

C-Sharp current and future trends

Erik Borgenvik and Martin Saegebrecht

24 may 2006

Contents

0.1	Introduction	1
0.2	The birth of C#	2
0.3	.NET Framework	3
0.3.1	Typecategories	4
0.3.2	Class library	4
0.3.3	MSIL and JIT	5
0.3.4	Garbage collection	5
0.4	The first version of C#	6
0.4.1	Development	6
0.4.2	Pointers and Unsafe code	6
0.4.3	Delegates	7
0.4.4	Reflection	8
0.5	The second version of C#	9
0.5.1	Generics	9
0.5.2	Partial Types	10
0.5.3	Nullable Types	10
0.5.4	Anonymous Methods	11
0.5.5	Iterator	13
0.6	Design patterns in .NET	14
0.6.1	Observer pattern	14
0.6.2	Iterator pattern	16
0.7	C# future	17
0.7.1	Extension methods	17
0.7.2	Lambda Expressions	18
0.7.3	Implicit variables	19
0.7.4	Object initializers	20
0.7.5	Anonymous Types	20

Abstract

What is a perfect programming language? There isn't a simple answer to that question, because all languages has there own different qualities. Some people say a perfect language is a language which is flexible and easy to understand. A language with a rich class library and with a high abstract level makes it easier for the programmer to create flexible programs. *C# (C-Sharp)* is such a language. It has a rich class library and takes care of a lot of things a programmer normally has to handle himself like memory deallocation etc. Are there any drawbacks then? Higher abstraction conceals a lot of underlying syntax from the programmer. Of course this makes it easier to create datatypes but there will also follow some complexity because the programmer will be more dependent of the built in types and loses a bit freedom to solve problems in his own way. Some parts of *C#* is really good and saves the programmer a lot of time while other parts just seem to increase the complexity and actually makes it harder to write code that is understandable. The report will cover features like *delegates*, *anonymous types*, *nullable types* and *lambda expressions* among with other interesting topics. To make you come up to your own opinion of *C#*, the report contains the most significant parts of the language.

0.1 Introduction

This report will give you deeper knowledge about the object oriented programming language C#. It covers the most significant programming parts of the language and gives a deeper knowledge about the underlying framework. We assume that you are familiar with the basic principles of object oriented programming and understand classes, abstract data types, structures etc. At the end you can also read about the future of C# with the new upcoming version 3.0.

0.2 The birth of C#

In the end of the 90's Microsoft started a big project meant to create a whole new infrastructure for software development and distributed systems. At that time Microsoft's development tools were all working and executing independently from each other. The task was to develop one new model for objected oriented programming and another model for the execution of component based software. The project was called .NET (dotnet) which you can read more about in the following chapter. Another important milestone in the .NET project was to separate the type system (how variables are being created etc.) from the language so that all languages that support the type system could fit into .NET.

1996 Anders Hejlsberg was recruited by Microsoft. As a chief Architect he and his group created C# as a part of the .NET project which were presented in the year 2000 and became an ISO standard in 2003. This language was created to support the same type system that .NET is using. C# is a modern objected oriented language which includes the most popular mechanisms from Java and C++ and has also got some influences from Visual Basic.

C# is simple to use, in comparison to C++, multiple inheritance from classes has been removed. Constants, variables and arrays must always belong to a class or a struct. Almost everything in C# are either classes and objects, even the classic primitive types like int, float, etc can be seen as objects.

0.3 .NET Framework

The .NET Framework is a develop and execution environment that allows different programming languages and libraries to work together seamlessly to create Window-based applications which are easier to build, manage, deploy and integrate with other networked systems.

Dotnet is based on the international standard called *CLI (Common Language Infrastructure)* which is central for libraries and languages to be able work together. Microsoft's implementation of CLI is called *CLR (Common Language Runtime)* which provides that service.

The CLI standard which languages like C# are based on contains the following principles

- **Common Type System (CTS):** Defines how types are declared, used and managed in the runtime environment. It also defines rules that languages must follow which ensure that objects written in different languages can interact with each other. According to CTS there are only three kinds of datatypes: classes, structs and interfaces.
- **Common Language Specification (CLS):** To fully interact with other objects regardless of the language they were implemented in there must be a set of simple types that all languages can understand and which they can include in there interfaces.
- **Common Intermediate Language (CIL):** A format for metainformation which after it has been JIT (Just In Time) - compiled becomes code for the CPU.
- **Virtual Execution System (VES):** Defines the environment of the execution, how classes should be loaded and automatic garbage collection.

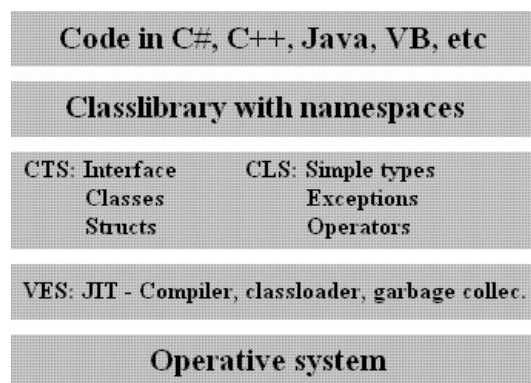


Figure 1: CLI model.

0.3.1 Typecategories

When programming with CLI it is possible to use any language that supports the CTS/CLS. According to CLS the language are separated into two categories called consumers and extenders. Consumers only uses functions stored in the framework, but they don't create any new datatypes and don't have to support the object oriented mechanisms. Extenders are able to create datatypes according to the CLS and possible also according to the CTS. C# is an extender and supports both CLS and everything in CTS.

If the framework are suppose to be independent from the choice of language there must be a set of simple types that all languages have to understand and not use any other types in there public interfaces. Therefore CLS types are as follows:

CLS

- **bool**: can only be true or false
- **char**: 16 - bits character
- **unsigned int8**: 8 bits non - negative integer
- **int16**: 16 - bits integer
- **int32**: 32 - bits integer
- **int64**: 64 bits integer
- **float32**: 32 bits floating number with precision of 7 numbers
- **float64**: 64 bits floating number with a precision of 15 numbers

One important thing to note here is that the CLS - types are *simple types*, not *primitive types*. Primitive types are variables not objects and they can only contain a value. Objects have both values and behaviors.

As we mentioned, in CTS there are only three kinds of datatypes where class is the most central (the other two are structs and interfaces). The big difference between a class and a struct is that a struct can be created on the stack which is not the case for class-objects. Interfaces are classes without implementation, only named methods.

0.3.2 Class library

The entire class library is a big deriving hierarchy where the base class is called Object. All datatypes inherits from Object. It is the root of the type hierarchy and supports all classes in the .NET Framework.

The library is also divided into different namespaces where the root is called System. Namespaces are thought to give datatypes a division which corresponds to applicationareas. For example most things that has to do with threads and synchronization is placed in the System.Threading namespace.

Other example of namespaces is:

- **System.Timers:** allows you to trigger an event on a specific time interval.
- **System.Security:** Contains base classes for permission.
- **System.Drawing:** provides graphic functionality.

0.3.3 MSIL and JIT

When compiling code .NET translates the source code to a CPU - independent language called Microsoft Intermediate Language (MSIL) which can be converted to native code. MSIL includes instructions for loading, storing, initializing, control flow among other things. Before code can be run, MSIL must be converted to CPU - code usually by a Just - In - Time - compiler. This type of compiler has the benefit of compiling code whenever it is needed instead of compiling the whole program at once. .NET supplies one or more JIT- compilers for each computer-architecture it supports.

To describe the benefits that MSIL is providing we first have to explain the term *metadata*. Metadata is binary information that describes your program that is stored either in a common language runtime portable executable (PE) file or in memory. When you compile your code into a PE file, metadata is inserted into one portion of the file, while your code is converted to *MSIL (Microsoft intermediate language)* and inserted into another portion of the file. Every type and member defined and referenced in a module or assembly is described within metadata. When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on. Metadata allows .NET languages to describe themselves automatically in a language-neutral manner.

0.3.4 Garbage collection

One interesting quality that VES is providing is the automatic garbage collection. Objects that don't have a reference will be erased automatically. Without garbage collection the programmer has to the job him self for example by calling the function delete in C++.

Example:

```
MyClass obj = new MyClass();  
Obj = null;
```

When the reference is disappearing or being put to another value the object gets one less reference. When all references are gone the object is ready for garbage collection. To detect objects with no references is usually very hard because objects can be put together as a chain and it demands complicated algorithms to manage that with graph analyse. VES is supposed to handle those situations.

0.4 The first version of C#

0.4.1 Development

Developed as a part of Microsoft's .NET package C# was set to be an object oriented simplification of C++, although when released it also contained many features from Delphi, Visual Basic and Java. This was much due to chief designer Anders Hejlsberg who previously had been working on the development of Delphi and Visual J++.

An ongoing theme during the initial and continuous development of the language was to add simplicity over simplicity, Hejlsberg describes this in an interview as follows:

"No one ever argues that simplicity isn't good, but people define simplicity in a variety of ways. There's one kind of simplicity that I like to call simplicity. When you take something incredibly complex and try to wrap it in something simpler, you often just shroud the complexity. You don't actually design a truly simple system. In some ways you make it even more complex, because now the user has to understand what was omitted that they might sometimes need. That's simplicity. So to me, simplicity has to be true, in the sense that the further down you go the simpler it gets."

This thinking opened up new opportunities for C# whereas it now could also be used for *RAD (Rapid Application Deployment)* where usability and simplicity are highly valued. Many important design choices had to be made during the development of C#, one was not to use multiple-inheritance, the reason for this was to avoid complications with CLI's structure and at the same time make the user unable to struggle with multiple dependant classes when coding. Even though C# 1.0 lacks the use of multiple inheritance, a class can still implement an endless number of interfaces to it. One thing that was overlooked in the early version of C# was polymorphism, no support for templates as in C++ was made and generics was first to come in a later version.

0.4.2 Pointers and Unsafe code

Since C# is much like Java and use safe references you might think it doesn't support the use of pointers but that's in fact not all true. Even though C# was made to be safe and simple, pointers can still be used although in a somewhat limited fashion. Code that uses a user-defined pointer will be marked as unsafe and only the proper application with the right permission will be able to execute the unsafe code. When operating in unsafe code it is much like writing C code but in a C# environment. A common misconception is that unsafe code is bad and shouldn't be used but don't let the name unsafe fool you. Ironically unsafe code is very safe compared to a C or C++ program where basically the whole program would be marked as unsafe.

In C-Sharp you can strictly control the access by using the keyword unsafe and the execution engine also makes sure the code isn't executed in an untrusted environment. However unsafe code shouldn't be used unless it is really called

for. As mentioned earlier C# doesn't directly allow manipulation of pointers and direct memory management but this still needs to be done. When you read about VES in the previous chapter you were informed about the automatic garbage collection. When needed the garbage collection frees up memory and thus managed memory doesn't have to be freed explicitly by the user.

0.4.3 Delegates

Delegation is one of the more interesting features C# 1.0 has to offer. Using a delegate enables the user to pass-on the execution to another object, in C++ this could be solved by the use of function pointers. Delegates however has some distinct advantages over function pointers, they not only offer the full use of object orientation, but also encapsulates an object instance and method. To better understand delegates and how they work we will look at a first example where a delegate is made to fit the correct methods.

```
delegate int D(int i, double d);

class A
{
    public static int M1(int a, double b) {...}
}
class B
{
    public static int M1(int f, double g) {...}
    public static void M2(int k, double l) {...}
    public static int M3(int g) {...}
}
```

In order for a delegate to be instantiated and work properly it has to have the correct parameters and return value. The delegate D used here has a return type of int and takes two parameters of type int and double. This delegate can now be used on either M1 in class A or M1 in class B since both those methods fulfill the type requirements. Methods M2 and M3 in class B however have either a wrong return type or parameter type and therefore can't be used when the delegate is instantiated.

A delegate can consist of an instance method or static method but a delegate can also consist of another delegate. This allows the user to create a chain of delegates which can then be executed in order. The '+' or '+=' operator is used to combine two or more delegate into a sequence which will then be executed when the original delegate is called. Here's an example of how it works:

```

delegate void D(int x);

class A
{
    public static void M1(int i) {...}
    public void M2(int i) {...}
}
class B
{
    static void Main() {
        D cd1 = new D(A.M1);
        A a = new A();
        D cd2 = new D(a.M2);
        D cd3 = cd1 + cd2;
        cd3 += cd1;
    }
}

```

Here we create 3 new delegates `cd1`, `cd2` and `cd3`. The first delegate `cd1` is created from the static method `M1`, thereafter `cd2` is created through instance method `M2` and finally we create a delegate `cd3` from the two delegates we just made. Last we add the `cd1` delegate to `cd3` once again using the `'+='` operator. Note that the type rules still apply when using a set of delegates and thus methods with incorrect parameters or return type can't be used. Just as you can add delegates you can also remove them using the `'-='` operator. In the case where more then one of the same delegate is existent in a set just like `cd1` in the previous example, then the last added delegate of `cd1` will be removed. If you were to use a return value in the methods of a set of delegates the output value would be of the last executing delegate.

0.4.4 Reflection

Another interesting feature C-Sharp 1.0 offers is the use of reflection. For people new to this, reflection or *Computational reflection* enables a program to observe and modify its behavior or structure at runtime. This can prove to be extremely useful in some situations such as file loading at runtime. This is the same principal you read about in the dotnet chapter about MSIL.

0.5 The second version of C#

0.5.1 Generics

Parameterized types, also known as generics is essentially the biggest change from C# 1.0 In short, generics is the ability to have different parameters on a type, much like C++ templates but without any complications. You might say, yes but this can already be made using the object type. This is true but using object does come with some drawbacks. Consider the following example of an object implemented stack.

```
Stack stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();
```

The Customer object in this case is pushed onto the stack and that is fine since any object can be used. Although later when it's time to pop you explicitly have to cast it to a Customer since the stack itself doesn't know what specific object was pushed on in the first place. In the end it will lower performance because of the type checking made at runtime.

The same applies when the input is a value of a value type, such as a number. The the number will automatically be boxed and later has to be unboxed using cast. This will lead to dynamic memory allocations being used as well as type checking at run-time.

Let's now look at how generics work and the advantages it comes with. We start off by looking at an example of a Stack, implemented using generics.

```
class Stack<T>
{
    T[] items;
    int count;
    public void Push(T item) {...}
    public T Pop() {...}
}
```

> In this example we use a type parameter called T witch act as a placeholder until a type is specified. Unlike using the object type we now aren't forcing any type conversions, instead we accept the data type for what it is and store it. When we instanciate the stack as in the example below, T will simply be replaced with the int type.

```
Stack<int>stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

We now not only eliminate the problem with having to type cast back and forth, we also get a much greater storage efficiency as we now use an int array

instead of an object array. Generics will also promote strong typing in the sense that trying to push an int onto a stack with Customers objects will result in a compile time error.

Also there is no limitation to how many type parameters that can be used. We only used T in the example above but some situations might call for the use of two or more parameters.

0.5.2 Partial Types

Many people agree on the fact that splitting up related code into two different sections or even place related code in two different files is an outrage and ultimately just a step in the wrong direction. Partial Types however was created and added into C# 2.0 for just this reason. It was mostly because of the extensive use of source code generators among programmers and designers. Partial types allow the user to separate the generated and user written code and thus simplifying and speeding up development. The designers are now free to generate new code without overriding the code added by programmers. Partial types can also be used to restructure very big classes into smaller sections for a better overview and possibly easier maintenance. The code below shows a simple example of what a partial type could look like

```
partial class Door
{
    private bool isOpen;
    public Door() {...};
}
partial class Door
{
    public void Open(int Key); {...}
    public void Lock(int key) {...};
}
```

When compiled these partial classes will be merged in to one class.

0.5.3 Nullable Types

Using and handling none-value types and null values in a language has always been a bit troublesome. Different solutions exist but they all have flaws. For example you can use predefined null types but this would only work for existing types. Another solution is to use a certain value to represent null such as -1, but this only works as long as you don't use that special value. C-Sharps nullable types however solve this problem by using a ? operator. Here's an example of how the ? type modifier works.

```
int? x = null;
int? y = 10;
if(x.HasValue == true) return x;
if(y.HasValue == true) return y;
```

First of all two nullable types `x` and `y` are declared using the `?` modifier. A nullable type consist of a boolean type plus the underlying value type, 'int' in this example. The boolean value of `x` is then called by using the `HasValue` command which returns `true` if it's a none-null type and `false` if it's a null type. Since `x` is set to `null` `HasValue` will return `false` and thus `x` won't be returned. The opposite happens for `y`, `HasValue` will return `true` since `y` has a value of 10 and thus `y` will be returned.

Nullable types can be implicitly converted from nullable type to a none-null type and the other way around. They can also be used with other operators just as normal types. The following examples will show how nullable, non-nullable types and operators work together.

```
int? x = null;
int? y = null;
if(x == y)
    int? z = x + 1;
```

In this second example the nullable type `x` has a value of 0 and `y` is still null. Then we try to compare the types using `>` and will end up getting a false expression. The same would happen if we were to use two null values. On the other hand `z` will now consist of `x + 1` and not `null` since `x` now has a value of 0. Appart from the `?` type modifier a new null coalescing operator `'??'` exist. The expression `x ?? y` should be interpreted as follows:

```
if(x != null)
    return x;
else
    return y;
```

This operator can be specially convinient when dealing with nullable and non-nullable types as this final example will show:

```
int? x = GetNullableInt();
int? y = GetNullableInt();
int? z = x ?? y;
int i = z ?? -1;
```

You can see the result of `x ?? y` is a nullable value `z`, but the result of `z ?? -1` is a non-nullable value. In other words the `??` operator in the latter case will remove the `?` from the type and at the same time supplies a default value to use in case `z` is `null`.

0.5.4 Anonymous Methods

An anonymous method is anonymous in the sence that it lacks a name, so the question arises how do you run such a method if you can't call it by it's

name. The answer is though a delegate. An anonymous method might use a delegate to execute but seem to act far from it. Instead of delegating the work elsewhere the code gets executed in line with the rest of the code or so it might seem. In truth C# converts the code within the anonymous method into a unique method within a unique class. The delegate is then set to reference the new uniquely named method. In order for this to work you need to fulfill the requirements of delegates of course. Here's an example of a click event done using delegates but without an anonymous method:

```
class A
{
    ListBox listBox;
    TextBox textBox;
    Button button;

public A()
{
    listBox = new ListBox(...);
    textBox = new TextBox(...);
    button = new Button(...);
    button.Click += new EventHandler(AddClick);
}

    void addClick(System.Object o, System.EventArgs e)
    {
        listBox.Items.Add(textBox.Text);
    }
}
```

Click in this case is a delegate which we reference to the method `addClick`, where we then add the current text in a `textbox` to a `listbox`. Now to do this without using the `addClick` method we can use an anonymous method. The example below will show how this is done.

```
class A
{
    ListBox listBox;
    TextBox textBox;
    Button button;

public A()
{
    listBox = new ListBox(...);
    textBox = new TextBox(...);
    button = new Button(...);
    button.Click += delegate(sender, e)
    {
        listBox.Items.Add(textBox.Text);
    };
}
```

What we do here is simply add brackets after the keyword `delegate` and then input the code we want to execute. As you can see we can still use and manipulate the variables just like if it was a normal method. Note that the parameters `sender` and `e` of type `Object` and `EventArgs` has to be passed even with an anonymous method this is because the click delegate is constructed of those two parameters and thus according to C# delegation rules we have to pass them along when we use the click delegate.

0.5.5 Iterator

A new form of iterator was introduced along with C# 2.0. The new iterator supports the `foreach` statement and also uses a new `yield` keyword. For the iterator to work properly you will need to implement a `GetEnumerator` method from `System.Collections.IEnumerator`. This is also where the new keyword `yield` comes to use. Let's look at an example before we start and discussion how the `yield` statement is used.

```
publicclass List : System.Collections.IEnumerable
{
    Int items[] = new int[5] { 1, 2, 3, 4, 5 };

    publicSystem.Collections.IEnumerator GetEnumerator()
    {
        for (int i = 0; i < items.Length; i++)
        {
            yield return items[i];
        }
    }
}

class MainClass
{
    static void Main()
    {
        List list = new List();
        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
    }
}
```

Here we first create a list class which contains five items, but the interesting part is of course the `GetEnumerator` method where we basically decide how to iterate our items. By implementing this method we can now decide how the

items should be iterated using `yield return` and `yield break`. In this example we simply choose to go through all the items in the list and return them one after another using `yield return`. We then use the `foreach` statement in the main class to output all the data on the screen. By implementing the `GetEnumerator` method accordingly, using the keyword `yield` and including the `IEnumerable` interface we get a fully `Enumerable` type. This means that we don't have to implement the `MoveNext`, `Current` or `Reset` methods since `C#` already has provided us with them and on top of that we now have the `foreach` statement at our disposal. The `yield` statement can be used more than once in the iterator as this second example of `GetEnumerator` will show.

```
public System.Collections.IEnumerator GetEnumerator()  
{  
    yield return "Hello";  
    yield return "there";  
    yield return "!";  
}
```

Using `foreach` or calling `MoveNext` manually will result in the first item in the iterator body, "Hello" in this case. The next iteration will resume after the previous `yield` statement and then go on until it reaches a `yield break` statement or the end of the body is encountered. Something to remember when using the new iterator is that calling the `Reset` method will result in `Current` returning a none-defined value. To avoid an exception being raised we therefore have to call `MoveNext` before the first value in a list can be used.

0.6 Design patterns in .NET

In this chapter we will talk about which design patterns are used in the *BCL* (*Base Class Library*) of .NET. Within the .NET framework the use of design patterns are so prevalent that it has become a built in part of the programming languages themselves instead of just being represented by classes.

To describe this more briefly we will give you two examples of how to use the observer- and iterator pattern which is supported by `C#` and other languages.

0.6.1 Observer pattern

A common scenario when classes have to interact with each other occurs when a class (**observer**) needs to know if something has changed in another class (**subject**). One way to fix this problem is to let the observers implement an interface with a method called `notify`.

```

public interface observer
{
    Void notify();
}

```

All the subjects need a list of interested observers to call notify on when its state changes. Therefore the subject must derive from a `subjectbaseclass` to handle that task.

```

public abstract class subjectbaseclass
{
    private ArrayList observers = new ArrayList();

    public void add(observer o)
    {
        observers.add(o);
    }

    public void remove(observer o)
    {
        observers.remove(o);
    }

    public void notify()
    {
        foreach(observer o in observers)
        {
            o.notify();
        }
    }
}

```

This solution works but it is quite extensive to let all subjects in the system inherit from the `subjectbaseclass` and all observers implement the `observer` interface. Therefore a much simpler solution is provided in *C#* using delegates and events. In Windows, events arise when a button is pressed. If an `observer` needs to be notified about that event it is possible to couple one of its methods with a delegate. Principally this is the same as pointers to functions but on a higher abstract level.

```

public delegate void EventHandler();

public class Subject
{
    public EventHandler Event;

    public void raiseEvent()
    {
        EventHandler ev = Event;
        if (ev != null)
            ev();
    }
}

public class observer1
{
    public observer1(Subject s)
    {
        s.Event += new EventHandler(handleEvent);
    }

    public void handleEvent()
    {
        Console.WriteLine(" An event has occurred!");
    }
}

```

With this solution, whenever an event occurs the observer will be aware of it through the method applied to the delegate (`handleEvent`).

0.6.2 Iterator pattern

When working with collections of objects, you often need to traverse and visit each element in different orders. Therefore C# uses the interface `IEnumerable` and the class `IEnumerator`.

```

int[] values = new int[] { 1,9,4,3,2};
IEnumerator e = ((IEnumerable)values).GetEnumerator();

while(e.MoveNext())
{
    Console.Write(e.Current.ToString() + " ");
}

```

This iterator pattern lets you easily traverse a collection without expose the inner workings of the collection. It uses the interface `IEnumerable` which has one

single method `GetEnumerator()` that returns an object which implements `IEnumerator`. The `IEnumerator` class contains the code necessary to iterate through the collection. All of the collection classes in the `System.Collections` namespace, as well as arrays implement `IEnumerable` and can therefore be iterated all over. Other patterns used widely in .NET is Adapter-, Factory-, and Strategy pattern.

0.7 C# future

In October 2005 Microsoft proposed some new features that were meant to extend the current 2.0 version. The 3.0 version, also called *C# Orcas* is now under production and is suppose to contain those features. When reviewing the new promised features it seems as C# is taking a step towards functional style programming with the support for higher order functional style class libraries. The extended features are:

- **Extension methods**
- **Lambda expressions**
- **Implicitly typed local variables**
- **Object initializers**
- **Extended anonymous types**
- **Query expressions**
- **Expression trees**

This section will cover the first five.

0.7.1 Extension methods

You can say that extension methods are extended static methods. A static method is something you can call only from a class and not from an object.

Example:

```
Class X
{
    static int val
    static void getVal() {return val;}
}
```

The method `getVal` here is only possible to invoke like this:

```
int val2 = X.getVal();
```

Extension methods have the ability to be invoked from objects but they have to use the keyword `this`. Extension methods appear like additional methods on the types that are given by their first parameter. Extension methods can only be created in static classes.

Example:

```

Public static class Extensions
{
    static void addOne(this int val) {return val++;}
}

int val2 = 1;
int val3 = val2.addOne();

```

0.7.2 Lambda Expressions

As we mentioned in the beginning of the chapter, Microsoft wants to strive for more functional style syntax in their new upcoming version. Functional languages like Haskell, Lisp and ML are based on lambda calculus which can describe functions and their evolution.

With lambda expression it will be easier to create and describe anonymous methods. A lambda expression has the form:

```
(param) =>expr
```

`param` is the parameter to the function and the `expr` is the body of the function. The parameters of a lambda expression can also be explicitly or implicitly typed. In an explicitly parameter list the types are explicitly stated which is not the case for implicitly once.

Example:

```

Implicit type
x =>x + 1

```

```

Explicit type
(int x) =>x + 1

```

The main reason why lambda expression is a better way to create anonymous methods is because you don't have to create large code blocks for the function body. To illustrate this we will show two examples first with delegates and then with lambda expression.

Let's say we have a list of ints and we want to create a new list with only even numbers.

Example with anonymous methods using delegates:

```

List<int>list = new List<int>();
list.Add(1); list.Add(2); list.Add(3); list.Add(4);

```

```
List<int>evenNumbers = list.FindAll(delegate(int i) {return (i % 2) == 0;});
```

In this case we have to create a codeblock for the delegate and the types must be explicitly stated, in our case, `i` have to be declared as an integer. Lambda expression can save us some work. Example with anonymous methods using

lambda expression:

```
List<int>list = new List<int>();  
list.Add(1); list.Add(2); list.Add(3); list.Add(4);
```

```
List<int>evenNumbers = list.FindAll(i =>(i % 2) == 0);
```

As you can see lambda expression is more effective. We don't have to capsule syntax in a code block and the expression can be implicitly typed (we don't have to declare `i` as an integer).

0.7.3 Implicit variables

Implicitly typed local variables are declared using the type 'var'. The type witch will actually be used is determined by the expression where the variable was declared. For this to work implicitly declared variables must always be initialized upon declaration. The following examples will show what an implicitly typed variable might look like and what it corresponds to:

```
var i = 1;  
var s = "Hello";  
var d = 1.0;  
var numbers = new int[] { 1, 2, 3 };  
var strings = new[] { "Hello", "there", "!" };
```

These implicitly typed variables will be converted into the following when used:

```
int i = 5;  
string s = "Hello";  
double d = 1.0;  
int[] numbers = new int[] { 1, 2, 3 };
```

An implicitly typed variable or a set of variables must always be initialized by an expression and not by an object or anything else. That's why we have to use the 'new' expression when declaring a set of variables. The last case where an array containing strings is declared but without any string type is actually a special case where both object initializers and an anonymous type is used. These features and what they corresponds to will be explained more in detail later on. A unique occasion arises when there is a user defined type named `var` in existance, `C#` will then always use the user defined type over the actual implicit type. A final example will show the wrong use of implicitly typed variables:

```
var x;  
var y = 1, 2, 3;  
var z = null;
```

In the first case the declared variable `x` isn't initialized and therefore an error will occur. The variable `y` however is initialized but not with an expression and also generates an error. Last we try to set `z` to a null value and this will also result in an error since `C#` can't derive `null` to a specific type.

0.7.4 Object initializers

This new feature offers an alternative to the use of a constructor. With object initializers you can initialize member variables or elements in a collection where the object is being created. A first example will show how an initializer is used:

```
class Customer
{
    public string name;
    string address;

    public string Adress { get { return address; } set { adress = value; } }
}
```

This syntax example defines a initializer(Adress) for the class Customer. We can later use it when creating an object by adding bracketts and defining values like the example below.

```
var customer = new Customer{name = "John Doe", address = "123 23rd Street"};
```

As you might notice 'name' is also set at creation, this can be done because we have public access to it, this is not the case for the variable 'adress' which justfys the use of an initializer. An object initializer can only consist of one member variable and thus another object initializer would be needed if 'name' didn't have public access. It's important to remember that object initializers doesn't replace a constructor. A constructor can still be used as normal, if fact you can even use a constructor and object initializers at the same time. This is simply done by calling the constructor with it's parameters and then adding bracketts after, like the prevoius example.

0.7.5 Anonymous Types

The use of object initializers enables a user to create and use anonymous types. An anonymous type is basiclly a temporary class without any name or methods. An example will show how to create an anonymous type:

```
var a1 = new Name = "John Doe", Age = 32 ;
var a2 = new Name = "Jane Doe", Age = 24 ;
if(a1 == a2)
    a1 = a2;
```

Here we actually don't have two anonymous types but two instances of the same anonymous type. If an anonymous type has the same number of parameters, with matching names and in the same order, an instance of the same type is created. This enables you to compare anoymous types or assagin new vaules to instances of the same type as shown in the example. Anonymous types can be confusing to read and hard to keep track of at times and they are therefore best put to use in temporary situations.

Bibliography

- [1] Anders Forsberg. *Programmering i C#*. studentlitteratur, 2005
- [2] codersource. <http://www.codersource.net>
- [3] wikipedia. <http://en.wikipedia.org>
- [4] microsoft. <http://msdn.microsoft.com>
- [5] microsoft. <http://msdn2.microsoft.com>
- [6] microsoft. <http://blogs.msdn.com>
- [7] gotdotnet. <http://www.gotdotnet.com>
- [8] genamics. <http://genamics.com>
- [9] csharp-station. <http://www.csharp-station.com>