C# Tutorial

1.1. Getting started

1.1.1. Introduction

Welcome to this C# tutorial. With the introduction of the .NET framework, Microsoft included a new language called C# (pronounced C Sharp). C# is designed to be a simple, modern, general-purpose, object-oriented programming language, borrowing key concepts from several other languages, most notably Java.

C# could theoretically be compiled to machine code, but in real life, it's always used in combination with the .NET framework. Therefore, applications written in C#, requires the .NET framework to be installed on the computer running the application. While the .NET framework makes it possible to use a wide range of languages, C# is sometimes referred to as THE .NET language, perhaps because it was designed together with the framework.

C# is an Object Oriented language and does not offer global variables or functions. Everything is wrapped in classes, even simple types like int and string, which inherit from the System.Object class.

In the following chapters, you will be guided through the most important topics about C#.

1.1.2. Visual Studio Community

Visual Studio is the professional IDE (Integrated Development Environment) of choice for many .NET developers. It's created by Microsoft, who also makes the .NET framework as well as the C# programming language, so this makes perfect sense. Historically, VS (short for Visual Studio) has been expensive to use, but fortunately for you and me, Microsoft has offered a free version for individual developers for many years.

Previously, this free version came in separate versions for various tasks, e.g. Visual C# Express, Visual Web Developer and so on. However, now the name is simply Visual Studio Community and just like the Express versions, it's a lighter version of the professional version of Visual Studio. This means that you will be missing out on some functionality, but don't worry about it - the Community edition contains all but the most advanced features and it will be more than enough to learn C# through this tutorial.

1.1.2.1. Download Visual Studio Community

So, to get started with this tutorial, go ahead and download Visual Studio Community from visualstudio.com . Here's a direct link to the download page:

https://www.visualstudio.com/downloads/

As soon as you have downloaded and installed it, you are ready to proceed with the next articles, where we will create your very first C# application.

1.1.2.2. Not using Windows?

Don't worry, there's a version for macOS as well - just follow the link above and be sure to select the version of **Visual Studio Community for macOS**!

1.1.3. Hello, world!

If you have ever learned a programming language, you know that they all start with the "Hello, world!" example, and who are we to break such a fine tradition? Start Visual Studio Community (introduced in the last chapter), and select **File** -> **New** -> **Project**. From the project dialog, select the Console App (.NET framework). This is the most basic application type on a Windows system, but it's great for learning the language. Once you click Ok, Visual Studio creates a new project for you, including a file called Program.cs. This is where all the fun is, and it should look something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1
{
     class Program
     {
          static void Main(string[] args)
          {
          }
     }
}
```

Actually, all these lines don't really accomplish anything, or at least it may seem so. Try running the application by pushing F5 on your keyboard. This will make Visual Studio compile and execute your code, but as you will see, it doesn't do much. You will likely just see a black window launch and close again. That is because our application doesn't do anything yet. In the next chapter we will go through these lines to see what they are all about, but for now, we really would like to see some results, so let's pretend that we know all about C# and add a couple of lines to get some output. Inside the last set of { }, add these lines:

```
Console.WriteLine("Hello, world!");
Console.ReadLine();
```

The code of your first application should now look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace ConsoleApp1
{
     class Program
     {
          static void Main(string[] args)
          {
                    Console.WriteLine("Hello, world!");
                    Console.ReadLine();
          }
     }
}
```

Once again, hit F5 to run it, and you will see the black window actually staying, and even displaying our greeting to the world. Okay, so we added two lines of code, but what to they do? One of the nice things about C# and the .NET framework is the fact that a lot of the code makes sense even to the untrained eye, which this example shows.

The first line uses the Console class to output a line of text, and the second one reads a line of text from the console. Read? Why? Actually this is a bit of a trick, since without it, the application would just end and close the window with the output before anyone could see it.

The ReadLine command tells the application to wait for input from the user, and as you will notice, the console window now allows you to enter text. Press Enter to close it. Congratulations, you have just created your first C# application! Read on in the next chapter for even more information about what's actually going on.

1.1.4. Hello world explained

In the previous chapter, we tried writing a piece of text to the console, in our first C# application. To see some actual progress, we didn't go into much detail about the lines of code we used, so this chapter is an explanation of the Hello world example code. As you can probably see from the code, some of the lines look similar, so we will bring them back in groups for an individual explanation. Let's start with the shortest and most common characters in our code: The { and }. They are often referred to as curly braces, and in C#, they mark the beginning and end of a logical block of code. The curly braces are used in lots of other languages, including C++, Java, JavaScript and many others. As you can see in the code, they are used to wrap several lines of code which belong together. In later examples, it will be clearer how they are used.

Now let's start from the beginning:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

using is a keyword, highlighted with blue by the editor. The using keyword imports a namespace, and a namespace is a collection of classes. Classes brings us some sort of functionality, and when working with an advanced IDE like Visual Studio, it will usually create parts of the trivial code for us. In this case, it created a class for us, and imported the namespaces which are required or expected to be used commonly. In this case, 5 namespaces are imported for us, each containing lots of useful classes. For instance, we use the Console class, which is a part of the System namespace.

On the other hand, we don't really use the *System.Linq* namespace yet (for example), so if you're a purist, you may choose to remove this line, but it won't make much of a difference at this point.

As you can see, we even get our own namespace:

```
namespace ConsoleApp1
```

The namespace ConsoleApp1 is now the main namespace for this application, and new classes will be a part of it by default. Obviously, you can change this, and create classes in another namespace. In that case, you will have to import this new namespace to use it in your application, with the using statement, like any other namespace.

Next, we define our class. Since C# is truly an Object Oriented language, every line of code that actually does something, is wrapped inside a class. In this case, the class is simply called Program:

```
class Program
```

We can have more classes, even in the same file. For now, we only need one class. A class can contain several variables, properties and methods, concepts we will go deeper into later on. For now, all you need

to know is that our current class only contains one method and nothing else. It's declared like this:

```
static void Main(string[] args)
```

This line is probably the most complicated one in this example, so let's split it up a bit. The first word is **static**. The static keyword tells us that this method should be accesible without instantiating the class, but more about this in our chapter about classes.

The next keyword is **void**, and tells us what this method should return. For instance, it could be an integer or a string of text, but in this case, we don't want our method to return anything (C# uses the keyword *void* to express the concept of *nothing*).

The next word is **Main**, which is simply the name of our method. This method is the so-called entry-point of our application, that is, the first piece of code to be executed, and in our example, the only piece to be executed.

Now, after the name of a method, a set of arguments can be specified within a set of parentheses. In our example, our method takes only one argument, called **args**. The type of the argument is a **string**, or to be more precise, an array of strings, but more on that later. If you think about it, this makes perfect sense, since Windows applications can always be called with an optional set of arguments. These arguments will be passed as text strings to our main method.

And that's it. You should now have a basic understanding of our first C# application, as well as the basic principles of what makes a console application work.

1.2. The Basics

1.2.1. Variables

A variable can be compared to a storage room, and is essential for the programmer. In C#, a variable is declared like this:

<data type> <name>;

An example could look like this:

string name;

That's the most basic version, but the variable doesn't yet have a value. You can assign one at a later point or at the same time as declaring it, like this:

```
<data type> <name> = <value>;
```

If this variable is not local to the method you're currently working in (e.g. a class member variable), you might want to assign a visibility to the variable:

<visibility> <data type> <name> = <value>;

And a complete example:

```
private string name = "John Doe";
```

The visibility part is related to classes, so you can find a more complete explanation of it in the chapter about classes. Let's concentrate on the variable part with an example where we actually use a couple of them:

```
using System;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               string firstName = "John";
               string lastName = "Doe";
```

```
            Console.WriteLine("Name: " + firstName + " " + lastName);

            Console.WriteLine("Please enter a new first name:");
            firstName = Console.ReadLine();

            Console.WriteLine("New name: " + firstName + " " +
lastName);

            Console.ReadLine();
        }
    }
}
```

Okay, a lot of this has already been explained, so we will jump directly to the interesting part. First of all, we declare a couple of variables of the string type. A string simply contains text, as you can see, since we give them a value straight away. Next, we output a line of text to the console, where we use the two variables. The string is made up by using the + characters to "collect" the different parts.

Next, we ask the user to enter a new first name, and then we use the ReadLine() method to read the user input from the console and enter it into the firstName variable. Once the user presses the Enter key, the new first name is assigned to the variable, and in the next line we output the name presentation again, to show the change. We have just used our first variable and the single most important feature of a variable: The ability to change its value at runtime.

Another interesting example is doing math. Here is one, based on a lot of the same code we have just used:

```
int number1, number2;

Console.WriteLine("Please enter a number:");
number1 = int.Parse(Console.ReadLine());

Console.WriteLine("Thank you. One more:");
number2 = int.Parse(Console.ReadLine());

Console.WriteLine("Adding the two numbers: " + (number1 + number2));

Console.ReadLine();
```

Put this in our Main method, and try it out. The only new "trick" we use here, is the int.Parse() method. It simply reads a string and converts it into an integer. As you can see, this application makes no effort to validate the user input, and if you enter something which is not a number, an exception will be raised. More

about those later.

1.2.1.1. Variables & scope

So far, we have only used local variables, which are variables defined and used within the same method. In C#, a variable defined inside a method can't be used outside of this method - that's why it's called local. If you're familiar with other programming languages, you may also know about global variables, which can be accessed from more places, but C# doesn't support the concept of global variables. Instead, you can define a field on a class, which can be accessed from all the methods of this class. Allow me to demonstrate this with an example:

```
using System;

namespace VariableScope
{
     class Program
     {
          private static string helloClass = "Hello, class!";

          static void Main(string[] args)
          {
               string helloLocal = "Hello, local!";
               Console.WriteLine(helloLocal);
               Console.WriteLine(Program.helloClass);
               DoStuff();
          }

          static void DoStuff()
          {
               Console.WriteLine("A message from DoStuff: " +
Program.helloClass);
          }
     }
}
```

Notice the **helloClass** member, declared on the class scope instead of inside a method - this will allow us to access it from both our *Main()* method as well as our own *DoStuff()* method. That is not true for our **helloLocal** variable, which has been declared inside the Main() method and can therefore only be used inside of this specific method.

The concept of differentiating between where a variable has been declared is called **scoping** and it prevents your code from becoming a huge mess of variables which can be changed from too many places. Another technique that helps us with this is called member *visibility* (in this case illustrated with the private

keyword), which we'll discuss in the chapter about classes.

1.2.1.2. Summary

Variables allow you to store data of various types, e.g. text strings, numbers or custom objects. There are local variables, which are accessible inside of the method in which it has been defined, and then there are class fields, which can be accessed from all the methods of the class and even outside of the class, if the *visibility* permits it.

1.2.2. Data types

Data types are used everywhere in a programming language like C#. Because it's a strongly typed language, you are required to inform the compiler about which data types you wish to use every time you declare a variable, as you will see in the chapter about variables. In this chapter we will take a look at some of the most used data types and how they work.

bool is one of the simplest data types. It can contain only 2 values - false or true. The bool type is important to understand when using logical operators like the if statement.

int is short for integer, a data type for storing numbers without decimals. When working with numbers, int is the most commonly used data type. Integers have several data types within C#, depending on the size of the number they are supposed to store.

string is used for storing text, that is, a number of chars. In C#, strings are immutable, which means that strings are never changed after they have been created. When using methods which changes a string, the actual string is not changed - a new string is returned instead.

char is used for storing a single character.

float is one of the data types used to store numbers which can contain decimals.

1.2.2.1. Summary

These are just the most basic data types in C# and I only told you the very basic stuff about them, because it's a pretty dry subject and you may want to see some data types in action before you read more about them. So, move on to the next article, where we'll be using variables to contain data of various types!

1.2.3. Functions

A function allows you to encapsulate a piece of code and call it from other parts of your code. You may very soon run into a situation where you need to repeat a piece of code, from multiple places, and this is where functions come in. In C#, they are basically declared like this:

```
<visibility> <return type> <name>(<parameters>)
{
          <function code>
}
```

To call a function, you simply write its name, an open parenthesis, then parameters, if any, and then a closing parenthesis, like this:

```
DoStuff();
```

Here is an example of our DoStuff() function:

```
public void DoStuff()
{
     Console.WriteLine("I'm doing something...");
}
```

The first part, public, is the visibility, and is optional. If you don't define any, then the function will be private. More about that later on. Next is the type to return. It could be any valid type in C#, or as we have done it here, void. A void means that this function returns absolutely nothing. Also, this function takes no parameters, as you can see from the empty set of parentheses, so it's actually just a tad bit boring. Let's change that:

```
public int AddNumbers(int number1, int number2)
{
     int result = number1 + number2;
     return result;
}
```

We've changed almost everything. The function now returns an integer, it takes two parameters (both integers), and instead of outputting something, it makes a calculation and then returns the result. This means that we can add two numbers from various places in our code, simply by calling this function, instead of having to write the calculation code each time. While we don't save that much time and effort in this small example, you better believe that you will learn to love functions, the more you use C#. This function is called like this:

```
int result = AddNumbers(10, 5);
Console.WriteLine(result);
```

As mentioned, this function actually returns something, and it has to, because we told C# that it's supposed to do so. When declaring anything else than void as a return type, we are forcing our self to return something. You can try removing the return line from the example above, and see the compiler complain:

'AddNumbers(int, int)': not all code paths return a value

The compiler is reminding us that we have a function which doesn't return something, although we promised. And the compiler is pretty clever! Instead of removing the line, try something like this:

```
public int AddNumbers(int number1, int number2)
{
     int result = number1 + number2;
     if(result > 10)
     {
          return result;
     }
}
```

You will see the exact same error - but why? Because there is no guarantee that our if statement will evaluate to true and the return line being executed. You can solve this by having a second, default like return statement in the end:

```
public int AddNumbers(int number1, int number2)
{
     int result = number1 + number2;
     if(result > 10)
     {
          return result;
     }
     return 0;
}
```

This will fix the problem we created for ourselves, and it will also show you that we can have more than one return statement in our function. As soon as a return statement is reached, the function is left and no more code in it is executed. In this case, it means that as long as the result is higher than 10, the "return 0" is never reached.

1.2.4. Function parameters

In the previous chapter, we had a look at functions. We briefly discussed parameters, but only briefly. While parameters are very simple and straight forward to use, there are tricks which can make them a lot more powerful.

The first thing that we will take a look at, is the out and ref modifiers. C#, and other languages as well, differ between two parameters: "by value" and "by reference". The default in C# is "by value", which basically means that when you pass on a variable to a function call, you are actually sending a copy of the object, instead of a reference to it. This also means that you can make changes to the parameter from inside the function, without affecting the original object you passed as a parameter.

With the ref and the out keyword, we can change this behavior, so we pass along a reference to the object instead of its value.

1.2.4.1. The ref modifier

Consider the following example:

```
static void Main(string[] args)
{
     int number = 20;
     AddFive(number);
     Console.WriteLine(number);
     Console.ReadKey();
}

static void AddFive(int number)
{
     number = number + 5;
}
```

We create an integer, assign the number 20 to it, and then we use the AddFive() method, which should add 5 to the number. But does it? No. The value we assign to number inside the function, is never carried out of the function, because we have passed a copy of the number value instead of a reference to it. This is simply how C# works, and in a lot of cases, it's the preferred result. However, in this case, we actually wish to modify the number inside our function. Enter the ref keyword:

```
static void Main(string[] args)
{
     int number = 20;
     AddFive(ref number);
     Console.WriteLine(number);
     Console.ReadKey();
```

```
}

static void AddFive(ref int number)
{
     number = number + 5;
}
```

As you can see, all we've done is added the ref keyword to the function declaration as well as to the call function. If you run the program now, you will see that the value of number has now changed, once we return from the function call.

1.2.4.2. The out modifier

The out modifier works pretty much like the ref modifier. They both ensure that the parameter is passed by reference instead of by value, but they do come with two important differences: A value passed to a ref modifier has to be initialized before calling the method - this is not true for the out modifier, where you can use un-initialized values. On the other hand, you can't leave a function call with an out parameter, without assigning a value to it. Since you can pass in un-initialized values as an out parameter, you are not able to actually use an out parameter inside a function - you can only assign a new value to it.

Whether to use out or ref really depends on the situation, as you will realize once you start using them. Both are typically used to work around the issue of only being able to return one value from a function, with C#.

Using the out modifier is just like using the ref modifier, as shown above. Simply change the ref keyword to the out keyword. In the example above, also remember to remove the value assigned to number in the method and declare it in the call function instead.

1.2.4.3. The params modifier

So far, all of our functions have accepted a fixed amount of parameters. However, in some cases, you might need a function which takes an arbitrary number of parameters. This could of course be done by accepting an array or a list as a parameter, like this:

```
static void GreetPersons(string[] names) { }
```

However, calling it would be a bit clumsy. In the shortest form, it would look like this:

```
GreetPersons(new string[] { "John", "Jane", "Tarzan" });
```

It is acceptable, but it can be done even smarter, with the params keyword:

```
static void GreetPersons(params string[] names) { }
```

Calling it would then look like this:

```
GreetPersons("John", "Jane", "Tarzan");
```

Another advantage of using the params approach, is that you are allowed to pass zero parameters to it as well. Functions with params can even take other parameters as well, as long as the parameter with the params keyword are the last one. Besides that, only one parameter using the params keyword can be used per function. Here is a last and more complete example:

```
static void Main(string[] args)
{
     GreetPersons(0);
     GreetPersons(25, "John", "Jane", "Tarzan");
     Console.ReadKey();
}

static void GreetPersons(int someUnusedParameter, params string[] names)
{
     foreach(string name in names)
          Console.WriteLine("Hello, " + name);
}
```

1.2.5. Code Comments

When writing code, you will quickly get used to the fact that pretty much any character or word you enter will have a special meaning. For instance, you will see a lot of **keywords** in C#, like *class*, *namespace*, *public* and many more. You will also see that the compiler makes sure that you are using these keywords, as well as your own variables and methods, in the correct way. C# is a fairly strict language and the compiler will help you make sure that everything is entered the way it should be. However, you do have a single possibility to write whatever you like, thanks to the concept of **code comments**.

You may have already experienced comments in some of the code you have seen, be it C# or any other programming language - the concept of comments in code is pretty universal. The way they are written varies a lot though, so let's have a look at the type of comments you can use in your C# code.

1.2.5.1. Single-line comments

The most basic type of comment in C# is the single-line comment. As the name indicates, it turns a single line into a comment - let's see how it might look:

```
// My comments about the class name could go here...
class Program
{
          ......
```

So that's it: Prefix your lines with two forward slashes (//) and your text goes from something the compiler will check and complain about, to something the compiler completely ignores. And while this only applies to the prefixed line, you are free to do the same on the next line, essentially using single-line comments to make multiple comment lines:

```
// My comments about the class name could go here...
// Add as many lines as you would like
// ...Seriously!
class Program
{
          ......
```

1.2.5.2. Multi-line comments

In case you want to write multiple lines of comments, it might make more sense to use the multi-line comment variant that C# offers. Instead of having to prefix every line, you just enter a start and stop character sequence - everything in between is treated as comments:

```
/*
My comments about the class name could go here...
Add as many lines of comments as you want
     ...and use indentation, if you want to!
```

```
*/
class Program
{
          ......
```

Use the *start sequence* of forward-slash-asterisk (/*), write whatever you like, across multiple lines or not, and then end it all with the *end sequence* of asterisk-forward-slash (*/). In between these markers, you can write whatever you want.

As with pretty much any other programming related subject, whether to use multiple single-line comments or one multi-line comment is often debated. Personally, I use both, for various situations - in the end, it's all up to you!

1.2.5.3. Documentation comments

Documentation Comments (sometimes referred to as XML Documentation Comments) looks like regular comments, but with embedded XML. Just like with regular comments, they come in two forms: Single-line and multi-line. You also write them the same way, but with an extra character. So, single-line XML Documentation Comments uses three forward slashes (///) instead of two, and the multi-line variant gets an extra asterisk added in the start delimiter. Let's see how it looks:

```
class User
{
     /// <summary>
     /// The Name of the User.
     /// </summary>
     public string Name { get; set; }

     /**
     * <summary>The Age of the User.</summary>
     */
     public string Age { get; set; }
}
```

Here you can see both variants - single-line and multi-line. The result is the same, but the first variant tends to be the most commonly used for documentation comments.

Documenting your types and their members with documentation comments is a pretty big subject, and therefore it will be covered more in depth in a later article, but now you know how they look!

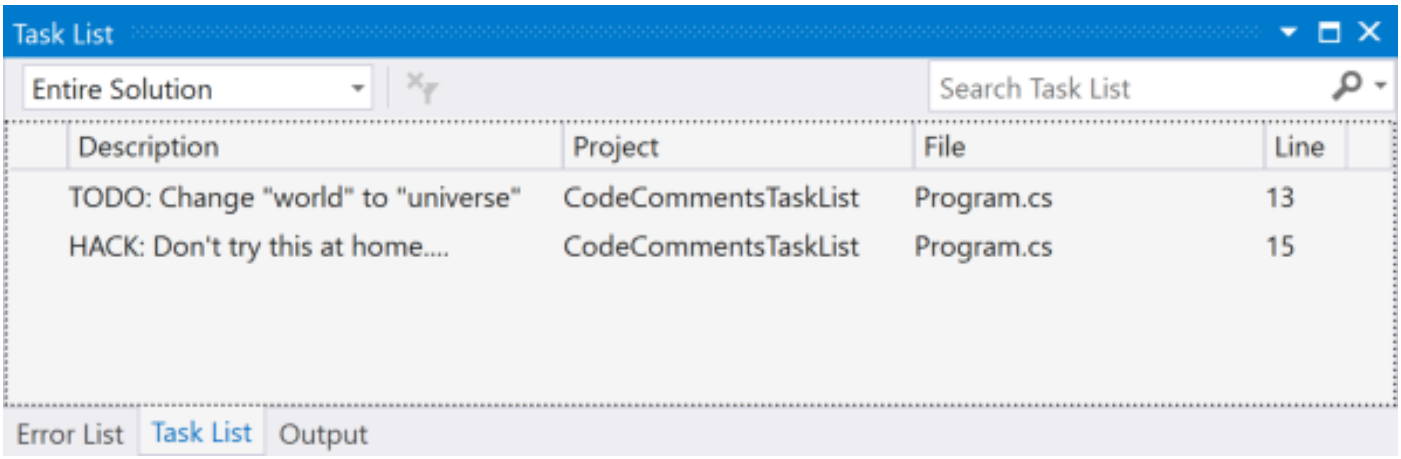
1.2.5.4. Code comments & the Task list

If you're using Visual Studio, you can actually get some help tracking your code comments. In the Task List window (access it from the menu **View** > **Task List**) your comments will appear if they use the special, but

very simple Task List comment syntax:

```
//TODO: Change "world" to "universe"
Console.WriteLine("Hello, world!");
//HACK: Don't try this at home....
int answerToLife = 42;
```

So if the single-line comment is immediately followed by TODO or HACK, it will appear in the Task List of Visual Studio, like this:



And there are more types - depending on the version of Visual Studio you're using, it will respond to some or all of the following comment tokens:

- TODO
- HACK
- NOTE
- UNDONE

You can even add your own tokens, if you want to - just follow the steps described in this article.

1.2.5.5. Summary

Code comments are extremely useful in documenting your code or for the purpose of leaving clues to your self or your potential colleagues on how stuff works. As an added benefit, they're great when you need to test something quickly - just copy a line and comment out the original line and you can see how it works now. If you're not happy with the result, you can just delete the new line and uncomment the original line and you're back to where you started.

And don't worry about the end-user snooping through your comments - they are, as already mentioned, completely ignored by the compiler and therefore not in anyway included in your final DLL or EXE file. Code comments are your personal free-space when programming, so use it in any way you want to.

1.3. Control Structures

1.3.1. The if statement

One of the single most important statements in every programming language is the if statement. Being able to set up conditional blocks of code is a fundamental principal of writing software. In C#, the if statement is very simple to use. If you have already used another programming language, chances are that you can use the if statement in C# straight away. In any case, read on to see how it's used. The if statement needs a boolean result, that is, true or false. In some programming languages, several datatypes can be automatically converted into booleans, but in C#, you have to specifically make the result boolean. For instance, you can't use if(number), but you can compare a number to something, to generate a true or false, like we do later on.

In the previous chapter we looked at variables, so we will expand on one of the examples to see how conditional logic can be used.

```
using System;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               int number;

               Console.WriteLine("Please enter a number between 0 and
10:");
               number = int.Parse(Console.ReadLine());

               if(number > 10)
                    Console.WriteLine("Hey! The number should be 10 or
less!");
               else
                    if(number < 0)
                         Console.WriteLine("Hey! The number should be 0
or more!");
                    else
                         Console.WriteLine("Good job!");

               Console.ReadLine();
```

```
            }
        }
}
```

We use 2 if statements to check if the entered number is between 0 and 10, and a companion of the if statement: The else keyword. Its meaning should be obvious to anyone speaking English - it simply offers an alternative to the code being executed if the condition of the if statement is not met.

As you may have noticed, we don't use the { and } characters to define the conditional blocks of code. The rule is that if a block only contains a single line of code, the block characters are not required. Now, this seems like a lot of lines to simply check a number, doesn't it? It can be done with fewer lines of code, like this:

```
if((number > 10) || (number < 0))
     Console.WriteLine("Hey! The number should be 0 or more and 10 or
less!");
else
     Console.WriteLine("Good job!");
```

We put each condition in a set of parentheses, and then we use the || operator, which simply means "or", to check if the number is either more than 10 OR less than 0. Another operator you will be using a lot is the AND operator, which is written like this: &&. Could we have used the AND operator instead? Of course, we simply turn it around a bit, like this:

```
if((number <= 10) && (number >= 0))
     Console.WriteLine("Good job!");
else
     Console.WriteLine("Hey! The number should be 0 or more and 10 or
less!");
```

This introduces a couple of new operators, the "less than or equal to" and the "greater than or equal to".

1.3.2. The switch statement

The switch statement is like a set of if statements. It's a list of possibilities, with an action for each possibility, and an optional default action, in case nothing else evaluates to true. A simple switch statement looks like this:

```
int number = 1;
switch(number)
{
     case 0:
          Console.WriteLine("The number is zero!");
          break;
     case 1:
          Console.WriteLine("The number is one!");
          break;
}
```

The identifier to check is put after the switch keyword, and then there's the list of case statements, where we check the identifier against a given value. You will notice that we have a break statement at the end of each case. C# simply requires that we leave the block before it ends. In case you were writing a function, you could use a return statement instead of the break statement.

In this case, we use an integer, but it could be a string too, or any other simple type. Also, you can specify the same action for multiple cases. We will do that in the next example too, where we take a piece of input from the user and use it in our switch statement:

```
Console.WriteLine("Do you enjoy C# ? (yes/no/maybe)");
string input = Console.ReadLine();
switch(input.ToLower())
{
     case "yes":
     case "maybe":
          Console.WriteLine("Great!");
          break;
     case "no":
          Console.WriteLine("Too bad!");
          break;
}
```

In this example, we ask the user a question, and suggest that they enter either yes, no or maybe. We then read the user input, and create a switch statement for it. To help the user, we convert the input to lowercase before we check it against our lowercase strings, so that there is no difference between lowercase and uppercase letters.

Still, the user might make a typo or try writing something completely different, and in that case, no output will be generated by this specific switch statement. Enter the default keyword!

```
Console.WriteLine("Do you enjoy C# ? (yes/no/maybe)");
string input = Console.ReadLine();
switch(input.ToLower())
{
     case "yes":
     case "maybe":
          Console.WriteLine("Great!");
          break;
     case "no":
          Console.WriteLine("Too bad!");
          break;
     default:
          Console.WriteLine("I'm sorry, I don't understand that!");
          break;
}
```

If none of the case statements has evaluated to true, then the default statement, if any, will be executed. It is optional, as we saw in the previous examples.

1.3.3. Loops

Another essential technique when writing software is looping - the ability to repeat a block of code X times. In C#, they come in 4 different variants, and we will have a look at each one of them.

1.3.3.1. The while loop

The while loop is probably the most simple one, so we will start with that. The while loop simply executes a block of code as long as the condition you give it is true. A small example, and then some more explanation:

```
using System;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               int number = 0;

               while(number < 5)
               {
                    Console.WriteLine(number);
                    number = number + 1;
               }

               Console.ReadLine();
          }
     }
}
```

Try running the code. You will get a nice listing of numbers, from 0 to 4. The number is first defined as 0, and each time the code in the loop is executed, it's incremented by one. But why does it only get to 4, when the code says 5? For the condition to return true, the number has to be less than 5, which in this case means that the code which outputs the number is not reached once the number is equal to 5. This is because the condition of the while loop is evaluated before it enters the code block.

1.3.3.2. The do loop

The opposite is true for the do loop, which works like the while loop in other aspects through. The do loop evaluates the condition after the loop has executed, which makes sure that the code block is always executed at least once.

```
int number = 0;
do
{
Console.WriteLine(number);
number = number + 1;
} while(number < 5);
```

The output is the same though - once the number is more than 5, the loop is exited.

1.3.3.3. The for loop

The for loop is a bit different. It's preferred when you know how many iterations you want, either because you know the exact amount of iterations, or because you have a variable containing the amount. Here is an example of the for loop.

```
using System;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               int number = 5;

               for(int i = 0; i < number; i++)
                    Console.WriteLine(i);

               Console.ReadLine();
          }
     }
}
```

This produces the exact same output, but as you can see, the for loop is a bit more compact. It consists of 3 parts - we initialize a variable for counting, set up a conditional statement to test it, and increment the counter (++ means the same as "variable = variable + 1").

The first part, where we define the i variable and set it to 0, is only executed once, before the loop starts. The last 2 parts are executed for each iteration of the loop. Each time, i is compared to our number variable - if i is smaller than number, the loop runs one more time. After that, i is increased by one.

Try running the program, and afterwards, try changing the number variable to something bigger or smaller than 5. You will see the loop respond to the change.

1.3.3.4. The foreach loop

The last loop we will look at, is the foreach loop. It operates on collections of items, for instance arrays or other built-in list types. In our example we will use one of the simple lists, called an ArrayList. It works much like an array, but don't worry, we will look into it in a later chapter.

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               ArrayList list = new ArrayList();
               list.Add("John Doe");
               list.Add("Jane Doe");
               list.Add("Someone Else");

               foreach(string name in list)
                    Console.WriteLine(name);

               Console.ReadLine();
          }
     }
}
```

Okay, so we create an instance of an ArrayList, and then we add some string items to it. We use the foreach loop to run through each item, setting the name variable to the item we have reached each time. That way, we have a named variable to output. As you can see, we declare the name variable to be of the string type – you always need to tell the foreach loop which datatype you are expecting to pull out of the collection. In case you have a list of various types, you may use the object class instead of a specific class, to pull out each item as an object.

When working with collections, you are very likely to be using the foreach loop most of the time, mainly because it’s simpler than any of the other loops for these kind of operations.

1.4. Classes

1.4.1. Introduction to C# classes

In lots of programming tutorials, information about classes will be saved for much later. However, since C# is all about Object Oriented programming and thereby classes, we will look at a basic introduction to the most important features now.

First of all, a class is a group of related methods and variables. A class describes these things, and in most cases, you create an instance of this class, now referred to as an object. On this object, you use the defined methods and variables. Of course, you can create as many instances of your class as you want to. Classes, and Object Oriented programming in general, is a huge topic. We will cover some of it in this chapter as well as in later chapters, but not all of it.

In the Hello world chapter, we saw a class used for the first time, since everything in C# is built upon classes. Let's expand our Hello world example with a class we build on our own:

```
using System;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               Car car;

               car = new Car("Red");
               Console.WriteLine(car.Describe());

               car = new Car("Green");
               Console.WriteLine(car.Describe());

               Console.ReadLine();

          }
     }

     class Car
     {
          private string color;
```

```
        public Car(string color)
        {
            this.color = color;
        }

        public string Describe()
        {
            return "This car is " + Color;
        }

        public string Color
        {
            get { return color; }
            set { color = value; }
        }
    }
}
```

Okay, lots of new stuff here, but almost all of it is based on things we've already used earlier in this tutorial. As you can see, we have defined a new class, called Car. It's declared in the same file as our main application, for an easier overview, however, usually new classes are defined in their own files. It defines a single variable, called color, which of course is used to tell the color of our car. We declared it as private, which is good practice - accessing variables from the outside should be done using a property. The Color property is defined in the end of the class, giving access to the color variable.

Besides that, our Car class defines a constructor. It takes a parameter which allows us to initialize Car objects with a color. Since there is only one constructor, Car objects can only be instantiated with a color. The Describe() method allows us to get a nice message with the single piece of information that we record about our car. It simply returns a string with the information we provide.

Now, in our main application, we declare a variable of the type Car. After that, we create a new instance of it, with "Red" as a parameter. According to the code of our class, this means that the color red will be assigned as the color of the car. To verify this, we call the Describe() method, and to show how easily we can create several instances of the same class, we do it again, but with another color. We have just created our first functional class and used it.

In the following chapters, concepts like: properties, constructors, and visibility will be explained in more depth.

1.4.2. Properties

Properties allow you to control the accessibility of a class's variables, and is the recommended way to access variables from the outside in an object oriented programming language like C#. In our chapter on classes, we saw the use of a property for the first time, and the concept is actually quite simple. A property is much like a combination of a variable and a method - it can't take any parameters, but you are able to process the value before it's assigned to our returned variable. A property consists of 2 parts, a get and a set method, wrapped inside the property:

```
private string color;

public string Color
{
     get { return color; }
     set { color = value; }
}
```

The get method should return the variable, while the set method should assign a value to it. Our example is as simple as it gets, but it can be extended. Another thing you should know about properties is the fact that only one method is required - either get or set, the other is optional. This allows you to define read-only and write-only properties. Here is a better example of why properties are useful:

```
public string Color
{
     get
{
          return color.ToUpper();
}
     set
{
     if(value == "Red")
               color = value;
     else
               Console.WriteLine("This car can only be red!");
     }
}
```

Okay, we have just made our property a bit more advanced. The color variable will now be returned in uppercase characters, since we apply the ToUpper() method to it before returning it, and when we try to set the color, only the value "Red" will be accepted. Sure, this example is not terribly useful, but it shows the potential of properties.

1.4.3. Constructors and destructors

Constructors are special methods, used when instantiating a class. A constructor can never return anything, which is why you don't have to define a return type for it. A normal method is defined like this:

```
public string Describe()
```

A constructor can be defined like this:

```
public Car()
```

In our example for this chapter, we have a Car class, with a constructor which takes a string as argument. Of course, a constructor can be overloaded as well, meaning we can have several constructors, with the same name, but different parameters. Here is an example:

```
public Car()
{

}

public Car(string color)
{
     this.color = color;
}
```

A constructor can call another constructor, which can come in handy in several situations. Here is an example:

```
public Car()
{
     Console.WriteLine("Constructor with no parameters called!");
}

public Car(string color) : this()
{
     this.color = color;
     Console.WriteLine("Constructor with color parameter called!");
}
```

If you run this code, you will see that the constructor with no parameters is called first. This can be used for instantiating various objects for the class in the default constructor, which can be called from other constructors from the class. If the constructor you wish to call takes parameters, you can do that as well. Here is a simple example:

```
public Car(string color) : this()
{
     this.color = color;
     Console.WriteLine("Constructor with color parameter called!");
}

public Car(string param1, string param2) : this(param1)
{

}
```

If you call the constructor which takes 2 parameters, the first parameter will be used to invoke the constructor that takes 1 parameter.

1.4.3.1. Destructors

Since C# is garbage collected, meaing that the framework will free the objects that you no longer use, there may be times where you need to do some manual cleanup. A destructor, a method called once an object is disposed, can be used to cleanup resources used by the object. Destructors doesn't look very much like other methods in C#. Here is an example of a destructor for our Car class:

```
~Car()
{
     Console.WriteLine("Out..");
}
```

Once the object is collected by the garbage collector, this method is called.

1.4.4. Method overloading

A lot of programming languages support a technique called default/optional parameters. It allows the programmer to make one or several parameters optional, by giving them a default value. It's especially practical when adding functionality to existing code.

For instance, you may wish to add functionality to an existing function, which requires one or more parameters to be added. By doing so, you would break existing code calling this function, since they would now not be passing the required amount of parameters. To work around this, you could define the newly added parameters as optional, and give them a default value that corresponds to how the code would work before adding the parameters.

Default parameters were introduced in C# version 4.0, but up until that, C# coders have been using a different technique, which basically does the same, called method overloading. It allows the programmer to define several methods with the same name, as long as they take a different set of parameters. When you use the classes of the .NET framework, you will soon realize that method overloading is used all over the place. A good example of this, is the Substring() method of the String class. It has an extra overload, like this:

```
string Substring (int startIndex)
string Substring (int startIndex, int length)
```

You can call it with either one or two parameters. If you only call it with one parameter, the length parameter is assumed to be the rest of the string, saving us time whenever we simply want to get the last part of a string.

So, by defining several versions of the same function, how do we avoid having the same code several places? It's actually quite simple: We let the simple versions of the method make the complex version of it do all the work. Consider the following example:

```
class SillyMath
{
     public static int Plus(int number1, int number2)
     {
          return Plus(number1, number2, 0);
     }

     public static int Plus(int number1, int number2, int number3)
     {
          return number1 + number2 + number3;
     }
}
```

We define a Plus method, in two different versions. The first one takes two paramaters, for adding two

numbers, while the second version takes three numbers. The actual work is done in the version that takes three numbers - if we only wish to add two, we call the three parameter version, and simply use 0 as the third paramater, acting as a default value. I know, I know, it's a silly example, as indicated by the name of the class, but it should give you an idea about how it all works.

Now, whenever you feel like doing advanced math by adding a total of four numbers (just kidding here), it's very simple to add a new overload:

```
class SillyMath
{
     public static int Plus(int number1, int number2)
     {
          return Plus(number1, number2, 0);
     }

     public static int Plus(int number1, int number2, int number3)
     {
          return Plus(number1, number2, number3, 0);
     }

     public static int Plus(int number1, int number2, int number3, int
number4)
     {
          return number1 + number2 + number3 + number4;
     }
}
```

The cool thing about this, is that all your existing calls to the Plus method will continue working, as if nothing had been changed. The more you use C#, the more you will learn to appreciate method overloading.

1.4.5. Visibility

The visibility of a class, a method, a variable or a property tells us how this item can be accessed. The most common types of visibility are private and public, but there are actually several other types of visibility within C#. Here is a complete list, and although some of them might not feel that relevant to you right now, you can always come back to this page and read up on them:

public - the member can be reached from anywhere. This is the least restrictive visibility. Enums and interfaces are, by default, publicly visible.

protected - members can only be reached from within the same class, or from a class which inherits from this class.

internal - members can be reached from within the same project only.

protected internal - the same as internal, except that classes which inherit from this class can reach its members; even from another project.

private - can only be reached by members from the same class. This is the most restrictive visibility. Classes and structs are by default set to private visibility.

So for instance, if you have two classes: Class1 and Class2, private members from Class1 can only be used within Class1. You can't create a new instance of Class1 inside of Class2, and then expect to be able to use its private members.

If Class2 inherits from Class1, then only non-private members can be reached from inside of Class2.

1.4.6. Static members

As we saw in a previous chapter, the usual way to communicate with a class is to create a new instance of the class and then work on the resulting object. In most cases, this is what classes are all about; the ability to instantiate multiple copies of the same class and then use them differently in some way. However, in some cases, you might like to have a class which you may use without instantiating it, or at least a class where you can use members of it without creating an object for it. For instance, you may have a class with a variable that always remains the same no matter where and how it's used. This is called a static member, because it remains the same.

A class can be static, and it can have static members, both functions and fields. A static class can't be instantiated, so in other words, it will work more as a grouping of related members than an actual class. You may choose to create a non-static class instead, but let it have certain static members. A non-static class can still be instantiated and used like a regular class, but you can't use a static member on an object of the class. A static class may only contain static members.

First, here is an example of a static class:

```
public static class Rectangle
{
     public static int CalculateArea(int width, int height)
     {
          return width * height;
     }
}
```

As you can see, we use the static keyword to mark the class as static, and then we use it again to mark the method, CalculateArea, as static as well. If we didn't do that, the compiler would complain, since we can't have a non-static member of a static class.

To use this method, we call it directly on the class, like this:

```
Console.WriteLine("The area is: " + Rectangle.CalculateArea(5, 4));
```

We could add other helpful methods to the Rectangle class, but perhaps you are wondering why we are passing on width and height to the actual method, instead of storing it inside the class and then pulling them from there when needed? Because it's static! We could store them, but only one set of dimensions, because there is only one version of a static class. This is very important to understand.

Instead, we can make the class non-static, and then have the CalculateArea as a utility function on this class:

```
public class Rectangle
{
```

```
        private int width, height;

        public Rectangle(int width, int height)
        {
            this.width = width;
            this.height = height;
        }

        public void OutputArea()
        {
            Console.WriteLine("Area output: " + Rectangle.CalculateArea(
this.width, this.height));
        }

        public static int CalculateArea(int width, int height)
        {
            return width * height;
        }
}
```

As you can see, we have made the class non-static. We have also added a constructor, which takes a width and a height and assigns it to the instance. Then we have added an OutputArea method, which uses the static method to calculate the area. This is a fine example of mixing static members with non-static members, in a non-static class.

A common usage of static classes, although frowned upon by some people, are utility/helper classes, where you collect a bunch of useful methods, which might not belong together, but don't really seem to fit elsewhere either.

1.4.7. Inheritance

One of the absolute key aspects of Object Oriented Programming (OOP), which is the concept that C# is built upon, is inheritance, the ability to create classes which inherits certain aspects from parent classes. The entire .NET framework is built on this concept, with the "everything is an object" as a result of it. Even a simple number is an instance of a class, which inherits from the System.Object class, although .NET helps you out a bit, so you can assign a number directly, instead of having to create a new instance of e.g. the integer class.

This subject can be a bit difficult to comprehend, but sometimes it help with some examples, so let's start with a simple one of those:

```
public class Animal
{
     public void Greet()
     {
          Console.WriteLine("Hello, I'm some sort of animal!");
     }
}

public class Dog : Animal
{

}
```

First, we define an Animal class, with a simple method to output a greeting. Then we define a Dog class, and with a colon, we tell C# that the Dog class should inherit from the Animal class. The beautiful thing about this is that it makes sense in the real world as well - a Dog is, obviously, an Animal. Let's try using the classes:

```
Animal animal = new Animal();
animal.Greet();
Dog dog = new Dog();
dog.Greet();
```

If you run this example, you will notice that even though we have not defined a Greet() method for the Dog class, it still knows how to greet us, because it inherits this method from the Animal class. However, this greeting is a bit anonymous, so let's customize it when we know which animal it is:

```
public class Animal
{
     public virtual void Greet()
     {
```

```
            Console.WriteLine("Hello, I'm some sort of animal!");
      }
}

public class Dog : Animal
{
      public override void Greet()
      {
            Console.WriteLine("Hello, I'm a dog!");
      }
}
```

Besides the added method on the Dog class, you should notice two things: I have added the virtual keyword to the method on the Animal class, and on the Dog class, I use the override keyword.

In C#, you are not allowed to override a member of a class unless it's marked as virtual. If you want to, you can still access the inherited method, even when you override it, using the base keyword.

```
public override void Greet()
{
      base.Greet();
      Console.WriteLine("Yes I am - a dog!");
}
```

Methods are not the only thing to get inherited, though. In fact, pretty much all class members will be inherited, including fields and properties. Just remember the rules of visibility, as discussed in a previous chapter.

Inheritance is not only from one class to another - you can have a whole hierarchy of classes, which inherits from eachother. For instance, we could create a Puppy class, which inherits from our Dog class, which in turn inherits from the Animal class. What you can't do in C#, is to let one class inherit from several other classes at the same time. Multiple inheritance, as it's called, is not supported by C#.

1.4.8. Abstract classes

Abstract classes, marked by the keyword abstract in the class definition, are typically used to define a base class in the hierarchy. What's special about them, is that you can't create an instance of them - if you try, you will get a compile error. Instead, you have to subclass them, as taught in the chapter on inheritance, and create an instance of your subclass. So when do you need an abstract class? It really depends on what you do.

To be honest, you can go a long way without needing an abstract class, but they are great for specific things, like frameworks, which is why you will find quite a bit of abstract classes within the .NET framework it self. A good rule of thumb is that the name actually makes really good sense - abstract classes are very often, if not always, used to describe something abstract, something that is more of a concept than a real thing.

In this example, we will create a base class for four legged animals and then create a Dog class, which inherits from it, like this:

```
namespace AbstractClasses
{
      class Program
      {
            static void Main(string[] args)
            {
                  Dog dog = new Dog();
                  Console.WriteLine(dog.Describe());
                  Console.ReadKey();
            }
      }

      abstract class FourLeggedAnimal
      {
            public virtual string Describe()
            {
                  return "Not much is known about this four legged animal!"
;
            }
      }

      class Dog : FourLeggedAnimal
      {

      }
}
```

If you compare it with the examples in the chapter about inheritance, you won't see a big difference. In fact, the abstract keyword in front of the FourLeggedAnimal definition is the biggest difference. As you can see, we create a new instance of the Dog class and then call the inherited Describe() method from the FourLeggedAnimal class. Now try creating an instance of the FourLeggedAnimal class instead:

```
FourLeggedAnimal someAnimal = new FourLeggedAnimal();
```

You will get this fine compiler error:

Cannot create an instance of the abstract class or interface 'AbstractClasses.FourLeggedAnimal'

Now, as you can see, we just inherited the Describe() method, but it isn't very useful in it's current form, for our Dog class. Let's override it:

```
class Dog : FourLeggedAnimal
{
     public override string Describe()
     {
          return "This four legged animal is a Dog!";
     }
}
```

In this case, we do a complete override, but in some cases, you might want to use the behavior from the base class in addition to new functionality. This can be done by using the base keyword, which refers to the class we inherit from:

```
abstract class FourLeggedAnimal
{
     public virtual string Describe()
     {
          return "This animal has four legs.";
     }
}


class Dog : FourLeggedAnimal
{
     public override string Describe()
     {
          string result = base.Describe();
          result += " In fact, it's a dog!";
          return result;
```

```
    }
}
```

Now obviously, you can create other subclasses of the FourLeggedAnimal class - perhaps a cat or a lion? In the next chapter, we will do a more advanced example and introduce abstract methods as well. Read on.

1.4.9. More abstract classes

In the previous chapter, we had a look at abstract classes. In this chapter, we will expand the examples a bit, and throw in some abstract methods as well. Abstract methods are only allowed within abstract classes. Their definition will look like a regular method, but they have no code inside them:

```
abstract class FourLeggedAnimal
{
     public abstract string Describe();
}
```

So, why would you want to define an empty method that does nothing? Because an abstract method is an obligation to implement that very method in all subclasses. In fact, it's checked at compile time, to ensure that your subclasses has this method defined. Once again, this is a great way to create a base class for something, while still maintaining a certain amount of control of what the subclasses should be able to do. With this in mind, you can always treat a subclass as its baseclass, whenever you need to use methods defined as abstract methods on the baseclass. For instance, consider the following example:

```
namespace AbstractClasses
{
     class Program
     {
          static void Main(string[] args)
          {
               System.Collections.ArrayList animalList = new
System.Collections.ArrayList();
               animalList.Add(new Dog());
               animalList.Add(new Cat());
               foreach(FourLeggedAnimal animal in animalList)
                    Console.WriteLine(animal.Describe());
               Console.ReadKey();
          }
     }

     abstract class FourLeggedAnimal
     {
          public abstract string Describe();
     }


     class Dog : FourLeggedAnimal
     {
```

```
        public override string Describe()
        {
            return "I'm a dog!";
        }
    }

    class Cat : FourLeggedAnimal
    {
        public override string Describe()
        {
            return "I'm a cat!";
        }
    }
}
```

As you can see, we create an ArrayList to contain our animals. We then instantiate a new dog and a new cat and add them to the list. They are instantiated as a Dog and a Cat respectively, but they are also of the type FourLeggedAnimal, and since the compiler knows that subclasses of that class contains the Describe() method, you are actually allowed to call that method, without knowing the exact type of animal. So by typecasting to the FourLeggedAnimal, which is what we do in the foreach loop, we get access to members of the subclasses. This can be very useful in lots of scenarios.

1.4.10. Interfaces

In previous chapters, we had a look at abstract classes. Interfaces are much like abstract classes and they share the fact that no instances of them can be created. However, interfaces are even more conceptual than abstract classes, since no method bodies are allowed at all. So an interface is kind of like an abstract class with nothing but abstract methods, and since there are no methods with actual code, there is no need for any fields. Properties are allowed though, as well as indexers and events. You can consider an interface as a contract - a class that implements it is required to implement all of the methods and properties. However, the most important difference is that while C# doesn't allow multiple inheritance, where classes inherit more than a single base class, it does in fact allow for implementation of multiple interfaces!

So, how does all of this look in code? Here's a pretty complete example. Have a look, perhaps try it out on your own, and then read on for the full explanation:

```
using System;
using System.Collections.Generic;

namespace Interfaces
{
     class Program
     {
          static void Main(string[] args)
          {
               List<Dog> dogs = new List<Dog>();
               dogs.Add(new Dog("Fido"));
               dogs.Add(new Dog("Bob"));
               dogs.Add(new Dog("Adam"));
               dogs.Sort();
               foreach(Dog dog in dogs)
                    Console.WriteLine(dog.Describe());
               Console.ReadKey();
          }
     }

     interface IAnimal
     {
          string Describe();

          string Name
          {
               get;
               set;
```

```
        }
    }

    class Dog : IAnimal, IComparable
    {
        private string name;

        public Dog(string name)
        {
            this.Name = name;
        }

        public string Describe()
        {
            return "Hello, I'm a dog and my name is " + this.Name;
        }

        public int CompareTo(object obj)
        {
            if(obj is IAnimal)
                return this.Name.CompareTo((obj as IAnimal).Name);
            return 0;
        }

        public string Name
        {
            get { return name; }
            set { name = value; }
        }
    }
}
```

Let's start in the middle, where we declare the interface. As you can see, the only difference from a class declaration, is the keyword used - interface instead of class. Also, the name of the interface is prefixed with an I for Interface - this is simply a coding standard, and not a requirement. You can call your interfaces whatever you want, but since they are used like classes so much that you might have a hard time telling the difference in some parts of your code, the I prefix makes pretty good sense.

Then we declare the Describe method, and afterwards, the Name property, which has both a get and a set keyword, making this a read and writeable property. You will also notice the lack of access modifiers (public, private, protected etc.), and that's because they are not allowed in an interface - they are all public

by default.

Next up is our Dog class. Notice how it looks just like inheriting from another class, with the colon between the class name and the class/interface being subclassed/implemented. However, in this case, two interfaces are implemented for the same class, simply separated by a comma. You can implement as many interfaces as you want to, but in this case we only implement two - our own IAnimal interface, and the .NET IComparable interface, which is a shared interface for classes that can be sorted. Now as you can see, we have implemented both the method and the property from the IAnimal interface, as well as a CompareTo method from the IComparable interface.

Now you might be thinking: If we have to do all the work our self, by implementing the entire methods and properties, why even bother? And a very good example of why it's worth your time, is given in the top of our example. Here, we add a bunch of Dog objects to a list, and then we sort the list. And how does the list know how to sort dogs? Because our Dog class has a CompareTo method that can tell how to compare two dogs. And how does the list know that our Dog object can do just that, and which method to call to get the dogs compared? Because we told it so, by implementing an interface that promises a CompareTo method! This is the real beauty of interfaces.

1.4.11. Namespaces

In one of the first articles, we briefly discussed namespaces. You probably recognize the keyword, because it's found in most files containing C# code, usually almost in the top. A namespace is essentially a way to group a set of types, e.g. classes, in a named space of its own. When Visual Studio generates a new project for you, it also generates a default namespace in which it places your first file (at least this is true for the Console App project type). It could look like this:

```
using System;

namespace MyProject
{
class Program
{
     static void Main(string[] args)
     {
               // More code below this....
```

In this case, the namespace "MyProject" is now a part of the application and when using its classes outside of it, you will have to prefix the class name with the namespace name. You see the exact same thing when you want to use something burried deep down in the .NET framework, like this:

```
System.IO.File.ReadAllText("test.txt");
```

In this case, we use the *ReadAllText() method* found on the *File class* which exists in the *System.IO namespace*. Of course, it would be tedious to write such a long name each time you wanted to use a class from a namespace, so C# allows you to "import" an entire namespace into the scope of your file with a *using statement*. Again, you might already know them, because you can usually find them at the top of your C# files. For the example above, if we would need the File class more than once, it would make sense to import the System.IO namespace with a using statement like this:

```
using System;
using System.IO;
// More using statements here...
```

1.4.11.1. Why do you need namespaces?

If you have just started programming, you might wonder what we need namespaces for. Why not just put all your classes in the same namespaces so that they are always accessible? You have a valid point, but only if your project is very small. As soon as you start adding more and more classes, it makes very good sense to separate them into namespaces. It simply makes it easier for you to find your code, especially if you places your files in corresponding folders - in fact, if you add a folder to your project and then add a class to it, Visual Studio will automatically put it in a corresponding namespace. So, if you create a folder in *MyProject* called *MyFolder*, classes added to this folder will, by default, be placed in a namespace called

MyProject.MyFolder.

A great example of why namespaces are needed is the .NET framework itself. Just think if ALL the classes in the framework were just floating around in a global namespace - it would be a mess! Instead, they have organized them nicely, with System as the root namespace for most classes and then sub-namespaces like **System.IO** for input/output stuff, **System.Net** for network related stuff and **System.Net.Mail** for mail-related stuff.

1.4.11.2. Name Conflicts with Namespaces

As mentioned, namespaces are also there to encapsulate your types (usually classes), so that they can exist within their own domain. This also means that you are free to create classes with the same name as the ones found elsewhere in your project or even in the .NET framework. For instance, you might decide that you need a File class of your own. As we saw in the previous examples, such a class already exists in the System.IO namespace, but you are free to create one in your own namespace, like this:

```
using System;

namespace MyProject.IO
{
class File
{
     public static void HelloWorld()
     {
          Console.WriteLine("Hello, world!");
     }
}
}
```

Now when you want to use it in your project, e.g. in your Program.cs Main method (if you're working on a Console App, like I am), you can either write the full name:

```
MyProject.IO.File.HelloWorld();
```

But you can also import the namespace, like you can with any other namespace (built-in or user-defined), thanks to the using statement. Here's a more complete example:

```
using System;
using MyProject.IO;

namespace MyProject
{
     class Program
```

```
    {
        static void Main(string[] args)
        {
            File.HelloWorld();
        }
    }
}
```

So far, so good! However, what if you also want to use the File class from the System.IO namespace? Well, this is where the trouble starts, because if you import that namespace as well, with a using statement, the compiler no longer knows which File class you're referring to - our own or the one from the System.IO namespace. This can be solved by only importing one of the namespaces (ideally the one from which you use the most types) and then fully qualifying the name of the other one, like in this example:

```
using System;
using System.IO;

namespace MyProject
{
    class Program
    {
        static void Main(string[] args)
        {
            MyProject.IO.File.HelloWorld();
        }
    }
}
```

But it's a bit cumbersome to type each time, especially if your class is even deeper nested in namespaces, e.g. MyProject.FileStuff.IO.File. Fortunately, C# has a solution for that.

1.4.11.3. Using Alias Directive

To shorten the name of the namespace a lot, you can import the namespace under a different name, with a Using Alias Directive. Notice how I do just that in the next example:

```
using System;
using System.IO;
using MyIO = MyProject.IO;

namespace MyProject
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            File.ReadAllText("test.txt");
            MyIO.File.HelloWorld();
        }
    }
}
```

The magic happens in the third line, where I pull in the MyProject.IO namespace and give it a shorter name (MyIO), which can then be used when we want to access types from it. At this point, we're not saving a lot of keystrokes, but again you should imagine even longer names and levels of namespaces, and believe me, they can get quite long and nested.

1.4.11.4. Summary

Namespaces gives you the opportunity to encapsulate your types into "named spaces", which allows you to get a better structure in your code, as well as have multiple classes with the same name, as long as they exist in separate namespaces.

1.4.12. Constants (the const keyword)

So far, we have dealt a lot with variables and as the name implies, variables can always be changed. The opposite of that is a constant, introduced in C# with the keyword *const*. When declaring a constant, you have to immediately assign a value to it and after that, NO changes can be made to the value of this constant. This is great when you have a value which doesn't ever change, and you want to make sure that it's not manipulated by your code, even by accident.

You will find many constants in the framework itself, e.g. in the Math class, where a constant for PI has been defined:

```
Console.WriteLine(Math.PI);
```

But of course, the interesting part is to declare some constants of our own. A constant can be defined in the scope of a method, like this:

```
static void Main(string[] args)
{
     const int TheAnswerToLife = 42;
     Console.WriteLine("The answer to life, the universe and everything:
" + TheAnswerToLife);
}
```

However, most constants are declared on the class level, so that they can be accessed (but not changed, of course) from all methods of the class and even outside of the class, depending on the visibility. A constant will act like a static member of the class, meaning that you can access it without instantiating the class. With that in mind, let's try a full example where two constants are defined - a private and a public constant:

```
using System;

namespace Constants
{
     class Program
     {
          static void Main(string[] args)
          {
               Console.WriteLine("The fake answer to life: " +
SomeClass.TheFakeAnswerToLife);
               Console.WriteLine("The real answer to life: " +
SomeClass.GetAnswer());
          }
     }
```

```
    class SomeClass
    {
        private const int TheAnswerToLife = 42;
        public const int TheFakeAnswerToLife = 43;

        public static int GetAnswer()
        {
            return TheAnswerToLife;
        }
    }
}
```

Notice how I define a class (SomeClass) with two constants. The first is private, so it can only be access from the class it self, but the other one is public. So, in our main program code, I access both constants differently - first directly, since the fake answer is publicly available, and secondly with the help of the GetAnswer() method.

1.4.12.1. Which types can be used as a constant?

Since constants has to be declared immediately and can't be changed later on, the value you assign to a constant has to be a constant expression and the compiler must be able to evaluate the value already at compile time. This means that numbers, boolean values and strings can be used just fine for a constant, while e.g. a DateTime object can't be used as a constant.

Since the compiler needs to know the value immediately, it also means that there are some limitations to what you can do when setting the value. For instance, these are perfect examples of what you CAN do:

```
const int a = 10;
const float b = a * 2.5f;

const string s1 = "Hello, world!";
const string s2 = s1 + " How are you?";
```

On the other hand, you can't use the result of a method call or a non-constant class member, since these are not constant expressions. Here comes a couple of examples of what you CAN'T do:

```
// NOT possible:
const int a = Math.Cos(4) * 2;
// Possible:
const string s1 = "Hello, world!";
// NOT possible:
const string s2 = s1.Substring(0, 6) + " Universe";
```

The difference lies in what the compiler can be expected to know when it reaches your code, e.g. numbers, strings and other constants, in contrast to what it has to execute to get the value for.

1.4.12.2. A constant alternative: The readonly field

If you're looking for a slightly less restrictive version of a class constant, you may want to have a look at the *readonly* keyword. It's not available on the method level, but it can be used on the class level, to define a field which can only be modified during declaration or the execution of the constructor method of the class. So, as soon as the object is available for use, the *readonly* field will have the same value forever and can't be modified by the user. Let's try it out:

```
class SomeClass
{
     private readonly DateTime rightNow;
     public readonly DateTime later = DateTime.Now.AddHours(2);

     public SomeClass()
     {
          this.rightNow = DateTime.Now;
     }
}
```

So, we have two readonly fields: The first is private, the second one is public (we usually have properties for that, but bear with me here). The first is declared without a value (we can do that with readonly fields, unlike with constants), while the other one is initialized immediately. You will also notice that we're using the DateTime class as the data type, and we assign a non-constant value to it. In other words, we do a lot of stuff that we can't do with constants, making readonly fields a nice alternative to constants.

Notice how I assign a value to the *rightNow* field in the constructor of the SomeClass class. As already mentioned, this is the last chance to assign a value to a readonly field. After that, whether you are in a method inside the defining class or outside, you will get a compile error if you try to assign a value to a readonly field.

1.4.12.3. Summary

A constant can be defined either inside the scope of a method or on the class level. It allows you to define a value which is known already at compile time and which can't be changed later on. Typical types used for constants are integers, floats, strings, and booleans. If you're looking for more flexibility, try the readonly field, as described above.

1.4.13. Partial Classes

If you have worked with C#, or perhaps even another programming language, you are used to the fact that the name of a class has to be unique - there cannot be two classes with the same name, unless they are in different namespaces. However, at one point, Microsoft decided to change this, with the introduction of something called partial classes.

When you define your class with the partial keyword, you or someone else is allowed to extend the functionality of your class with another class, which also needs to be declared as partial. This is useful in the following situations:

- When you have a very large class - you can then keep it in multiple files, to make it easier to work with various parts of the classes. For instance, you could have all the properties in one file and all the methods in another file, while still just having one class.
- When you work with a designer, like the one in Visual Studio - for instance with WinForms, where all the automatically generated designer code can be kept in one file, while your code is kept in another file.

Let me illustrate this with an example. In my project, I have the usual Program.cs, found in a console app. Besides that, I have added two files: PartialClass1.cs and PartialClass2.cs. Here are the files and their contents:

PartialClass1.cs

```
using System;

namespace PartialClasses
{
     public partial class PartialClass
     {
          public void HelloWorld()
          {
               Console.WriteLine("Hello, world!");
          }
     }
}
```

PartialClass2.cs

```
using System;

namespace PartialClasses
{
     public partial class PartialClass
```

```
    {
        public void HelloUniverse()
        {
            Console.WriteLine("Hello, universe!");
        }
    }
}
```

Notice that both classes are defined with the *partial* keyword and have the same names. Also notice that each of them define a method - HelloWorld() and HelloUniverse(). In our Program.cs we can now use this class as if it was defined in only one place, just like any other class:

```
using System;

namespace PartialClasses
{
    class Program
    {
        static void Main(string[] args)
        {
            PartialClass pc = new PartialClass();
            pc.HelloWorld();
            pc.HelloUniverse();
        }
    }
}
```

1.4.13.1. Summary

With partial classes, you can split your classes into multiple files, either because the class definition is very large or when the tools you work with benefits from it, like with the Visual Studio designer for WinForms.

1.5. Collections

1.5.1. Arrays

Arrays works as collections of items, for instance strings. You can use them to gather items in a single group, and perform various operations on them, e.g. sorting. Besides that, several methods within the framework work on arrays, to make it possible to accept a range of items instead of just one. This fact alone makes it important to know a bit about arrays.

Arrays are declared much like variables, with a set of [] brackets after the datatype, like this:

```
string[] names;
```

You need to instantiate the array to use it, which is done like this:

```
string[] names = new string[2];
```

The number (2) is the size of the array, that is, the amount of items we can put in it. Putting items into the array is pretty simple as well:

```
names[0] = "John Doe";
```

But why 0? As it is with so many things in the world of programming, the counting starts from 0 instead of 1. So the first item is indexed as 0, the next as 1 and so on. You should remember this when filling the array with items, because overfilling it will cause an exception. When you look at the initializer, setting the array to a size of 2, it might seem natural to put item number 0, 1 and 2 into it, but this is one item too much. If you do it, an exception will be thrown. We will discuss exceptions in a later chapter.

Earlier, we learned about loops, and obviously these go great with arrays. The most common way of getting data out of an array, is to loop through it and perform some sort of operation with each value. Let's use the array from before, to make a real example:

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               string[] names = new string[2];
```

```
            names[0] = "John Doe";
            names[1] = "Jane Doe";

            foreach(string s in names)
                Console.WriteLine(s);

            Console.ReadLine();
        }
    }
}
```

We use the foreach loop, because it's the easiest, but of course we could have used one of the other types of loop instead. The for loop is good with arrays as well, for instance if you need to count each item, like this:

```
for(int i = 0; i < names.Length; i++)
    Console.WriteLine("Item number " + i + ": " + names[i]);
```

It's actually very simple. We use the Length property of the array to decide how many times the loop should iterate, and then we use the counter (i) to output where we are in the process, as well as get the item from the array. Just like we used a number, a so called indexer, to put items into the array, we can use it to get a specific item out again.

I told you earlier that we could use an array to sort a range of values, and it's actually very easy. The Array class contains a bunch of smart methods for working with arrays. This example will use numbers instead of strings, just to try something else, but it could just as easily have been strings. I wish to show you another way of populating an array, which is much easier if you have a small, predefined set of items that you wish to put into your array. Take a look:

```
int[] numbers = new int[5] { 4, 3, 8, 0, 5 };
```

With one line, we have created an array with a size of 5, and filled it with 5 integers. By filling the array like this, you get an extra advantage, since the compiler will check and make sure that you don't put too many items into the array. Try adding a number more - you will see the compiler complain about it.

Actually, it can be done even shorter, like this:

```
int[] numbers = { 4, 3, 8, 0, 5 };
```

This is short, and you don't have to specify a size. The first approach may be easier to read later on though.

Let's try sorting the array - here's a complete example:

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               int[] numbers = { 4, 3, 8, 0, 5 };

               Array.Sort(numbers);

               foreach(int i in numbers)
                    Console.WriteLine(i);

               Console.ReadLine();
          }
     }
}
```

The only real new thing here is the Array.Sort command. It can take various parameters, for various kinds of sorting, but in this case, it simply takes our array. As you can see from the result, our array has been sorted. The Array class has other methods as well, for instance the Reverse() method. You can look it up in the documentation to see all the features of the Array class.

The arrays we have used so far have only had one dimension. However, C# arrays can be multidimensional, sometimes referred to as arrays in arrays. Multidimensional arrays come in two flavors with C#: Rectangular arrays and jagged arrays. The difference is that with rectangular arrays, all the dimensions have to be the same size, hence the name rectangular. A jagged array can have dimensions of various sizes. Multidimensional arrays are a heavy subject, and a bit out of the scope of this tutorial.

1.5.2. Lists

C# has a range of classes for dealing with lists. They implement the IList interface and the most popular implementation is the generic list, often referred to as List<T>. The T specifies the type of objects contained in the list, which has the added benefit that the compiler will check and make sure that you only add objects of the correct type to the list - in other words, the List<T> is type-safe.

List is much like the ArrayList class, which was the go-to List choice before C# supported generic lists. Therefore you will also see that the List can do a lot of the same stuff as an Array (which also implements the IList interface by the way), but in a lot of situations, List is simpler and easier to work with. For instance, you don't have to create a List with a specific size - instead, you can just create it and .NET will automatically expand it to fit the amount of items as you add them.

As mentioned, the T stands for type and is used to specify the type of objects you want the list to contain. In our first example, I will show you how to create a list which should contain strings:

```
List<string> listOfStrings = new List<string>();
```

This creates an empty list, but adding stuff to it afterwards is very easy with the *Add* method:

```
listOfStrings.Add("a string");
```

However, if you try to add something that is not a string, the compiler will immediately complain about it:

```
listOfStrings.Add(2);

Error CS1503 Argument 1: cannot convert from 'int' to 'string'
```

1.5.2.1. Initializing a list with items

In the above example, we just created a list and then we added an item to it. However, C# actually allows you to create a list AND add items to it within the same statement, using a technique called collection initializers. Let's see how it's done:

```
List<string> listOfNames = new List<string>()
{
     "John Doe",
     "Jane Doe",
     "Joe Doe"
};
```

The syntax is quite simple: Before the usual ending semicolon, we have a set of curly brackets, which in turn holds a list of the values we want to be present in the list from the start. Since this is a list of strings, the initial objects we provide should of course be of the string type. However, the exact same can be accomplished for list of other types, even if we're using our own classes, as I will demonstrate in the next

example.

1.5.2.2. Working with the items

There are several ways to work with the items of a generic list and to show you some of them, I have created a larger example:

```
using System;
using System.Collections.Generic;

namespace Lists
{
     class Program
     {
          static void Main(string[] args)
          {
               List<User> listOfUsers = new List<User>()
               {
                    new User() { Name = "John Doe", Age = 42 },
                    new User() { Name = "Jane Doe", Age = 34 },
                    new User() { Name = "Joe Doe", Age = 8 },
               };

               for(int i = 0; i < listOfUsers.Count; i++)
               {
                    Console.WriteLine(listOfUsers[i].Name + " is " +
listOfUsers[i].Age + " years old");
               }
               Console.ReadKey();
          }
     }

     class User
     {
          public string Name { get; set; }

          public int Age { get; set; }
     }
}
```

Let's start from the bottom, where we define a simple class for holding information about a User - just a name and an age. Back to the top part of the example, where I have changed our list to use this User class

instead of simple strings. I use a collection initializer to populate the list with users - notice how the syntax is the same as before, just a bit more complex because we're dealing with a more complex object than a string.

Once we have the list ready, I use a *for* loop to run through it - to know how many iterations we're going to do, I use the *Count* property of the List. On each iteration, I access the user in question through the indexer of the list, using the square bracket syntax (e.g. listOfUsers[i]). Once I have the user, I output name and age.

1.5.2.3. Adding, Inserting & Removing items

We already tried adding a single item to a list, but there are more options for doing this. First of all, you can Insert an item instead of adding it - the difference is that while the *Add* method always adds to the end of the list, the *Insert* method allows you to insert an item at a specific position. Here's an example:

```
List<string> listOfNames = new List<string>()
{
     "Joe Doe"
};
// Insert at the top (index 0)
listOfNames.Insert(0, "John Doe");
// Insert in the middle (index 1)
listOfNames.Insert(1, "Jane Doe");
```

We start the list of with just one item, but then we insert two more items, first at the top of the list and then in the middle. The first parameter of the Insert method is the index where we want to insert the item. Be careful though - an exception will be thrown if you try to insert an item at index 3, if the list has less items!

1.5.2.4. Adding multiple items

Just like we have the Add and Insert methods for adding a single item, there are also corresponding methods for adding and inserting multiple items. They are called *AddRange()* and *InsertRange()* and accepts any type of collection which implements the IEnumerable interface as a parameter - this could be e.g. an array of items or another list, which items you want to add or insert into the current list.

As an example on the Range methods, let's do something fun - we combine the AddRange method with a collection initializer to add several new names to an existing list in a single statement:

```
listOfNames.AddRange(new string[]
                {
                     "Jenna Doe",
                     "Another Doe"
                });
```

We simply create an array of strings on-the-fly and immediately append its items to our list of names from the previous example.

1.5.2.5. Removing items

There are currently three methods at your disposal when you want to remove one or several items from a list: *Remove()*, *RemoveAt()* and *RemoveAll()*.

The **Remove()** method takes just one parameter: The item you want to remove. This is great for e.g. a list of strings or integers, because you can simply just write the item you want to remove. On the other hand, if you have a list of complex objects, you would have to find that object first, to have a reference you could pass to the Remove() method.¨Let's deal with that later - here's a very basic example on how you can remove a single item with the Remove() method:

```
List<string> listOfNames = new List<string>()
{
     "John Doe",
     "Jane Doe",
     "Joe Doe",
     "Another Doe"
};

listOfNames.Remove("Joe Doe");
```

The Remove() method simply iterates through the list until it finds the first instance of the object you specified for removal, and them removes it - it only removes one instance, and if you specify an item in the list which doesn't exist, no error is thrown. The method returns *true* if it was able to remove an item and *false* if it wasn't.

The **RemoveAt()** method takes advantage of the fact that the generic list is index based by allowing you to remove an item based on its index/position in the list. For instance, you could remove the first item from the list like this:

```
listOfNames.RemoveAt(0);
```

Or the last item in the list like this:

```
listOfNames.RemoveAt(listOfNames.Count - 1);
```

Again, this only removes a single item and this time, you should be careful when providing the index of the item to be removed - if you use an index that falls out of bounds (lower than 0 or higher than the amount of items) an exception will be thrown! So, unless you're sure of what you're doing, you might want to wrap the RemoveAt() method in a try-catch block for handling the exception (explained in detail elsewhere in this tutorial). The RemoveAt() method doesn't return anything, so you will have to check the amount of items in

the list before and after the call, to decide if it was successful - on the other hand, if you know that you have an index that exists in the list, which you should always make sure of, then you can always expect RemoveAt() to be successful.

The **RemoveAll()** is the most complex of the remove-methods, but definitely also the most powerful. It takes a delegate to a method as its parameter and this method then decides whether an item should be removed or not by returning true or false. This allows you to apply your own logic when removing items and it also allows you to remove more than one item at a time. Delegates will be treated elsewhere in this tutorial, because it's a big and complex subject, but I still want you to get a feel of how cool the RemoveAll method is, so here's an example:

```
List<string> listOfNames = new List<string>()
{
     "John Doe",
     "Jane Doe",
     "Joe Doe",
     "Another Doe"
};

listOfNames.RemoveAll(name =>
{
     if (name.StartsWith("J"))
          return true;
     else
          return false;
});
```

In this example, we use an anonymous method (again too complex to be explained here, but it will be treated in another chapter) as a parameter for the RemoveAll method. Our anonymous method is pretty simple - it will be called for each item in the list and have a parameter called *name*, which is of course the current item in the iteration. It looks at this name and if it starts with "J", *true* is returned - otherwise *false*. The RemoveAll() method uses this response (true or false) to decide if each item should be removed or not. In the end, this leaves our initial list with just one Doe member left: Another Doe.

1.5.2.6. Sorting List Items

So far, the items in the list we have worked with has just been used in the order in which they were added to the list. However, you may want to have the items sorted in a specific way, e.g. alphabetically in the case of our list-of-names. The List<T> has a Sort() method which we can use for this:

```
List<string> listOfNames = new List<string>()
{
     "John Doe",
```

```
        "Jane Doe",
        "Joe Doe",
        "Another Doe"
};
listOfNames.Sort();
foreach (string name in listOfNames)
        Console.WriteLine(name);
```

As you will see from the output, the items in the list have now been sorted alphabetically, and if you want it in descending order instead (from Z to A), simply call the **Reverse()** method after you perform the sort:

```
listOfNames.Sort();
listOfNames.Reverse();
```

So sorting a list was pretty easy, right? Well, it was mainly so easy because we have a list of strings and the .NET framework knows exactly how to compare two strings. If you have a list of numbers, .NET will, of course, know how to sort that as well. On the other hand, you might have a list of custom objects (since the List<T> can contain any object) which .NET doesn't have a chance of knowing how to compare. There are several solutions to this problem, e.g. implementing the IComparable interface or using LINQ (we'll look into both later in this tutorial), but as a quick-fix, we can also just provide a method for the Sort() method to call, to learn how two items stack up against each other, like this:

```
using System;
using System.Collections.Generic;

namespace ListSort
{
        class Program
        {
                static void Main(string[] args)
                {
                        List<User> listOfUsers = new List<User>()
                        {
                                new User() { Name = "John Doe", Age = 42 },
                                new User() { Name = "Jane Doe", Age = 39 },
                                new User() { Name = "Joe Doe", Age = 13 },
                        };
                        listOfUsers.Sort(CompareUsers);
                        foreach (User user in listOfUsers)
                                Console.WriteLine(user.Name + ": " + user.Age + "
years old");
                }
```

```
        public static int CompareUsers(User user1, User user2)
        {
            return user1.Age.CompareTo(user2.Age);
        }
    }

    public class User
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
}
```

This added quite a bit of code to our example, but it's actually not too complicated. If we start from the bottom, I have created a very simple User class, consisting of a name and an age. In the middle, I have declared a method called *CompareUsers()* - it takes two users as parameters and then returns an integer, which will indicate whether one item is "smaller", "the same" or "larger" (-1, 0 or 1). These values will be used by the Sort() method to move the items around so that the order of items will match what we want. In this case, I simply use the Age property for comparison, essentially leaving us with a list of users sorted by their age.

1.5.2.7. Summary

This article is one of the longer ones in this tutorial, but hopefully you learned a lot about lists, because the more programming you do, the more you will realize how important lists and dictionaries are. Speaking of dictionaries, we'll discuss them in the next article.

1.5.3. Dictionaries

Dictionaries in C# all implement the IDictionary interface. There are several Dictionary types, but the most commonly used is the generic Dictionary, often referred to as Dictionary<TKey, TValue> - it holds a type-specific key and a corresponding type-specific value. This is basically what separates a Dictionary from a List - the items of a list comes in a specific order and can be accessed by a numerical index, where items in a Dictionary is stored with a unique key, which can then be used to retrieve the item again.

We'll dig more into what dictionaries are all about, but first, let's look at a simple example, to show you what it's all about:

```
Dictionary<string, int> users = new Dictionary<string, int>();
users.Add("John Doe", 42);
users.Add("Jane Doe", 38);
users.Add("Joe Doe", 12);
users.Add("Jenna Doe", 12);

Console.WriteLine("John Doe is " + users["John Doe"] + " years old");
```

Meet the Doe family, who are once again the test-data for an example in this tutorial. Dad, Mom and the twins Joe and Jenna. Notice how I define the Dictionary to use a string as the key (in this case, the name) and an integer for the value (in this case an age). The cool thing is that this allows us to access an item from the dictionary using the key (name), instead of a numerical index. You can see that in action in the last line of the example - between a set of square brackets, we simply specify the key, to access the value (the age).

There are two important things to know here: First of all, the key has to be unique. In this case, where we use a string as a key, it means that you can't add two users with the exact same name. That makes pretty good sense, because when you can use the key to reference a single item, the key can only point to one value. On the other hand, this makes this example a bit too simple, because obviously you can't have a collection of users where duplicate names can't exist, but bear with me for now.

The other thing you will have to remember, when accessing values from a dictionary, is that the key has to exist in the collection, when you try to access its value. In other words, my example was only safe because I was in full control of what the list contained - in most situations, you should always check if the key exists, before trying to use it - like this:

```
string key = "John Doe";
if(users.ContainsKey(key))
     Console.WriteLine("John Doe is " + users[key] + " years old");
```

If you try to add a key which already exists, or if you try to access a value with a key which *doesn't* exists, .NET will throw an exception. This might sound obvious, but both are actually very commonly found in the exception logs around the world.

1.5.3.1. Working with the items

Accessing a single item can be very useful, but what if you want to loop through the collection and e.g. look for something? In that case, the first thing you should be aware of is that items in the list are obviously not just a simple object - instead, the Dictionary hold items of the type *KeyValuePair<TKey, TValue>*. The T's are the types you have used to declare the list - in this case a string and an integer. So looping through the collection with a *foreach* loop would look something like this:

```
Dictionary<string, int> users = new Dictionary<string, int>()
{
     { "John Doe", 42 },
     { "Jane Doe", 38 },
     { "Joe Doe", 12 },
     { "Jenna Doe", 12 }
};
foreach (KeyValuePair<string, int> user in users)
{
     Console.WriteLine(user.Key + " is " + user.Value + " years old");
}
```

The first thing you might notice is that I've used the opportunity to change the way we initialize the dictionary - instead of manually adding each item after instantiating the dictionary, we use the collection initializer syntax, as described in a previous article.

With that out of the way, we now have the same dictionary as before - let's loop through it. As you can see, I use a foreach loop and I have declared the type of item I expect as a KeyValuePair<string, int> - the exact same two types used to declare the dictionary. Now we can access both the key and the value (name and age) for each item, and we use that to generate a simple string of information about the person to the Console.

1.5.3.2. The order of items

The above example brings us to a very important point: Unlike a List, where the basic order of items is determined by numerical indexes, the order of items in a dictionary is non-deterministic - you simply can't rely on the items being in a certain order, not even when you manually add each item individually. Sure, when you run the above example, you will likely experience that the items are in the exact same order as we added them, but **that's not guaranteed**, and as soon as you have a dictionary with more items and start adding and removing items, the order will likely change.

The Dictionary class doesn't come with a Sort() method, because in the end, even if you sort it, the order might change as soon as you start working with it again. Instead, you can use the *OrderBy()* and *OrderByDescending()* methods from LINQ (more on that in another chapter) to get a sorted copy of the Dictionary. This also allows you to sort either by the key or by the value, which could be useful in our example, to get the users out in the order of their age:

```
Dictionary<string, int> users = new Dictionary<string, int>()
{
     { "John Doe", 42 },
     { "Jane Doe", 38 },
     { "Joe Doe", 12 },
     { "Jenna Doe", 12 }
};
foreach (KeyValuePair<string, int> user in users.OrderBy(user =>
user.Value))
{
     Console.WriteLine(user.Key + " is " + user.Value + " years old");
}
```

If you run the example, you will see that even though the dictionary is declared with the items in one order, we can easily get them out in another order, as we see fit. Just remember that you are responsible for ensuring that you get the items out in the order you want, because the .NET framework stores them exactly as it sees fit.

1.6. Data types

1.6.1. Introduction

In a previous article, we briefly discussed the concept of data types. Because C# is a strongly typed language, there are a LOT of data types that you want to know at some point. To get you started, we talked a bit about the most basic stuff, like strings, integers and booleans. Because data types is a very theoretic and slightly boring subject to start with, I wanted to show you a bit of programming magic before diving further into the subject.

However, now the time has come to learn more about data types. During the next articles, I will dig a lot deeper into each data type, so you know which data type to use for various tasks and how to work with them.

At this point, you may choose to read each article, skip some of them or skip them all entirely for now, because they are very theoretic and depending on how you like to learn stuff, it might still be a bit too theoretical for now. If so, don't worry about it - just move on and then return when you want to know more about one or several of the C# data types.

1.6.2. Booleans

The *bool* (boolean) data type is one of the simplest found in the .NET framework, because it only has two possible values: false or true. You can declare a boolean variable like this:

```
bool isAdult;
```

By default, the value of a bool is false, but you can of course change that - either when you declare the variable or later on:

```
bool isAdult = true;
```

Working with a boolean value usually means checking its current state and then reacting to it, e.g using an if-statement:

```
bool isAdult = true;
if (isAdult == true)
Console.WriteLine("An adult");
else
Console.WriteLine("A child");
```

But actually, it can be done a bit shorter, because when you check a boolean value, you can omit the true part - C# will understand this example in the exact same way:

```
bool isAdult = true;
if (isAdult)
Console.WriteLine("An adult");
else
Console.WriteLine("A child");
```

Whether you use the explicit approach or not is usually just a matter of taste. You can of course check for false as well - either by switching the keyword true with the keyword false, or by negating the variable with the exclamation mark operator:

```
bool isAdult = true;
if (!isAdult)
Console.WriteLine("NOT an adult");
else
Console.WriteLine("An adult");
```

The if-statement now basically asks "is the variable isAdult the opposite of true?", thanks to the exclamation mark which is also known as the logical negation operator.

1.6.2.1. Type conversion

It's not very often that you will find the need to convert a boolean into another type, because it's so simple. However, you may need to convert between an integer and a boolean because booleans are sometimes represented as either 0 (false) or 1 (true). For this purpose, I recommend the built-in Convert class, which can help you with most conversion tasks. Simply use the ToBoolean() method to convert an integer to a boolean, and the ToInt32() method if you want to go the other way. Here's an example:

```
int val = 1;
bool isAdult = Convert.ToBoolean(val);
Console.WriteLine("Bool: " + isAdult.ToString());
Console.WriteLine("Int: " + Convert.ToInt32(isAdult).ToString());
```

1.6.2.2. Summary

The *bool* data type can only have two values - false or true. It's easy to check with an if statement and is often the return type of many methods.

1.6.3. Integers

Computers and numbers go hand in hand, so when programming, you will very often find your self working with numbers in many forms. One of the most commonly used types in C# is the integer. The word "integer" is Latin for "whole", which makes sense, because an integer is a number with no fractional part - a whole number.

1.6.3.1. Int32 - the default integer

As we'll discuss in a short while, C# comes with many different integer types, but the one you will likely be using most of the time, is the Int32 - a 32 bit integer. It can be declared like this:

```
Int32 number;
```

However, since this really is the most used integer type in C#, it has a shortcut - you can just write "int" and C# will automatically know that you're talking about an Int32. You can of course also assign a value to it, in the same statement where you declare it:

```
int number = 42;
```

Notice the difference in casing though - *Int32* refers to a class, so it starts with an uppercase I, while *int* is a keyword, so it starts with a lowercase i.

The Int32 is a so-called signed integer, which can hold a number in the range of -2,147,483,648 to 2,147,483,647. You can verify this by accessing the Int32.MinValue and Int32.MaxValue constants. If you need larger numbers, or if you already know that the numbers you will be working with will not ever reach the limits of an integer, you can choose another type.

1.6.3.2. Integer types

As already mentioned, there are a lot of integer types in C#. In fact, too many to mention them all in this article, but here are the most common ones that you will likely run into from time to time:

- byte - an unsigned integer which can hold a number between 0 and 255.
- short - a 16 bit signed integer, which can hold a value between -32,768 and 32,767. Also known under its formal name Int16.
- long - a 64 bit integer, which can hold a value between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. Also known under its formal name Int64.

If you need to hold a bigger number than the *long* type allows, you can use a *ulong* instead - it's an unsigned integer, which can hold a value up to 18,446,744,073,709,551,615. On the other hand, because it's an unsigned integer, it doesn't allow negative values. Likewise, you can find unsigned versions of the other integer types which works the same way, like *uint* and *ushort*.

1.6.3.3. Math with integers

When working with integers and numbers in general, you might be looking to do some calculations. This comes very easy in most programming languages, including C#, where you can use all the most common mathematical operators as they are. For instance, you can do a calculation like this:

```
int a = 42;
int b = 8;

Console.WriteLine(a + b);
```

The number 50 will be printed in the console. Notice how you could deal with this calculation without doing anything special - just specify the variables and the mathematical plus operator and you're good to go.

The same goes if you want to use other mathematical operators, and you can of course use numbers which are not already defined as a variable in there as well:

```
int a = 42;
int b = 8;

Console.WriteLine(200 - ((a + b) * 2));
```

Notice the extra set of parentheses - just like in regular math, you will have to deal with the order of operations.

1.6.3.4. Integers and division

Now that we have added, subtracted and multiplied, let's talk a bit about division. Consider the following example:

```
int a = 10;
int b = 3;

Console.WriteLine(a / b);
```

You can probably calculate that in your head and so you know the result should be 3.33333333333. But if you run the example, you will see something strange - the result is instead 3. The reason is that if you do a division between two integers, C# will also return an integer, and since an integer can't contain any fractions, the result is just rounded (in this case down to 3). So if you want a result with fractions, you should either declare one of the variables as a floating point variable (more on those later) or typecast one of them on the fly:

```
int a = 10;
int b = 3;

Console.WriteLine((float)a / b);
```

1.6.3.5. Type conversion/casting

Let's talk more about how you can convert from and to an integer. First of all, C# allows for implicit and explicit conversions between various integer types. An example of an implicit conversion could look like this:

```
int a = 10;
long b = a;
```

In other words, if you have an integer, you may treat it as a long. This is possible because C# knows that the value of an integer can fit inside a long. On the other hand, the other way around might not be true - therefore, you will have to use an explicit conversion for that:

```
long a = 10;
int b = (int)a;
```

This is of course to make you aware that you are doing something that potentially could go wrong - if you have a *long* value which exceeds the capacity of a regular integer, you're in trouble!

1.6.3.6. Summary

Working with numbers, like integers and floating points, is something you will be doing a lot in pretty much any programming task. With your new knowledge about integers and how to work with them, it's time to move on to floating point values in the next article.

1.6.4. Floating points

Floating-point numbers are numbers that have fractional parts (usually expressed with a decimal point). You might wonder why there's isn't just a single data type for dealing with numbers (fractions or no fractions), but that's because it's a lot faster for the computer to deal with whole numbers than with numbers containing fractions. Therefore it makes sense to distinguish between them - if you know that you will only be dealing with whole numbers, pick an integer data type. Otherwise, use one of the floating point data types, as described in this article.

Using a floating point value is just as easy as using an integer, even though there are quite a few more concerns with floating point values, which we'll discuss later. For now, let's see what it looks like when declaring one of the most commonly used floating point data type: the *double*.

```
double number;
```

Just like an integer, you can of course assign a value to it at the same time as declaring it:

```
double number = 42.0;
```

The same goes for the float and decimal types, which will discuss in just a second, but here, the notation is slightly different:

```
double doubleVal = 42.0;
float floatVal = 42.0f;
decimal decimalVal = 42.0m;
```

Notice the "f" and "m" after the numbers - it tells the compiler that we are assigning a float and a decimal value. Without it, C# will interpret the numbers as double, which can't be automatically converted to either a float or decimal.

1.6.4.1. float, double or decimal?

Dealing with floating point values in programming has always caused a lot of questions and concerns. For instance, C# has at least three data types for dealing with non-whole/non-integer numbers:

- float (an alias for System.Single)
- double (an alias for System.Double)
- decimal (an alias for System.Decimal)

The underlying difference might be a bit difficult to understand, unless you have a lot of knowledge about how a computer works internally, so let's stick to the more practical stuff here.

In general, the difference between the float, double and decimal data types lies in the precision and therefore also in how much memory is used to hold them. The float is the least expensive one - it can represent a number with up to 7 digits. The double is more precise, with up to 16 digits, while the decimal is the most precise, with a whooping maximum of 29 digits.

You might wonder what you need all that precision for but the answer is "math stuff". The classic example to understand the difference is to divide 10 with 3. Most of us will do that in the head and say that the result is 3.33, but a lot of people also know that it's not entirely accurate. The real answer is 3.33 followed by an amount of extra 3's - how many, when doing this calculation with C#, is determined by the data type. Check out this example:

```
float a = 10f / 3;
Console.WriteLine(a);
double b = 10d / 3;
Console.WriteLine(b);
decimal c = 10m / 3;
Console.WriteLine(c);
```

I do the exact same calculation, but with different data types. The result will look like this:

```
a: 3.333333
b: 3.33333333333333
c: 3.3333333333333333333333333333
```

The difference is quite clear, but how much precision do you really need for most tasks?

1.6.4.2. How to choose

First of all, you to consider how many digits you need to store. A float can only contain 7 digits, so if you need a bigger number than that, you may want to go with a double or decimal instead.

Second of all, both the float and double will represent values as an approximation of the actual value - in other words, it might not be comparable down to the very last digit. That also means that once you start doing more and more calculations with these variables, they might not be as precise anymore, which basically means that two values which should check out as equals will suddenly not be equal anyway.

So, for situations where precision is the primary concern, you should go with the *decimal* type. A great example is for representing financial numbers (money) - you don't want to add 10 amounts in your bookkeeping, just to find out that the result is not what you expected. On the other hand, if performance is more important, you should go with a *float* (for small numbers) or a *double* (for larger numbers). A *decimal* is, thanks to its extra precision, much slower than a float - some tests shows that its up to 20 times slower!

1.6.4.3. Summary

When dealing with floating point values, you should use a *float* or a *double* data type when precision is less important than performance. On the other hand, if you want the maximum amount of precision and you are willing to accept a lower level of performance, you should go with the decimal data type - especially when dealing with financial numbers.

If you want to know more about the underlying differences between these data types, on a much more

detailed level, you should have a look at this very detailed article: What Every Computer Scientist Should Know About Floating-Point Arithmetic

1.6.5. The Char type

The System.Char data type is used to hold a single, unicode character. C# has an alias for it, called *char*, which you can use when declaring your char variables:

```
char ch;
```

And of course, you can immediately assign a value to it, if you want to. In C#, a char is surrounded by a set of single quotes:

```
char ch = 'H';
```

Since a string (which we'll discuss in the next chapter) is basically just a range of characters, .NET actually uses a list of char's to represent a string. That also means that you can pull out a single char from a string, or iterate over a string and get each character as a char data type:

```
string helloWorld = "Hello, world!";
foreach(char c in helloWorld)
{
     Console.WriteLine(c);
}
```

Under the helmet, a char is a numerical value, where each character has a specific number in the Unicode "alphabet". As of writing, there are more than 130.000 different Unicode characters, ranging from the Latin/western alphabet to historical scripts. So, in C#, you can very easily go from a char data type to its numeric representation, as illustrated by this slightly extended version of the previous example:

```
string helloWorld = "Hello, world!";
foreach(char c in helloWorld)
{
     Console.WriteLine(c + ": " + (int)c);
}
```

It will simply output the character, followed by the numerical representation, simply by casting the char to an integer. This also means that you can just as easily go the other way around: From a number to a character. But why would you do that? Well, there are a lot of characters which are not directly available on most keyboards, e.g. the copyright (©) character. You can instead use a Unicode lookup table, find the numeric version of the character you need, and then just make it into a char:

```
char ch = (char)169;
Console.WriteLine(ch);
```

1.6.5.1. Char helper methods

The Char class has some really cool helper methods, which can help you determine the type of char you're currently dealing with. This is very useful in a lot of situations, e.g. when validating input. Here's an example:

```
Console.WriteLine("Enter a single number:");
char ch = Console.ReadKey(true).KeyChar;
if (Char.IsDigit(ch))
     Console.WriteLine("Thank you!");
else
     Console.WriteLine("Wrong - please try again!");
```

I simply read the first key pressed by the user and then use the Char.IsDigit() method to see if it's a number or not. And there are many methods like this one, to check the type of the character. We can use this to do some very simple string validation:

```
Console.WriteLine("Write your name:");
string name = Console.ReadLine();
bool isValid = true;
for(int i = 0; i < name.Length; i++)
{
char ch = name[i];
if((i == 0) && ((!Char.IsLetter(ch)) || (!Char.IsUpper(ch))))
{
     Console.WriteLine("The first character has to be an uppercase
letter!");
     isValid = false;
     break;
}
if(Char.IsDigit(ch))
{
     Console.WriteLine("No digits allowed!");
     isValid = false;
     break;
}
}
if (isValid)
Console.WriteLine("Hello, " + name);
```

We simply loop through the name entered by the user and use various versions of the Is* methods to check that the input matches our simple requirements. And there are several other useful methods, like the *Char.IsLetterOrDigit()* method. For a full list, have a look at the documentation.

1.6.5.2. Summary

A *char* data type (alias for System.Char) is used to represent a single Unicode character. To represent more than one char, you use a string, which is basically just a list of chars. In the next article, we'll discuss strings.

1.6.6. Strings

A *string* is a piece of text. It's usually consists of 2 characters or more, since if it's just one, you should consider using a *char* instead. However, strings can be empty as well or even null, since it's a reference type. A string can be declared much like the other data types we have worked with already:

```
string s;
```

Or if you want to assign a value to it immediately:

```
string name = "John Doe";
```

Anything within a set of double-quotes in C# is considered a string, like in the example above - surrounding one or several characters between double quotes is simply how you tell the compiler that the stuff between it should be interpreted as a string instead of e.g. keywords and commands.

1.6.6.1. Strings are immutable

In C#, strings are immutable, which basically means that once they are created, they can't be changed. That's obviously not very practical in daily use, so the framework helps us - instead of having to keep declaring new strings to make changes, it simply creates a new string for you each time you change the existing one. This makes it a seamless process, but it can also make your code less efficient without you even noticing it. Here's an example to illustrate it:

```
string numbers = "";
for (int i = 0; i < 10000; i++)
     numbers += i.ToString();
```

In this case, we loop 10.000 times, each time appending the current index to the string. With the knowledge you just gained, you now know that instead of altering the existing string to include the latest number, a new string is created and then assigned to the old variable, leaving the old value to be cleaned up by the framework. And this happens 10.000 times! Instead, it's generally recommended to use a so-called StringBuilder if you know that you will be building a string over several operations:

```
StringBuilder numbers = new StringBuilder();
for (int i = 0; i < 10000; i++)
     numbers.Append(i);
Console.WriteLine(numbers.ToString());
```

1.6.6.2. Basic string operations

With all that about the strings being immutable said, you will still find your self manipulating and working with simple strings a lot, and don't worry about it - unless you are working inside a giant loop, it's likely no problem! Here are some of the basic stuff you can do with strings:

You can **concatenate** two or more strings simply by "adding" them (using the plus operator):

```
string name = "John" + " " + "Doe";
```

You can of course do the same with variables, where you mix double-quoted strings and variables which are either strings or which can be turned into strings (using the ToString() method found on all objects). However, a more "clean" way of doing that is with the **Format** method found on the String class:

```
string name = "John Doe";
int age = 42;
string userString = String.Format("{0} is {1} years old and lives in
{2}", name, age, "New York");
```

Notice how I use numbered placeholders ({0}, {1} etc.) and then supply the values for it as parameters to the method. Just remember that the indexes and amount of placeholders has to match the parameters you pass!

The **Length** property allows you to check the current length of a string, e.g. for validation purposes. The Length property is also very useful in combination with other properties and methods, e.g. the **Substring()** and the **IndexOf()** methods. The Substring method allows you to retrieve a part of the string, while the IndexOf method allows you to find the first index of a given char/string. Allow me to illustrate with an example:

```
string name = "John Doe";
int indexOfSpace = name.IndexOf(' ') + 1;
string lastName = name.Substring(indexOfSpace, name.Length -
indexOfSpace);
Console.WriteLine(lastName);
```

Quick explanation: We define a name, then we use the IndexOf() method to find the first position of a space character. Then we use the Substring() method to get everything after the space character by supplying a start position as well as a length.

Another cool helper method on the String class is the Replace() method. It allows you to take a string and then run a search/replace operation on it, like this:

```
string name = "John Doe";
Console.WriteLine(name.Replace("John", "Jane"));
```

The **Replace()** method is not strict at all - if the string you're searching for (the first parameter) is not present, then nothing will happen (no exceptions are thrown or anything like that). If it is present, it will be replaced with the second parameter. However, if you want to check before replacing, you can use the **Contains()** method:

```
string name = "John Doe";
if (name.Contains("John"))
     Console.WriteLine(name.Replace("John", "Jane"));
else
     Console.WriteLine("John was not found!");
```

Sometimes you want to know if a string starts or ends with a specific char or string. For that, the String class can help you with the **StartsWith()** and **EndsWith()** methods, which works just as the name indicates:

```
string name = "John Doe";
if ((name.StartsWith("John")) && (name.EndsWith("Doe")))
     Console.WriteLine("Hello, Mr. Doe!");
```

There are even more cool String methods, and there are definitely more ways of using them than illustrated by these short examples. If you want to know more about it, have a look at the MSDN documentation for the String class.

1.6.6.3. Verbatim Strings & Escaping

When defining a string, you will soon see that certain characters have special purposes. The most important example of this is the double quote it self, because since it's used to mark the start and end of a string to the compiler, how can you use it inside your string? The most simple answer to this is escaping, where you signal to the compiler that a special character should be treated literally instead of its usual function. Here's an example:

```
Console.WriteLine("What do you mean by \"Hello, world\" ??");
```

When executed, the result will look like this:

```
What do you mean by "Hello, world" ??
```

In other words, we simply use backslashes before a double quote, to indicate that this is NOT the end of the string, but instead, we actually want a double quote. So... what if we actually want a backslash and not just use it to escape another character? Well then you will have to escape that too - with another backslash:

```
Console.WriteLine("Right here \\ Right now");
```

Resulting in:

```
Right here \ Right now
```

The reason is that the backslash is not just used to escape double quotes - it's also used to prefix a range of characters to give them special meanings. For instance, \n is a new line, \t is a tab character and so on.

You can find a nice list of escape sequences here.

1.6.6.4. Verbatim strings

As an alternative to all this escaping, you can use a verbatim string. It's like a regular string declaration, but prefixed with the @ character, and inside it, all characters are treated literally:

```
Console.WriteLine(@"In a verbatim string \ everything is literal: \n & \t");
```

The output will look just like the input:

```
In a verbatim string \ everything is literal: \n & \t
```

There's just one exception to this rule: A double quote still has to be escaped, but that makes pretty good sense, because otherwise, how would the compiler know if you're trying to end the string or not? In a verbatim string however, a double quote is not escaped with a backslash but instead with another double quote, like this:

```
Console.WriteLine(@"What do you mean by ""Hello, world"" ??");
```

The result looks like this:

```
What do you mean by "Hello, world" ??
```

1.6.6.5. Summary

Strings are SUCH an important topic for all programmers - you will likely be doing some sort of string processing for much of the time you spent writing code. Fortunately, C# has all the tools you will need to do it, as I have tried to illustrate in this article.

1.6.7. Working with Dates & Time

C# comes with a really great struct for working with dates and time - it's called DateTime. It's not actually a data type, but I have included it in this chapter, because you will often find your self working with dates and/or time using the DateTime struct - sometimes just as much as you work with strings and numbers.

Let's start by instantiating a new DateTime object:

```
DateTime dt = new DateTime();
Console.WriteLine(dt.ToString());
```

The result is pretty boring though: 01-01-0001 00:00:00. This corresponds with the DateTime.MinValue field, but DateTime has more helping properties - the most interesting one is the DateTime.Now:

```
DateTime dt = DateTime.Now;
Console.WriteLine(dt.ToString());
```

This leaves us with the current date and time, which is often very relevant - we will see that in some of the upcoming examples. However, in a lot of situations, you are probably looking to represent a specific date and time - fortunately for us, the DateTime has several constructors to help us with just that. Here's an example:

```
DateTime dt = new DateTime(2042, 12, 24);
Console.WriteLine(dt.ToString());
```

The order of the parameters in this constructor is Year, Month, Day, so this will give us a DateTime object with a date representing Christmas Eve in the year 2042.

1.6.7.1. DateTime - with or without time?

But what about time? Well, if you don't specify one, it will default to 00:00:00, as you can see from our previous example. You can easily specify time as well though:

```
DateTime dt = new DateTime(2042, 12, 24, 18, 42, 0);
Console.WriteLine(dt.ToString());
```

Now the DateTime has time as well, in this case at 18:42. Notice that the hours should be specified as a number between 0 and 23, since we use the 24 hour clock for the input, but don't worry if you want it as a 12 hour clock with AM/PM - that's easily done in the output, as we'll discuss later.

The remaining question is: What to do if you're only interested in the date part? The obvious answer would be to use the Date class instead of the DateTime class, but actually, no such thing exists . Instead, the DateTime class has a Date property:

```
DateTime dt = new DateTime(2042, 12, 24, 18, 42, 0);
```

```
DateTime date = dt.Date;
Console.WriteLine(date.ToString());
```

As you will see however, the Date property is also a DateTime object, where the time part has been set to 00:00:00. This might be a bit confusing, but it also makes pretty good sense, because the DateTime should always just serve as a container, holding the data that you can output to the user in many ways and forms. We have already used the basic ToString() method, but there are many more ways to get the output of a DateTime object.

1.6.7.2. DateTime output

Getting the output of a DateTime is one of the most important aspects, but it can also be a complicated task - it really depends on your needs. So far, we have seen the very basic version of the output: The simple call to the ToString() method. This will output date and time based on the current culture of the application, which is, by default, based on the system settings. This means that if you just call the ToString() method, the date and time will be presented to the user in a way that makes sense to them no matter where in the world they are, because as you are probably aware, the format of time and especially the date varies a LOT from region to region.

If you are looking to get more control of the output, there are several ways to do that. The quickest way is to use one of the To* methods, e.g. the ToShortDateString() method:

```
Console.WriteLine(DateTime.Now.ToShortDateString());
```

This will output a short version of the date, omitting the time part completely. You can also use the ToLongDateString() method:

```
Console.WriteLine(DateTime.Now.ToLongDateString());
```

The output will vary, based on the region settings, but usually a long date includes a text representation of the month instead of a numeric representation.

If you're looking to get more control, perhaps because you want to adapt your date/time info to a specific region, you can use one of the overloads of the ToString() method, e.g. by specifying exactly which culture/region you are targeting:

```
var usCulture = new System.Globalization.CultureInfo("en-US");
Console.WriteLine(DateTime.Now.ToString(usCulture.DateTimeFormat));
```

The CultureInfo class allows you to get an entire box of info about a language, e.g. how they write dates and time (the CultureInfo class will be covered later on in this tutorial). We can then pass this information on to the DateTime class, to let it know that we want a visual representation of the date and time to match this language. But what if we want to be even more specific about the output?

1.6.7.3. Standard Date & Time Format Strings

To gain even more control of the output, you can use the standard date & time format strings provided by the .NET framework. It's one letter, used to represent a way to display the date and/or time. For the full list, I suggest that you have a look at the documentation, but for now, here's a couple of examples:

```
DateTime dt = new DateTime(2042, 12, 24, 18, 42, 0);

Console.WriteLine("Short date pattern (d): " + dt.ToString("d"));
Console.WriteLine("Long date pattern (D): " + dt.ToString("D"));
Console.WriteLine("Full date/time pattern (F): " + dt.ToString("F"));
Console.WriteLine("Year/month pattern (y): " + dt.ToString("y"));
```

The output will look like this:

```
Short date pattern (d):                    24-12-2042
Long date pattern (D):           24. december 2042
Full date/time pattern (F):      24. december 2042 18:42:00
Year/month pattern (y):                    december 2042
```

This gives you a bit more control of the output, but sometimes that's not enough - in those cases, you need custom format strings.

1.6.7.4. Custom Date & Time Format Strings

For full control, we have custom format strings. They are a combination of letters and special characters which you pass on to the ToString() method, to show the *exact* format you want for your date and/or time - this includes where you want the date/time parts, how you want them and what kind of separators you would like to use. Obviously this leaves you with a lot of options, so for the full list of available format specifiers, please check out the documentation, but let's see how it works straight away:

```
DateTime dt = new DateTime(2042, 12, 24, 18, 42, 0);

Console.WriteLine(dt.ToString("MM'/'dd yyyy"));
Console.WriteLine(dt.ToString("dd.MM.yyyy"));
Console.WriteLine(dt.ToString("MM.dd.yyyy HH:mm"));
Console.WriteLine(dt.ToString("dddd, MMMM (yyyy): HH:mm:ss"));
Console.WriteLine(dt.ToString("dddd @ hh:mm tt",
System.Globalization.CultureInfo.InvariantCulture));
```

The output will look something like this:

```
12/24 2042
24.12.2042
```

```
12.24.2042 18:42
onsdag, december (2042): 18:42:00
Wednesday @ 06:42 PM
```

In general, one or several *d's* will get you some version of the day, one or several *M's* will give you the month (lowercase *m* is for minutes) and a number of *y's* will give you the year. To fully understand what these format specifiers do, I still suggest that you have a look at the documentation, but hopefully the above example has given you a good idea of just how powerful the custom format strings are.

1.6.7.5. Parsing dates

So far, we have worked with dates defined directly in code, but you will probably quickly run into a situation where you need to work with a date specified by the user. This is a surprisingly complicated subject because there are so many ways of writing a date. The .NET framework can help you with this, because it supports all the cultures, as illustrated in previous examples, but you still need to help the user specify the date in the format you expect. After that, you can use the Parse() method to turn a user-specified string into a DateTime object, like this:

```
var usCulture = new System.Globalization.CultureInfo("en-US");
Console.WriteLine("Please specify a date. Format: " +
usCulture.DateTimeFormat.ShortDatePattern);
string dateString = Console.ReadLine();
DateTime userDate = DateTime.Parse(dateString,
usCulture.DateTimeFormat);
Console.WriteLine("Date entered (long date format):" +
userDate.ToLongDateString());
```

This example is actually pretty cool because it shows just how much the .NET framework can help us, as long as we have an instance of the CultureInfo class that we want to use (US English in this case - and don't worry, we will discuss the CultureInfo class in much greater detail later on). As soon as we have this, we use it for letting the user know which format we expect. Then we use it again in the 4th line, to actually parse the input. In the last line, we output the user-specified date, this time just using the format specified by the system.

However, be aware that the Parse() method is very strict - if the user doesn't enter the date in the expected format, it will throw an exception. For that reason, it's usually a good idea to use the **TryParse()** method instead - it does the exact same thing, but it allows you to check if the date could be parsed or not, and it doesn't throw an exception. Here's a revised version of the previous example:

```
var usCulture = new System.Globalization.CultureInfo("en-US");
Console.WriteLine("Please specify a date. Format: " +
usCulture.DateTimeFormat.ShortDatePattern);
string dateString = Console.ReadLine();
```

```
DateTime userDate;
if (DateTime.TryParse(dateString, usCulture.DateTimeFormat,
System.Globalization.DateTimeStyles.None, out userDate))
     Console.WriteLine("Valid date entered (long date format):" +
userDate.ToLongDateString());
else
     Console.WriteLine("Invalid date specified!");
```

1.6.7.6. Summary

Working with dates and time is very important but also very complex. Lucky for us, the .NET framework is there to help us in almost all situations with the brilliant DateTime class.

1.6.8. Nullable types

NULL literally means nothing - a variable that doesn't yet hold a value. At this point of the tutorial, you may have already dealt with variables which weren't initialized, whether on purpose or not - if so, you also know that you always have to make sure that a variable has a value before you try to access this value. If not, you will likely be met with a NullReferenceException.

NULL's are especially relevant when dealing with your own objects and strings. On the other hand, when dealing with numbers, e.g. an integer, you will see that it always has a (default) value. For an integer, the default/fallback value is 0 (zero), which is never to be confused with NULL - they are not at all the same, even though some non-statically typed programming languages might treat them the same. 0 is a number - NULL is *nothing*.

So, if you ever find your self wanting to have a number which represents a non-defined/NULL value, you might feel stuck, because you can't assign null to an integer variable. Unless of course that integer is defined as nullable - a special language construct created for situations just like this. You define a nullable variable by post-fixing the type with a questionmark. Here's an example to illustrate the difference:

```
int notNullable = null; // Will cause an error from the compiler

int? nullable = null; // Just fine - it's nullable!
```

This is valid for all so-called value types like integers, floats, bools and structs. Strings and objects on the other hand are reference types and can't be declared as nullable, because they are, by default, nullable.

1.6.8.1. Checking a nullable for null

So, now that you have defined a variable which might be null, checking whether that is the case is obviously important. You can do this in two ways: Just compare the variable to the null kewyord, like you would for any other types, or use the *HasValue* property which a nullable object inherits from the System.Nullable struct. Here's an example:

```
int? nullable = null;
if (nullable == null)
Console.WriteLine("It's a null!");
if (!nullable.HasValue)
Console.WriteLine("It's a null!");
```

The result is the same, so use whichever method you find most readable and easy to understand. Since our value can now be null, you should always check before using it - otherwise, you might run into an Exception.

1.6.8.2. Using the value of a nullable

From the System.Nullable, a nullable object also inherits the *Value* property. This can be used to retrieve

the actual value of the nullable object. However, for simple comparison operations, e.g. using the == and the != operators, C# lets you omit the Value property and just directly compare the nullable object. In other words, both of these examples accomplish the same thing:

```
int? nullable = 42;

if (nullable.Value == 42)
Console.WriteLine("It's 42!");

if (nullable == 42)
Console.WriteLine("It's 42!");
```

Nullable objects always come from a base data type, e.g. an integer as in the previous examples. While these data types may have a default value, the default value of a nullable is always null. That's why you have to check for null references before trying to use the value, as described previously. However, a nullable type inherits a very nice helper method which you can use: *GetValueOrDefault()*. It will return the value of the nullable object, unless it's null, in which case it will return the default value of the underlying type. So, for a nullable integer, it would return 0, for a nullable boolean it would return *false* and so on. This allows you to handle both the checking and the retrieval of the value in a single statement:

```
if ((nullable.HasValue) && (nullable.Value == 42))
     Console.WriteLine("It's 42!");

if(nullable.GetValueOrDefault() == 42)
     Console.WriteLine("It's 42!");
```

As you can see, the latter example is shorter and easier to read, while accomplishing the same thing.

1.6.8.3. Summary

Value types in C#, like integers and booleans, always have a default value. If you want to work around this behavior, e.g. because you need to differentiate between 0 and null, or because you need to know whether the user has actively selected "*false*" for something or if the variable just holds the default value of *false*, then you can make the variable nullable by postfixing the type with a ? (question mark).

1.6.9. Implicitly typed variables (the var keyword)

As of C# version 3.0, you can use the *var* keyword on the left side of a variable declaration, instead of explicitly declaring the type of the variable. This is only possible inside a method - not on the class level, where you always have to specify the type. Let's see how it looks when you use the var keyword:

```
int age = 42; // Explicitly typed variable

var name = "John Doe"; // Implicitly typed variable
```

Two variables - the one is declared with the int type, while the other one is declared with the var keyword, instead of specifying it as a string. Notice that I'm assigning a value in both cases, because while this is not required when declaring a type, it IS required when using the var keyword. The reason is that the C# compiler will infer the type from the right part of the statement - it simply looks at what you're trying to assign to the variable during the compile and then changes the var keyword into the appropriate type.

This also means that there is no overhead when using the var keyword - it's just as fast during runtime as an explicitly declared variable, because that's essentially what it is when the .NET framework executes your code.

Our first example is very trivial - there's not a whole lot of time saved in writing "var" instead of "string". However, sometimes you will be declaring much more complex types, either as a new variable or as the local result from a function (user-defined or from the framework). There's a chance to save a fair amount of keystrokes in an example like this one:

```
Dictionary<int, List<string>> dict1 = new Dictionary<int, List<string>
>();

var dict2 = new Dictionary<int, List<string>>();
```

The result will be exactly the same, but you definitely save some keystrokes in the latter example, and since you specify the type on the right side in both cases, there's pretty much no loss in readability of your code when using the *var* keyword in an example like this.

You can also use the var keyword when declaring a local variable as the result of a method call:

var s = DateTime.Now.ToString();

Again, it's faster and it's still pretty clear what happens and which type the variable will hold (a string). That might not be in a case like this though:

```
var myVar = SomeMethodWithANameWhichDoesntIndicateTheReturnType();
```

In a situation like this, it's not clear at all what the variable will contain and you might be sacrificing the readability of your code. You might want to consider use an explicitly typed variable here.

1.6.9.1. The var Keyword & Anonymous Types

So far, the examples we have seen with the var keyword has been mostly from the "syntactical sugar" department - they are nice to have and shorter to type, but not really a requirement. However, when working with anonymous types (more on them later), it makes sense to declare your objects with the var keyword:

```
var myObj = new
{
     Name = "John Doe",
     Age = 42
};
Console.WriteLine(myObj.Name);
```

In this case, you really need the var keyword to later access the fields of the anonymous type, as shown in this example.

1.6.9.2. Summary

The *var* keyword allows you to declare a local variable (inside a method or a loop) without explicitly stating the type - instead, the C# compiler infers the type from the right part of the declaration. This can be really convenient in a lot of situations, but it might also make your code slightly less readable. You can use it, or stick to explicitly declaring the variable types - whatever you prefer, but you need it for anonymous types, as already explained.

1.6.10. The dynamic Type

In C# version 4.0, Microsoft introduced a new type: the dynamic. It's actually a static type, but unlike other static types, the (potential) members of a dynamic object is not checked by the compiler.This will give you some of the advantages of the dynamically/weakly typed languages, while maintaining the advantages of a strongly typed language in all other cases.

Declaring a dynamic is like declaring any other type - simply use the *dynamic* keyword instead of the data type:

```
dynamic d1;
dynamic d2 = "A string";
dynamic d3 = 42;
```

Now you have three different objects - the first is really nothing at this point (null), while the second is a string and the third an integer. The interpreter will automatically decide this at runtime, based on what you assign to the variables. That also means that the compiler won't check up on what you do with these variables, as illustrated by this next example:

```
dynamic d1;
dynamic d2 = "A string";
dynamic d3 = 42;
Console.WriteLine(d2.Length);
Console.WriteLine(d3.Length);
```

Strings have a length property, to decide how long the string is, but an integer doesn't. In this case, I try to use this property on both variables, and the compiler won't complain about it - if these variables had been declared as a string and an integer, the compiler would have intercepted. But since the types are dynamic, it just compiles and run. However, as soon as the interpreter reaches the last line, an exception will be thrown, because obviously you can't access a property which doesn't exist.

This also shows the danger of the dynamic types - you have to be aware of what you're doing and make sure that you spell every property and method call properly, because the compiler won't double check your work, basically allowing you to publish code which fails every time.

1.6.10.1. A dynamic object

The dynamic keyword can, of course, be used for more complex types than integers and strings. A great example of this is when it's used to hold an anonymous object, like this:

```
dynamic user = new
{
     Name = "John Doe",
     Age = 42
```

```
};
Console.WriteLine(user.Name + " is " + user.Age + " years old");
```

This allows you to create an object without defining a class for it first. The *dynamic* keyword can be used to hold it, but so can the *var* keyword, which might be better suited in a lot of situations.

You might think that since the type is dynamic, you can just add properties to it later on, like this:

```
dynamic user = new
{
     Name = "John Doe",
     Age = 42
};
user.HomeTown = "New York";
```

Since no compile-time checking is done on a dynamic type, this code actually runs, because the compiler doesn't actually validate the existence of the *HomeTown* property, but as soon as the last line is reached, an exception will be thrown. While a dynamic type IS very dynamic, it doesn't allow you to dynamically add new properties to it. For that, you can use an ExpandoObject, which we'll discuss in the next article.

1.6.10.2. Changing type

A variable declared as a dynamic is not typeless at all. Instead, C# will treat it internally as whatever kind of object you have assigned to it. In our very first example, you will see how we can declare a variable as a dynamic, assign either a string or an integer to it, and then start using properties belonging to these types. You can, however, easily change the type of a dynamic variable - just assign a new value to it. Here's an example:

```
dynamic user = new
{
     Name = "John Doe",
     Age = 42
};
Console.WriteLine(user.GetType() + ": " + user.Name + " is " + user.Age
+ " years old");
user = "Jane Doe";
Console.WriteLine(user.GetType() + ": String.Length = " + user.Length);
```

Notice how we use the dynamic *user* variable: First it holds an anonymous object and then we assign a simple string to it instead. We validate it by using the GetType() method found on all C# objects, and in both cases, we access properties found on the type we currently have (first Name/Age properties of the anonymous object, then the Length property found on string objects).

1.6.10.3. Summary

The dynamic keyword allows you to use variables which are not type-specific - instead, they act as the kind of data which they hold. You can easily assign a new value to a dynamic variable, and if the type of value you are assigning is not the same as it currently holds, the dynamic variable will simply change type automatically.

Dynamic variables are not checked by the compiler, allowing you to access properties which may or may not be present. This makes them very flexible, but it also makes your applications a lot more vulnerable to errors. Especially for this reason, you may want to restrict the usage of the dynamic keywords to situations where you can't accomplish what you want without using it. Examples of when it makes good sense to use the dynamic keyword are for COM interop and when interacting with data formats like JSON and XML.

1.6.11. The ExpandoObject

As we saw in a previous article, we can use the dynamic keyword to hold an object where we get to define the properties, without having to define the class first. What we *can't* do with a dynamic object is dynamically adding properties to it after the object has been initialized. If you need this specific ability, C# comes with a solution for you: The *ExpandoObject*. Let's jump straight to an example, so you can see how easy it is to use:

```
dynamic user = new System.Dynamic.ExpandoObject();
user.Name = "John Doe";
user.Age = 42;
user.HomeTown = "New York";
Console.WriteLine(user.Name + " is " + user.Age + " years old and lives
in " + user.HomeTown);
```

You will notice that I declare the object with the dynamic type, even though I instantiate an ExpandoObject. The reason is that if the object was declared as an ExpandoObject, the compiler would check it and immediately complain about the lack of the properties that we invent for it (Name, Age and so on). We can prevent this by declaring it as a dynamic, which as we learned in the previous article, will prevent the compiler from checking for the existence of the properties we use.

A cool thing is that an ExpandoObject can of course have properties which are also ExpandoObject's, allowing you to make complex types on the fly, like in this example:

```
dynamic user = new System.Dynamic.ExpandoObject();
user.Name = "John Doe";
user.Age = 42;

user.HomeTown = new System.Dynamic.ExpandoObject();
user.HomeTown.Name = "New York";
user.HomeTown.ZipCode = 10001;

Console.WriteLine(user.Name + " is " + user.Age + " years old and lives
in " + user.HomeTown.Name + " [" + user.HomeTown.ZipCode + "]");
```

I simply change the HomeTown property from a string to an ExpandoObject, and then add properties to it, in this case the name and the zip code of the city. But it doesn't have to end with that - we can even add methods to the object, again on the fly, using some pretty sophisticated tricks:

```
user.DescribeUser = (Func<String>)(() => {
          return user.Name + " is " + user.Age + " years old and lives
in " + user.HomeTown.Name + " [" + user.HomeTown.ZipCode + "]";
});
```

```
Console.WriteLine(user.DescribeUser());
```

Pretty neat stuff, but what IS an ExpandoObject really? It implements several interesting interfaces, but one of them is IDictionary<string, object> - this means that underneath all the syntactical sugar, your ExpandoObject is basically just a dictionary which holds object values based on string keys. That also means that iterating over an ExpandoObject is just as easy as iterating through a regular Dictionary. This is very practical:

```
dynamic user = new System.Dynamic.ExpandoObject();
user.Name = "John Doe";
user.Age = 42;

foreach (KeyValuePair<string, object> kvp in user)
{
     Console.WriteLine(kvp.Key + ": " + kvp.Value);
}
```

1.6.11.1. Summary

The ExpandoObject type allows you to define objects on the fly and then add properties to it whenever you want to. Since it's basically a dynamic type, it inherits the same advantages and disadvantages as we discussed in the previous article. As a bonus though, the ExpandoObject implements the INotifyPropertyChanged interface, which you'll definitely appreciate if you're using e.g. WPF for your application.

1.6.12. Anonymous Types

So far, we have learned that objects come from a class. A class is declared with a number of fields, properties and/or methods, and then you can create an instance of this class as an object. However, with the introduction of anonymous types, you no longer have to declare a class before creating an object. Don't worry, classes are not dead at all, because anonymous types (or objects) comes with several limitations, but for some situations, they're really great!

An anonymous type is initialized using the *new* operator, in combination with an object initializer - in that regard, it's very much like instantiating a class, only you leave out the class name. Also, since there's no class behind the object, you must use the *var* keyword when retrieving the reference to your object. This might sound complicated, but the following example should demonstrate to you that it's not complicated at all:

```
var user = new
{
          Name = "John Doe",
          Age = 42
};
Console.WriteLine(user.Name + " - " + user.Age + " years old");
```

That's it - we now have an object with information (name and age) about a user. Anonymous types are great for a lot of situations, especially when you just need to return something quickly, with more complexity than just a string or a number. Anonymous types allow you to make up properties on the fly, without worrying about declaring a class first and then alter this class when your need changes. But as mentioned, there are several limitations that you need to be aware of when you consider using an anonymous type over defining a class:

- Unlike a real class, an anonymous type can't have fields or methods - only properties
- Once the object has been initialized, you can't add new properties to it
- Properties are readonly - as soon as the object has been initialized, you can't change their values

But with that said, anonymous types are still extremely practical for a lot of tasks. A common usage scenario is when you have a complex object (from a defined class) and you need to simplify it, e.g. because you have to keep the object as small as possible to send it to a browser or perhaps because the full object has sensitive information that you don't want to expose to the consumer. Anonymous types are great for this, as illustrated in this next example:

```
using System;
using System.IO;

namespace AnonymousTypes
{
     class Program
```

```
    {
        static void Main(string[] args)
        {
            string pathOfExe =
System.Reflection.Assembly.GetEntryAssembly().Location;
            FileInfo fileInfo = new FileInfo(pathOfExe);
            var simpleFileInfo = new
            {
                Filename = fileInfo.Name,
                FileSize = fileInfo.Length
            };
            Console.WriteLine("File name: " + simpleFileInfo.Filename
+ ". Size: " + simpleFileInfo.FileSize + " bytes");
        }
    }
}
```

The first line will simply get us the filename of the currently executing file, that is, our own application. We use it to create an instance of the FileInfo class, which will then contain a LOT of information about this specific file. We don't want all that information, so we create a simplified version of it with an anonymous type, using the information from the original FileInfo instance. In the last line, we use this information for some basic output.

You have probably noticed that we give a name to each of the properties defined in our anonymous type (*Filename* and *FileSize*) - that makes pretty good sense, since we want to access them later on. However, when basing our object on information from an existing object, we can actually leave out our own name and let the compiler just use the name of the property we assign to it, like this:

```
var simpleFileInfo = new
{
    fileInfo.Name,
    fileInfo.Length
};
```

Now, we have two properties called *Name* and *Length*, instead of *Filename* and *FileSize*. A pretty nice convenience, if you don't care about the names or perhaps more likely: If you actually prefer the exact same names as the original object.

1.6.12.1. Summary

Anonymous types allow you to quickly instantiate an object with one or several properties. These properties are readonly, and you can't add more properties at a later point.

1.7. Operators

1.7.1. Introduction

Operators are basically symbols, either single characters or a specific combination of several characters, which specify operations (math, obviously, but also indexing, function calls and so on) to perform in an expression. If you have read the previous articles in this tutorial, you have already seen several operators, like the assignment operator (a single equal sign) used when you learned about variables and so on.

So far, we haven't really talked about these operators, because they have been mostly self-explanatory. However, that is not the case for all operators in C# (and believe me, there are a lot), and even for those who makes sense when seeing it, it's nice to get a proper name for it and perhaps even learn a bit more about how it works internally.

In the next several articles, we'll do just that: we will look into many of the C# operators, both the simple and the not-so-simple, talk about what they do and show you how you can use them. You will find some very valuable information in these articles, which will give you a better understanding of what goes on when your code is interpreted, but it will also be a bit more theoretical than what you have previously read. If it turns out to be too much for you right now, then feel free to skip some of it and return later.

1.7.2. Comparison operators

C# has a lot of operators and several of them are used to compare values. This is obviously a very common task when programming - to check how two or more values relate to each other. In this chapter, we will look into these operators used for comparing values - you probably already know some of them, but have a look anyway and see if you learn something new!

1.7.2.1. The equality operator: ==

Comparing two values can obviously be done in many ways, but to check if they are in fact equals, you can use the double-equal-sign (==) operator. Let me show you how:

```
int val1 = 42;
int val2 = 42;
if(val1 == val2)
          Console.WriteLine(val1 + " is equal to " + val2);
```

Notice how I use not one but two equal signs, right after eachother - this is important, because if you just use a single equal sign, I will be assigning a value instead of comparing it.

1.7.2.2. The NOT equal operator: !=

Sometimes you need to check if two values are non-equal instead of equal. C# has an operator for that - you just replace the first equal sign with an exclamation mark. Here's the example from before, but using the not equal operator instead:

```
int val1 = 42;
int val2 = 43;
if(val1 != val2)
          Console.WriteLine(val1 + " is NOT equal to " + val2);
```

1.7.2.3. Smaller- and bigger-than operators: < and >

Especially when comparing numbers, you will often find your self wanting to see whether one value is bigger or smaller than the other. We will use the greater-than and less-than symbols, like this:

```
int val1 = 42;
int val2 = 43;
if(val1 > val2)
          Console.WriteLine(val1 + " is larger than " + val2);
else
{
          if(val1 < val2)
                    Console.WriteLine(val1 + " is smaller than " +
val2);
```

```
        else
                Console.WriteLine(val1 + " is equal to " + val2);
}
```

1.7.2.4. Smaller/bigger than or equal to: <= and >=

In the above example, we check if a value is smaller or bigger than another, but sometimes, instead of just smaller/bigger, you want to see if something is smaller-than-or-equal-to or bigger-than-or-equal-to. In that case, just put an equal sign after the smaller/bigger-than operator, like this:

```
int val1 = 42;
if (val1 >= 42)
        Console.WriteLine("val1 is larger than or equal to 42");
if (val1 <= 42)
        Console.WriteLine("val1 is smaller than or equal to 42");
```

1.7.2.5. Summary

Comparing stuff is such an essential task in programming, but fortunately, C# has a wide selection of operators to help you, as shown in this article. However, sometimes comparing two objects are not as simple as comparing two numbers - for that, C# allows you to write your own, custom methods for doing stuff like comparison. We will look into that in the article about operator overloading.

1.7.3. Increment/decrement operators

When dealing with values, especially the numeric kind, you will often find the need to add or subtract by 1. This is of course very easy to do – you just take the value, add 1 and then assign it back to the original variable. Here's an example:

```
int userAge = 41;
userAge = userAge + 1;
```

The *userAge* variable now contains the value of 42 instead of 41 – easy! You can of course do the same thing when you want to subtract by 1:

```
userAge = userAge - 1;
```

1.7.3.1. Postfix increment/decrement

However, C# has a much shorter operator for taking care of this job: The increment/decrement operator. It simply consists of two plus or two minus signs right after each other. Here's the first example, rewritten to use this operator:

```
int userAge = 41;
userAge++;
Console.WriteLine("Age of user: " + userAge);
```

The result is exactly the same, but look how much shorter it is! And we can make it even shorter, because this operator can of course be used inside of an existing statement:

```
int userAge = 41;
Console.WriteLine("Age of user: " + userAge++);
```

But wait a minute – when testing this, you will see that the userAge is still printed as 41. Why? Because we're using the postfix version, which is evaluated AFTER the surrounding statement. In other words, the line is printed before the value is incremented. This will be the behavior you want in some cases, but it's not optimal for this specific usecase. What you need is the prefix increment operator.

1.7.3.2. Prefix increment/decrement

This operator looks exactly like the postfix version, but instead of appearing after the variable, it will appear before, instructing the interpreter to evaluate it before evaluating the surrounding statement:

```
int userAge = 41;
Console.WriteLine("Age of user: " + ++userAge);
```

This will have the desired effect, where the userAge is incremented before the line is printed. Now, it may look a bit confusing with three + signs, but the first is for the string concatenation, while the last two are the

actual operator. If you prefer it, you can wrap the increment statement with parentheses to make it clear what belongs together:

```
Console.WriteLine("Age of user: " + (++userAge));
```

That's all up to you to decide – it works exactly the same!

And of course, you can use the decrement operator in exactly the same way as illustrated above. Here's the postfix version:

```
int userAge = 41;
Console.WriteLine("Age of user: " + (userAge--));
```

And the prefix version:

```
int userAge = 41;
Console.WriteLine("Age of user: " + (--userAge));
```

1.7.3.3. Summary

So, these were the increment and decrement operators. They are great for a very specific use case: When you want to increment or decrement a value by 1. In the next chapter, we will look into the addition assignment operator, which looks a lot like this one, but has a slightly broader use.

As with several other C# operators, this one falls under the term "syntactical sugar" - the same task can be accomplished without this specific operator, but with it, your code becomes shorter. Whether it becomes more readable is very subjective – some people like them, while others feel that their code becomes easier to read and understand without them. It's really all to you!

1.7.4. Addition assignment operators

We have previously looked into the increment/decrement operator which simply adds or subtracts 1 to/from a value, but in most cases, you probably want more flexibility in how much you're looking to add or subtract. For this, we can use the addition assignment operator. Without it, adding to a value looks like this:

```
int userAge = 38;
userAge = userAge + 4;
```

Not really very long or complicated, but since we're always looking into ways we can make our code even shorter, we can use the addition assignment operator instead:

```
int userAge = 38;
userAge += 4;
```

Notice the difference: Instead of re-stating the name of the value, to indicate that we're looking to add something to it and assign it back to the very same variable, we say it all with the operator += (plus-equals). You can of course do the same when you want to subtract a value:

```
int userAge = 42;
userAge -= 4;
```

This probably seems obvious, but what might be less obvious is that you can do it with multiplication and division and it's just as easy:

```
int userAge = 42;

userAge *= 2;
Console.WriteLine("User age: " + userAge);

userAge /= 2;
Console.WriteLine("User age: " + userAge);

Console.ReadKey();
```

1.7.4.1. Adding to strings

So far, we have worked exclusively with numbers, but the addition assignment operator can be used for e.g. strings as well, in exactly the same way. Let me illustrate it with a similar set of examples – first without the addition assignment operator:

```
string userName = "John";
userName = userName + " Doe";
```

Sure it's short and concise, but with the addition assigment operator, we can make it even shorter:

```
string userName = "John";
userName += " Doe";
```

Nice and easy!

1.7.4.2. Summary

As with several other C# operators, this one falls under the term "syntactical sugar" - the same task can be accomplished without this specific operator, but with it, your code becomes shorter. Whether it becomes more readable is very subjective – some people like them, while others feel that their code becomes easier to read and understand without them. It's really all to you!

1.7.5. The NULL coalescing operator

The ?? operator is also called the "null-coalescing operator" because it allows you to check for a NULL value and assign a fallback value in one line of code. This might seem trivial to do without this operator, but consider the following example:

```
string userSuppliedName = null;

if (userSuppliedName == null)
Console.WriteLine("Hello, Anonymous!");
else
Console.WriteLine("Hello," + userSuppliedName);
```

You should think of the variable *userSuppliedName* as something that comes from the user, e.g. from a dialog or a data file - something that could result in the value being NULL. We must deal with this by checking the value before using it, in this case for displaying the name to the user.

In the above example, we use the classical if-else approach, but with the null-coalescing operator, we can do it much shorter, in a single line:

```
Console.WriteLine("Hello, " + (userSuppliedName ?? "Anonymous"));
```

In a single statement, we ask the interpreter to use the *userSuppliedName* variable if it has a value – if not, we supply a fallback value, in this case the name "Anonymous". Short and simple!

1.7.5.1. Summary

As with all "syntactical sugar", like the null-coalescing operator, it's always a compromise between keeping the code short and readable. Some find that these operators make it harder to read and navigate the code, while others love how short and simple it is. In the end, it's all up to you!

1.7.6. The String Interpolation Operator

Elsewhere in this tutorial, you'll find a lengthy description of the string data type, because dealing with text is such an important task. In this article we'll focus on a special operator for working with strings, allowing you to do quite a few extra tricks when declaring strings. It's called string interpolation (introduced in C# 6.0) and it will allow you to place special markers in your string, which the interpreter will later replace with the relevant values. It works much like the String.Format() method we already discussed, but the syntax is more readable and convenient.

String Interpolation can be achieved very easily - just prefix the string with a $ character. As always, we'll jump straight to an example:

```
string name = "John Doe";
int age = 42;

Console.WriteLine(name + " is " + age + " years old");
Console.WriteLine($"{name} is {age} years old");
```

In the last two lines, we do the exact same thing: first in the traditional way, where we concatenate a string using our two variables, and then using string interpolation. Notice how it allows us to inject variables directly into the string by surrounding it with curly braces. And even though this happens inside a string, the compiler will actually check the variables you're trying to inject!

Also, the variables you use doesn't have to be simple types - you can use properties from complex objects as well, just like you could if you were doing regular string concatenation. Here's an example:

```
var user = new
{
     Name = "John Doe",
     Age = 42
};
Console.WriteLine($"{user.Name} is {user.Age} years old");
```

1.7.6.1. Format Strings

By default, the variables you include will be turned into the required string representation by calling the ToString() method on them. But sometimes you might be looking for a bit more control over how each variable is displayed, and fortunately, this is very simple to do, thanks to format strings. Simply put a colon (:) after the variable and then enter a format string to be used, like in this example:

```
double daysSinceMillenium = (DateTime.Now - new DateTime(2000, 1,
1)).TotalDays;
Console.WriteLine($"Today is {DateTime.Now:d} and
{daysSinceMillenium:N2} days have passed since the last millennium!");
```

The result will look something like this, depending on your system settings for dates and numbers:

```
Today is Friday, June 29, 2018 and 6,754.49 days have passed since the
last millenium!
```

Notice how the date is represented in the long date format (as specified with the "D" format string) and the number is nicely formatted with thousand separators and two decimal numbers, thanks to the "N2" numeric format string.

If you want even more control, you can change from the default format strings to custom format strings, e.g. for dates:

```
Console.WriteLine($"Today is {DateTime.Now:yyyy-MM-dd}");
```

There are many more ways of formatting a DateTime, that you may use in an interpolated string. For all the options, please consult the documentation.

1.7.6.2. Beyond variables

Now we have included variables and even properties of objects in our strings and we have seen how easy we can format the resulting strings. But string interpolation goes beyond that, because you can actually have entire C# expressions inside your strings, as long as they result in something that can be appended to a string. A great example of that is the classic "with or without the ending s"-situation, where you have to create a string that's either "1 year old" or "2 year**s** old". You can do that directly with string interpolation, simply by using the ternary operator inside the string:

```
string name = "John Doe";
int age = 42;

Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old");
```

Notice how I have inserted a simple if-then-else statement into the string, inside a set of parentheses. We can do that because the result of the expression is a string - either an empty one or an "s". You can do math as well:

```
Console.WriteLine($"5 + 5 * 2 = {((5 + 5) * 2)}");
```

Again, as long as the result of the expression can be turned into a string, you can use it within your interpolated string.

1.7.6.3. String Interpolation & Escaping

You won't have worked with string interpolation for long before a very obvious question comes to mind: How can I include characters which have a specific meaning, e.g. curly braces, and have them treated literally? The answer to that is usually escaping - the act of prefixing/postfixing the character with another

character, to negate the special purpose. This is true for string interpolation as well, where curly braces should be written twice if you want them to be treated literally, like this:

```
Console.WriteLine($"Insert {name} between curly braces: {{name here}}");
```

The result:

```
Insert John Doe between curly braces: {name here}
```

1.7.6.4. Summary

If you have ever looked at classical string concatenation and thought of it as too clumsy, you'll love string interpolation! It was introduced in C# version 6.0 and allows you to write text which includes variables in a very natural and easy-to-read way. They're also really cool in allowing you to integrate expressions directly into the strings, but be careful - at a certain point, your interpolated string might be so advanced that it becomes harder to read than the alternatives.

1.8. LINQ

1.8.1. Introduction

LINQ, short for **Language-Integrated Query**, is a technology built into the .NET framework which will allow you to easily query and manipulate data across various sources. In other words, you can work with data in the same way, no matter if the source is a simple list, a dictionary, an XML file or a database table. LINQ comes in two syntax flavors: The Query and the Method syntax. We will discuss both of them in depth in the next article, but to get you excited about LINQ, allow me to give you a quick demonstration of how elegant LINQ can query a simple data source:

```
var names = new List<string>()
{
"John Doe",
"Jane Doe",
"Jenna Doe",
"Joe Doe"
};

// Get the names which are 8 characters or less, using LINQ
var shortNames = from name in names where name.Length <= 8 orderby
name.Length select name;

foreach (var name in shortNames)
Console.WriteLine(name);
```

In just one line, using the LINQ **query syntax**, I can ask for all names in the list which are 8 characters (or less) long and have them sorted by length! That's just ONE line of code and it only shows a small fraction of what LINQ is capable of! In the next articles, I will go through all the amazing possibilities you get when using LINQ, but first, we should talk a bit about how and when a LINQ query is executed.

1.8.1.1. Deferred execution

Most programmers are used to seeing the code being executed line after line, so it might come as a surprise to you that a LINQ query is not executed as soon as the line with the query part is hit. Instead, LINQ is executed as soon as you start using the data, e.g. when you iterate over it or call a method like ToList() or Count(). That also means that you can build a query over several lines, where you perform multiple operations, but the data is not actually fetched until you need it. Allow me to illustrate with an extended version of our previous example, where we switch to the **method syntax** instead of the previous query syntax:

```
var names = new List<string>()
```

```
{
     "John Doe",
     "Jane Doe",
     "Jenna Doe",
     "Joe Doe"
};

// Get the names which are 8 characters or less, using LINQ
var shortNames = names.Where(name => name.Length <= 8);
// Order it by length
shortNames = shortNames.OrderBy(name => name.Length);
// Add a name to the original list
names.Add("Zoe Doe");

// Iterate over it - the query has not actually been executed yet!
// It will be as soon as we hit the foreach loop though!
foreach (var name in shortNames)
     Console.WriteLine(name);
```

I have added some comments to the code, so it should be easy to follow. Also, notice that I have added a bit of proof in there: After doing the query, I add a name to the original list. Now, if the query had already been executed, this would of course not make it into the result, but it does, because it's added after the query but BEFORE we actually use the data, which happens in the *foreach* loop.

This is not of great importance when querying a list with 4 or 5 names, but imagine that instead, we had a list of 10 million entries or perhaps a remote database somewhere. In that case, it's very important that we don't execute multiple queries when we can just execute one query.

1.8.1.2. Summary

LINQ allows you to simply and natively query various data sources using the same syntax - either the query syntax or the method syntax, which we'll discuss in the next article.

1.8.2. LINQ: Query Syntax vs. Method syntax

In the previous article, we got a quick glimpse of how both LINQ syntaxes look, but let's just get them on the table again right next to each other, so you are sure of what we'll be discussing in this article:

```
var listOfNames = new List<string>()
{
     "John Doe",
     "Jane Doe",
     "Jenna Doe",
     "Joe Doe"
};

// Query syntax
var qNames = from name in listOfNames where name.Length <= 8 select name
;
// Method syntax
var mNames = listOfNames.Where(name => name.Length <= 8);
```

While the two lines will accomplish the exact same thing, the syntactical difference is quite clear: The **query syntax** looks more like other query languages, e.g. SQL and less like a regular C# statement. The **method syntax**, on the other hand, looks very much like the rest of the C# code you have seen in this tutorial.

So, the biggest difference is definitely just the syntax and therefore, you will see both variations used when reading C# articles and finding answers to questions you might have. It seems that the query syntax is slightly more popular because some people find it easier to read than the method syntax - these might be people who are used to expressing data retrieval operations in SQL anyway. On the other hand, if you have never used a query language like SQL before, but you have some C# experience, then you might find the method syntax easier to read. Also, for at least a couple of operations, like counting the items in the result of the query, you will have to use the method syntax.

1.8.2.1. Lambda Expressions

In the method syntax example, you might notice something that you haven't seen before, as a parameter to the Where() method. It looks like this:

```
name => name.Length <= 8
```

This is called a Lambda Expression and while it's not a specific piece of LINQ functionality, it's used A LOT in the world of LINQ. Therefore you will also see it a lot in the following articles, and while I will go into greater details about Lambda Expressions in another article, this is a good time to get a very basic understanding of how they work.

It's actually quite simple. A Lambda Expression has a left side and a right side, divided by the **=>** operator (not to be confused with the "greater than or equal to"-operator, which looks like this: >=).

On the **left side**, we have the input parameter(s) - there can be several, but in our example, there's just one, which is the *name* variable. In this case, it comes from the Where() method, from iterating over our list of names and we get to pick the name. I chose "name", because that's what you'll find in a list of names, but you could also call it "x", "z", "foobar" or anything else.

On the **right side**, we have the expression/statement part. This is where we generate the expected result - in this case, a boolean value telling the Where() method whether to include the name in question or not. We answer that with an expression, saying that we want to include the name if it has a length of 8 (or less) characters. We can (but are not required to) use the input (left side) to determine this. Again, we refer to it as "name", but we could have called it "x" instead - in that case, the statement would have looked like this:

```
var mNames = listOfNames.Where(x => x.Length <= 8);
```

Lambda Expressions are about more than just LINQ queries, but as mentioned, they are important when using LINQ - you will see that in the next articles. If you want to know more about Lambda Expressions in general, have a look elsewhere in this tutorial where the topic will be covered.

1.8.2.2. Summary

LINQ comes in two syntactical flavors: The Query and the Method syntax. They can do almost the same, but while the query syntax is almost a new language within C#, the method syntax looks just like regular C# method calls. In this tutorial, **we will mostly be using the method syntax**, when we discuss all the possible operations available with LINQ in the next articles.

1.8.3. Filtering data: the Where() method

One of the most basic (but also most powerful) operations you can perform on a set of data is to filter some of it out. We already saw a glimpse of what you can do with the **Where()** method in the LINQ introduction article, but in this article, we'll dig a bit deeper. We already discussed how many LINQ methods can use a Lambda Expression to performs its task and the Where() method is one of them - it will supply each item as the input and then you will supply the logic that decides whether or not the item is included (return true) or excluded (return false) from the final result. Here's a basic example:

```
List<int> numbers = new List<int>()
{
     1, 2, 4, 8, 16, 32
};
var smallNumbers = numbers.Where(n => n < 10);
foreach (var n in smallNumbers)
     Console.WriteLine(n);
```

In this example, each number is checked against our expression, which will return true if the number is smaller than 10 and false if it's 10 or higher. As a result, we get a version of the original list, where we have only included numbers below 10, which is then outputted to the console.

But the expression doesn't have to be as simple as that - we can easily add more requirements to it, just like if it was a regular if-statement:

```
List<int> numbers = new List<int>()
{
     1, 2, 4, 8, 16, 32
};
var smallNumbers = numbers.Where(n => n > 1 && n != 4 && n < 10);
foreach (var n in smallNumbers)
     Console.WriteLine(n);
```

We specify that the number has to be greater than 1, but not the specific number 4, and smaller than 10.

You can of course also use various methods call in your expression - as long as the final result is a boolean value, so that the Where() method knows whether you want the item in question included or not, you're good to go. Here's an example:

```
List<int> numbers = new List<int>()
{
     1, 2, 4, 7, 8, 16, 29, 32, 64, 128
};
List<int> excludedNumbers = new List<int>()
```

```
{
     7, 29
};
var validNumbers = numbers.Where(n => !excludedNumbers.Contains(n));
foreach (var n in validNumbers)
     Console.WriteLine(n);
```

In this example, we declare a second list of numbers - sort of a black-list of numbers which we don't want to be included! In the Where() method, we use the **Contains()** method on the black-list, to decide whether a number can be included in the final list of numbers or not.

And of course, it works for more complex objects than numbers and strings, and it's still very easy to use. Just have a look at this example, where we use objects with user information instead of numbers, and use the Where() method to get a list of users with names starting with the letter "J", at the age of 39 or less:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqWhere2
{
     class Program
     {
          static void Main(string[] args)
          {
               List<User> listOfUsers = new List<User>()
               {
                    new User() { Name = "John Doe", Age = 42 },
                    new User() { Name = "Jane Doe", Age = 34 },
                    new User() { Name = "Joe Doe", Age = 8 },
                    new User() { Name = "Another Doe", Age = 15 },
               };

               var filteredUsers = listOfUsers.Where(user =>
user.Name.StartsWith("J") && user.Age < 40);
               foreach (User user in filteredUsers)
                    Console.WriteLine(user.Name + ": " + user.Age);
          }


          class User
```

```
        {
            public string Name { get; set; }
            public int Age { get; set; }

        }
    }
}
```

And just for comparison, here's what the where operation would look like if we had used the **query based syntax** instead of the method based:

```
// Method syntax
var filteredUsers = listOfUsers.Where(user => user.Name.StartsWith("J")
&& user.Age < 40);

// Query syntax
var filteredUsersQ = from user in listOfUsers where user.Name.StartsWith
("J") && user.Age < 40 select user;
```

1.8.3.1. Chaining multiple Where() methods

We discussed this briefly in the introduction to LINQ: The actual result of a LINQ expression is not realized until you actually need the data, e.g. when you loop over it, count it or iterate over it (as we do in our examples). That also means that you chain multiple Where() methods together, if you feel that's easier to read - in very complex expressions, it definitely can be! Here's a modified version of our previous example:

```
List<int> numbers = new List<int>()
{
    1, 2, 4, 8, 16, 32
};
var smallNumbers = numbers.Where(n => n > 1).Where(n => n != 4).Where(n
=> n < 10);
foreach (var n in smallNumbers)
    Console.WriteLine(n);
```

The result is exactly the same, and while the first version might not have been complex enough to justify the split into multiple Where() method calls, you will likely run into situations where it makes good sense to do so. I want to emphasize that this doesn't cost extra, in terms of performance, because the actual "where" operation(s) are not carried out until the part where we loop over the result - by then, the compiler and interpreter will have optimized your query to be as fast as possible, no matter how you wrote it.

1.8.3.2. Summary

With the **Where()** method, you can easily filter out unwanted items from your data source, to create a subset of the original data. Remember that it is indeed a new set of data you get - the original data source will be untouched unless you specifically override the original variable.

1.8.4. Sorting data: the OrderBy() & ThenBy() methods

So, now that we have learned through the previous article how to get data from the data source with LINQ and filter it with the Where() method, the next step could be to sort the data. We have used lists of objects, either numeric or based on e.g. a User class, so the order we got the items in was the same order that they were added to the list. However, as we have previously talked about, your data source for LINQ operations might as well be an XML document or a database. Therefore, the ability to properly sort the data, once we have the data we need, is crucial. Fortunately for us, LINQ has several easy-to-use methods for sorting data - let's try a basic example first:

```
List<int> numbers = new List<int>()
{
     1, 7, 2, 61, 14
};
List<int> sortedNumbers = numbers.OrderBy(number => number).ToList();
foreach (int number in sortedNumbers)
     Console.WriteLine(number);
```

That was easy, right? Just call the **OrderBy()** method and supply the object or the member of the object to sort by, and a sorted list will be returned. And of course you can do it just as easy with strings, as we'll see in the next example, but let's get the items in descending (from largest to smallest/from Z to A) order:

```
List<string> cityNames = new List<string>()
{
     "Amsterdam", "Berlin", "London", "New York"
};
List<string> sortedCityNames = cityNames.OrderByDescending(city =>
city).ToList();
foreach (string cityName in sortedCityNames)
     Console.WriteLine(cityName);
```

We do the exact same thing as before, except that we use the **OrderByDescending()** method instead of the OrderBy() method. But of course you can get your list of integers and strings sorted easily - that's a piece of cake! However, thanks to LINQ, it's pretty much just as easy to sort more complex objects. Here's an example:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqOrder2
{
     class Program
```

```
    {
        static void Main(string[] args)
        {
            List<User> listOfUsers = new List<User>()
            {
                new User() { Name = "John Doe", Mail =
"john@doe.com", Age = 42 },
                new User() { Name = "Jane Doe", Mail =
"jane@doe.com", Age = 34 },
                new User() { Name = "Joe Doe", Mail = "joe@doe.com"
, Age = 8 },
                new User() { Name = "Another Doe", Mail =
"another@doe.com", Age = 15 },
            };

            List<User> usersByAge = listOfUsers.OrderBy(user =>
user.Age).ToList();
            foreach (User user in usersByAge)
                Console.WriteLine(user.Name + ": " + user.Age + "
years");
        }

        class User
        {
            public string Name { get; set; }
            public string Mail { get; set; }
            public int Age { get; set; }
        }
    }
}
```

This is of course a more complete example, with the User class included and the list of users initialized, but as you can see, the actual sorting is still very short and precise: When calling the OrderBy() method, we still just supply a parameter name and then use this parameter to access the Age property of the User objects. The result is a perfectly age-sorted list! But what if we want to order by more than one property?

1.8.4.1. ThenBy() and ThenByDescending()

In the example above, we sorted the list of users by their age, but what if there were several users of the same age? A pretty common scenario, even in our tiny example - just imagine that Jane and John are the same age and their kids are twins. In that case, it would be practical to control the order even after the data source had been arranged by age. For that, we can use the **ThenBy()** and **ThenByDescending()** methods.

They do just what the name indicates: Control the order after the initial sort. We can use that to get the list of users sorted, first by age and then alphabetically by name:

```
List<User> listOfUsers = new List<User>()
{
     new User() { Name = "John Doe", Mail = "john@doe.com", Age = 42 },
     new User() { Name = "Jane Doe", Mail = "jane@doe.com", Age = 42 },
     new User() { Name = "Joe Doe", Mail = "joe@doe.com", Age = 8 },
     new User() { Name = "Jenna Doe", Mail = "another@doe.com", Age = 8
},
};

List<User> sortedUsers = listOfUsers.OrderBy(user =>
user.Age).ThenBy(user => user.Name).ToList();
foreach (User user in sortedUsers)
     Console.WriteLine(user.Name + ": " + user.Age + " years");
```

Quite simple but very effective! You can even chain multiple ThenBy() method calls, in case your data is more complex than the data from our test case. And of course, you can mix and match the OrderBy(), OrderByDescending(), ThenBy() and ThenByDescending() methods in whatever way you need:

```
List<User> sortedUsers = listOfUsers.OrderBy(user =>
user.Age).ThenByDescending(user => user.Name).ToList();
foreach (User user in sortedUsers)
Console.WriteLine(user.Name + ": " + user.Age + " years");
```

We're mostly using the method-based syntax of LINQ in this tutorial, but as always, I will take one of the examples in the article and show you how it would look with the query syntax - here's the latest example, including a **LINQ query syntax** version:

```
// Method syntax
List<User> sortedUsers = listOfUsers.OrderBy(user =>
user.Age).ThenByDescending(user => user.Name).ToList();

// Query syntax
List<User> sortedUsersQ = (from user in listOfUsers orderby user.Age
ascending, user.Name descending select user).ToList();
```

As you can see, the syntax is slightly different - the direction (ascending or descending) is specified directly after the field to order by (ascending is actually implicit, but I included it to show you the difference). Also, there's no "ThenBy" - instead, you just separate multiple sort instructions with a comma. Of course, in the end, both queries will give you the same result.

1.8.4.2. Summary

Using the **OrderBy()** and **ThenBy()** methods (as well as their "descending" counterparts), you can easily get your data sorted just the way you want it. And remember, just like any other LINQ method, the actual data source is not manipulated - instead, you get a sorted copy of the original data source, which you can work with.

1.8.5. Limiting data: the Take() & Skip() methods

So far, in this LINQ chapter of the tutorial, we have discovered several ways of working with data sources using LINQ. Now the time has come to see how we can limit the amount of data to work with. This is especially useful when using a database as a data source, because it often involves huge amounts of rows, which are resource consuming to fetch.

The methods we will discuss in this article are called **Take()** and **Skip()**, and in combination, they are great for doing stuff like pagination on a website. In fact, they are often used together, but they can of course be used alone as well. The **Take()** method will get you X number of items from the data source, while **Skip()** will allow you to ignore the first X items. A simple example could look like this:

```
List<string> names = new List<string>()
{
     "John Doe",
     "Jane Doe",
     "Joe Doe",
     "Jenna Doe",
};
var middleNames = names.Skip(1).Take(2).ToList();
foreach (var name in middleNames)
     Console.WriteLine(name);
```

We create a simple list of names and then, for the output, we skip the first name (Skip(1)) and then we take the next two names (Take(2)), basically leaving us with only the two middle names of the list.

1.8.5.1. Basic pagination with Skip() and Take()

As you can see, both the Take() and Skip() methods are very simple to use, but they are more interesting to demonstrate with more data than we have previously used, so I have taken the liberty of creating a slightly more complex example, which will better demonstrate how these methods can help you. First, here's the code for it:

```
using System;
using System.Globalization;
using System.Linq;
using System.Xml.Linq;

namespace LinqTakeSkip1
{
     class Program
     {
          static void Main(string[] args)
```

```
          {
               CultureInfo usCulture = new CultureInfo("en-US");
               XDocument xDoc = XDocument.Load(
"http://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml");
               var cubeNodes = xDoc.Descendants().Where(n =>
n.Name.LocalName == "Cube" && n.Attribute("currency") != null).ToList();
               var currencyRateItems = cubeNodes.Select(node => new
               {
                    Currency = node.Attribute("currency").Value,
                    Rate = double.Parse(node.Attribute("rate").Value,
usCulture)
               });


               int pageSize = 5, pageCounter = 0;
               var pageItems = currencyRateItems.Take(pageSize);
               while(pageItems.Count() > 0)
               {
                    foreach (var item in pageItems)
                         Console.WriteLine(item.Currency + ": " +
item.Rate.ToString("N2", usCulture));
                    Console.WriteLine("Press any key to get the next
items...");
                    Console.ReadKey();
                    pageCounter++;
                    // Here's where we use the Skip() and Take()
methods!
                    pageItems = currencyRateItems.Skip(pageSize *
pageCounter).Take(pageSize);
               }
               Console.WriteLine("Done!");

          }
     }
}
```

That's quite a bit of code, but let's run through it. In the first half, we parse a publically available XML source of currency exchange rates. This gives me the opportunity to briefly show you LINQ to XML, which is a very cool part of the LINQ eco-system. We will discuss LINQ to XML in another article, but for now, just know that we use it to pull the important nodes out of the XML source and put it into anonymous objects, consisting of the name and current exchange rate of the currency, which we'll use later on.

We now have a nice data source of currency information in the currencyRateItems variable. In the last half of the example, we use this source to do some simple pagination - we simply pull out 5 entries and then asks the user to press a key to get the next 5 (or however many there are left in the source). We do that by extracting the first 5 entries and then we use a while loop to continuously pull out the next 5 entries, until the source is empty. Getting the *next* 5 entries is done with a combination of **Skip()** and **Take()** - the entire basis of this article.

1.8.5.2. Summary

The Skip() and Take() methods are very simple to use, but nevertheless very useful in a lot of situations. As mentioned, they are often used together, but especially the Take() method can just as well be used alone.

1.8.6. Data transformations: the Select() method

So far in this LINQ chapter of the tutorial, we've worked with simple data sources, e.g. a list of integers, strings or simple objects like the User class. We will continue to do so because it's very practical when showing you examples of the various LINQ methods, but keep in mind that with LINQ, the data source might as well be a complex XML document or a huge database.

In this article, we will talk about the Select() method, which allows you to take the data from the data source and shape it into something else. That might be more obviously useful with larger and more complex data sources, like the ones I mention above, but bear with me anyway, while I try to show you how the Select() method works and what you can do with it. As usual, we'll go straight to an example:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqSelect1
{
class Program
{
     static void Main(string[] args)
     {
          List<User> listOfUsers = new List<User>()
          {
               new User() { Name = "John Doe", Age = 42 },
               new User() { Name = "Jane Doe", Age = 34 },
               new User() { Name = "Joe Doe", Age = 8 },
               new User() { Name = "Another Doe", Age = 15 },
          };

          List<string> names = listOfUsers.Select(user =>
user.Name).ToList();

          foreach (string name in names)
               Console.WriteLine(name);
     }

     class User
     {
          public string Name { get; set; }
          public int Age { get; set; }
     }
```

```
}
}
```

Notice how I can take a list of objects, in this case of the type User, and then use the Select() method to shape this list into a new type of list. In this example, I turn the list of objects into a list of strings containing the names of the users. This is extremely practical in so many situations, where you need your data to look differently, or when you only need a subset of it. Of course, it also works the other way around - you can easily create a list of User objects from a list of names (you will have to manually add their age later though):

```
List<User> listOfUsers = new List<User>()
{
     new User() { Name = "John Doe", Age = 42 },
     new User() { Name = "Jane Doe", Age = 34 },
     new User() { Name = "Joe Doe", Age = 8 },
     new User() { Name = "Another Doe", Age = 15 },
};

List<string> names = listOfUsers.Select(user => user.Name).ToList();

List<User> users = names.Select(name => new User { Name = name
}).ToList();

foreach (User user in users)
     Console.WriteLine(user.Name);
```

Notice how I can create new objects with the Select() method - it's an extremely powerful feature, which allows you to do pretty much anything with your data, on the fly! A common usage for this functionality is to make less complex versions of an object, e.g. to return over a web service as JSON or XML. Imagine that you have a User class with a LOT of properties (Birthday, Gender, Mail, Address, Country etc.), but you only want to return a limited set of these properties - that might also make sense, security-wise, to make sure that you don't return common User properties like Username and Password. Here's a simplified example - just imagine a much more complex User class:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqSelect2
{
     class Program
     {
```

```
        static void Main(string[] args)
        {
            List<User> listOfUsers = new List<User>()
            {
                new User() { Name = "John Doe", Mail =
"john@doe.com", Age = 42 },
                new User() { Name = "Jane Doe", Mail =
"jane@doe.com", Age = 34 },
                new User() { Name = "Joe Doe", Mail = "joe@doe.com"
, Age = 8 },
                new User() { Name = "Another Doe", Mail =
"another@doe.com", Age = 15 },
            };

            var simpleUsers = listOfUsers.Select(user => new
            {
                Name = user.Name,
                Age = user.Age
            });
            foreach (var user in simpleUsers)
                Console.WriteLine(user.Name);
        }

        class User
        {
            public string Name { get; set; }
            public int Age { get; set; }
            public string Mail { get; set; }
        }
    }
}
```

In this example, we use the Select() method to return a list of anonymous types, which contains only a subset of the properties found on the data source. We then do the classical output of these objects to the console, but you should just imagine us returning them to a website or somewhere else, where it's important to keep the result as simple as possible, without revealing any secrets like passwords etc.

As we already discussed, I will mostly be using the method syntax of LINQ in these examples, but of course the operations we perform in this example can be expressed with the LINQ query syntax as well:

```
// Method syntax
```

```
var simpleUsers = listOfUsers.Select(user => new
{
     Name = user.Name,
     Age = user.Age
});

// Query syntax
var simpleUsersQ = (from user in listOfUsers
                         select new
                         {
                              Name = user.Name,
                              Age = user.Age
                         }).ToList();

foreach (var user in simpleUsersQ)
     Console.WriteLine(user.Name);
```

I hope this gives you a clearer idea of which syntax you prefer, but the result is of course the same!

1.8.6.1. Summary

The Select() method allows you to transform and shape data from your data source into new variants, for instance by selecting only a single property or returning objects which only contains a subset of the properties found on the source object.

1.8.7. Grouping data: the GroupBy() Method

So far, we have worked mostly with lists of data. We have sorted it, limited it and shaped it into new objects, but one important operation is still missing: Grouping of data. When you group data, you take a list of something and then divide it into several groups, based on one or several properties. Just imagine that we have a data source like this one:

```
var users = new List<User>()
{
     new User { Name = "John Doe", Age = 42, HomeCountry = "USA" },
     new User { Name = "Jane Doe", Age = 38, HomeCountry = "USA" },
     new User { Name = "Joe Doe", Age = 19, HomeCountry = "Germany" },
     new User { Name = "Jenna Doe", Age = 19, HomeCountry = "Germany" },
     new User { Name = "James Doe", Age = 8, HomeCountry = "USA" },
};
```

A flat list of user objects, but it could be interesting to group these users on e.g. their home country or their age. With LINQ, this is very easy, even though the use of the GroupBy() method can be a bit confusing in the beginning. Let's have a look at how it works:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqGroup
{
     class Program
     {
          static void Main(string[] args)
          {
               var users = new List<User>()
               {
                    new User { Name = "John Doe", Age = 42, HomeCountry
= "USA" },
                    new User { Name = "Jane Doe", Age = 38, HomeCountry
= "USA" },
                    new User { Name = "Joe Doe", Age = 19, HomeCountry =
"Germany" },
                    new User { Name = "Jenna Doe", Age = 19, HomeCountry
= "Germany" },
                    new User { Name = "James Doe", Age = 8, HomeCountry
= "USA" },
```

```
            };
            var usersGroupedByCountry = users.GroupBy(user =>
user.HomeCountry);
            foreach(var group in usersGroupedByCountry)
            {
                Console.WriteLine("Users from " + group.Key + ":");

                foreach(var user in group)
                    Console.WriteLine("* " + user.Name);
            }
        }

        public class User
        {
            public string Name { get; set; }

            public int Age { get; set; }

            public string HomeCountry { get; set; }
        }
    }
}
```

The resulting output will look something like this:

```
Users from USA:
* John Doe
* Jane Doe
* James Doe
Users from Germany:
* Joe Doe
* Jenna Doe
```

The example might seem a bit long, but as you'll soon realize, most of it is just preparing the data source. Remember that all of the data could just as well come from an XML document or a database - it's just easier to demonstrate with an object data source that you can use as-is.

The interesting part is when we create the **usersGroupedByCountry** variable. We make it by calling the **GroupBy()** method on our data source, supplying the parameter we want to group the data by. In this case, I want the users grouped by their home country, so that's the property I supply to the GroupBy() method. The result is an object with a Key property, which holds the value of the property we grouped by (HomeCountry, in this case), as well as all the objects which belong to the group. We use that in the next

lines to iterate over the groups we just created, and for each group, we print the Key (HomeCountry) and then iterate and print all the User objects from the group.

1.8.7.1. Custom group keys

As you can see, grouping by an existing property is easy-peasy, but as you might have come to know by now, the LINQ methods are very flexible. It's just as simple to create your own, custom groups, based on whatever you like - an example of that could be the following, where we create groups based on the first two letters of the user's name:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqGroup
{
     class Program
     {
          static void Main(string[] args)
          {
               var users = new List<User>()
               {
                    new User { Name = "John Doe", Age = 42, HomeCountry
= "USA" },
                    new User { Name = "Jane Doe", Age = 38, HomeCountry
= "USA" },
                    new User { Name = "Joe Doe", Age = 19, HomeCountry =
"Germany" },
                    new User { Name = "Jenna Doe", Age = 19, HomeCountry
= "Germany" },
                    new User { Name = "James Doe", Age = 8, HomeCountry
= "USA" },
               };
               var usersGroupedByFirstLetters = users.GroupBy(user =>
user.Name.Substring(0, 2));
               foreach(var group in usersGroupedByFirstLetters)
               {
                    Console.WriteLine("Users starting with " + group
.Key + ":");
                    foreach(var user in group)
                         Console.WriteLine("* " + user.Name);
               }
```

```
        }

        public class User
        {
            public string Name { get; set; }

            public int Age { get; set; }

            public string HomeCountry { get; set; }
        }
    }
}
```

We simply call the Substring() method on the name, to get the two first letters, and then LINQ creates the groups of users based on it. The result will look something like this:

```
Users starting with Jo:
* John Doe
* Joe Doe
Users starting with Ja:
* Jane Doe
* James Doe
Users starting with Je:
* Jenna Doe
```

So as you can see, we are free to call a method inside the GroupBy() method - in fact, we can do pretty much whatever we want in there, as long as we return something that LINQ can use to group the items. We can even create a method which returns a new piece of information about the item and then use it to create a group, as we do in the next example:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqGroup
{
    class Program
    {
        static void Main(string[] args)
        {
            var users = new List<User>()
```

```
                {
                    new User { Name = "John Doe", Age = 42, HomeCountry
= "USA" },
                    new User { Name = "Jane Doe", Age = 38, HomeCountry
= "USA" },
                    new User { Name = "Joe Doe", Age = 19, HomeCountry =
"Germany" },
                    new User { Name = "Jenna Doe", Age = 19, HomeCountry
= "Germany" },
                    new User { Name = "James Doe", Age = 8, HomeCountry
= "USA" },
                };
                var usersGroupedByAgeGroup = users.GroupBy(user =>
user.GetAgeGroup());
                foreach(var group in usersGroupedByAgeGroup)
                {
                    Console.WriteLine(group.Key + ":");
                    foreach(var user in group)
                        Console.WriteLine("* " + user.Name + " [" +
user.Age + " years]");
                }
            }

            public class User
            {
                public string Name { get; set; }

                public int Age { get; set; }

                public string HomeCountry { get; set; }

                public string GetAgeGroup()
                {
                    if (this.Age < 13)
                        return "Children";
                    if (this.Age < 20)
                        return "Teenagers";
                    return "Adults";
                }
            }
```

```
    }
}
```

Notice how I have implemented a **GetAgeGroup()** method on the User class. It returns a string that defines the age group of the user and we simply call it in the GroupBy() method to use it as a group key. The result will look like this:

```
Adults:
* John Doe [42 years]
* Jane Doe [38 years]
Teenagers:
* Joe Doe [19 years]
* Jenna Doe [19 years]
Children:
* James Doe [8 years]
```

I choose to implement the GetAgeGroup() method on the User class, because it might be useful in other places, but sometimes you just need a quick piece of logic to create the groups, not to be re-used elsewhere. In those situations, you are free to supply the logic directly to the GroupBy() method, as a lambda expression, like this:

```
var usersGroupedByAgeGroup = users.GroupBy(user =>
                {
                    if (user.Age < 13)
                        return "Children";
                    if (user.Age < 20)
                        return "Teenagers";
                    return "Adults";
                });
```

The result is of course the same!

1.8.7.2. Grouping by a composite key

So far, the keys of our groups have just been a single value, e.g. a property or the result of a method call. However, you are free to create your own keys which contain several values - these are called composite keys. A usage example could be if we wanted to group our users based on both their home country and their age, like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace LinqGroup2
{
     class Program
     {
          static void Main(string[] args)
          {
               var users = new List<User>()
               {
                    new User { Name = "John Doe", Age = 42, HomeCountry
= "USA" },
                    new User { Name = "Jane Doe", Age = 38, HomeCountry
= "USA" },
                    new User { Name = "Joe Doe", Age = 19, HomeCountry =
"Germany" },
                    new User { Name = "Jenna Doe", Age = 19, HomeCountry
= "Germany" },
                    new User { Name = "James Doe", Age = 8, HomeCountry
= "USA" },
               };

               var usersGroupedByCountryAndAge = users.GroupBy(user =>
new { user.HomeCountry, user.Age });
               foreach(var group in usersGroupedByCountryAndAge)
               {
                    Console.WriteLine("Users from " + group
.Key.HomeCountry + " at the age of " + group.Key.Age + ":");
                    foreach (var user in group)
                         Console.WriteLine("* " + user.Name + " [" +
user.Age + " years]");
               }
          }

          public class User
          {
               public string Name { get; set; }

               public int Age { get; set; }

               public string HomeCountry { get; set; }
```

```
            }
        }
}
```

Notice the syntax we use in the GroupBy() method - instead of supplying a single property, we create a new anonymous object, which contains the HomeCountry and the Age properties. LINQ will now create groups based on these two properties and attach the anonymous object to the Key property of the group. We are free to use both properties when we iterate over the groups, as you can see. The result will look something like this:

```
Users from USA at the age of 42:
* John Doe [42 years]
Users from USA at the age of 38:
* Jane Doe [38 years]
Users from Germany at the age of 19:
* Joe Doe [19 years]
* Jenna Doe [19 years]
Users from USA at the age of 8:
* James Doe [8 years]
```

As always, we have used the LINQ Method syntax through this article, but allow me to supply you with a comparison example on how it could be done with the LINQ Query syntax:

```
// Method syntax
var usersGroupedByCountryAndAge = users.GroupBy(user => new {
user.HomeCountry, user.Age });
// Query syntax
var usersGroupedByCountryAndAgeQ = from user in users group user by new
{ user.HomeCountry, user.Age } into userGroup select userGroup;
```

1.8.7.3. Summary

As you can probably see from the examples in this article, the GroupBy() method of LINQ is extremely powerful. It really allows you to use your data in new ways, with very little code. Previously this would either be very cumbersome or require a relational database, but with LINQ, you can use whatever data source you'd like and still get the same, easy-to-use functionality.

1.9. Working with Culture & Regions

1.9.1. Introduction

The time where you created a piece of software to be executed on just a couple of local computers are pretty much gone. Thanks to globalization and the Internet in particular, software today is generally used all around the world, on a very wide range of devices. This means that your code needs to handle a lot of culture-specific cases, like dealing with numbers and dates in another format than what you expect. For instance, did you know that in many countries, a number with fractions (e.g. 1.42) doesn't use a period but a comma as the decimal separator (e.g. 1,42)? And are you aware that in many countries, the day is written before the month in a date, while others write the year first?

Dealing with all this can be a major hassle, but fortunately for us, the .NET framework has several classes which can help us deal with these situations. The most commonly used is the CultureInfo class, which we'll discuss in the next article, but .NET also offers classes for working with regions and even specific calendars (you do know that even the calendar is not the same around the world, right?).

This topic is especially important if you're working on an application which should support multiple languages, but even if that's not the case, you still have to deal with the fact that it might be used on a device which doesn't use the same notations for e.g. dates and numbers. To illustrate just how important that is, consider this example:

```
string inputNumber = "1.425";
double usNumber = double.Parse(inputNumber, CultureInfo.GetCultureInfo(
"en-US"));
double germanNumber = double.Parse(inputNumber,
CultureInfo.GetCultureInfo("de-DE"));
Console.WriteLine(usNumber.ToString() + " is not the same as " +
germanNumber);
```

Think of the *inputNumber* variable as something we just received from the user of the application, e.g. something typed in a text field on a web form. We use the double.Parse() method to turn it into a float, but we pass in a second parameter of the CultureInfo type - if we didn't pass in this parameter, the system settings would be used, which could then be English, German or something completely different. Now notice the output:

```
1,425 is not the same as 1425
```

Quite true! Our number value just got a thousand times bigger, because in Germany, they use a comma as the decimal separator, while a period acts as the thousands separator. This would be a major problem in most applications, but the good news is that while the example illustrates the problem, it also illustrates the solution: You always have to be in control of how you receive input and then deal with it accordingly, because as you can see, thanks to the CultureInfo class, .NET is capable of parsing a number (and dates

too!) in any of the possible formats, as long as you tell it what to expect.

1.9.1.1. Summary

Dealing with cultural and regional differences is perhaps even more important when programming than in real life. Fortunately for us, the .NET framework can be a huge help in these regards, as long as you know where to look. In the next couple of articles, we'll discuss the tools offered by the .NET framework for dealing with these differences.

1.9.2. Application Culture & UICulture

As we discussed in the previous article, dealing with culture is very important, especially when working with dates and numbers. For that reason, your application will always have an instance of the CultureInfo class defined as its "CurrentCulture" - a fallback instance for all the situations where you don't specifically state which culture should be used e.g. for outputting a number. Unless you change this property, which exists on the CultureInfo class, it will be the same as the culture used by your operating system. Here's a simple way to check it:

```
Console.WriteLine("Current culture: " +
CultureInfo.CurrentCulture.Name);
```

In this article, we will use the CultureInfo class all the time, so don't forget to include it with the using statement, as shown in the top of the example.

The example will output the culture used by your application, e.g. "en-US" for a computer running with the English language in the United States. If you live in Germany and your computer uses the German language, the result will instead be "de-DE". In other words, the two first letters specify the language, while the last two letters specify the country or region.

However, it might very well be that you want more control of which culture is used by your application. For instance, if your application uses the English language all over the place, does it make sense to output numbers in a German or Swedish format just because the computer of your user specifies it? Perhaps it does, but if not, you can very easily specify a new default/fallback culture, using the CurrentCulture property again:

```
CultureInfo.CurrentCulture = new CultureInfo("en-US");
Console.WriteLine("Current culture: " +
CultureInfo.CurrentCulture.Name);

float largeNumber = 12345.67f;
Console.WriteLine("Number format (Current culture): " +
largeNumber.ToString());

CultureInfo germanCulture = new CultureInfo("de-DE");
Console.WriteLine("Number format (German): " +
largeNumber.ToString(germanCulture));
```

We basically go in and overrule the CurrentCulture property by setting it to an en-US culture. Then we output it, along with a large, floating point number. You will see that the result is a number formatted the way it is in English. In the last couple of lines, we illustrate how you can, of course, overrule the fallback culture by passing another CultureInfo instance to the ToString() method - in this case the number will be outputted in a German format as well. The resulting output of this example should look something like this:

```
Current culture: en-US
Number format (Current culture): 12345.67
Number format (German): 12345,67
```

1.9.2.1. CurrentCulture vs. CurrentUICulture

You may notice that the CultureInfo also has a property called **CurrentUICulture**. This property is only relevant if you're using resource files for localizing a user interface - in that case, your application will know which versions of the resource files should be loaded, based on the CurrentUICulture property. For all other purposes, including the formatting of numbers, dates and so on, you should use the **CurrentCulture** property.

1.9.2.2. CurrentCulture and Threads

We have not really talked about threads yet, but they are basically a concept which will allow your application to work on several things at the same time. When a .NET application is started, a single thread is created and only this thread will be used unless you specifically create a new one, using one of the many multi-threading strategies of the framework. I mention it here because it's also very relevant when it comes to the fallback culture of your application. In fact, *CultureInfo.CurrentCulture* is basically a shortcut to the **Thread.CurrentThread.CurrentCulture** property, which means that whenever you define the CurrentCulture, it's only valid for the current thread.

Before .NET framework version 4.5, you would have to manually specify the culture of each new thread you created. However, in .NET 4.5, the **CultureInfo.DefaultThreadCurrentCulture** property was introduced. When you set it, each new thread created will use this culture as well, and it's just as easy to use as the CurrentCulture property:

```
CultureInfo.DefaultThreadCurrentCulture = new CultureInfo("de-DE");
```

But what about the existing thread? Well actually, if you have not already defined another culture for the *CurrentCulture* property, setting the *DefaultThreadCurrentCulture* property will also be applied to the *CurrentCulture* property. In other words, it makes sense to use the *DefaultThreadCurrentCulture* instead of *CurrentCulture* if you plan on using multiple threads in your application - it will take care of all scenarios.

1.9.2.3. Summary

By using the *CultureInfo.CurrentCulture* and/or the *CultureInfo.DefaultThreadCurrentCulture* properties, you can define a fallback culture for your application - it will always be used whenever a number or a date is being outputted, unless you specifically overrule this in each situation.

1.9.3. The CultureInfo class

In the last couple of articles, we have talked about how useful the CultureInfo class is when you need full control of how numbers and dates are displayed in your application. We have also talked about how you can verify and modify which culture your application should use as a fallback. With all that in place, it's time to dig deeper into the actual CultureInfo class, to see how we can take full advantage of it.

A quick reminder before we get started: The CultureInfo class is part of the *System.Globalization* namespace, so be sure to import that whenever you try the examples:

```
using System.Globalization;
```

1.9.3.1. Neutral and specific cultures

In the previous examples in this chapter, we have only used specific cultures, which is a culture that specifies both a language and a country/region. An example of this is the **en-US** culture, which clearly states that the desired language should be English and the region is the US. An alternative to that is the **en-GB** culture, which is the same language (English) but with Great Britain as the region instead of the US.

There will be times when these differences are important to you, in which case you should use these region-specific versions of the CultureInfo class. On the other hand, there will also be situations where English is just a language and you don't want to tie this language to a specific country or region. For this, the .NET framework defines so-called neutral cultures, which only specifies a language. In fact, both en-US and en-GB inherit from such a neutral culture (which makes perfect sense, since they share the same language!) and you can access it from the **Parent** property. Let me illustrate with an example:

```
CultureInfo enGb = new CultureInfo("en-GB");
CultureInfo enUs = new CultureInfo("en-US");
Console.WriteLine(enGb.DisplayName);
Console.WriteLine(enUs.DisplayName);
Console.WriteLine(enGb.Parent.DisplayName);
Console.WriteLine(enUs.Parent.DisplayName);
```

Not a terribly useful example, but it should give you a better idea of the internal structure of the CultureInfo class. The output should look like this:

```
English (United Kingdom)
English (United States)
English
English
```

1.9.3.2. Getting the right CultureInfo

From our previous examples, you have seen that we can fetch the desired CultureInfo class by passing the language-country/region identifier to the constructor of the class. But since you might be looking for a

neutral culture, as described above, you could also just pass in a language identifier:

```
CultureInfo en = new CultureInfo("en");
```

The .NET framework will then return an English, region-neutral CultureInfo instance for you. For a full list of possible language and/or language-country/region identifiers, I suggest you check out the MSDN documentation.

Another way of identifying a specific culture is with the so-called LCID (LoCale ID). You will find it as a property on existing CultureInfo instances, but if you know the ID, you can also use it to instantiate a CultureInfo object. For instance, the LCID for en-US is 1033:

```
CultureInfo enUs = new CultureInfo(1033);
```

However, in most situations, it's much easier to use the language-country/region specifier, as previously demonstrated.

1.9.3.3. Getting a list of available Cultures

We can now get a specific culture and use it for various purposes, but perhaps you need a list of the available cultures, e.g. to allow the user to select a language and/or country/region. Fortunately, the .NET framework makes that easy for us as well - here's an example:

```
CultureInfo[] specificCultures =
CultureInfo.GetCultures(CultureTypes.SpecificCultures);
foreach (CultureInfo ci in specificCultures)
     Console.WriteLine(ci.DisplayName);
Console.WriteLine("Total: " + specificCultures.Length);
```

As you can tell from the first line of code, I use the **GetCultures** static method on the CultureInfo class to get a list of cultures. It requires the **CultureTypes** parameter, which specifies which kind of cultures you're looking for. In this case, I have asked for the specific cultures, which, as we talked about previously, are the cultures which are tied to both a language and a country/region. That's quite a long list, by the way - on this computer, I get a total of 563 available cultures!

But perhaps you're more interested in the neutral cultures? That would, for instance, make perfect sense if you were building a list of available languages, while not caring which country or region they were related to. Doing it is as simple as changing the CultureTypes parameter:

```
CultureInfo[] neutralCultures =
CultureInfo.GetCultures(CultureTypes.NeutralCultures);
foreach (CultureInfo ci in neutralCultures)
     Console.WriteLine(ci.DisplayName);
Console.WriteLine("Total: " + neutralCultures.Length);
```

By doing so, you will also see that there not quite as many neutral cultures as there are specific cultures - on my computer/.NET framework version, the result is a total of 280 neutral cultures.

1.9.3.4. Important properties & methods of CultureInfo

Once you have an instance of the CultureInfo class, you immediately get access to a very wide range of usable properties and methods. These members can help you accomplish many useful things in regards to culture - let's have a look at some of them!

1.9.3.5. DateTimeFormat

With the DateTimeFormat property, you get access to information about how date and time should be formatted, as well as a lot of useful information about the calendar for the given culture. A nice example of this is the **FirstDayOfWeek** and **CalendarWeekRule** properties - they can tell you on which day a week starts (usually Sunday or Monday) and how the first calendar week of the year is decided (e.g. just the first day or the first full week):

```
CultureInfo enUs = new CultureInfo("en-US");
Console.WriteLine("First day of the: " +
enUs.DateTimeFormat.FirstDayOfWeek.ToString());
Console.WriteLine("First calendar week starts with: " +
enUs.DateTimeFormat.CalendarWeekRule.ToString());
```

Try changing the CultureInfo instance to your own culture or another culture you know, to see how these properties vary!

Another cool thing is that you can get information about the month and day names for the specific culture using properties like **MonthNames** and methods like **GetMonthName()**. Here's a quick example:

```
CultureInfo enUs = new CultureInfo("en-US");

foreach (string monthName in enUs.DateTimeFormat.MonthNames)
     Console.WriteLine(monthName);
Console.WriteLine("Current month: " +
enUs.DateTimeFormat.GetMonthName(DateTime.Now.Month));
```

And the exact same thing can be accomplished for days, using the **DayNames** property and the **GetDayName()** method:

```
CultureInfo enUs = new CultureInfo("en-US");

foreach (string dayName in enUs.DateTimeFormat.DayNames)
     Console.WriteLine(dayName);
Console.WriteLine("Today is: " +
```

```
enUs.DateTimeFormat.GetDayName(DateTime.Now.DayOfWeek));
```

There are many more useful properties and methods on the DateTimeFormat property, e.g. *DateSeparator* , *YearMonthPattern* and so on. Have a look for your self - there might very well be a solution to your date/time related problem hidden in there: DateTimeFormatInfo documentation.

1.9.3.6. NumberFormat

Just like the DateTimeFormat has information about dates, you can access information on how the specific culture treats numbers from the **NumberFormat** property. This information is used each time you ask for a visual representation of a number, e.g. when converting it to a string and writing it to the console, but you can also access the information yourself, by using the properties and methods on the NumberFormat property - here's an example:

```
CultureInfo enUs = new CultureInfo("en-US");
Console.WriteLine(enUs.DisplayName + ":");
Console.WriteLine("NumberGroupSeparator: " +
enUs.NumberFormat.NumberGroupSeparator);
Console.WriteLine("NumberDecimalSeparator: " +
enUs.NumberFormat.NumberDecimalSeparator);

CultureInfo deDe = new CultureInfo("de-DE");
Console.WriteLine(deDe.DisplayName + ":");
Console.WriteLine("NumberGroupSeparator: " +
deDe.NumberFormat.NumberGroupSeparator);
Console.WriteLine("NumberDecimalSeparator: " +
deDe.NumberFormat.NumberDecimalSeparator);
```

We use the **NumberGroupSeparator** and the **NumberDecimalSeparator** properties to get information about how a number is displayed (e.g. 1,000.00 or 1.000,00) for the English and German cultures. If you have a look, you will find matching properties for currencies (**CurrencyGroupSeparator** and **CurrencyDecimalSeparator**) as well as percentages (**PercentGroupSeparator** and **PercentDecimalSeparator**).

Speaking of currency, the NumberFormat property can also tell you which symbol a given culture uses to display a monetary amount - simply use the **CurrencySymbol** property:

```
CultureInfo enUs = new CultureInfo("en-US");
Console.WriteLine(enUs.DisplayName + " - currency symbol: " +
enUs.NumberFormat.CurrencySymbol);
CultureInfo deDe = new CultureInfo("de-DE");
Console.WriteLine(deDe.DisplayName + " - currency symbol: " +
deDe.NumberFormat.CurrencySymbol);
```

```
CultureInfo ruRu = new CultureInfo("ru-RU");
Console.WriteLine(ruRu.DisplayName + " - currency symbol: " +
ruRu.NumberFormat.CurrencySymbol);
```

All these properties are nice to know about, but in most situations, you won't have to deal with them, since C# will silently use the information to format numbers, percentages and currencies for you, as long as you specify the right format string when you turn the number into a string.

1.9.3.7. Names & identifiers

Lastly, let's have a look at the properties which represents the CultureInfo instance. We have already used some of them, e.g. **Name** and **DisplayName**, but how do they actually work? First, here's a list of available properties used to identify a CultureInfo:

- Name will identify a CultureInfo in the languagecode-country/region-code format, e.g. "en-US" for English in the US, en-GB for English in Great Britain and so on. If no country/region is specified, only the first part will be returned, e.g. "en" for English.
- TwoLetterISOLanguageName will do pretty much the same as Name, but it will only return the language code, no matter if a country/region has been specified or not. For instance, "en" will be returned for both "en-US" and "en-GB". The letters returned are specified in the ISO 639-1 standard.
- ThreeLetterISOLanguageName works much like TwoLetterISOLanguageName, but it returns three letters instead of two, as specified by the ISO 639-2 standard.
- EnglishName will return the name of the language (in English). If a country/region has been specified, this will be appended to the result, within a set of parentheses.
- NativeName will return the name of the language (in the language specified by the CultureInfo instance). If a country/region has been specified, this will be appended to the result, within a set of parentheses.

1.9.3.8. Summary

As you can gather from the length of this article, dealing with culture in general is not a simple task. Fortunately for us, the .NET framework makes it a whole lot easier with the CultureInfo class. It's used silently all across your application when formatting numbers and dates, but it's good for you to know how it works so that you can modify the behavior if needed. Hopefully this article has taught you most of what you need to know about the CultureInfo class.

1.9.4. The RegionInfo class

In the previous article on the CultureInfo class, we discussed the country/region part of it a bit, but we can actually do way more region-based stuff with one of the other classes in the *System.Globalization* namespace: The **RegionInfo** class. It will contain a lot of useful information about a specific region (usually a country), e.g. the name and symbol of their currency, whether they use the metric system or not and so on.

1.9.4.1. Obtaining a RegionInfo instance

To get access to regional info, you need an instance of the RegionInfo class. It has a constructor which can take an ISO 3166 code or the languagecode/region-country code of the region (e.g. "en-US"). Here's an example:

```
RegionInfo regionInfo = new RegionInfo("en-US");
Console.WriteLine(regionInfo.EnglishName);
```

This also means that if you already have a reference to a CultureInfo class, you can easily use this to make sure that you get the matching RegionInfo. And as we learned in a previous article, your application always has a fallback CultureInfo instance that you can reference:

```
RegionInfo regionInfo = new RegionInfo(CultureInfo.CurrentCulture.Name);
Console.WriteLine(regionInfo.EnglishName);
```

With that in place, let's have a look at some of the useful features of the RegionInfo class.

1.9.4.2. Important properties of the RegionInfo class

We already checked out the EnglishName property - it simply returns the name of the region, in English. But of course there's more good stuff in there - for instance several properties related to currency:

```
RegionInfo regionInfo = new RegionInfo("sv-SE");
Console.WriteLine(regionInfo.CurrencySymbol);
Console.WriteLine(regionInfo.ISOCurrencySymbol);
Console.WriteLine(regionInfo.CurrencyEnglishName);
Console.WriteLine(regionInfo.CurrencyNativeName);
```

Using **CurrencySymbol**, **ISOCurrencySymbol**, **CurrencyEnglishName** and/or **CurrencyNativeName**, we get the information we need to output monetarily related messages. The result will look like this (in this case for Swedish/Sweden):

```
kr
SEK
Swedish Krona
Svensk krona
```

You can also easily check if the given region uses the metric system, using the **IsMetric** property:

```
RegionInfo regionInfo = new RegionInfo(CultureInfo.CurrentCulture.Name);
Console.WriteLine("Is the metric system used in " +
regionInfo.EnglishName + "? " + (regionInfo.IsMetric ? "Yes" : "No"));
```

That leaves us with all the identity related properties:

- Name will get you the ISO 3166 code which identifies the language and country/region, e.g. "en-US"_ for English/United States.
- DisplayName will get you the full name of the country/region in the localized .NET framework version.
- EnglishName will get you the full name of the country/region in English.
- NativeName will get you the full name of the country/region in the given language, e.g. "United States" for en-US or "Deutschland" for de-DE.
- TwoLetterISORegionName will get you the two-letter ISO 3166 code for the country/region, e.g. "US" for the United States or "DE" for Germany.
- ThreeLetterISORegionName will get you the three-letter ISO 3166 code for the country/region, e.g. "USA" for the United States or "DEU" for Germany.

Obviously, these properties will come in handy when you have to display information about a country/region, as we'll see in our next example.

1.9.4.3. Getting a list of countries with RegionInfo

In a previous article, I showed you how to get a list of all the defined cultures in the .NET framework, which basically leaves us with a list of language-country/region combinations. We can use that in combination with the RegionInfo class to get a list of countries/regions:

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;

namespace RegionInfoCountries
{
     class Program
     {
          static void Main(string[] args)
          {
               CultureInfo[] cultures =
CultureInfo.GetCultures(CultureTypes.SpecificCultures);
```

```
            List<RegionInfo> countries = new List<RegionInfo>();
            foreach (CultureInfo ci in cultures)
            {
                RegionInfo regionInfo = new RegionInfo(ci.Name);
                if (countries.Count(x => x.EnglishName ==
regionInfo.EnglishName) <= 0)
                    countries.Add(regionInfo);
            }
            foreach (RegionInfo regionInfo in countries.OrderBy(x =>
x.EnglishName))
                Console.WriteLine(regionInfo.EnglishName);
        }
    }
}
```

So, this example is a bit longer than the other examples in this article, but let me break it down for you. We start by getting a list of all available, specific cultures - as we learned in a previous article, specific cultures are the ones who define a language AND a region/country. We loop through this list and on each iteration, we use the CultureInfo instance to create a corresponding RegionInfo instance. We check if a country with that name has already been added to our list and if it hasn't, we add it. When the loop has finished, we now have a complete list of the countries defined by the .NET framework, which we can loop through and output to the console, or do pretty much anything else that we might find useful.

This is a great example of what you can accomplish with the combination of CultureInfo and RegionInfo, but allow me to emphasize that this is not a complete and accurate list of countries. It is instead a complete list of countries defined in the version of the .NET framework you're using, basically meaning that some countries could be missing or may have changed their name since the version was released. So, if you need a list of countries which is 100% accurate and up-to-date, you will probably have to create and maintain it yourself.

1.9.4.4. Summary

The RegionInfo class is basically an extension of the CultureInfo class with even more useful information about a specific country/region. It allows you to know more about the identity and currency of a specific region, and it can help you get a list of countries/regions.

1.10. Regular Expressions (Regex)

1.10.1. Introduction

We have previously talked a lot about strings - the ability to process and manipulate text is so important for all programmers out there! But while chopping up strings with the SubString method or doing some simple string-to-string replace operations can be considered simple string processing, string processing with Regular Expressions (usually shortened to Regex) is the extreme version!

First of all, Regular Expressions is not a Microsoft/.NET invention. In fact, Regular Expressions were invented way before the .NET framework, as a way of expressing a search pattern. This search pattern can then be used to carry out search or search-replace operations on a piece of text. Your first thought might be that you don't need another "language" to search through a string, but just wait and see what you can do with Regular Expressions!

The cool thing about Regular Expressions is that they are supported by pretty much every programming language out there, and while there are small variations, you can use the same regex across these programming languages to accomplish the same thing. The .NET framework has a very nice implementation of Regular Expressions, centered around the *Regex* class found in the *System.Text.RegularExpressions* namespace.

1.10.1.1. Summary

With Regular Expressions you can define a search pattern to perform search and search/replace operations on a string. The .NET framework can easily work with your Regular Expressions, as we will discover in the next articles, where we will work with the *Regex* class as well as helper classes like the *Match* and *MatchCollection* classes.

1.10.2. Searching with the Regex Class

As we discussed in the previous article, Regular Expressions allow you to define search patterns for working with strings. To process this search pattern, the .NET framework comes with a very versatile class: The Regex class. In this article, we will define some search patterns and use them with the Regex class, but please bear in mind that the syntax of Regular Expressions can be quite complicated and that this is a C# tutorial and not a Regex tutorial. Instead, I will use some simple Regex patterns to demonstrate how you work with them in C#. If you want to know more about Regular Expression, I can recommend this Regular Expression Tutorial.

1.10.2.1. The IsMatch() Method

In this first example, I'll use one of the most basic methods of the Regex class called IsMatch. It simply returns true or false, depending on whether there is one or several matches found in the test string:

```
string testString = "John Doe, 42 years";
Regex regex = new Regex("[0-9]+");
if (regex.IsMatch(testString))
     Console.WriteLine("String contains numbers!");
else
     Console.WriteLine("String does NOT contain numbers!");
```

We define a test string and then we create an instance of the Regex class. We pass in the actual Regular Expression as a string - in this case, the regex specifies that we're looking for a number of any length. We then output a line of text depending on whether the regex is a match for our test string. Pretty cool, but in most cases, you're looking to actually do something with the match(es) - for this, we have the Match class.

1.10.2.2. The Match Class & Method

In this next example, we'll capture the number found in the test string and present it to the user, instead of just verifying that it's there:

```
string testString = "John Doe, 42 years";
Regex regex = new Regex("[0-9]+");
Match match = regex.Match(testString);
if (match.Success)
     Console.WriteLine("Number found: " + match.Value);
```

We use the same regex and test string as before. I call the **Match()** method, which will return an instance of the Match class - this will happen whether or not a match is actually found. To ensure that a match *has* been found, I check the **Success** property. Once I'm sure that a match has been found, I use the **Value** property to retrieve it.

The Match class contains more useful information than just the matched string - for instance, you can easily find out where the match was found, how long it is and so on:

```
string testString = "John Doe, 42 years";
Regex regex = new Regex("[0-9]+");
Match match = regex.Match(testString);
if (match.Success)
     Console.WriteLine("Match found at index " + match.Index + ".
Length: " + match.Length);
```

The **Index** and **Length** properties are used here to display information about the location and length of the match.

1.10.2.3. Capture Groups

In the first couple of examples, we have just found a single value in our search string, but Regular Expressions can, of course, do a lot more than that! For instance, we can find both the name and the age in our test string, while sorting out the irrelevant stuff like the command and the "years" text. Doing stuff like that is a piece of cake for Regular Expressions, but if you're not familiar with the syntax, it might seem very complicated, but let's give it a try anyway:

```
string testString = "John Doe, 42 years";
Regex regex = new Regex(@"^([^,]+),\s([0-9]+)");
Match match = regex.Match(testString);
if (match.Success)
     Console.WriteLine("Name: " + match.Groups[1].Value + ". Age: " +
match.Groups[2].Value);
```

I have modified the regex so that it looks for anything that is NOT a comma - this value is placed in the first capture group, thanks to the surrounding parentheses. Then it looks for the separating comma and after that, a number, which is placed in the second capture group (again, thanks to the surrounding parentheses). In the last line, I use the Groups property to access the matched groups. I use index 1 for the name and 2 for the age since it follows the order in which the match groups were defined in the regex string (index 0 contains the entire match).

1.10.2.4. Named Capture Groups

As soon as the regex becomes more advanced/longer than the one we just used, numbered capture groups might become unmanageable because you constantly have to remember the order and index of them. Fortunately for us, Regular Expressions and the .NET framework supports named capture groups, which will allow you to give each group a name in the regex and then reference it in the Groups property. Check out this re-written example, where we use named groups instead of numbered:

```
string testString = "John Doe, 42 years";
Regex regex = new Regex(@"^(?<name>[^,]+),\s(?<age>[0-9]+)");
Match match = regex.Match(testString);
if (match.Success)
```

```
    Console.WriteLine("Name: " + match.Groups["name"].Value + ". Age: "
+ match.Groups["age"].Value);
```

It works exactly as it does before, but you can now use logical names to lookup the matched values instead of having to remember the correct index. This might not be a big difference in our simple example, but as mentioned you will definitely appreciate it when your Regular Expressions grows in complexity and length.

1.10.2.5. The MatchCollection Class

The Match class is the way to go if you only want to work with a single match (remember that a match can contain multiple values, as we saw in the previous examples), but sometimes you want to work with several matches at once. For this, we have the **Matches()** method which will return a MatchCollection class. It will contain all matched values, in the order in which they were found. Let's have a look at how it can be used:

```
string testString = "123-456-789-0";
Regex regex = new Regex(@"([0-9]+)");
MatchCollection matchCollection = regex.Matches(testString);
foreach (Match match in matchCollection)
    Console.WriteLine("Number found at index " + match.Index + ": " +
match.Value);
```

I have changed the regex and the test string, compared to the previous examples. We now have a test string which contains several numbers and a regex which specifically looks for strings consisting of one or more numbers. We use the **Matches()** method to get a MatchCollection from our regex, which contains the matches found in the string. In this case, there are four matches, which we output one after another with a *foreach* loop. The result will look something like this:

```
Number found at index 0: 123
Number found at index 4: 456
Number found at index 8: 789
Number found at index 12: 0
```

If no matches were found, an empty MatchCollection would have been returned.

1.10.2.6. Summary

With help from the Regex class, along with the Match and MatchCollection classes, we can easily do very advanced string matching. The Regular Expression syntax might seem very complex, but once you learn it, you will have a very strong tool. Even if you don't want to invest the time in learning the regex syntax, you can often find expressions for specific needs, created by other programmers, with a simple Google search. As soon as you have written or borrowed the regex string, you can use it for your own purpose with the techniques and classes demonstrated in this article.

But searching is only a part of the fun - you can also do some very cool search/replace operations with

Regular Expressions. We will look into this in one of the next articles.

1.10.3. Search/Replace with the Regex Class

We have already discussed the Regex class and how to use it when we want to search through a string in a previous article. Regular Expressions are great for that, but another use case is when you want to carry out search/replace operations, where you want to look for a specific pattern and replace it with something else. The String class already has a Replace() method, but this is only good for doing simple searches. When using Regular Expressions, you can use the power of regex searches and even use captured groups as part of the replace string. Does it sound complicated? Don't worry, we'll start with a simple example and then slowly work toward more advanced use cases.

As in the previous article, all examples assume that you have imported the RegularExpressions namespace, like this:

```
using System.Text.RegularExpressions;
```

With that in place, let's try working with Regular Expression based string replacement. We'll use the **Replace()** method found on the **Regex** class:

```
string testString = "<b>Hello, <i>world</i></b>";
Regex regex = new Regex("<[^>]+>");
string cleanString = regex.Replace(testString, "");
Console.WriteLine(cleanString);
```

This example displays a very simplified approach to removing HTML tags from a string. We match anything that is surrounded by a set of angle brackets (<>) and then we use the Replace() method to replace each occurrence with an empty string, basically removing the HTML tags from the test string.

1.10.3.1. Replacing with Captured Values

But let's say that you don't actually want to remove them, but instead, you want to transform the tags into something that will not be interpreted by a browser, e.g. by replacing the angle brackets (<>) with square brackets ([]). This is where Regular Expressions really show their power, because it's actually very easy, as illustrated by this slightly rewritten version of our previous example:

```
string testString = "<b>Hello, <i>world</i></b>";
Regex regex = new Regex("<([^>]+)>");
string cleanString = regex.Replace(testString, "[$1]");
Console.WriteLine(cleanString);
```

I actually just changed two minor details: I added a set of parentheses to the regex, to create a capture group, essentially capturing the value between the angle brackets into the first capture group. In the **Replace()** method I reference this using the special notation **$1**, which basically just means capture group number 1. With that in place, our output will now look like this:

```
[b]Hello, [i]world[/i][/b]
```

1.10.3.2. Named Capture Groups

You can of course do the exact same thing when using named capture groups (discussed in the previous article), like this:

```
string testString = "<b>Hello, <i>world</i></b>";
Regex regex = new Regex("<(?<tagName>[^>]+)>");
string cleanString = regex.Replace(testString, "[${tagName}]");
Console.WriteLine(cleanString);
```

When using named capture groups, just use the **${name-of-capture-group}** notation.

1.10.3.3. Using a MatchEvaluator method

But if we want even more control over how the value is replaced? We can use a MatchEvaluator parameter for this - it's basically just a reference (delegate) to a method which will be called each time a replacement is to be made, allowing you to modify the replacement value before it's used. Let's stick with the HTML tags example we have already used a couple of times, but this time, we take control over which HTML tags are used. Here's the complete example:

```
using System;
using System.Text.RegularExpressions;

namespace RegexSearchReplaceMethod
{
     class Program
     {
          static void Main(string[] args)
          {
               string testString = "<b>Hello, <i>world</i></b>";
               Regex regex = new Regex("<(?<tagName>[^>]+)>");
               string cleanString = regex.Replace(testString,
ProcessHtmlTag);
               Console.WriteLine(cleanString);
          }

          private static string ProcessHtmlTag(Match m)
          {
               string tagName = m.Groups["tagName"].Value;
               string endTagPrefix = "";
               if(tagName.StartsWith("/"))
               {
                    endTagPrefix = "/";
```

```
                tagName = tagName.Substring(1);
            }
            switch (tagName)
            {
                case "b":
                    tagName = "strong";
                    break;
                case "i":
                    tagName = "em";
                    break;
            }
            return "<" + endTagPrefix + tagName.ToLower() + ">";
        }
    }
}
```

The first part of the example looks exactly as it did before, but instead of supplying a replacement string, we pass on a reference to our *ProcessHtmlTag()* method. As mentioned, this method is called each time a replacement is about to be made, with the Match in question as a parameter. This means that in our MatchEvaluator method, we have all the information about the match so that we can act accordingly. In this case, we use this opportunity to make the tags more semantic by replacing the bold (b) tag with a strong tag and the italic (i) tag with an emphasis (em) tag. No matter if the tag is changed or not, we turn it into lowercase.

Using a MatchEvaluator parameter is obviously very powerful and this is just a simple example of what can be accomplished.

1.10.3.4. Summary

Search/replace operations becomes very powerful when you use Regular Expressions and even more so when you use the MatchEvaluator parameter, where the possibilities for manipulating strings become almost endless.

1.11. Misc

1.12. Debugging

1.12.1. Introduction to debugging

When you get past the most basic "Hello world!" examples, your code will reach a level of complexity where you can't necessarily figure out what's going on just by running it. What you need, is some black magic, which allows you to open the virtual hood of your application while it's running and see what's going on. Debugging is that magical tool, and as soon as you learn the most basic steps of it, you will wonder how you ever lived without it. It's a tool that every good programmer should understand, simply because it's almost impossible to fix bugs in complex code without it.

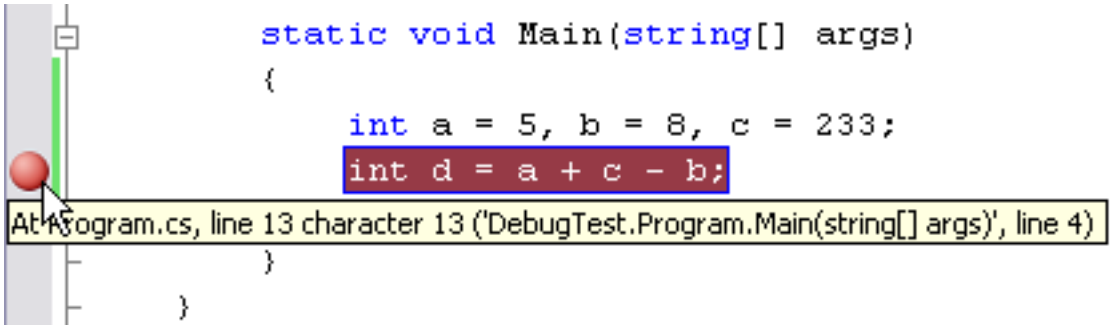
The most basic type of debugging, which is still being used by even advanced programmers, is sometimes called "print debugging" - a simple procedure, where you make your application print a piece of text or a number somewhere, allowing you to see which part of your code has been reached and what your variables contain. With C#, you can use the Console.Write() method, to output the contents of a variable or a simple status message, which will be printed to the console. That can be enough for some situations, but if you're using a nice IDE like Visual Studio or one of the Express versions, you have much stronger tools to your disposal, and they are every bit as easy to use, once you learn the most basic principles. In the next couple of chapters, we will guide you through the debugging possibilities of your IDE and after that, you will be a much stronger programmer.

1.12.2. Breakpoints

The very first thing about debugging that you need to know, is the breakpoint. It actually does exactly what the name implies - it marks a point in your code where the execution will take a break (and no, it won't actually break your code, don't worry). Placing a breakpoint in Visual Studio or one of the Express versions, is as simple as left-clicking in the gutter, which is the grey are to the left of your code. Once you click it, you will get a shiny, red circle as a reward - this circle marks where the debugger will halt when you execute your application. You better have a look for your self, and to see the effect, we will use the following piece of code:

```
namespace DebugTest
{
     class Program
     {
          static void Main(string[] args)
          {
               int a = 5, b = 8, c = 233;
               int d = a + c - b;
               Console.WriteLine(d);
          }
     }
}
```

Now, can you predict the result just from looking at the code? Probably, and if not, you could just get out the old calculator and do the math, but that's not really the point. Just imagine the amount of code being much bigger, and let's debug the thing! Place a breakpoint by clicking in the left gutter - your IDE should now look something like this:



Okay, you're ready to start your first debugging session. As soon as you have placed the breakpoint, you can just run your application like you normally would - from the menu, the toolbar or by pressing F5. What happens now is that the application is executed just like normal, but as soon as a line with a breakpoint is reached, the execution is stopped right before that line would be executed. In this case, it means that the variables a, b and c will have a value, but d will only have it's default value (which is 0 for an integer), since it won't be set before the line with the breakpoint has been evaluated. Now, here comes the cool part - try hovering your mouse over the different variables - the IDE will tell you what they contain. As mentioned, the

d variable will have it's default value, but let's change that, by moving forward in the execution. In the next chapter, I will show you how to navigate around your code, while it's being executed.

1.12.3. Stepping through the code

In this chapter, we will look into stepping through your code, which is another very essential part of debugging. For the purpose, I have written this simple application:

```
namespace DebugTest
{
	class Program
	{
		static void Main(string[] args)
		{
			int a = 5;
			a = a * 2;
			a = a - 3;
			a = a * 6;
			Console.WriteLine(a);
		}
	}
}
```

It simply manipulates the variable "a" a couple of times and the outputs the final result. Try placing a breakpoint, as described in the previous chapter, on the very first line where a is used (and declared). Now run the application. The execution is stopped and you can hover your mouse over the a, to ensure that what we learned in the previous chapter is in fact true: The variable only contains the default value, because the code that assigns the value (in this case 5), has not been executed yet, but let's change that. From the Debug menu, select the "Step over" option, or even better, use the keyboard shortcut F10. The execution will now proceed to the next relevant line and if you hover your mouse over the a variable, you will now see that it has a value. Try again, and you will see the value change according to the lines being executed one by one, until you have reached the end.

Okay, so that was pretty basic, but also very useful, as you will realise once you start writing more complicated code. In this example, the flow of the code was very simple, since we stayed within a single function, but what if your code starts spreading over multiple classes and/or functions? Try this example:

```
namespace DebugTest
{
	class Program
	{
		static void Main(string[] args)
		{
			int a = 5;
			int b = 2;
			int result = MakeComplicatedCalculation(a, b);
```

```
            Console.WriteLine(result);
        }

        static int MakeComplicatedCalculation(int a, int b)
        {
            return a * b;
        }
    }
}
```

Place a breakpoint on the first line of the Main method and run the application. Now use the "Step over" function to step through each line. As you will see, it moves over the function call without any notice - that's simply how debugging works. Now, try again, from the start, and once you have stepped to the line with the MakeComplicatedCalculation() call, select Debug -> Step into, or use the keyboard shortcut F11. The debugger now steps into the first possible function call, allowing you to step through that function as well. As you can probably imagine, this allows you to step through a complicated block of code, while only entering the function calls that interests you.

If you step into a function and then realise that you would rather return to the previous context, you use the very logically named option from the Debug menu called "Step out" (keyboard shortcut is Shift+F11). This will return you to your previous context, which obviously means that you can step into as many function calls as you want to, and then find your way back by using the step out option.

1.12.4. The tool windows

When debugging in Visual Studio, the tool windows in the bottom of the screen will change and new windows will be revealed (unless you have turned them off). The windows are called something along the lines of "Locals", "Watch", "Call stack" and "Immediate window" and they are all related to the debugging experience. In this chapter we will look into each of them and show you what they can do for you.

1.12.4.1. Locals

This window is the most simple of them all. When a breakpoint is hit, all local variables will be listed here, allowing you to get a quick overview of their name, type and value. You can even right-click in the grid and select "Edit value", to give a variable a new value. This allows you to test your code under other conditions than the current ones.

1.12.4.2. Watch

The Watch window is a bit like the Locals window, only here you get to decide which variables are tracked, local or global. You can add variables to watch over by dragging them from the code window, from the Locals window or by writing its name on the last, empty line. Your variables will stay in the Watch window until you remove it again, but will only be updated when you are debugging within the current scope. For instance, a variable in function A will not be updated when you are stepping through function B. Just like with the Locals window, you can right-click a watched variable and select "Edit value" to change the current value of the variable.

1.12.4.3. Call Stack

The Call Stack window will show you the current hierarchy of called functions. For instance, if function A calls function B which calls function C which then calls function D, the Call Stack window will show it, and you will be able to jump to each of the function declarations. You can also see which parameters were passed to each function. In the simple examples we have worked with so far, this might seem pointless, since keeping track of which function calls which function is trivial, but as soon as your code reaches a higher level of complexity and you have function in classes calling function in other classes, the Call Stack can be a real life saver.

1.12.4.4. Immediate window

The Immediate window is probably the most useful of them all. It allows you to execute custom lines of code in the current context of the debugger. This allows you to check variables, alter their values or just simply test a line of code. You simply type it into the window, hit Enter and the line will be executed. Type a variable name, and its value will be printed. Set a variable value by writing a = 5. The result, if any, will be printed and any changes you make, will be reflected when you continue the execution of the code. The Immediate window is like a C# terminal, where you can enter code and see the results immediately - once you get used to it, you might become addicted. I know I am.

1.12.5. Advanced breakpoints

In a previous chapter, we set the first breakpoint and it was good. However, there is actually more to breakpoints than that, at least if you're using Visual Studio. Unfortunately, it seems that Microsoft has disabled these extra debugging features in some of their Express versions, but don't worry: They are certainly nice to have, but you can get by without them. However, for those with access to Visual Studio, here is the most interesting breakpoint related features. You get access to them by setting a breakpoint, right-clicking it with the mouse and then selecting the desired function.

1.12.5.1. Condition

This option allows you to specify a condition that has to be true or changed, for the breakpoint to be hit. This can be really useful when dealing with more advanced code, where you want the execution to stop only under certain circumstances. For instance, you might have a loop that iterates a bunch of time before the relevant code is reached - in a situation like that, you could simply place a breakpoint and then configure an appropriate condition. Here is a rather boring example, which will show you how it works:

```
static void Main(string[] args)
{
     for(int i = 0; i < 10; i++)
          Console.WriteLine("i is " + i);
}
```

Set a breakpoint on the line where we do output to the console. Now run the application - the breakpoint is triggered each time the loop iterates. But perhaps that's not what we want. Maybe we only want it to be hit when i equals 4 (the 5th iteration). Do that by defining a simple condition like this:

```
i == 4
```

The breakpoint will now get a little, white plus inside it and when you run the application, it will only break when the i variable equals 4. You can also use the "has changed" option to instruct the debugger to only halt the execution if the result of the above statement has changed, for instance from false to true.

1.12.5.2. Hit count

With this dialog, you can define an alternative condition, based on the amount of times the breakpoint has been hit. For instance, you can decide that your breakpoint should not halt the execution until it has been hit a certain amount of times. There are various options to control this, depending on what you need, and during debug time, you can check this dialog to see how many times the breakpoint has been hit so far.

1.12.5.3. When hit...

Using this dialog, you can define alternative behaviour for when your breakpoint is hit. This can come in handy in lots of situations, where you don't want the execution to stop, but simply get a status message printed or a macro activated. It allows you to define a custom message which will be printed, where you can include all kinds of information about the execution. For advanced users, the option to get a specific macro

executed when a breakpoint is hit will also be useful.

1.13. Advanced topics

1.13.1. Enumerations

Enumerations are special sets of named values which all maps to a set of numbers, usually integers. They come in handy when you wish to be able to choose between a set of constant values, and with each possible value relating to a number, they can be used in a wide range of situations. As you will see in our example, enumerations are defined above classes, inside our namespace. This means we can use enumerations from all classes within the same namespace.

Here is an example of a simple enumeration to show what they are all about.

```
public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday }
```

All of these possible values correspond to a number. If we don't set them specifically, the first value is equal to 0, the next one to 1, and so on. The following piece of code will prove this, as well as show how we use one of the possible values from the enum:

```
using System;

namespace ConsoleApplication1
{
     public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday }

     class Program
     {
          static void Main(string[] args)
          {
               Days day = Days.Monday;
               Console.WriteLine((int)day);
               Console.ReadLine();
          }
     }
}
```

The output will be zero, because the Monday value maps directly to the number zero. Obviously we can change that - change the line to something like this:

```
public enum Days { Monday = 1, Tuesday, Wednesday, Thursday, Friday,
```

```
Saturday, Sunday }
```

If you run our code again, you will see that the Monday now equals 1 instead of 0. All of the other values will be one number higher as well as a result. You can assign other numbers to the other values as well. Because of the direct mapping to a number, you can use numbers to get a corresponding value from the enumeration as well, like this:

```
Days day = (Days)5;
Console.WriteLine(day);
Console.ReadLine();
```

Another cool feature of enumerations is the fact that you can a string representation of the values as well. Change the above example to something like this:

```
static void Main(string[] args)
{
     string[] values = Enum.GetNames(typeof(Days));
     foreach(string s in values)
          Console.WriteLine(s);

     Console.ReadLine();
}
```

The Enum class contains a bunch of useful methods for working with enumerations.

1.13.2. Exception handling

In every program, things go wrong sometimes. With C#, we're blessed with a good compiler, which will help us prevent some of the most common mistakes. Obviously it can't see every error that might happen, and in those cases, the .NET framework will throw an exception, to tell us that something went wrong. In an earlier chapter, about arrays, I described how we would get an exception if we tried to stuff too many items into an array. Let's bring the example:

```
using System;
using System.Collections;

namespace ConsoleApplication1
{
     class Program
     {
          static void Main(string[] args)
          {
               int[] numbers = new int[2];

               numbers[0] = 23;
               numbers[1] = 32;
               numbers[2] = 42;

               foreach(int i in numbers)
                    Console.WriteLine(i);
               Console.ReadLine();
          }
     }
}
```

Okay, try running this example, and you will see what I'm talking about. Do you see what we're doing wrong? We have defined an array of integers with room for 2 items, yet we try to use 3 spaces in it. Obviously, this leads to an error, which you will see if you try to run this example. When run inside Visual Studio, the IDE gives us some options for the exception, but if you try to execute the program by simply double-clicking the EXE file, you will get a nasty error. If you know that an error might occur, you should handle it. This is where exceptions are used. Here is a slightly modified version of the code from above:

```
int[] numbers = new int[2];
try
{
     numbers[0] = 23;
     numbers[1] = 32;
```

```
        numbers[2] = 42;

        foreach(int i in numbers)
            Console.WriteLine(i);
}
catch
{
        Console.WriteLine("Something went wrong!");
}
Console.ReadLine();
```

Let me introduce to you your new best friend when it comes to error handling: the try..catch block. Try running the program now, and see the difference - instead of Visual Studio/Windows telling us that a serious problem occurred, we get to tell our own story. But wouldn't it be nice if we could tell what went wrong? No problem:

```
catch(Exception ex)
{
        Console.WriteLine("An error occured: " + ex.Message);
}
```

As you can see, we have added something to the catch statement. We now tell which exception we want caught, in this case the base of all exceptions, the Exception. By doing so, we get some information about the problem which caused the exception, and by outputting the Message property, we get an understandable description of the problem.

As I said, **Exception** is the most general type of exception. The rules of exception handling tells us that we should always use the least general type of exception, and in this case, we actually know the exact type of exception generated by our code. How? Because Visual Studio told us when we didn't handle it. If you're in doubt, the documentation usually describes which exception(s) a method may throw. Another way of finding out is using the Exception class to tell us - change the output line to this:

```
Console.WriteLine("An error occured: " + ex.GetType().ToString());
```

The result is, as expected, IndexOutOfRangeException. We should therefore handle this exception, but nothing prevents us from handling more than one exception. In some situations you might wish to do different things, depending on which exception was thrown. Simply change our catch block to the following:

```
catch(IndexOutOfRangeException ex)
{
        Console.WriteLine("An index was out of range!");
}
catch(Exception ex)
```

```
{
     Console.WriteLine("Some sort of error occured: " + ex.Message);
}
```

As you can see, we look for the IndexOutOfRangeException first. If we did it the other way around, the catch block with the Exception class would get it, because all exceptions derive from it. So in other words, you should use the most specific exceptions first.

One more thing you should know about concerning exceptions is the finally block. The finally block can be added to a set of catch blocks, or be used exclusively, depending on your needs. The code within the finally block is always run - exception or no exception. It's a good place if you need to close file references or dispose objects you won't need anymore. Since our examples have been pretty simple so far, we haven't really been in need of any cleanup, since the garbage collector handles that. But since will likely run into situations where you need the finally block, here is an extended version of our example:

```
int[] numbers = new int[2];
try
{
     numbers[0] = 23;
     numbers[1] = 32;
     numbers[2] = 42;

     foreach(int i in numbers)
          Console.WriteLine(i);
}
catch(IndexOutOfRangeException ex)
{
     Console.WriteLine("An index was out of range!");
}
catch(Exception ex)
{
     Console.WriteLine("Some sort of error occured: " + ex.Message);
}
finally
{
     Console.WriteLine("It's the end of our try block. Time to clean
up!");
}
Console.ReadLine();
```

If you run the code, you will see that both the first exception block and the finally block is executed. If you remove the line that adds the number 42 to the array, you will see that only the finally block is reached.

Another important part you should know about exceptions, is how they impact the method in which the exceptions occur. Not all unhandled exceptions are fatal for your application, but when they aren't, you should not expect the remaining code of the method to be executed. On the other hand, if you do handle the exception, only the lines after the try block will be executed. In our example, the loop that outputs the values of the array is never reached, because the try block goes straight to the catch/finally block(s) once an exception is thrown. However, the last line, where we read from the console to prevent the application from exiting immediately, is reached. You should always have this in mind when you construct try blocks.

1.13.3. Structs

The C# struct is a lightweight alternative to a class. It can do almost the same as a class, but it's less "expensive" to use a struct rather than a class. The reason for this is a bit technical, but to sum up, new instances of a class is placed on the heap, where newly instantiated structs are placed on the stack. Furthermore, you are not dealing with references to structs, like with classes, but instead you are working directly with the struct instance. This also means that when you pass a struct to a function, it is by value, instead of as a reference. There is more about this in the chapter about function parameters.

So, you should use structs when you wish to represent more simple data structures, and especially if you know that you will be instantiating lots of them. There are lots of examples in the .NET framework, where Microsoft has used structs instead of classes, for instance the Point, Rectangle and Color struct.

First I would like to show you an example of using a struct, and then we will discuss some of the limitations of using them instead of classes:

```
class Program
{
     static void Main(string[] args)
     {
          Car car;

          car = new Car("Blue");
          Console.WriteLine(car.Describe());

          car = new Car("Red");
          Console.WriteLine(car.Describe());

          Console.ReadKey();
     }
}

struct Car
{
     private string color;

     public Car(string color)
     {
          this.color = color;
     }

     public string Describe()
```

```
        {
            return "This car is " + Color;
        }

        public string Color
        {
            get { return color; }
            set { color = value; }
        }
}
```

The observant reader will notice that this is the exact same example code as used in the introduction to classes, besides the change from a class to a struct. This goes to show how similar the two concepts are. But how do they differ, besides the technical details mention in the beginning of this chapter?

First of all, fields can't have initializers, meaning that you can't declare a member like this:

```
private string color = "Blue";
```

If you declare a constructor, all fields must be assigned to before leaving the constructor. A struct does come with a default constructor, but as soon as you choose to define your own, you agree to initialize all fields in it. That also means that you can't declare your own paramaterless constructor - all struct constructors has to take at least one parameter. In our example above, we did in fact assign a value to the color field. If we hadn't done that, the compiler would complain.

A struct can not inherit from other classes or structs, and classes can't inherit from structs. A struct does inherit from the Object class, but that's it for inheritance and structs. They do support interfaces though, meaning that your structs can implement custom interfaces.

1.14. XML

1.14.1. Introduction to XML with C#

XML is short for eXtensible Markup Language. It is a very widely used format for exchanging data, mainly because it's easy readable for both humans and machines. If you have ever written a website in HTML, XML will look very familiar to you, as it's basically a stricter version of HTML. XML is made up of tags, attributes and values and looks something like this:

```
<users>
<user name="John Doe" age="42" />
<user name="Jane Doe" age="39" />
</users>
```

As you can see, for a data format, this is actually pretty easy to read, and because it's such a widespread standard, almost every programming language has built-in functions or classes to deal with it. C# is definitely one of them, with an entire namespace, the System.Xml namespace, to deal with pretty much any aspect of XML. In the following chapters, we will look into using them, both for writing and reading XML. Read on!

1.14.2. Reading XML with the XmlReader class

There are mainly two methods for reading XML with C#: The XmlDocument class and the XmlReader class. XmlDocument reads the entire XML content into memory and then lets you navigate back and forward in it as you please, or even query the document using the XPath technology.

The XmlReader is a faster and less memory-consuming alternative. It lets you run through the XML content one element at a time, while allowing you to look at the value, and then moves on to the next element. By doing so, it obviously consumes very little memory because it only holds the current element, and because you have to manually check the values, you will only get the relevant elements, making it very fast. In this chapter, we will focus on the XmlReader class, and then move onto the XmlDocument class in the next chapter.

Let's try a little example, where we read an XML document containing currency rates. Fortunate for us, the European Central Bank has one that we can use. We could download it and read it from the harddrive, but actually, both the XmlReader and the XmlDocument classes can read XML from a remote URL just as well as from a local file. You can see the XML here (http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml) and here comes some code and then an explanation of it:

```
using System;
using System.Text;
using System.Xml;

namespace ParsingXml
{
     class Program
     {
          static void Main(string[] args)
          {
               XmlReader xmlReader = XmlReader.Create(
"http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml");
               while(xmlReader.Read())
               {
                    if((xmlReader.NodeType == XmlNodeType.Element) &&
(xmlReader.Name == "Cube"))
                    {
                         if(xmlReader.HasAttributes)
                              Console.WriteLine(xmlReader.GetAttribute(
"currency") + ": " + xmlReader.GetAttribute("rate"));

                    }
               }
               Console.ReadKey();
```

```
            }
        }
}
```

We start off by creating the XmlReader instance, using the static Create() method. It has several overloads, but the simplest of them just takes a URL pointing to the file. In a while loop, we call the Read() method on the XmlReader instance. It advances the reader to the next element and then returns true as long as there is more to read. Inside the loop, we can now use one of the many properties and methods on the XmlReader to access data from the current element.

In our case, we check the NodeType to see if we have an Element (the tag part), and if the name is "Cube". As you can see in the XML document we used, each currency rate is within an element called Cube, so this is of course what we're looking for - the rest is ignored. Once we have a Cube element, we do a formal check to see if it has attributes (which it should) and then we use the GetAttribute() method to read the two attribute values "currency" and "rate". We print them out to the Console and then we move on to the next element. The result should be a complete list of currencies and their current exchange rate.

As you can see, this is pretty simple to do, but mainly because we just need the data in the same order as they are read, without doing anything fancy to it. In the next chapter, we will do the same thing but using the XmlDocument class, so that you can see the difference.

Note: You should be aware that there are many ways in which the above code could fail and throw an exception, which you should of course handle. See the chapter on exceptions for more information.

1.14.3. Reading XML with the XmlDocument class

As described in the previous chapter, the XmlDocument is more memory consuming and possibly a bit slower than the XmlReader approach. However, for many purposes, the XmlDocument can be easier to work with and often require less code. Once the XML content has been read, you can read the data in a hierarchical way, just like the XML structure, with a root element which can have child elements, which can have child elements, and so on. In the previous chapter, we parsed XML data from the European Central Bank which could tell us about the current currency exchange rates, and we will do the same now, but using the XmlDocument class instead.

The XML can be found at the current URL (http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml) and the data we need is in the <cube> elements. In a tree structure, it looks something like this:

```
<gesmes:Envelope>
     [other child nodes]
     <Cube>
          <Cube time="2011-04-12">
               <Cube currency="USD" rate="1.4470"/>
               <Cube currency="JPY" rate="121.87"/>
               …
```

The gesmes:Envelope is our root element, which we can access using the DocumentElement property. We will then be able to access children of this node by using the ChildNodes collection property. In our example, we want the child nodes three levels below the root/document element. We can do that using the following code, which essentially does the same as the XmlReader based code in the previous chapter:

```
using System;
using System.Text;
using System.Xml;

namespace ParsingXml
{
     class Program
     {
          static void Main(string[] args)
          {
               XmlDocument xmlDoc = new XmlDocument();
               xmlDoc.Load(
"http://www.ecb.int/stats/eurofxref/eurofxref-daily.xml");
               foreach(XmlNode xmlNode in
xmlDoc.DocumentElement.ChildNodes[2].ChildNodes[0].ChildNodes)
                    Console.WriteLine(xmlNode.Attributes["currency"
].Value + ": " + xmlNode.Attributes["rate"].Value);
```

```
                Console.ReadKey();
            }
        }
}
```

As you can see, we access the Cube nodes by going down the ChildNodes hierarchy. From the DocumentElement (the root element), we ask for the third child node (zero-index based), then we ask for the first child node of that, and then we ask for the entire collection of child nodes. Obviously this is only possible because we know the structure of the XML document, and it's definitely not very flexible, pretty or easy to change later on. However, the way you navigate an XML document very much depends on the XML source and the data you need. For this example, the above will work just fine and even with a very limited amount of code, but for other purposes, you may want to use a bit more code to increase the readability.

Once we have a node with a currency rate, we access the two properties we're interested in and then output them to the console, just like the example in the previous chapter.

1.14.4. Working with the XmlNode class

In the previous chapter, we used the XmlDocument class to parse an XML file. A new class was introduced in the example, which is very essential to parsing XML with XmlDocument: The XmlNode class. The XML is basically parsed into an XmlNode which is the root element and then you can access the child elements using the ChildNodes property. However, the XmlNode class gives you access to a lot of other information as well, for instance the name of the tag, the attributes, the inner text and the XML itself. This chapter is a brief description of some of the more interesting aspects of the XmlNode class, which is important to know about because the XmlNode class is such a key concept when parsing XML with the XmlDocument class. In the following examples we will use the DocumentElement a lot, and while it is in fact of the type XmlElement, XmlElement does inherit from XmlNode, so it's essentially the same.

The **Name property** will simply give you the name of the node. For instance, the following example will output the text "user":

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<user name=\"John Doe\">A user node</user>");
Console.WriteLine(xmlDoc.DocumentElement.Name);
Console.ReadKey();
```

The **InnerText property** will hold the text contained within the starting and the ending tag, like this:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<test>InnerText is here</test>");
Console.WriteLine(xmlDoc.DocumentElement.InnerText);
Console.ReadKey();
```

The **InnerXml property** is a bit like the InnerText property, but while InnerText will strip out any XML within it, the InnerXml property obviously won't. The following example should illustrate the difference:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<users><user>InnerText/InnerXml is here</user></users>"
);
Console.WriteLine("InnerXml: " + xmlDoc.DocumentElement.InnerXml);
Console.WriteLine("InnerText: " + xmlDoc.DocumentElement.InnerText);
Console.ReadKey();
```

The **OuterXml property** is the same as the InnerXml, but it will include the XML of the node itself as well. The following example should illustrate the difference:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<users><user>InnerText/InnerXml is here</user></users>"
);
Console.WriteLine("InnerXml: " + xmlDoc.DocumentElement.InnerXml);
```

```
Console.WriteLine("OuterXml: " + xmlDoc.DocumentElement.OuterXml);
Console.ReadKey();
```

We worked with attributes in the previous chapter, but here is another example:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml("<user name=\"John Doe\" age=\"42\"></user>");
if(xmlDoc.DocumentElement.Attributes["name"] != null)
     Console.WriteLine(xmlDoc.DocumentElement.Attributes["name"].Value);
if(xmlDoc.DocumentElement.Attributes["age"] != null)
     Console.WriteLine(xmlDoc.DocumentElement.Attributes["age"].Value);
Console.ReadKey();
```

1.14.5. Using XPath with the XmlDocument class

In a previous chapter, we used the XmlDocument class to get out information from an XML file. We did it by using a range of calls to the ChildNodes property, which was somewhat simple because the example was very simple. It didn't do much good for the readability of our code though, so in this chapter we will look at another approach, which is definitely more powerful and yet easier to read and maintain. The technology we will be using for this is called XPath and is maintained by the same organization which created the XML standard. XPath is actually an entire query language, with lots possibilities, but since this is not an XPath tutorial, we will only look into some basic queries. However, even in its simplest forms, XPath is still powerful, as you will see in the following examples.

The XmlDocument class has several methods which takes an XPath query as a parameter and then returns the resulting XmlNode(s). In this chapter we will look into two methods: The SelectSingleNode() method, which returns a single XmlNode based on the provided XPath query, and the SelectNodes() method, which returns a XmlNodeList collection of XmlNode objects based on the provided XPath query.

We will try both of the above mentioned methods, but instead of using the currency information XML we tested in previous chapters, we will now try a new XML source. RSS feeds are essentially XML documents built in a specific way, to allow for a load of different news readers to parse and show the same information in their own way.

We will use an RSS feed from CNN, located at http://rss.cnn.com/rss/edition_world.rss, with news from across the world. If you open it in your browser, your browser may render this in a nice way, allowing you to get an overview of the feed and subscribe to it, but don't get fooled by that: Under the hood, it's just XML, which you will see if you do a "View source" in your browser. You will see that the root element is called "rss". The rss element usually has one or several "channel" elements, and within this element, we can find information about the feed as well as the "item" nodes, which are the news items we usually want.

In the following example, we will use the SelectSingleNode() method to get the title of the feed. If you look at the XML, you will see that there is a <title> element as a child element of the <channel> element, which is then a child element of the <rss> element, the root. That query can be described like this in XPath:

//rss/channel/title

We simply write the names of the element we're looking for, separated with a forward-slash (/), which states that the element should be a child to the element before the preceeding forward-slash. Using this XPath is as simple as this:

```
using System;
using System.Text;
using System.Xml;

namespace ParsingXml
{
    class Program
```

```
     {
          static void Main(string[] args)
          {
               XmlDocument xmlDoc = new XmlDocument();
               xmlDoc.Load("http://rss.cnn.com/rss/edition_world.rss");
               XmlNode titleNode = xmlDoc.SelectSingleNode(
"//rss/channel/title");
               if(titleNode != null)
                    Console.WriteLine(titleNode.InnerText);
               Console.ReadKey();
     }
     }
}
```

We use the SelectSingleNode() method to locate the <title> element, which simply takes our XPath as a string parameter. We then check to make sure that it returned a result, and if it did, we print the InnerText of the located node, which should be the title of the RSS feed.

In the next example, we will use the SelectNodes() method to find all the item nodes in the RSS feed and then print out information about them:

```
using System;
using System.Text;
using System.Xml;

namespace ParsingXml
{
     class Program
     {
          static void Main(string[] args)
          {
               XmlDocument xmlDoc = new XmlDocument();
               xmlDoc.Load("http://rss.cnn.com/rss/edition_world.rss");
               XmlNodeList itemNodes = xmlDoc.SelectNodes(
"//rss/channel/item");
               foreach(XmlNode itemNode in itemNodes)
               {
                    XmlNode titleNode = itemNode.SelectSingleNode(
"title");
                    XmlNode dateNode = itemNode.SelectSingleNode(
"pubDate");
```

```
                    if((titleNode != null) && (dateNode != null))
                         Console.WriteLine(dateNode.InnerText + ": " +
titleNode.InnerText);
               }
               Console.ReadKey();
     }
     }
}
```

The SelectNodes() method takes an XPath query as a string, just like we saw in the previous example, and then it returns a list of XmlNode objects in a XmlNodeList collection. We iterate through it with a foreach loop, and from each of the item nodes, we ask for a child node called title and pubDate (published date) using the SelectSingleNode() directly on the item node. If we get both of them, we print out the date and the title on the same line and then move on.

In our example, we wanted two different values from each item node, which is why we asked for the item nodes and then processed each of them. However, if we only needed the e.g. the titles of each item, we could change the XPath query to something like this:

//rss/channel/item/title

It will match each title node in each of the item nodes. Here's the query with some C# code to make it all happen:

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load("http://rss.cnn.com/rss/edition_world.rss");
XmlNodeList titleNodes = xmlDoc.SelectNodes("//rss/channel/item/title");
foreach(XmlNode titleNode in titleNodes)
     Console.WriteLine(titleNode.InnerText);
Console.ReadKey();
```

1.14.6. Writing XML with the XmlWriter class

In previous chapters we have dealt with reading XML, but now it's time to start writing it as well. Since XML is simply text, you could just start writing out XML tags to a file and give it an xml extension, but it's more easy and safer to let the .NET framework handle it, and just like with reading XML, there are at least two different options: The XmlWriter approach and the XmlDocument approach. This article will focus on the first approach, and then we'll look into writing XML with the XmlDocument in the next chapter.

The difference between the two is once again mostly related to memory consumption - XmlWriter uses less memory than XmlDocument, which is only an issue if you write very big files though. Another important difference is that when using XmlDocument, you can read an existing file, manipulate it and then write back the changes. With XmlWriter, you will have to write the entire document from scratch each time. This is not necessarily a problem though, so as always, it really comes down to your situation as well as your personal preferences.

Here's an example of writing XML using the XmlWriter class:

```
using System;
using System.Text;
using System.Xml;

namespace WritingXml
{
     class Program
     {
          static void Main(string[] args)
          {
               XmlWriter xmlWriter = XmlWriter.Create("test.xml");

               xmlWriter.WriteStartDocument();
               xmlWriter.WriteStartElement("users");

               xmlWriter.WriteStartElement("user");
               xmlWriter.WriteAttributeString("age", "42");
               xmlWriter.WriteString("John Doe");
               xmlWriter.WriteEndElement();

               xmlWriter.WriteStartElement("user");
               xmlWriter.WriteAttributeString("age", "39");
               xmlWriter.WriteString("Jane Doe");

               xmlWriter.WriteEndDocument();
               xmlWriter.Close();
```

```
            }
        }
}
```

The above code will create the following XML:

```
<users>
 <user age="42">John Doe</user>
 <user age="39">Jane Doe</user>
</users>
```

So, we start by creating an instance of the XmlWriter class. It takes at least one parameter, in this case the path to where the XML should be written, but comes in many variations for various purposes. The first thing we should do is to call the WriteStartDocument() method. After that, we write a start element called "users". The XmlWriter will translate this into <users>. Before closing it, we write another start element, "user", which will then become a child of "users". We then proceed to adding an attribute (age) to the element, using the WriteAttributeString() method, and then we write the inner text of the element, by calling the WriteString() method. We then make sure to close the first "user" element with a call to the WriteEndElement() method.

This process is repeated to add another user, except that we don't call WriteEndElement() like we did before. In fact, it should be called twice, since we have an open "users" element too, but the XmlWriter will do that for us when we call the WriteEndDocument() method.

To get the XmlWriter to write our data to the disk, we call the Close() method. You can now open the file, "test.xml", in the same directory where the EXE file of your project is, usually in the bin\debug directory.

And that's really all it takes to write a simple XML file. In the next chapter, we will do the same, but using the XmlDocument class.

1.14.7. Writing XML with the XmlDocument class

In the previous chapter, we wrote XML using the XmlWriter class. However, for some situations, especially when doing updates of existing XML, using the XmlDocument class can come in handy. You should be aware of the higher memory consumption though, mainly for large XML documents. Here is some code:

```
using System;
using System.Text;
using System.Xml;
using System.Xml.Serialization;

namespace WritingXml
{
     class Program
     {
          static void Main(string[] args)
          {
               XmlDocument xmlDoc = new XmlDocument();
               XmlNode rootNode = xmlDoc.CreateElement("users");
               xmlDoc.AppendChild(rootNode);

               XmlNode userNode = xmlDoc.CreateElement("user");
               XmlAttribute attribute = xmlDoc.CreateAttribute("age");
               attribute.Value = "42";
               userNode.Attributes.Append(attribute);
               userNode.InnerText = "John Doe";
               rootNode.AppendChild(userNode);

               userNode = xmlDoc.CreateElement("user");
               attribute = xmlDoc.CreateAttribute("age");
               attribute.Value = "39";
               userNode.Attributes.Append(attribute);
               userNode.InnerText = "Jane Doe";
               rootNode.AppendChild(userNode);

               xmlDoc.Save("test-doc.xml");
          }
     }
}
```

And the resulting XML:

```
<users>
 <user age="42">John Doe</user>
 <user age="39">Jane Doe</user>
</users>
```

As you can see, this is a bit more object oriented than the XmlWriter approach, and it does require a bit more code. What we do is that we instantiate an XmlDocument object, which we will use to create both new elements and attributes, using the CreateElement() and CreateAttribute() methods. Each time we do that, we append the elements, either directly to the document, or to another element. You can see this in the example, where the root element ("users") is appended directly to the document, while the user elements are appended to the root element. Attributes are of course appended to the elements where they belong, using the Append() method on the Attributes property. The entire XML document is written to the disk on the last line, where we use the Save() method.

Now, as already mentioned, this requires a bit more code than when using the XmlWriter, but imagine a situation where you just need to go into an existing XML document and change a few values. Using the XmlWriter approach, you would have to first read all the information using an XmlReader, store it, change it, and then write the entire information back using the XmlWriter. Because the XmlDocument holds everything in memory for you, updating an existing XML file becomes a lot simpler. The following example opens the "test-doc.xml" file that we created in the previous chapter and increases every user's age by one. See how easy it is:

```
using System;
using System.Text;
using System.Xml;
using System.Xml.Serialization;

namespace WritingXml
{
     class Program
     {
          static void Main(string[] args)
          {
               XmlDocument xmlDoc = new XmlDocument();
               xmlDoc.Load("test-doc.xml");
               XmlNodeList userNodes = xmlDoc.SelectNodes("//users/user"
);
               foreach(XmlNode userNode in userNodes)
               {
                    int age = int.Parse(userNode.Attributes["age"
].Value);
                    userNode.Attributes["age"].Value = (age +
```

```
1).ToString();
                }
                xmlDoc.Save("test-doc.xml");
    }
    }
}
```

We load the XML file and ask for all the <user> nodes. We then iterate through them, read the age attribute into an integer variable, and then we write the value back to the node and attribute, after increasing the value by 1. At last, we save the document back to the same file and if you open it up, you will see that our users all just had a birthday. Pretty cool!

1.15. C# 3.0

1.15.1. Introduction to C# 3.0

With the release of Microsoft .NET framework 3.5 and Visual Studio 2008, codenamed "Orcas", a bunch of new features were added to the C# language, under the name C# version 3. In the following chapters, I will show you some of the cool, new stuff that Microsoft added, in a constant effort to make it easier and faster for all their programmers to write good code.

Please be aware that you will need at least version 3.5 of the framework installed, as well as a version 2008 of your favourite IDE, either Visual Studio or one of the Express versions, to compile and take advantage of the examples.

1.15.2. Automatic properties

A real pain in the neck for all programmers writing object oriented code has always been declaring public properties for all the private fields. This is a lot of tedious work, especially because almost all properties will be a simple get and set mapping to the private field, without anything clever added, like this:

```
private string name;

public string Name
{
     get { return name; }
     set { name = value; }
}
```

With a simple property like that, we could pretty much just as well have declared the field as public and used it directly, instead of adding the extra layer of a property. However, the guidelines of OOP tells us to do it this way, and most of us resists the temptation of doing it the easy way. With C# 3.0 we don't have to deal with this dilemma anymore! The above example can now be written like this instead:

```
public string Name
{
     get;
     set;
}
```

Or using even less space, like this:

```
public string Name { get; set; }
```

No field declaration, and no code to get and set the value of the field. All of that is handled automatically by the compiler, which will automatically create a private field and populate the get and set method with the basic code required to read and write the field. From the outside, this will look just like a regular property, but you saved a lot of extra keystrokes and your class will be less bloated. Of course, you can still use the old way, as shown in our first example - this is simply an extra feature that you can use, if you feel like it.

1.15.3. Object Initializers

With C# 3.0, initializing both objects and collections have become much easier. Consider this simple Car class, where we use the automatic properties described in a previous chapter:

```
class Car
{
     public string Name { get; set; }
     public Color Color { get; set; }
}
```

Now, in C# 2.0, we would have to write a piece of code like this to create a Car instance and set its properties:

```
Car car = new Car();
car.Name = "Chevrolet Corvette";
car.Color = Color.Yellow;
```

It's just fine really, but with C# 3.0, it can be done a bit more cleanly, thanks to the new object initializer syntax:

```
Car car = new Car { Name = "Chevrolet Corvette", Color = Color.Yellow };
```

As you can see, we use a set of curly brackets after instantiating a new Car object, and within them, we have access to all the public properties of the Car class. This saves a bit of typing, and a bit of space as well. The cool part is that it can be nested too. Consider the following example, where we add a new complex property to the Car class, like this:

```
class Car
{
     public string Name { get; set; }
     public Color Color { get; set; }
     public CarManufacturer Manufacturer { get; set; }
}

class CarManufacturer
{
     public string Name { get; set; }
     public string Country { get; set; }
}
```

To initialize a new car with C# 2.0, we would have to do something like this:

```
Car car = new Car();
car.Name = "Corvette";
car.Color = Color.Yellow;
car.Manufacturer = new CarManufacturer();
car.Manufacturer.Name = "Chevrolet";
car.Manufacturer.Country = "USA";
```

With C# 3.0, we can do it like this instead:

```
Car car = new Car {
               Name = "Chevrolet Corvette",
               Color = Color.Yellow,
               Manufacturer = new CarManufacturer {
                    Name = "Chevrolet",
                    Country = "USA"
               }
          };
```

Or in case you're not too worried about readability, like this:

```
Car car = new Car { Name = "Chevrolet Corvette", Color = Color.Yellow,
Manufacturer = new CarManufacturer { Name = "Chevrolet", Country = "USA"
} };
```

Just like with the automatic properties, this is syntactical sugar - you can either use it, or just stick with the old, fashioned way of doing things.

1.15.4. Collection Initializers

Just like C# 3.0 offers a new way of initializing objects, a new syntax for initializing a list with a specific set of items added to it, has been included. We can use the Car class from the last chapter:

```
class Car
{
     public string Name { get; set; }
     public Color Color { get; set; }
}
```

If we wanted to create a list to contain a range of cars, we would have to do something like this with C# 2.0:

```
Car car;
List<Car> cars = new List<Car>();

car = new Car();
car.Name = "Corvette";
car.Color = Color.Yellow;
cars.Add(car);

car = new Car();
car.Name = "Golf";
car.Color = Color.Blue;
cars.Add(car);
```

Using object initializers, we could do it a bit shorter:

```
List<Car> cars = new List<Car>();
cars.Add(new Car { Name = "Corvette", Color = Color.Yellow });
cars.Add(new Car { Name = "Golf", Color = Color.Blue});
```

However, it can be even simpler, when combined with collection initializers:

```
List<Car> cars = new List<Car>
{
new Car { Name = "Corvette", Color = Color.Yellow },
     new Car { Name = "Golf", Color = Color.Blue}
};
```

Or in the one-line version, which does exactly the same:

```
List<Car> cars = new List<Car> { new Car { Name = "Corvette", Color =
```

```
Color.Yellow }, new Car { Name = "Golf", Color = Color.Blue} };
```

10 lines of code has been reduced to a single, albeit a bit long, line, thanks to object and collection initializers.

1.15.5. Extension Methods

Another cool feature of C# 3.0 is Extension Methods. They allow you to extend an existing type with new functionality, without having to sub-class or recompile the old type. For instance, you might like to know whether a certain string was a number or not. The usual approach would be to define a function and then call it each time, and once you got a whole lot of those kind of functions, you would put them together in a utility class, like this:

```
public class MyUtils
{
     public static bool IsNumeric(string s)
     {
          float output;
          return float.TryParse(s, out output);
     }
}
```

Now you could check a string by executing a line of code like this:

```
string test = "4";
if (MyUtils.IsNumeric(test))
     Console.WriteLine("Yes");
else
     Console.WriteLine("No");
```

However, with Extension Methods, you can actually extend the String class to support this directly. You do it by defining a static class, with a set of static methods that will be your library of extension methods. Here is an example:

```
public static class MyExtensionMethods
{
     public static bool IsNumeric(this string s)
     {
          float output;
          return float.TryParse(s, out output);
     }
}
```

The only thing that separates this from any other static method, is the "this" keyword in the parameter section of the method. It tells the compiler that this is an extension method for the string class, and that's actually all you need to create an extension method. Now, you can call the IsNumeric() method directly on strings, like this:

```
string test = "4";
if (test.IsNumeric())
     Console.WriteLine("Yes");
else
     Console.WriteLine("No");
```

1.16. File handling

1.16.1. Reading and writing files

In this chapter, we will look into reading and writing simple files with C#. Fortunately for us, C# makes it very easy. The File class, from the Syste.IO namespace comes with pretty much everything we could possibly want, making it very easy to do simple reading and writing of a file.

In our first example, we will construct an extremely minimalistic text editor. In fact, it is so simple that we can only read one file and then write new content to it, and only a single line of text at a time. But it shows just how easy it is to use the File class:

```
using System;
using System.IO;

namespace FileHandlingArticleApp
{
     class Program
     {
          static void Main(string[] args)
          {
               if(File.Exists("test.txt"))
               {
                    string content = File.ReadAllText("test.txt");
                    Console.WriteLine("Current content of file:");
                    Console.WriteLine(content);
               }
               Console.WriteLine("Please enter new content for the
file:");
               string newContent = Console.ReadLine();
               File.WriteAllText("test.txt", newContent);
          }
     }
}
```

You will notice that we use the File class in three places: We use it to check if our file exists, we use the ReadAllText() method to read the content of the file, and we use the WriteAllText() method to write new content to the file. You will notice that I'm not using absolute paths, but just a simple filename. This will place the file in the same directory as the executable file, which is fine for now. Other than that, the example should be easy enough to understand: We check for the file, if it exists, we read its content and output it to the console. Then we prompt the user for new content, and once we have it, we write it to the

file. Obviously that will overwrite the previous content, but for now, that's just fine. We could however use the AppendAllText method instead. Try changing the WriteAllText line to this instead:

```
File.AppendAllText("test.txt", newContent);
```

If you run it, you will see that the new text is added to the existing text instead of overwriting it. As simple as that. But we still get only one line of text per execution of our application. Let's be a bit creative and change that. Replace the last lines in our example with this:

```
Console.WriteLine("Please enter new content for the file - type exit and
press enter to finish editing:");
string newContent = Console.ReadLine();
while(newContent != "exit")
{
     File.AppendAllText("test.txt", newContent + Environment.NewLine);
     newContent = Console.ReadLine();
}
```

As you can see, we instruct the user to enter the word exit when they wish to stop editing the file, and until they do just that, we append the user input to the file and then prompt for a new line. We also append a newline character, the Environment.NewLine, to make it look like actual lines of text.

However, instead of writing to the file each time, a more pretty solution would probably look something like this:

```
Console.WriteLine("Please enter new content for the file - type exit and
press enter to finish editing:");
using(StreamWriter sw = new StreamWriter("test.txt"))
{
     string newContent = Console.ReadLine();
     while(newContent != "exit")
     {
          sw.Write(newContent + Environment.NewLine);
          newContent = Console.ReadLine();
     }
}
```

The usage of Streams is a bit out of the scope of this chapter, but the cool thing about it in this example is that we only open the file once, and then write the changes to it once we're satisfied. In this case, we're taking advantage of the using() statement of C#, which ensures that the file reference is closed once it goes out of scope, which is when it's block of { } is done. If you don't use the using() statement, you will have to manually call the Close() method on the StreamWriter instance.

1.16.2. Manipulating files and directories

In the previous chapter, we looked into reading and writing text with simple files. We used the File class for this, but it does a lot more than just reading and writing. When combined with the Directory class, we can perform pretty much any filesystem operation, like renaming, moving, deleting and so on.

This chapter will provide numerous examples of doing just those things. The explanations will be limited, since the methods used are pretty simple and easy to use. You should be aware of two things: First of all, make sure that you import the System.IO namespace like this:

```
using System.IO;
```

Also, be aware that we are not doing any exception handling here. We will check that files and directories exist before using it, but there's no exception handling, so in case anything goes wrong, the application will crash. Exception handling is generally a good idea when doing IO operations. For more information, please read the exception handling chapter in this tutorial.

In all of the examples, we use simple file and directory names, which will have to exist in the same directory as the generated EXE file of your application. In the project settings you can see where your EXE file is generated to.

1.16.2.1. Deleting a file

```
if(File.Exists("test.txt"))
{
     File.Delete("test.txt");
     if(File.Exists("test.txt") == false)
          Console.WriteLine("File deleted...");
}
else
     Console.WriteLine("File test.txt does not yet exist!");
Console.ReadKey();
```

1.16.2.2. Deleting a directory

```
if(Directory.Exists("testdir"))
{
     Directory.Delete("testdir");
     if(Directory.Exists("testdir") == false)
          Console.WriteLine("Directory deleted...");
}
else
     Console.WriteLine("Directory testdir does not yet exist!");
Console.ReadKey();
```

If testdir is not empty, you will receive an exception. Why? Because this version of Delete() on the Directory class only works on empty directories. It's very easy to change though:

```
Directory.Delete("testdir", true);
```

The extra parameter makes sure that the Delete() method is recursive, meaning that it will traverse subdirectories and delete the files of it, before deleting the directories.

1.16.2.3. Renaming a file

```
if(File.Exists("test.txt"))
{
     Console.WriteLine("Please enter a new name for this file:");
     string newFilename = Console.ReadLine();
     if(newFilename != String.Empty)
     {
          File.Move("test.txt", newFilename);
          if(File.Exists(newFilename))
          {
               Console.WriteLine("The file was renamed to " +
newFilename);
               Console.ReadKey();
          }
     }
}
```

You will notice that we use the Move() method to rename the file. Why not a Rename() method? Because there are no such method, since moving and renaming is essentially the same thing.

1.16.2.4. Renaming a directory

Doing the same thing with a directory is just as easy:

```
if(Directory.Exists("testdir"))
{
     Console.WriteLine("Please enter a new name for this directory:");
     string newDirName = Console.ReadLine();
     if(newDirName != String.Empty)
     {
          Directory.Move("testdir", newDirName);
          if(Directory.Exists(newDirName))
          {
               Console.WriteLine("The directory was renamed to " +
```

```
newDirName);
                Console.ReadKey();
          }
     }
}
```

1.16.2.5. Creating a new directory

Creating a brand new directory is easy too - just use the **CreateDirectory()** method on the Directory class, like in this example:

```
Console.WriteLine("Please enter a name for the new directory:");
string newDirName = Console.ReadLine();
if(newDirName != String.Empty)
{
     Directory.CreateDirectory(newDirName);
     if(Directory.Exists(newDirName))
     {
          Console.WriteLine("The directory was created!");
          Console.ReadKey();
     }
}
```

1.16.2.6. Reading & Writing Files

As a final example, I will show you how the File class makes it very easy to read from and write to a file. This can be done in a whole lot of ways with C#, but the Read* and Write* methods, found on the File class, are probably the easiest ones to use. There are three versions: **WriteAllBytes()**, **WriteAllLines()** and **WriteAllText()**, with corresponding Read methods. The simplest one to use is the last one, which will take a simple string as its input. Let me illustrate how they work with a simple example:

```
string fileContents = "John Doe & Jane Doe sitting in a tree...";
File.WriteAllText("test.txt", fileContents);

string fileContentsFromFile = File.ReadAllText("test.txt");
Console.WriteLine(fileContentsFromFile);
```

Notice how few lines of code I need to write something to a file and then read it back! The first parameter to both methods is the path to where the text should be written to and read from. Normally, you would specify a full path here, but to make the example more clear, I only specify a filename, resulting in the file being written to the same directory as the EXE file.

1.16.2.7. Summary

As you can see, the File and Directory classes are a huge help when you need to work with files and directories. They can help you do most of the operations needed, and if you need to do even more advanced stuff, these classes and their methods will usually serve as great building blocks to expand on.

1.16.3. File and directory information

The File and the Directory classes, which we have used in the previous couple of chapters, are great for direct file and directory manipulation. However, sometimes we wish to get information on them instead, and once again, the System.IO namespace comes to our rescue: The FileInfo and DirectoryInfo classes. In this chapter, we will look into some of the ways to use these two classes.

1.16.3.1. The FileInfo class

First, let's explore a simple way to use the FileInfo class.

```
static void Main(string[] args)
{
     FileInfo fi = new
FileInfo(System.Reflection.Assembly.GetExecutingAssembly().Location);
     if(fi != null)
          Console.WriteLine(String.Format("Information about file: {0},
{1} bytes, last modified on {2} - Full path: {3}", fi.Name, fi.Length,
fi.LastWriteTime, fi.FullName));
     Console.ReadKey();
}
```

We create a new instance of the FileInfo class. It takes one parameter, which is the path to the file we want information about. We could have just specified a filename, but for fun, I thought it would be cool to get information about the actual application we are working on, that is, the EXE file that our project is compiled into. Since we don't have access to the Application project in a Console application (it's part of the WinForms assembly), we use a bit of Reflection to get the path to the current assembly. This is all way out of the scope of this particular chapter, but at least now you know.

Once we have a FileInfo instance, we output all sorts of information about it. Try running the application and you will see. All very nice and easy, and if you look at the FileInfo class, you will see that it offers even more information, as well as shortcuts to the methods found on the File class - and why not? We have a reference to the file anyway with the FileInfo instance, so we might as well get the same options as on the File class.

1.16.3.2. The DirectoryInfo class

Now, information about a single file is just fine, but using the DirectoryInfo class, we can get information about all files and directories within a directory, which is obviously a very common scenario. Let me show you with a simple example:

```
DirectoryInfo di = new
DirectoryInfo(Path.GetDirectoryName(System.Reflection.Assembly.GetExecut
ingAssembly().Location));
if(di != null)
```

```
{
     FileInfo[] subFiles = di.GetFiles();
     if(subFiles.Length > 0)
     {
          Console.WriteLine("Files:");
          foreach(FileInfo subFile in subFiles)
          {
               Console.WriteLine(" " + subFile.Name + " (" +
subFile.Length + " bytes)");
          }
     }
     Console.ReadKey();
}
```

Instead of a FileInfo instance, we create a DirectoryInfo instance. We use the same trick to get the path of the executing file, and then the GetDirectoryName() method from the Path class, to get the directory part of the path only. We use the GetFiles() method to get an array of FileInfo instances, each representing a file in the directory. We then loop through it, printing out each filename and size.

Perhaps we want the directories as well. It's just as easy:

```
DirectoryInfo[] subDirs = di.GetDirectories();
if(subDirs.Length > 0)
{
     Console.WriteLine("Directories:");
     foreach(DirectoryInfo subDir in subDirs)
     {
          Console.WriteLine(" " + subDir.Name);
     }
}
```

In some situations, you might only want files or directories with a specific name or file extension. Fortunately, FileInfo and DirectoryInfo has some pretty good support for that as well.

This will give us all files in the directory with a .exe extension:

```
FileInfo[] subFiles = di.GetFiles("*.exe");
```

This will give us all the directories which have the word "test" somewhere in the name:

```
DirectoryInfo[] subDirs = di.GetDirectories("*test*");
```

We can even find both files and directories recursively, which means that it will search in subdirectories of subdirectories of.... the originial directory:

```
FileInfo[] subFiles = di.GetFiles("*.exe", SearchOption.AllDirectories);
```

To only search the toplevel directory, the code would have to look like this:

```
FileInfo[] subFiles = di.GetFiles("*.exe",
SearchOption.TopDirectoryOnly);
```

1.16.3.3. Summary

Using the FileInfo and DirectoryInfo classes, we can easily discover information about the file system on the current computer.

1.17. Data Streams

1.18. Reflection

1.18.1. Reflection introduction

Wikipedia says that "In computer science, reflection is the process by which a computer program can observe and modify its own structure and behaviour". This is exactly how Reflection in C# works, and while you may not realize it at this point, being able to examine and change information about your application during runtime, offers huge potential. Reflection, which is both a general term, as well as the actual name of the reflection capabilities in C#, works very, very well, and it's actually not that hard to use. In the next couple of chapters, we will go more into depth about how it works and provide you with some cool examples, which should show you just how useful Reflection is.

However, to get you started and hopefully interested, here is a small example. It solves a question that I have seen from many newcomers to any programming language: How can I change the value of a variable during runtime, only by knowing its name? Have a look at this small demo application for a solution, and read the next chapters for an explanation of the different techniques used.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
     class Program
     {
          private static int a = 5, b = 10, c = 20;

          static void Main(string[] args)
          {
               Console.WriteLine("a + b + c = " + (a + b + c));
               Console.WriteLine("Please enter the name of the variable
that you wish to change:");
               string varName = Console.ReadLine();
               Type t = typeof(Program);
               FieldInfo fieldInfo = t.GetField(varName,
BindingFlags.NonPublic | BindingFlags.Static);
               if(fieldInfo != null)
```

```
                {
                    Console.WriteLine("The current value of " +
fieldInfo.Name + " is " + fieldInfo.GetValue(null) + ". You may enter a
new value now:");
                    string newValue = Console.ReadLine();
                    int newInt;
                    if(int.TryParse(newValue, out newInt))
                    {
                        fieldInfo.SetValue(null, newInt);
                        Console.WriteLine("a + b + c = " + (a + b +
c));
                    }
                    Console.ReadKey();
                }
            }
        }
}
```

Try running the code and see how it works. Besides the lines where we use the actual Reflection, it's all very simple. Now, go to the next chapter for some more theory on how it works.

1.18.2. The right Type

The Type class is the foundation of Reflection. It serves as runtime information about an assembly, a module or a type. Fortunately, obtaining a reference to the Type of an object is very simply, since every class that inherits from the Object class, has a GetType() method. If you need information about a non-instantiated type, you may use the globally available typeof() method, which will do just that. Consider the following examples, where we use both approaches:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
     class Program
     {
          static void Main(string[] args)
          {
               string test = "test";
               Console.WriteLine(test.GetType().FullName);
               Console.WriteLine(typeof(Int32).FullName);
               Console.ReadKey();
          }
     }
}
```

We use the GetType() method on our own variable, and then we use the typeof() on a known class, the Int32. As you can see, the result in both cases is a Type object, for which we can read the FullName property.

At some point, you might even only have the name of the type that you're looking for. In that case, you will have to get a reference to it from the proper assembly. In the next example, we get a reference to the executing assembly, that is, the assembly from where the current code is being executed from, and then we list all of it's types:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Assembly assembly = Assembly.GetExecutingAssembly();
            Type[] assemblyTypes = assembly.GetTypes();
            foreach(Type t in assemblyTypes)
                Console.WriteLine(t.Name);
            Console.ReadKey();
        }
    }

    class DummyClass
    {
        //Just here to make the output a tad less boring :)
    }
}
```

The output will be the name of the two declared classes, Program and DummyClass, but in a more complex application, the list would probably be more interesting. In this case, we only get the name of the type, but obviously we would be able to do a lot more, with the Type reference that we get. In the next chapters, I will show you a bit more on what we can do with it.

1.18.3. Instantiating a class

So far, we have worked with .NET types or objects already instantiated. But with Reflection, we can actually do the instantiation at runtime as well, knowing the name of the class we wish to instantiate. There are several ways of doing it, but I prefer getting a reference to the constructor that I wish to use, invoke it, and then use the returned value as my instance. Here's an example of doing just that. Code first, then I will explain it all:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;

namespace ReflectionTest
{
     class Program
     {
          static void Main(string[] args)
          {
               Type testType = typeof(TestClass);
               ConstructorInfo ctor =
testType.GetConstructor(System.Type.EmptyTypes);
               if(ctor != null)
               {
                    object instance = ctor.Invoke(null);
                    MethodInfo methodInfo = testType.GetMethod(
"TestMethod");
                    Console.WriteLine(methodInfo.Invoke(instance, new
object[] { 10 }));
               }
               Console.ReadKey();
          }
     }

     public class TestClass
     {
          private int testValue = 42;

          public int TestMethod(int numberToAdd)
          {
               return this.testValue + numberToAdd;
          }
```

```
    }
}
```

I have defined a simple class for testing this, called TestClass. It simply contains a private field and a public method. The method returns the value of the private property, with the value of the parameter added to it. Now, what we want is to create a new instance of this TestClass, call the TestMethod and output the result to the console.

In this example, we have the luxury of being able to use the typeof() directly on the TestClass, but at some point, you may have to do it solely by using the name of the desired class. In that case, you can get a reference to it through the assembly where it is declared, as demonstrated in the chapter about Type.

So, with a Type reference to the class, we ask for the default constructor by using the GetConstructor() method, passing System.Type.EmptyTypes as a parameter. In case we wanted a specific constructor, we would have to provide an array of Type's, each defining which parameter the constructor we were looking for, would take.

Once we have a reference to the constructor, we simply call the Invoke() method to create a new instance of the TestClass class. We pass null as the parameter to Invoke(), since we're not looking to specify any parameters. We use the GetMethod(), along with the name of the method we want, to get the TestMethod() function, and then we once again use the magic of the Invoke() method to call this function. This time we need to specify a parameter, in the form of an array of objects. We do that on-the-fly, specifying the number 10 as the only parameter we need, and we then output the result of the method invocation. All of this through the magic of Reflection!

1.18.4. A Reflection based settings class

Okay, so I thought that I would end this part of the tutorial about Reflection, with a cool and useful example. It's a bit bigger than the usual examples here at the site, but hopefully you will find it really useful. It uses a bunch of the stuff that we have looked into during the last couple of chapters, so hopefully you can keep up.

A common scenario when creating any sort of application, is the desire to save the users settings. When you get several settings, you will probably create a Settings class, which will handle loading and saving of the desired settings. Each time you need a new setting in your Settings class, you will have to update the Load() and Save() methods, to include this new setting. But hey, why not let the Settings class discover its own properties and then load and save them automatically? With Reflection, it's quite easy, and if you have read the other chapters in the Reflection section of this tutorial, you should be able to understand the following example.

To make it fit better into a small example, I am saving information about a person instead of application settings, but hopefully you will get the general idea anyway. Please be aware that using Reflection WILL be slower than reading and writing known properties manually, so you should consider when to use it and when to opt for a faster approach! Also, in our example, we use a simple text file for storing even simpler values, only separated by a | (pipe character). If you're using this for real world stuff, you will probably want a better format for your data, perhaps XML. And of course, there is not much error handling, so you should probably add some of that as well.

Okay, let's get started. First, our Person class, which you can simply rename to Settings or something like that, to make it more useful to you:

```
public class Person
{
     private int age = -1;
     private string name = String.Empty;

     public void Load()
     {
          if(File.Exists("settings.dat"))
          {
               Type type = this.GetType();

               string propertyName, value;
               string[] temp;
               char[] splitChars = new char[] { '|' };
               PropertyInfo propertyInfo;

               string[] settings = File.ReadAllLines("settings.dat");
               foreach(string s in settings)
```

```
                {
                    temp = s.Split(splitChars);
                    if(temp.Length == 2)
                    {
                        propertyName = temp[0];
                        value = temp[1];
                        propertyInfo = type.GetProperty(propertyName);
                        if(propertyInfo != null)
                            this.SetProperty(propertyInfo, value);
                    }
                }
            }
        }

        public void Save()
        {
            Type type = this.GetType();
            PropertyInfo[] properties = type.GetProperties();
            TextWriter tw = new StreamWriter("settings.dat");
            foreach(PropertyInfo propertyInfo in properties)
            {
                tw.WriteLine(propertyInfo.Name + "|" +
propertyInfo.GetValue(this, null));
            }
            tw.Close();
        }

        public void SetProperty(PropertyInfo propertyInfo, object value)
        {
            switch(propertyInfo.PropertyType.Name)
            {
                case "Int32":
                    propertyInfo.SetValue(this, Convert.ToInt32(value),
null);
                    break;
                case "String":
                    propertyInfo.SetValue(this, value.ToString(), null);
                    break;
            }
        }
```

```
        public int Age
        {
            get { return age; }
            set { age = value; }
        }

        public string Name
        {
            get { return name; }
            set { name = value; }
        }
}
```

Okay, there's a lot of stuff, I know. But I will help you through the entire class. First of all, we have a couple of private fields, for holding information about our person. In the bottom of the class, we have the corresponding public properties which uses the private fields of course.

We also have a Load() method. It looks for the file settings.dat, and if it exists, it reads the entire file and places each line of it in an array of strings. Now, we run through each setting line, and splits it up into two parts: A property name and a value part. If both is present, we simply use the Type object to get the property with the property name, and then we set the value for it, using our own SetProperty method.

The SetProperty() method looks at the type of the property about to be changed, and then acts correspondingly. Right now, it only supports integers and strings, but as you probably realize, extending this support would be quite easy.

The Save() method gets an array of PropertyInfo instances, one for each of the defined properties on the Person class, and then uses a TextWriter to write each property, and its value, to the data file.

Now we just need some code to use this class. This small application will try to load the person from the settings file, and if it doesn't succeed, the user is prompted for the information:

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person();
        person.Load();
        if((person.Age > 0) && (person.Name != String.Empty))
        {
            Console.WriteLine("Hi " + person.Name + " - you are " +
```

```
person.Age + " years old!");
          }
          else
          {
               Console.WriteLine("I don't seem to know much about you.
Please enter the following information:");
               Type type = typeof(Person);
               PropertyInfo[] properties = type.GetProperties();
               foreach(PropertyInfo propertyInfo in properties)
               {
                    Console.WriteLine(propertyInfo.Name + ":");
                    person.SetProperty(propertyInfo,
Console.ReadLine());
               }
               person.Save();
               Console.WriteLine("Thank you! I have saved your
information for next time.");
          }
          Console.ReadKey();
     }
}
```

Everything here is pretty trivial, except for the part where we ask the user for information. Once again, we use Reflection, to get all the public properties of the Person class, and then ask for each of them.

As a reader exercise, I suggest that you extend the Person class to include more information. Simply add more properties to it, and you will see that this information gets saved and loaded too.