debug runtime errors
```

Problems with handling data as objects

Demonstrated with the `object`-based `ArrayList` collection used in the previous lesson:

1. We must use casting (extra code)

```
ArrayList list = new ArrayList();
list.Add( new Person() );
...
```

```
Person i = (Person)list[0];
```

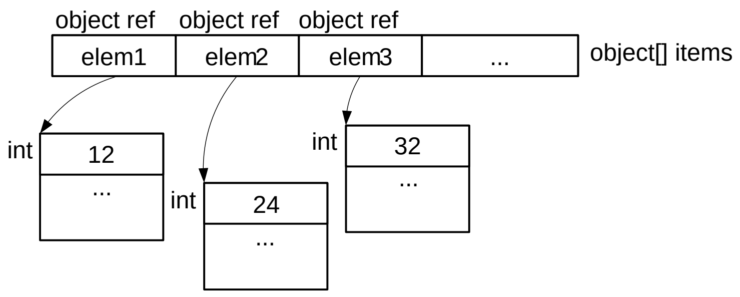
2. We will only know at runtime if there is a fault

```
ArrayList list = new ArrayList();
list.Add( new Person() );
...
Person i = (int)list[0];
```

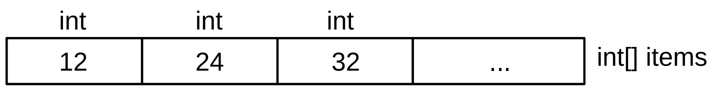
3. We can mix different types of objects, something we usually want to avoid

```
ArrayList list = new ArrayList();
list.Add( new Person() );
list.Add( 12 ); // This will build, but we rarely want to mix types in a List like this
```

4. For value types (e.g., simple `int`), boxing/unboxing causes a performance penalty. E.g., in the `Stack` example above, the elements are stored in an `object[]`, i.e., the elements of the array are references/pointers: an `int` variable can be included in this if the .NET runtime wraps (boxes) the `int` value in a heap-allocated object and makes a reference to it in the array:



But this solution is an order of magnitude slower and more space-consuming than using a simple **`int[] items`** array (as we are used to in C/C++):



The solution to the above problems (1-4) is to use generic types in .NET.

Generic class example

Let us transform the previous `object`-based `Stack` class into a generic one. In other words, define a new class type in which the type of elements it stores is not fixed, but **can be specified when the class is used (instantiated)**. The syntax is much cleaner than in C++, you don't need to use the `template` keyword when introducing generic/template parameters, you simply need to list the generic parameters separated by commas after the class name (in our case there is only one template parameter):

```
// Where there was object, we write T in the
class
public class Stack<T>
{
    readonly int size;
    int current = 0;
    T[] items;

    public Stack(int size) {
        this.size = size;
        items = new T[size];
    }
    public void Push(T item) {
        items[current++] = item;
    }
    public T Pop() {
        return items[current--];
    }
}
// During use, you must specify
```

```
// the type of generic parameters
Stack<int> intStack = new Stack<int>();
intStack.Push(1);
int i = intsStack.Pop();
// The following causes a build error, just
like we want ☺
intStack.Push("ss");
```

On the left side, the generic parameter is highlighted with **turquoise** **when declared** in the class name and highlighted in **green** **when used** inside the class. In this example, it is easy to make this distinction. However, if you see a more complex example and you delve deeper into the interpretation of the parameters, you should always have this split in mind (we **will use the colouring** in some places later).

Generic parameters can be given any name you like, in our example we could have used the more descriptive but longer name `ItemType` instead of `T`.

This is a "type-safe" solution, so the problems mentioned earlier do not occur.

Features of generic types*

C++ templates are not compiled/built by themselves. They are ‘unfolded’ during use, with separate code generated for each combination of template parameters. The generated code is then compiled. This has three unpleasant consequences:

- If you do not use a template, certain errors will not be detected during compilation.
- Risk of code bloat when used with many different parameter types.
- The source code of the template must be available at the time of compilation! Protecting the source code is not possible! ☹

In .NET, on the other hand, **generic types are translated to IL (intermediate) code during the build process**. And although for value types the compiler generates separate code for different types, **for reference types (classes) only one common code is generated**, so there is virtually no code bloat.

What can be generic?

in .NET, the following can be generic: class, function (member method), structure, interface, delegate. We've already seen examples for classes, now let's look at the others.

<u>Generic structure</u>

```
struct Point<T>
{
   public T X;
   public T Y;
}

Point<float> point;
point.X = 1.2f;
point.Y = 3.4f;
```

<u>Generic function</u>

Swap example (swapping two values):

```
class MyClass {
    public static void Swap<T>(ref T lhs, ref T rhs )
    {
            T temp; temp = lhs; lhs = rhs; rhs = temp;
    }
}

int a = 2, b = 3;
MyClass.Swap<int>(ref a, ref b);
MyClass.Swap(ref a, ref b ); // Equivalent to the previous line, no need to include the
type, it is inferred by the compiler from the context
```

Note: in the example we used the C# **`ref`** keyword: it can be used to pass a parameter by address (i.e. as a reference), similar to the way we use "&" in C/C++. In this case, no copy of the parameter is made during the call, but a pointer is passed to the original variable, and the original variable can be changed via the pointer! We built on this in the swap example above.

Generic interface

The following example shows part of the .NET built-in `IList`< T > interface :

```
interface IList<T> : ICollection< T>, IEnumerable< T>, IEnumerable {
    public void Add(T item);
    ..
}
```

Generic delegate

The following example shows the delegate type definition of the .NET built-in `Predicate`< T >:

```
delegate bool Predicate<T>(T obj);
```

.NET built-in generic types

.NET provides several built-in generic types. You only need to know by heart the ones that appear in other examples during the semester (highlighted in bold below), but it is definitely worth reviewing them as they are useful in your everyday work. Pl.:

Some generic collections:

- **List<** T **>:** Dynamically resizing array (like ArrayList in Java)
- LinkedList<T>
- HashSet<T>
- **Dictionary<K,V>:** Key-value pairs. K is the key’s type, needs a proper GetHashCode impl.
- SortedList<K,V>
- **Stack<T>**
- Queue<T>
- ReadOnlyCollection<T>

Some generic interfaces:

- **IEnumerable<** T **>:** Iterable only, forward one by one (GetEnumerator). Immutable.
- ICollection<T>: IEnumerable<T> + Add, Remove, Contains
- IList<T>: ICollection<T> + Count property, array index operator
- IReadOnlyCollection<T>: IEnumerable<T> + Count
- IReadOnlyList<T>: IReadOnlyCollection<T> + Count, array index operator
- Others: IDictionary<K,V>, IEnumerator<T>, IComparable<T>, IComparer<T>

We will look at examples of built-in generic delegates later in this document, when discussinglambda expressions.

Case study

Let's transform the universal, somewhat "ugly" `object`-based `HyperSort` example discussed in the previous lecture on delegates into a type-safe (generic) one. In the code below, changes are marked with highlighting and deleted parts are marked with strikethrough (explained in the lecture, see lecture demos under "01 Generic Types sort example").

```
delegate bool FirstIsSmallerDelegate<T>~~(object~~ T a, ~~object~~ T b);

// Here we can choose between making the class or just the method generic:
// we choose the latter.
class Sorter
{
    public static void HyperSort<T>(List< T> list, FirstIsSmallerDelegate<T> fis)
    {
        for(int i = 1; i < list.Count; i++) {
            for(int j = list.Count - 1; j >= i; j--) {
                if (fis(list[j], list[j - 1])) {
                    ~~object~~ T tmp = list[j];
                    list[j] = list[j - 1];
                    list[j - 1] = tmp;
                }
            }
        }
    }
}

class Programme
{
    static void Main(string[] args)
    {
        ~~ArrayList list = new ArrayList();~~
         List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        Sorter.HyperSort(list, FirstIsSmaller_Complex );
    }

    public static bool FirstIsSmaller_Complex~~(object~~ Complex a, ~~object~~ Complex b)
    {
        ~~Complex ca = (Complex)a;~~
        ~~Complex cb = (Complex)b;~~
        return a.Re < b.Re;
    }

}
```

Generic constraints (to be discussed at the end of the lesson, depending on time)

Let us write a generic collection class that supports the ordering of the elements it stores (`Sort` operation). In contrast to the previous delegate-based element comparison, here the elements must implement the `IComparable` interface, which has a `CompareTo` method to compare the given element with another element received as a parameter:

```
class SortableCollection<T>
{
    private T[] items;

    public void Sort()
    {
        for(int i = 1; i < items.Length; i++) {
            for(int j = items.Length - 1; j >= i; j--) {
                // If the order is not correct, swap
                if(items[i].CompareTo(items[i + 1]) > 0) {
                    T tmp = items[j];
                    items[j] = items[j - 1];
                    items[j - 1] = tmp;
                }
            }
        }
    }

    // ...
}
```

However, the above code does not build, the code highlighted in yellow is the one the compiler complains about. In our generic exercises so far, and also in this example, `T` can be of **any** type in use. That is, `T` can be `int`, `Complex`, or any custom type. However, for this type `T` in the example above, we called `CompareTo` on the highlighted line. That is, we assumed that `T` has such a method. However, there is no guarantee at the moment when the compiler compiles the above class to IL code that this is the case. In the case of C++, there was no problem because the template itself was not compiled, only after the given types were filled in! The compiler can only do one thing: in general, it will only allow the use of methods that are defined in some way for all types, e.g.: `operator=`, `ToString()` and those defined in the common base class of all types, `object`. Calling these methods should not cause an error. `CompareTo` is not included in the `object` base class, i.e. not supported by all types, so we get a compilation error.

What can be done in our example? We need to narrow down the range of types that can be applied to `T` to types that have a corresponding `CompareTo` operation. For example, if `T` is constrained to be only of a type that implements the `IComparable` interface, then it will certainly have a corresponding `CompareTo` operation. We've found our solution: **we can enforce this using .NET constraints**. Let's modify the above code by adding a constraint:

```
class SortableCollection< T> where T : IComparable
{
...
}
```

The highlighted part is new, use the **`where` keyword to specify constraints on the generic parameters**, separated by commas. The main options:

- **Interface and/or base class** constraints, as in the example, separated by commas, can be more than one.
- **new()**: the parameter must have a default constructor
- **struct**: the given parameter can only be a value type
- **class**: the given parameter can only be of reference type

You can read about other options here: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters.

Let's look at a more complex example (the example is not real, the .NET built-in `Dictionary< K , V>` class looks different):

```
class Dictionary<K,V>: IDictionary<K,V>
   where K: IComparable<K>
   where V: class, IKeyProvider<K>,IPersistable, new()
{
 ...
}
```

Let's interpret the above:

- The `Dictionary` class has two generic parameters, `K` and `V` (the key and value type).
- The class implements the generic `IDictionary` interface, parameterized with types `K` and `V`.
- `K`: can only be a type that implements `IComparable<K>`.
- `A`: can only be a reference type, must implement the `IKeyProvider<K>` and `IPersistable` interfaces, and must have a default constructor.

Lambda expression (lambda expression)

Introduction

Originally coming from the functional programming toolbox, support and use of lambda expressions has been introduced into the world of modern object-oriented languages (such as C#, Java, C++, etc.) several years ago. Nowadays, lambda terms are used in everyday work with the same frequency as interfaces or similar language elements.
In the following, we will learn about its concept and use from scratch, despite the basics being covered in the Java programming course in the previous semester. Compared to Java the basic concept is the same, but there are significant differences, such as:

- In Java it is based on interfaces (used for interfaces that have an abstract operation), in .NET it is based on delegates,
- In Java the operator is "->", in C# "=>".

They provide the possibility to define **anonymous functions**. To do this, use the lambda (declaration) operator, denoted by =>.
When you specify a lambda expression, you define an unnamed function with the parameter list to the left of the => operator and the implementation on the right. E.g.:

```
(int x, int y) => { return x+y; }
```

or in a much simpler form:

```
(x, y) => x + y
```

The syntax more precisely defined: the left side of the => operator is used to specify the parameters, and the right side is:

- **statement lambda** variant (first example above), in the case of {}, a sequence of statements, or a single statement. The syntax:
  `(input parameters) => { <sequence of instructions> }`
- **expression lambda** variant (second example above) without {} for a single expression. The syntax:
  `(input parameters) => expression`

Another question is **what is the type of a lambda expression**, i.e. what type of variable can be used to refer to a lambda expression. The answer: a **compatible delegate type**. E.g.:

```
delegate bool FirstIsSmallerDelegate(int a, int b); // Delegate type

// Store lambda in a local variable:
FirstIsSmallerDelegate fis =(int x, int y) => { return x+y; }
```

In the following chapters we will look at the details through a more comprehensive example.

Statement lambda

In this version, there can be any number of statements between the {} on the right side of =>, this is the general syntax:

```
(input parameters) => { <sequence of instructions>}
```

In practice, we rarely write more than 3-10 lines. Our starting example should be the final example from the previous Generic Types chapter in this document, the already genericized `HyperSort` function and its use (see "02a Lambda sort example" in the lecture demos). For clarity, we repeat the code here:

```
delegate bool FirstIsSmallerDelegate< T>(Ta,T b);

class Sorter
{
    public static void HyperSort< T>(List<T> list, FirstIsSmallerDelegate< T> fis)
    {
        for(int i = 1; i < list.Count; i++) {
            for(int j = list.Count - 1; j >= i; j--) {
                if (fis(list[j], list[j - 1])) {
                    T tmp = list[j];
                    list[j] = list[j - 1];
                    list[j - 1] = tmp;
                }
            }
        }
    }
}

class Programme
{
    static void Main(string[] args)
    {
        List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        Sorter.HyperSort(list, FirstIsSmaller_Complex);
    }

    public static bool FirstIsSmaller_Complex(Complex a, Complex b)
    {
        return a.Re < b.Re;
    }

}
```

So far, we have not used the term lambda. But we don't always want to define a "traditional" function with a name (`FirstIsSmaller_Complex` in our example) just to be able to pass some "code"/logic as a parameter to a function. In many cases there are more convenient ways to do this. In the following example we use a lambda expression:

```
delegate bool FirstIsSmallerDelegate< T>(Ta,T b);

class Sorter
{
    public static void HyperSort< T>(List<T> list, FirstIsSmallerDelegate< T> fis)
    {
        // Unchanged!
    }
}

class Programme
{
    ..
    static void Main(string[] args)
    {
        List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        ~~Sorter.HyperSort(list, FirstIsSmaller_Complex);~~
        Sorter.HyperSort(list, (Complex a, Complex b) => { return a.Re < b.Re;} );
    }

    ~~public static bool FirstIsSmaller_Complex(Complex a, Complex b)~~
    ~~{~~
    ~~    return a.Re < b.Re;~~
    ~~}~~
}
```

In the example, everything is unchanged and works exactly as before with the classic delegate solution, with one exception: here we did not introduce and use a separate function with a name (`FirstIsSmaller_Complex`), but defined the function without a name instead. This is in the form of a **lambda expression**. The `HyperSort` method is unchanged, since the lambda expression receives the existing `FirstIsSmallerDelegate` type and calls it in the usual way. Of course, you should also make sure that **only lambda expressions with a parameter list and return type compatible with the delegate type are allowed**!

Another simplification is that **the compiler can infer the type of the parameters in the vast majority of cases**, so we don't need to, and don't usually, specify them:

```
Sorter.HyperSort(list, (a, b) => { return a.Re < b.Re;} );
```

To sum up: the advantage of using a lambda expression is that **we didn't have to write a separate classical/named function for the sake of the logic/code referenced by delegate**, but we defined it locally using a lambda expression.

Expression lambda

If our lambda function **consists of a single statement**, we can use **expression lambda**, which allows us to use a more concise form of lambda than statement lambda above.

```
Sorter.HyperSort(list, (a, b) => a.Re < b.Re);
```

The syntax is: `(input parameters) => expression`
The right-hand side of A => without { }'s consists of a single expression, returning its value. "return" is not needed or even used, and ";" is not needed at the end, since it is an expression.

You can see **how simple and concise it is to** provide **"logic"/"code" to another function**, in this case `HyperSort`.

Other options

In the example above, lambda is passed to a function parameter. But of course, you can pass the value of a lambda expression to any (compatible) delegate variable, e.g., a member variable or a local variable. Example of a local variable:

```
static void Main(string[] args)
{
    FirstIsSmallerDelegate<Complex> fis = (a, b) => a.Re < b.Re;
    bool b = fis(new Complex(10, 20), new Complex(1, 2));
}
```

Built-in delegate types - Func generic delegate (important)

In the example above, we used the following delegate type that we defined:

```
delegate bool FirstIsSmallerDelegate(object a, object b);
```

In fact, it should not have been introduced at all. Because .NET has a number of **`Func`** built-in generic delegates. E.g.:

- **`Func<TResult>`** is a delegate type that has no parameters and returns `TResult`,
- **`Func<T1, TResult>`** is a delegate type that has a parameter of type `T1` and returns `TResult`
- **`Func<T1, T2, TResult>`** is a delegate type that has one parameter of type `T1` and one parameter of type `T2` and returns `TResult`,
- etc. for different parameter numbers.

Accordingly, instead of using the `FirstIsSmallerDelegate<T >` type above, we can use the `Func<T, T, bool>` type, which is more appropriate and removes the need to introduce the `FirstIsSmallerDelegate` type. Let us change the `HyperSort` function parameter accordingly (everything else unchanged!):

```
// Delete, no need
~~delegate bool FirstIsSmallerDelegate<T>(T a, T b);~~

class Sorter
{
    public static void HyperSort<T>(List< T> list, Func<T, T, bool> fis)
    {
        // Unchanged!
    }
}

// Unchanged!
class Programme
{
    ..
    static void Main(string[] args)
    {
        List<Complex> list = new List<Complex>();

        list.Add(new Complex(3, 2));
        list.Add(new Complex(1, 2));

        Sorter.HyperSort(list,(Complex a, Complex b) => { return a.Re < b.Re;} );
    }
}
```

Interpreting the `HyperSort` function header may be difficult at first (especially the two T parameters in the `Func<T, T, bool> >` parameter type). If we wanted to use `HyperSort` only for complex numbers, it would look like this:

```
    public static void HyperSort(List<Complex> list, Func<Complex, Complex, bool> fis)
```

It would expect a delegate with two `Complex` parameters (the two complex numbers to compare) and return `a boolean`. But we want a generic `HyperSort` that can be used with any type: so, as we learned in the Generic Types chapter, we introduce a parameter `T` (as usual in turquoise) and use it in our generic function (as usual in green) to get the solution:

```
    public static void HyperSort<T>(List< T> list,Func<T, T, bool> fis)
```

Now let's look at another example. Since `Func` is also a fully-fledged type/delegate, it can be a parameter/member variable/local variable. We have just seen an example of it acting as a parameter, now let's look at an example of a local variable:

```
Func<Complex, Complex, bool> fis2 = (a, b) => a.Re < b.Re;
Sorter.HyperSort(list, fis2);
```

Built-in delegate types - Action and generic Action delegate (important)

The `Func` delegate can be used when there is a return value. If there isn't (return type `void`), the different variants of the built-in **`Action`** delegate type can be used: E.g.:

- **`Action`** is a delegate type with no parameters (and returns `void`),
- **`Action<T1>`** is a delegate type that has one parameter of type `T1` (and returns void),
- **`Action<T1, T2>`** is a delegate type that has one parameter of type `T1` and one parameter of type `T2` (and returns `void`),
- etc. for different parameter numbers.

Example 1:

```
Action<string> greet = name =>
    {
        string greeting = $"Hello {name}!";
        Console.WriteLine(greeting);
    };
greet("World"); // Prints "Hello World"
```

Example 2:

```
Action<string, string> labelAndTextWriter =
    (label, text) => Console.WriteLine($"{label}:\t{text}");
labelAndTextWriter("Name", "Ennio Morricone");
labelAndTextWriter("Year of birth", "1928");
```

Syntax for specifying parameters in lambda expressions

If no parameter, empty brackets:

```
Action line = () => Console.WriteLine();
```

If there is one parameter, the bracket is optional (not usually printed):

```
Func<double, double> cube = x => x * x * x;
```

If there are multiple parameters, they are listed comma-separated between ( ), see examples above, respectively:

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

Remember: if the compiler can't infer the type of the parameters, then specify the types (in the previous example, it could):

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Literature

- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions

LINQ - Language-Integrated Query

LINQ is one of the most popular features of .NET/C# for developers. In its simplest form, it simplifies working with collections (e.g. filtering, sorting, projection, grouping, min/max/sum, etc.), but can also be used for XML processing and formulating database queries. We will only look at collections management with a few basic operations to get a taste, with more extensive use covered in later courses. We can work with two syntaxes: we only look at the so-called **fluent** syntax (the so-called query expression syntax is not covered in this course).

Let's look at a very simple example:

```
List<string> fruits = new List<string>() { "apple", "grape", "peach", "banana",
"pineapple" };

var fuits2 = fruits.Where(f => f.Length <= 5);

foreach(var f in fruits2)
    Console.WriteLine(f);
```

In the example, the `Where` operation defined on the collection is used to filter the elements. `Where` returns a collection that satisfies the condition specified in the lambda expression (in our example, we filter for strings no longer than 5). The behind-the-scenes implementation of `Where` is very simple: it calls a delegate/lambda for each element, passing the element as a parameter (`f` in the example), and takes the elements for which lambda returns `true` as output (`Where` is actually a generic function, `Func<T, bool>` expects a delegate as a parameter, in our example `Func<string, bool>`).

Let's look at a more complex example:

```
var fuits2 = fruits
    .Where(f => f.Length <= 5)  //Filtering
    .OrderBy(f => f.Length)     // Ordering
    .Select(f => $"Reverse: {f.Reverse()}, Length: {f.Length}"); // Projection
```

We could have written the whole thing in one line, but this is much more readable. In the example, the `fruit` collection was filtered with `Where`, then its output was sorted by length with `OrderBy`, and its output was projected /transformed with `Select`. For each element, `Select` creates a new object of any type specified by the lambda expression (in our example, a new string), and its output collection contains these objects.

The example illustrates that LINQ operations can be concatenated ("."). You can also see that it all "rhymes" a bit with SQL, only here we are working on objects rather than tables.

The `Where/OrderBy/Select/etc`. methods can be used on any .NET collection because they are defined on the `IEnumerable<T>` type, and in .NET, this interface is implemented by all collections.

These operations are returned with `IEnumerable<T>` (`T` is the element type): in essence, you get an iterator that produces the output stack during iteration (when you iterate through the output with a `foreach` operation). If we need to save the items in a new collection object, we can call `ToList()` or `ToArray()`. For example:

```
var fuits2 = fruits.Where(f => f.Length <= 5).ToList(); // fruits2 will be a List<string>
```

Other C# language tools

Partial classes - important!

A class can be defined in several separate .cs files. The compiler combines the parts into a class. A **`partial`** keyword must be used, without it you will get a build error. Example:

```
// Customer_a.cs
partial class Customer
{
    private int id;
    private string name;
    private List<Order> orders;

    public void SubmitOrder(Order order)
    {
        orders.Add(order);
    }
}
```

```
// Customer_b.cs
partial class Customer
{
    public bool HasOutstandingOrders()
    {
        return orders.Count > 0;
    }
}
```

Compile

```
// The translation process produces this
partial class Customer
{
    private int id;
    private string name;
    private List<Order> orders;

    public void SubmitOrder(Order order)
    {
        orders.Add(order);
    }

    public bool HasOutstandingOrders()
    {
        return orders.Count > 0;
    }
}
```

In the above example, the parts of the `Customer` class are contained in two files: the compiler combines the parts, resulting in a single `Customer` class with all its parts included.

One of the main applications of this feature is the separation of generated and handwritten code.

Expression-bodied members

Sometimes we write short functions or, in the case of properties, explicitly often short `get/set/init` (initializers are not discussed in this subject) definitions that consist of a **single expression**. In this case, the `get/set/init` trunk of a function or property can be specified using **expression-bodied members** syntax, using =>. This can be done whether or not there is a return value (return statement).

In the examples, we will see that the use of expression tags is nothing more than minor syntactic "sugar" to minimize the need to write boilerplate code.

Let's first look at a function example (assume that the class has a `yearOfBirth` member variable):

```
public int Age() => DateTime.Now.Year - yearOfBirth;
public void DisplayName() => Console.WriteLine(ToString());
```

As you can see, we have removed the {} brackets and the `return` statement, so the syntax is more concise.

Important: Although the => token is used here, it has nothing to do with the lambda expressions discussed earlier: it is simply that the same => token (symbol pair) is used by C# for two completely different purposes.

Example of specifying a property getter:

```
public int Age { get => DateTime.Now.Year - yearOfBirth; }
```

In fact, if you only have a getter for the property, you can omit the get keyword and the brackets.

```
public int Age => DateTime.Now.Year - yearOfBirth;
```

What distinguishes this from the similar syntax of the functions we have seen before is that we have not written the round brackets.

```
public int Age() => DateTime.Now.Year - yearOfBirth;
```

Object initializer

Initialization of public properties/member variables and constructor invocation can be combined using a syntax called object initializer. This is done by opening a block with brackets after the constructor call, where the value of the public properties/member variables can be specified, using the following syntax.

```
var p = new Person()
{
    Age = 17,
    Name = "Luke",
};
```

Properties/members are initialized after the constructor is run (if the class has a constructor). This syntax is also advantageous because it counts as one expression (as opposed to three, if you were to create an uninitialized `Person` object and then give values to the `Age` and `Name` members in two additional steps). This way, you can pass an initialized object directly as a function call parameter, without having to declare a separate variable.

```
void Foo(Person p)
{
    // do something with p
}

Foo(new Person() { Age = 17, Name = "Luke" });
```

As you can see in the examples above, it doesn't matter whether there is a comma after the last property or not.

For the time being, the benefit of using object initializers is less apparent. However, it is often unavoidable for certain language constructs, such as LINQ. These cases are not studied in the course but will be covered later in other subjects, so it is important to be familiar with object initializers.