|  |
| --- |
| EASJ Notes |
| Object-Oriented Programming With C# |
| A Gentle Introduction |

|  |
| --- |
| By Per Laursen  11-07-2017 |

[The Programming Process 5](#_Toc487969184)

[Software tools 7](#_Toc487969185)

[Microsoft Visual Studio 7](#_Toc487969186)

[ReSharper 8](#_Toc487969187)

[Git / GitHub 8](#_Toc487969188)

[Code organisation and Visual Studio basics 10](#_Toc487969189)

[Loading code into Visual Studio 10](#_Toc487969190)

[Code organisation 11](#_Toc487969191)

[Statements (and the cruel Master Syntax) 15](#_Toc487969192)

[Programming – fundamentals 19](#_Toc487969193)

[Types and variables 19](#_Toc487969194)

[Arithmetic 23](#_Toc487969195)

[A detour to Code Quality, part I 25](#_Toc487969196)

[Screen output and type conversions 28](#_Toc487969197)

[Logic 30](#_Toc487969198)

[The concept of functions 34](#_Toc487969199)

[Pre-OO programming 36](#_Toc487969200)

[Object-Oriented Programming - fundamentals 37](#_Toc487969201)

[The Object concept 37](#_Toc487969202)

[State and Behavior 37](#_Toc487969203)

[Public and private appearances 39](#_Toc487969204)

[The Class concept 41](#_Toc487969205)

[Using objects of an existing class 42](#_Toc487969206)

[A (very short) detour to Code Quality, part 2 46](#_Toc487969207)

[Final remarks on object creation 46](#_Toc487969208)

[Value types and Reference types 47](#_Toc487969209)

[Class definition elements 52](#_Toc487969210)

[Instance fields 53](#_Toc487969211)

[Properties 54](#_Toc487969212)

[Methods 59](#_Toc487969213)

[Constructors 62](#_Toc487969214)

[Class collaboration, and a bit about Abstraction 64](#_Toc487969215)

[Static – no object needed 66](#_Toc487969216)

[Programming – next level 69](#_Toc487969217)

[Conditional statements 69](#_Toc487969218)

[The if-statement 69](#_Toc487969219)

[The if-else statement 72](#_Toc487969220)

[The multi if-else statement 74](#_Toc487969221)

[The switch statement 76](#_Toc487969222)

[Repetition statements 78](#_Toc487969223)

[The while-loop 78](#_Toc487969224)

[The for-loop 82](#_Toc487969225)

[Debugging 84](#_Toc487969226)

[Data structures - fundamentals 87](#_Toc487969227)

[Arrays (and why you should not use them…) 87](#_Toc487969228)

[Lists 92](#_Toc487969229)

[Dictionary 95](#_Toc487969230)

[Enumerations 98](#_Toc487969231)

[Data structures – beyond List and Dictionary 100](#_Toc487969232)

[Run-time complexity 100](#_Toc487969233)

[Additional collection classes 104](#_Toc487969234)

[Recursion – iteration without loops 108](#_Toc487969235)

[Keeping your code DRY 112](#_Toc487969236)

[DRY and values 112](#_Toc487969237)

[DRY and instance fields 114](#_Toc487969238)

[DRY and methods 114](#_Toc487969239)

[DRY and classes (and a brief introduction to Inheritance) 117](#_Toc487969240)

[When does code become DRY? 120](#_Toc487969241)

[Object-oriented programming – Next level 122](#_Toc487969242)

[The has-a relationship (Composition) 122](#_Toc487969243)

[The is-a relationship (Inheritance) 122](#_Toc487969244)

[The inheritance mechanism 123](#_Toc487969245)

[The protected access level 124](#_Toc487969246)

[Constructors in derived classes 124](#_Toc487969247)

[Overriding methods 126](#_Toc487969248)

[Polymorphic behavior 127](#_Toc487969249)

[Calling base class mehods 129](#_Toc487969250)

[Abstract methods (and classes) 130](#_Toc487969251)

[Interfaces 131](#_Toc487969252)

[The Object class 133](#_Toc487969253)

[Exceptions 134](#_Toc487969254)

[Throwing and catching 135](#_Toc487969255)

[The finally block 137](#_Toc487969256)

[Rethrowing an exception 137](#_Toc487969257)

[Exceptions summary 138](#_Toc487969258)

[Generics – types as parameters 139](#_Toc487969259)

[Shortcomings of inheritance 139](#_Toc487969260)

[Using types as parameters 141](#_Toc487969261)

[Type constraints 143](#_Toc487969262)

[Type parameter variance 145](#_Toc487969263)

[The IComparable<T> and IComparer<T> interfaces 148](#_Toc487969264)

[Generic methods 152](#_Toc487969265)

[Functions as parameters 153](#_Toc487969266)

[A first attempt at parameterising code 153](#_Toc487969267)

[Lambda expressions 155](#_Toc487969268)

[Delegates 159](#_Toc487969269)

[Events 161](#_Toc487969270)

[GUI Development 164](#_Toc487969271)

[GUI and Object-Orientation 164](#_Toc487969272)

[Event-driven applications 164](#_Toc487969273)

[The duality of a GUI components 165](#_Toc487969274)

[What is XML (and XAML) 167](#_Toc487969275)

[XAML and Visual Studio – getting started 170](#_Toc487969276)

[Simple controls 175](#_Toc487969277)

[Button 175](#_Toc487969278)

[TextBlock 175](#_Toc487969279)

[TextBox 176](#_Toc487969280)

[Image 176](#_Toc487969281)

[Slider 176](#_Toc487969282)

[Layout controls 177](#_Toc487969283)

[Grid 177](#_Toc487969284)

[StackPanel 178](#_Toc487969285)

[Control properties 179](#_Toc487969286)

[Default properties 179](#_Toc487969287)

[Complex properties 180](#_Toc487969288)

[Attached properties 180](#_Toc487969289)

[Layout properties 180](#_Toc487969290)

[Using styles 181](#_Toc487969291)

[Data Binding 182](#_Toc487969292)

[Simple binding between GUI elements 182](#_Toc487969293)

[Data binding between GUI elements and model objects 183](#_Toc487969294)

[Collection Views and Data Binding 189](#_Toc487969295)

[The ListView control – getting started 189](#_Toc487969296)

[The ListView control – displaying items 191](#_Toc487969297)

[Defining a data template 193](#_Toc487969298)

[The ListView control – binding to SelectedItem 194](#_Toc487969299)

[The GridView 196](#_Toc487969300)

[Commands 197](#_Toc487969301)

[Deleting a domain object 197](#_Toc487969302)

[The ICommand interface 199](#_Toc487969303)

[The MVVM application architecture 205](#_Toc487969304)

[MVVM architecture – fundamental features 205](#_Toc487969305)

[MVVM – single domain object 206](#_Toc487969306)

[MVVM – collection of domain objects 212](#_Toc487969307)

[An Item view model class 212](#_Toc487969308)

[A Master view model class 213](#_Toc487969309)

[A Details view model class 215](#_Toc487969310)

[A MasterDetailsViewModel class 217](#_Toc487969311)

[Adding a Delete command to an MVVM-based view 221](#_Toc487969312)

[Generalising the classes 223](#_Toc487969313)

[Using files as persistent storage 225](#_Toc487969314)

# The Programming Process

If you have never tried something like programming before, it may seem like a mysterious activity – what is it really that we are doing? If we focus primarily on programming as a way of defining “business logic”, we are usually ***defining and manipulating a model of a small piece of the world***.

What does that mean more specifically? Suppose we wish to create a computer program – or **App[[1]](#footnote-1)**, for short – for administration of a school. Then we probably need to store and process certain information about students (and other things) in the App. What information is relevant to know about a student? Date of birth? Shoe size? That will depend entirely on the **requirements** to the App, which somebody will have to define.

The result will probably be that certain information is needed, while other

Infor­mation can be left out. So, the “model” of a student in the App may only contain certain information; the information which is relevant in relation to the requirements. Exactly which pieces of information we include will depend on the specific situation, i.e. the specific requirements.

Once we have figured out what information we need to include for each “concept” (student, teacher, classroom, course, …) in the model, we need to figure out how to repre­sent that information in our App. We get to that part very soon!

In almost all cases, the information will need to be processed in a certain way. A very simple pro­cess­ing could be just to show the information to a user of the App. A more complex processing could be to change the information. For a student, some information will probably never change (e.g. the date of birth), while some information will most likely change (e.g. the number of courses taken).

The way information changes can range from quite simple to very complex. The simplest change could simply be to change the current value to a new, given value. Say, when a student has taken one more course, the number of courses is increased by one. Other changes are more complex. If we e.g. store an average of the marks obtained for the exams passed by the student, then passing an additional exam will require a recalculation of the mark average. In any case, there will always be rulesfor how to process the information.

In programming, we then try to translatesuch rules (formulated in a human – or at least human-friendly – language) to instructions/logic written in a language the computer can understand; or more precisely: written in a language that

* We as humans can use to express such rules with relative ease, and
* The computer (using specialised software called a **compiler**) can translate into a language the computer hardware understands “directly”

A wide range of such **programming languages** exist. Some are quite obscure and only known to few, while others have gained widespread popularity. The program­ming language called **C#** (C-sharp) belongs to the latter category, and is used in these notes. C# is a so-called **Object-Oriented** language; other such languages are e.g. Java and C++.

So, translating our rules into the chosen programming language will result in writing a number of **statements**. A single statement usually performs a quite simple step of data processing, so any interesting program will contain a very large number of statements (many thousands, even millions…).

When we have a large body of statements, we need to organise them into larger units. One such unit is a **method**, which is a fairly small (less than 20) collection of statements performing a somewhat more complex data processing. Collections of methods can then be organised into even larger units called **classes**, and so forth. We will discuss such units of organisation later in these notes.

# Software tools

In order to write Apps using the C# language, we need some software tools to help us with this. These software tools should enable us to:

* Write C# code as easily as possible
* Write C# code of high quality
* Write C# code in collaboration with other programmers
* Help us to obtain – and maintain – an overview of the entire body of code, including all the various units of code organisation
* Translate our C# code into code that the computer can run directly
* Help us find and fix errors in our code, both when the code is written (syntax errors) and executed (logic errors)

Some of these features are somewhat fluffy (what does e.g. “high quality” mean in relation to C# code…?), but we will try to be more specific later in the notes.

## Microsoft Visual Studio

C# is invented by Microsoft, and their tool **Visual Studio** (or **MSVS** for short) can on its own help us with much of the above. You can obtain a free copy of **MSVS** from the school[[2]](#footnote-2). From now, we assume that you have a recent version of MSVS installed.

MSVS is a professional and commercial tool, with quite a lot of “bells and whistles”. It can therefore appear somewhat overwhelming at first glance. Fortunately, we only need to under­stand and use a fraction of the functionality:

* Understand the structure of a so-called solution or **project**
* Be able to navigate through the files in a project
* Understand the role of the files included in a project
* Add code to a project
* Compile, build and run a project
* Understand error messages from MSVS

You are of course encouraged to explore MSVS further during the course, but at first, it can be useful to try to maintain a certain degree of “tunnel vision”, and focus on the specific parts needed to get started.

## ReSharper

Even though Microsoft will probably claim that MSVS provides all of the features we outlined above, it is nonetheless possible to “extend” the functionality of MSVS by installing so-called **extensions** to MSVS, that will expand and/or improve on the built-in functionality of MSVS. These extensions can be created by anybody up to the task; one such extension is **ReSharper** (by JetBrains).

The ReSharper extension provides quite a lot of extra/improved functionality to MSVS, but it will not as such be visible; it will just seem like MSVS has been updated with extra functionality. The primary motivation for including ReSharper is that it provides a lot of functionality to help improve code quality (again, we defer a more concrete definition of “quality” to later in these notes).

Specifically, you should at this point – assuming you have already installed MSVS – obtain and install ReSharper[[3]](#footnote-3). We will later on return to the role of ReSharper in relation to code quality.

## Git / GitHub

The final “leg” of our features is the ability to collaborate with others when writing code. This is probably not a feature you will need until you embark on creating a larger App as part of a group project, but it is still useful to set up from the beginning, since you can also use it to safely experiment with code changes, without risking to lose previous work.

A general problem when several programmers are working on the same project, is to ensure that two (or more) programmers do not try to make changes to the same file simultaneously. A typical solution to this problem is to use a tool that allows a programmer to

* Obtain an exclusive “lock” on a particular file
* Edit and save the locked file
* Release the lock, allowing others to obtain the lock

In order for this to work, the files should be maintained at a central “repository” (or just “repo”) that all programmers can access. Typically, the tool will notify a pro­grammer if changes have been made to files in the shared project, such that the programmer can obtain the changed files from the repo. Also, such tools – which are generally known as **version control systems (VCS)** – will maintain a copy of all versions of all files in a project, making it possible to “roll back” a file to a previous version. This is a very useful feature, since it makes experimentation with e.g. large restructurings of a file completely safe, since you can always roll back to the previous version, if the restructuring proved unsuccessful. Version control systems usually also offer more sophisticated functionality with regards to version management, change tracking, etc..

A lot of VCSs exist today, and it may be hard to choose one over another. The VCS called **Git** has gained widespread popularity, and we have chosen to use Git as the VCS of choice as well. Also, Git integrates smoothly into Visual Studio. So, just as for ReSharper, you should at this point – assuming you have already installed MSVS – obtain and install Git[[4]](#footnote-4). We will later on return to the role of Git in relation to version management.

The term **GitHub** is very often mentioned in relation to Git. For now, you can simply think of GitHub as a free-of-charge storage facility in the “cloud”, where you can create and maintain a repo for your code. GitHub actually offers a lot of services in addition to the basic repository service, but we will not cover these service here.

# Code organisation and Visual Studio basics

We should now be up-and-running with MSVS, including the extensions covered in the previous section. We will now try to open a (very small) piece of code in Visual Studio, for two purposes:

* Investigating how code is organised
* Trying to load, navigate, edit and run the code

## Loading code into Visual Studio

For demonstration purposes, a small piece of code called **Sandbox** has been created. This piece of code should be available to you – exactly where will depend on the course, but your teacher can inform you about this ☺. With this information, you should be able to follow the below steps:

1. Start Visual Studio
2. From the menu, choose **File | Open Project**
3. Navigate to the folder called **Sandbox**, and go into the folder
4. Double-click on the file called **Sandbox.sln**
5. The code should now load – you will see some files in the window with the title ***Solution Explorer***

If you have completed the above steps successfully, you should see something similar to this in the ***Solution Explorer*** window:



What are we seeing here? In order to understand that, we need a basic under­stand­ing of **code organisation** in Visual Studio.

## Code organisation

The highest unit of organisation in MSVS is a **solution**. Remember that MSVS is an advanced, industrial-strength tool, that should be able to handle very complex software development tasks. This could imply that the entire “solution” to e.g. a school administration system would contain several applications (say, a smartphone App for students, a desktop App for staff, another desktop App for administrators, etc.). All these applications should be manageable as one integrated solution.

The next level is a **project**. A project will usually correspond to a single application. If you have several projects in a solution, you will still be able to modify, compile and run a single project, without involving the other projects.

With this information, we can already better understand what we just did, and what we see in the ***Solution*** (aha!) ***Explorer*** window.

First, we navigated to a folder called **Sandbox**. In MSVS, a solution is typically con­tained in a file folder with the same name as the solution it contains. So, this folder contains a solution called **Sandbox**.

Second, we entered the folder. The folder contained

* A file called **sandbox.sln**
* A folder called **Sandbox**

A file with the extension **.sln** is interpreted by MSVS as a **solution file** (.sln = solution) which holds information about a particular solution. The subfolders in the folder contains those projects that are part of the solution. So, the solution **Sandbox** thus contains a project called… **Sandbox**. It might seem confusing that we have a project with the same name as the solution it is part of, but this is actually very common for small solutions that only contain a single project. *We will only deal with single-project solutions in these notes and the associated exercises!*

Third, we double-clicked on the **sandbox.sln** file. This prompted MSVS to load in the **Sandbox** solution, which only contains the **Sandbox** project. This is what we see in the ***Solution Explorer*** window:



The project itself is where the really interesting stuff is. The stuff “hanging” under the **Sandbox** project in the ***Solution Explorer*** window is where the actual C# code is. Before dealing with actual code, we return again to the code structure discussion.

What is the next unit of organisation below **project**? The simple answer is **classes**. Classes is where the actual C# code resides, and we will deal with classes in great detail in these notes. A class is (usually) defined within a single file with the extension **.cs**. In the **Sandbox** project, there are thus two classes **Program** and **InsertCodeHere**, defined in the files **Program.cs** and **InsertCodeHere.cs**, respectively.

With this information, we should be able to understand almost everything we see in the ***Solution Explorer*** window. The two items called **Properties** and **References** are not interesting at this point, so they will remain a mystery…

For completeness, it should be mentioned that a unit of organisation between **project** and **class** actually exist, called **namespace**. A namespace is a way to organise classes that in some sense belong together, which can indeed be useful in projects with many classes. However, if you have very few classes in your project, the most common setup is to have just one namespace, with the same name as the project. We will not really utilise namespaces in these notes.

Returning to classes – which we advertised as containing actual C# code – the next unit of organisation is **methods**. A method is a collection of C# **statements**, that perform some useful task. We will deal quite a lot with methods later on. It should also be mentioned that classes can contain more than just methods, but in terms of code organisation, they can for now be thought of as containers for methods.

With statements, we have reached the end of the line of terms of code organisation, since statements are the “atoms” of code. Let us then review the entire hierarchy of code organisation:

A **solution** contains a number of

**projects**, that contain a number of

**namespaces**, that contain a number of

**classes**, that contain a number of

**methods**, that contain a number of

**statements**

A six-tiered hierarchy, no less! This may seem overwhelming, but try to compare it with a publication of a large body of text, say, the collected works of Kierkegaard (a Danish philosopher of some fame). Such a publication would probably be organised like this:

A **publication** contains a number of

**volumes**, that contain a number of

**chapters**, that contain a number of

**sections**, that contain a number of

**paragraphs**, that contain a number of

**sentences**, that contain a number of

**words**

A seven-tier hierarchy… This hopefully illustrates that the code organisation is as such quite meaningful, and not overly complex. However, the solutions we will encounter in relation to these notes will have a rather simple structure:

A **solution** that contains one

**project**, that contains one

**namespace**, that contain a few

**classes**, that contain a few

**methods**, that contain a few

**statements**

This also holds for the **Sandbox** project. Going forward, we will try to work in a bottom-up fashion, where we initially focus entirely on writing statements, without thinking too much in terms of methods and classes.

We conclude this first brief look at MSVS by trying to actually run the code we have loaded into MSVS, i.e. the **Sandbox** project. In general, code must be compiled and built, before it can be executed (running the code is often denoted as executing the code). Compiling and building C# code essentially consists of two activities, both performed by MSVS

1. Checking that the code obeys the syntax for the C# language
2. Translating the code into a language the computer can execute directly

The process is a bit more sophisticated than this, but we need not worry about that now. One thing to note is that even though the code passes successfully through these steps – and can therefore be executed – there is no guarantee whatsoever that the code behaves as we intend it to, i.e. that it complies with our requirements. We will return to this distinction later.

The **Sandbox** project does contain code that is ready-to-run, and we set the execution of the code in motion by either pressing ***F5*** on the keyboard, or clicking on the button containing a small green triangle in the toolbar:



If you are able to run the code successfully, you should see something like this appear on your screen:



This somewhat primitively looking window is a so-called **console application**. This is what most Apps looked liked before Windows and Macs became mainstream. It does look rather dull, and only allows input/output in character form, but using this style first enables us to postpone having to learn about GUI (GUI: Graphical User Interface) programming initially.

As advertised on the screen, the App does nothing more than print “Hello world!”, and now awaits that you press a key on the keyboard to terminate it. Once you do that, you have successfully loaded, compiled, built and executed your first C# appli­cation using Visual Studio!

## Statements (and the cruel Master Syntax)

As promised earlier, we will investigate the organisation of code in a bottom-up manner, starting with the simplest entity: the **statement**.

A statement can be thought of as a “code atom”, i.e. it is the fundamental code building block, but it also has an internal structure, which must follow certain rules. We cannot just mash up any sequence of characters and claim it to be a C# statement, just as we cannot throw together a random mix of electrons, neutrons and protons, and expect to end up with a stable, useful atom. Only certain combinations qualify as being statements.

More specifically, a statement is an instruction to the application about what to do next. This could be to

* Perform an arithmetic or logical calculation
* Read or write data to a file, the screen, etc..
* Control the “flow” of execution, i.e. choose between several alternatives about what to do next
* Etc.

You may already here sense that statements tend to fall into two board categories: statements that actually do something, like a calculation, visualisation or data trans­fer, and state­ments that control what to do next, depending on certain conditions. We shall see several examples from both categories in these notes.

Returning to MSVS; if you – assuming the **Sandbox** project is still loaded – double-click on the file **InsertCodeHere.cs** in the ***Solution Explorer*** window, a new window will open, showing the content of that file (there may be some small differences between the code below, and what you actually see in the window):

using System;

namespace Sandbox

{

class InsertCodeHere

{

public void MyCode()

{

// The FIRST line of code should be BELOW this line

Console.WriteLine("Hello world!");

// The LAST line of code should be ABOVE this line

}

}

}

This may look a bit overwhelming, with a lot of colors and odd characters posi­tioned strangely on the screen; for now, try to apply “tunnel vision”, and focus on a single of these lines:

Console.WriteLine("Hello world!");

This line is indeed a C# statement. It instructs the computer to print out the words ***Hello world!*** on the screen. However, if you are new to programming in general, that may be hard to figure out just by reading the line of code… The sentence ***Hello world!*** is indeed part of the line, but it is wrapped up in a lot of other stuff. What that “stuff” precisely means is not that important – we will learn to under­stand it later on.

To put it in more general terms: there will usually be a “gap” between what we intend the computer to do, and how we express that intention in a programming language. In human language, our intention could be written as:

Please display the words ***Hello world!*** on the screen.

In C#, that intention is expressed as

Console.WriteLine("Hello world!");

If somebody could make a compiler that could directly translate the human-language intention into executable code, the world wouldn’t need programmers… However, human language is inherently vague and ambiguous, while a computer needs very precise instructions! If you think about it, the human-language intention leaves many questions unanswered, like e.g.

* How should the words be displayed (where on the screen, color, size, etc.)?
* What “words” exactly. Only **Hello world!** or maybe **Hello world! on** or…?
* What should be done after displaying the words? Stop the application, wait for the user to do something, or…?

And this is just for an extremely simple intention! For more complex intentions, we need intermediate “stops” on the road from intention to code, like requirement specifications, designs, etc..

Another distinction between human-language intentions and C# code is the tole­rance for errors. Suppose we made some small spelling mistake in our intention description:

Please displlay the words ***Hello world!*** on the screen.

This small mistake would probably not hinder another human being in under­standing the intention. However, a similar mistake in the C# statement, like

Console.WriteLline("Hello world!");

will have catastrophic consequences (Try it! Open the file **InsertCodeHere.cs**, make the change, and try to run the application…). The compiler is absolutely unforgiving about errors! A C# statement has to strictly follow a predefined **syntax** for that parti­cular type of statement. Imagine that your mail application had a similar Draconian[[5]](#footnote-5) attitude towards errors, absolutely refusing to send a mail unless there are ZERO spelling and grammatical errors in the content…

The fact that C# programming (and most programming, actually) is a discipline that requires strict adherence to a given syntax, is something you need to come to terms with as an aspiring programmer. Fortunately, the software tools available now do a very good job in assisting you with getting the syntax right. If you try to modify or add code to the **Sandbox** project, you will notice that MSVS will provide suggestions to – and even point out errors – while you type! MSVS’s eagerness to help you can feel a bit intrusive and confusing at first, but once you get used to it, you will find it quite helpful.

If you tried to do the small change suggested above, you probably noticed that a wavy red line appeared below the incorrect piece of code. Such lines imply that something is wrong with the code! If you hover the mouse cursor over the line, an error mes­sage tooltip will pop up. If you have a larger section of code with many errors in it, you can get a full list of errors in your code by pressing ***F6*** in MSVS. Some error mes­sages may be hard to understand for a novice programmer, and the best advice is often simply to try to read the code again, and use your common sense to spot the problem.

In general, you should try to fix errors as soon as you see them! If you have more than one error in your code, some errors may actually not be true errors, but rather errors that occur because the previous code is incorrect. In that case, you should try to fix the errors from the top and downwards.

If we zoom out just a little bit in the code in the **InsertCodeHere.cs** file, you will notice some more lines of text:

// The FIRST line of code should be BELOW this line

Console.WriteLine("Hello, world");

// The LAST line of code should be ABOVE this line

Are these new lines also C# code? No, they are so-called **comments**. A comment is just a piece of text that MSVS does not consider to be code, and it will therefore ignore it when compiling the code. Comments are thus only present in order to help the human being working with the code. A single-line comment like the two above must always start with the symbol **//**. A multi-line comment must start with the symbol **/\***, and end with **\*/**.

The ability to add comments to code is very common in programming languages, and it is consider good practice to add comments to code that has a certain complexity[[6]](#footnote-6), to help those that might work with the code at a later time.

In this specific situation, the two comment lines have been added to outline the “sandbox” (hence the project name) within which we will play around for a while, to learn about the fundamentals of programming.

# Programming – fundamentals

We have already discussed what the programming process as such is all about – instructing the computer to do our bidding! Being a bit more specific, we can say that programming essentially deals with **representation** and **processing** of data.

Concerning processing of data, we can detail this activity further

* Obtain the data needed for a particular type of processing
* Store the data in the computer memory, in suitable data structures
* Perform the processing, according to certain business logic (algorithms)
* Store the resulting data in suitable data structures
* Enable the user to access the resulting data

The two main topics we need to delve into are thus:

* Data representation, using suitable data structures
* Data processing, using suitable algorithms (sequences of statements)

The first question to consider is then: What types of data do we wish to be able to represent and process? Possible types could be

* Numeric data (numbers)
* Text data
* Logical data
* Special-purpose data (pictures, music, etc..)

We will here focus on the first three types of data, since the last category typi­cally requires more advanced handling, which is beyond the scope of this text.

## Types and variables

From the computer’s perspective, the discussion about “types” of data is a bit meaningless, since all data is represented as sequences of **bits**, where the value of a bit is either 0 or 1. However, working directly at the bit level is somewhat obscure for human beings, so we like to group bits together in larger units, and interpret such a group in different ways.

The first level of bit grouping is to define a group of 8 bits as being a **byte**. We very rarely work at a finer level than the byte-level. As you may know, a modern PC will usually have between 4 and 16 Gigabytes of working memory (RAM). Each byte in the computer memory can be specified by its **address**. The address is just a counter starting from zero, up to the number of bytes in memory. When we place some data in the working memory of the computer, we are essentially just writing a sequence of bytes into a specific area of the memory, starting at some given address.

Suppose we want to store something more human-friendly than “raw” bit sequences in the computer memory, for instance an integer number (i.e. a number without any decimal part, like 12 or 704). How do we translate an integer number into bits? First, we can note that with 8 bits in a byte, we can create 28 (2 to the power of 8) different bits, like

00000000

00000001

00000010

00000011

00000100

00000101

…and so on, until

11111111

Hopefully, you can see a system here. If we choose to interpret the first bit sequence as the number 0, the next one as the number 1, the next one as the number 2 and so forth, we get:

00000000 = 0

00000001 = 1

00000010 = 2

00000011 = 3

00000100 = 4

00000101 = 5

…and so on, until

11111111 = 255

So, we can use a single byte to represent numbers from 0 to 255. Actually, since it is us that interpret the sequence of bits, we are free to choose a different starting num­ber than zero, if we also want to be able to store negative integers. Suppose we start from -128. We can then follow the same pattern as above, giving us a range from -128 to 127 (both included).

This is very nice, but what if we need to store larger numbers? We could then choose to use four bytes instead of just one to represent a number. That would give us a more impressive range, from –2147483648 to 2147483647. You could even use 8 bytes, giving you an even large range. A natural thought could then be *“well, lets be absolutely sure we have a large enough range! Lets use 64 bytes for each integer number!”*. For most applications, this would probably also work fine in this day and age, where even tiny devices have multi-gigabyte memories. Still, there was a time where memory was a more scarce resource, and there are indeed still many appli­cations today where you have to be quite careful w.r.t. memory consumption. A Machine Learning algorithm may need to represent billions of numbers, making it a highly relevant matter if each number takes up 4 or 8 bytes.

The general point, however, is this: Even though the computer stores everything as sequences of bits, we can fortunately make use of more human-friendly data types, that are available in C#. The compiler – and underlying code – will handle the details of the translation to bits for us.

A lot of these so-called primitive data types are available; we only present a few here, but feel free to look for additional information about other such types else­where. The names of the types are sometimes a bit obscure, which is often for historical reasons.

|  |  |  |
| --- | --- | --- |
| **Type name** | **Memory use** | **Description** |
| **int** | 4 bytes | A number without decimal part, like -98 or 6501.  Range: from –2147483648 to 2147483647 |
| **double** | 8 bytes | A number with decimal part, like 3.8716243456  Range: from about 10-308 to 10308  NB: Not completely precise! |
| **bool** | 4 bytes | A boolean value, either **true** or **false** |
| **string** | One byte per character | A sequence of characters, like ***“Hello!”***  NB: Strictly speaking not a primitive type! |

Notice how these four types match the three types of data (numeric, text and logical) we initially stated we want to be able to process. With that in place, we can begin to define so-called **variables** in C#.

A variable in C# - and programming languages in general – is just a piece of memory that we define as containing data of a certain type. We can then store and change the actual data as we wish, hence the term “variable”. The data in a variable will thus be located at a specific address in memory. However, instead of having to refer to that address directly, we assign a name to the variable as well. Creating such a variable in C# can look like this:

// Reserve space in memory for an int,

// refer to the address by the name “age”

int age;

// Store the value 24 in the address

// referred to by “age”

age = 24;

The first line of code is called a variable declaration, where we reserve some memory in the computer, with the intention of storing data of the type **int** in it. At this point, there is strictly speaking not any data in the variable yet (in practice, C# will set the value to zero initially). The next line – which is known as an assignment statement – puts the value 24 into the memory referred to by **age**.

If you are new to programming, but have some knowledge of mathematics, the second statement may seem confusing. You may think that this statement tries to compare **age** with 24 – which may be true or false, depending on the value of **age** – which is how to understand that statement in a mathematical context. However, in C# that statement is an action; we change the value contained in **age** to 24. We will soon see how to express a comparison of two values.

Suppose we added a third line of code after the first two:

age = 28;

This will change the value of (the data in) **age** to 28. What happened to the previous value of 24? That value is now irretrievably lost! A variable of this type can contain only one value of the specified type, so assigning a new value to the variable will overwrite the existing value.

Finally, it is considered good practice to initialise – i.e. assign an initial value – to a variable as part of the declaration statement, like this:

// Reserve space in memory for an int,

// refer to the address by the name “age”,

// and initialise the value to 24.

int age = 24;

If the initial value of an **int** variable should be 0, you should still write this explicitly, even though an **int** is initialised to 0 by default. By writing it explicitly, you remove any doubt about whether or not you simply forgot to initialise the variable…

## Arithmetic

A very important part of most programming tasks is **arithmetic**. Almost all program­ming languages support arithmetic, since much data processing has an arithmetic nature – we perform calculations. The specific syntax may vary somewhat between the different languages.

C# supports most common arithmetic operations, but there are certain operations that differ from “classic” arithmetic. We have already seen an assignment statement

int age;

age = 24;

Again, be aware that the second line means *“change the value of age to 24”*, and NOT *“compare the value of age to 24”*.

Below is an example of very simple arithmetic

age = 24 + 32; // Now age is 56

That is perhaps not the most mind-bending example, since we could just have written 56 directly on the right-hand-side of the **=** symbol. A bit more interesting is this:

age = age + 10;

Can we really assign to a variable, and use the variable itself on the right-hand-side as well? Yes, because the expression on the right-hand-side will be evaluated first – using the value currently assigned to **age** – and the resulting value is subsequently assigned to **age**. Suppose the value of **age** is 24 when the statement is reached. The first step is then to evaluate the right-hand-side, which is 24 + 10, i.e. 34. Next, the value 34 is assigned to **age**, thus replacing the previous value of 24.

We can have complex expressions on the right-hand-side of an assignment, involving several variables, say

double tax = incomeTax + housingTax + 0.5\*zoneTax;

assuming that the variables on the right-hand-side have been declared previously.

Doing addition, subtraction and multiplication with integer numbers is fairly straight­forward (even though integer overflow is a pitfall). Division can be slightly more tricky. Consider the below code:

int a = 7;

int b = 4;

int c = a / b; // a divided by b

The result is NOT 1.75 as you might expect, but 1. When doing arithmetic with inte­gers, the result will also be an integer. Also, there is no rounding of the result. It might seem more natural that **c** should become 2, but it doesn’t!

There are some non-standard operators in C#, for instance the “remainder” operator % (or “modulo”)

int a = 7 % 4;

The result of the above is 3 – the remainder when dividing 7 with 4 (integer division). The modulo operator is not something you will use very often, but it can come in handy when e.g. checking if, say, a number is even. We will see such examples later.

The usual rules for so-called operator precedence also apply in C#, so e.g.

int a = 2\*3 + 4; // This is 10, NOT 14

Use of parentheses is also allowed, and follows standard rules from mathematics. It is often a good idea to use parentheses to increase readability, even if they are not strictly necessary.

## A detour to Code Quality, part I

We are now at the brink of being able to write small pieces of C# code ourselves. Before that, it is now an appropriate time for an initial discussion of code quality, and setting up a few good habits to follow.

We have already seen that a first hurdle to pass for any C# application – even the size of just a few lines of code – is to be syntactically correct. If we cannot achieve that, we cannot even compile our code, let alone run it. So, this forms a (trivial) first criterion for high-quality code: it must be able to compile!

The next hurdle is usually much harder: the C# application should behave according to specification. For any real-life application, this is almost impossible to achieve! We might achieve a state where “almost everything” behaves as expected, but reaching an absolute 100 % is usually unrealistic. At some point, the effort to get closer to 100 % may not be worthwhile, since the remaining errors may be quite insignificant. Exactly when this break-even point is reached will be highly dependent on the type of application. A software control system for a nuclear weapon should (hopefully) be much closer to 100 % than a harmless mobile game needs to be… Regardless of this, the degree of compliance to requirements is also a relevant measure of quality.

Suppose now that the application can compile, run and seems to behave as specified (to a reasonable degree). Are we then done? Can we not increase the quality further? In some situations, we could actually say that “we are done”. We might just need the software to demonstrate that something is possible (a proof-of-concept), and discard the software after the demonstration. In a lot of other real-life situations, however, the code will need to be updated at a later time, perhaps very significantly. Software tends to have a long lifecycle, and will in many organisations outlast those employees that originally wrote it. A piece of software may thus “change hands” many times during its lifecycle. Such a change of hands will often come at a significant cost, since the new programmer will need to get acquainted with the software, before being able to safely and easily modify it. Therefore, we should strive to create code that is as easy as possible to maintain and extend, in order to reduce this cost.

Writing code that is “as easy as possible to maintain and extend” is somewhat sub­jective. What does this mean in practice? This is quite hard to pin down, and there are diverging opinions about it. Historically, it has to some extent been assumed that as long as you were careful about specifying and designing your application, the resulting code would automatically be of high quality. This has proven to be an illusion; in the real world, requirements may change rapidly, and we cannot up-front anticipate how the optimal end design will be. We must therefore do the best we can on the basis of the available information, but also be prepared to spend time on making quality improvements to our code, that do not add extra functionality. Over the years, some agreement has been reached concerning what such improvements might specifically be; the famous (in the programmer community) book **Refactoring[[7]](#footnote-7)** by Martin Fowler gave a first comprehensive presentation of a large number of so-called “refactorings” that can be applied to code, with the sole purpose of improving code structure, while keeping the functionality intact. These refactorings range from the very simple – as we shall see in a moment – to the quite sophisticated.

Just as we can probably never reach the “100 % compliance to requirements” level, we can probably never reach a “100 % perfectly structured code” level either. How­ever, improvements will still have value until some breakeven point. One of the simplest refactorings is simply called **Rename**.

Consider the code below:

double x = 25.00;

double y = 6.00;

double z = 0.08;

double t = x \* (1.00 + z) + y;

What does it do? You can probably see that some arithmetic calculation is going on, but it is hard to figure out what those variables actually mean.

Now compare it to this code:

double netPrice = 25.00;

double shipping = 6.00;

double tax = 0.08;

double totalPrice = netPrice \* (1.00 + tax) + shipping;

Now it should be clearer what this is all about; calculating the total price for a bought item, including tax and shipping. The logic of the two examples is identical, however. This may be too small an example to be convincing, but imagine that the calculation of the total price had been coded wrong (we assume the logic should be as above). Which of these two lines makes it easier to spot the error?

double t = x \* (1.00 + y) + z;

or

double totalPrice = netPrice \* (1.00 + shipping) + tax;

The simple practice of using descriptive names for variables (and other elements in your code) is a first good habit to get into!

You may also notice that besides using descriptive names, a distinctive style has also been used with regards to the choice of small and capital letters. For variables like the above (which we will later know as being so-called **local variables**), we will use a style known as **camelCase**. A word written in camelCase will

* Start with a lower-case letter
* Not contain underscores
* If the word is concatenated by several word, each word after the first word will start with a capital letter

All the four variables in the example above are examples of camelCase. We choose this style simply because it is recommended by Microsoft. Trying to use descriptive variable names written consistently in camelCase is a good starting point for an aspiring programmer!

At this point, you may start to sense what all these tools, functionalities and whatnot we have crammed onto the computer are actually good for. They all play a role in helping us get past the three hurdles of software development:

* Make the code compilable
* Make the code conform to requirements
* Make the code of high quality

We have only looked briefly at some of the tools, and not at all at others, so the full picture is probably far from clear yet. It may also be hard to distinguish if a certain feature comes from MSVS or e.g. ReSharper, but it doesn’t really matter. The entire suite of tools are designed to collaborate within the framework of MSVS.

## Screen output and type conversions

We still need a small piece of knowledge before we can be let loose on some code. We should now be capable of writing small pieces of code…but we cannot really present the results of e.g. an arithmetic calculation anywhere. It would be nice to

be able to print it on the screen, in a fashion similar to the ***“Hello world!”*** example.

Fortunately, that is not too hard to do, but requires some understanding of how data becomes “printable”. We have already seen that the code line

Console.WriteLine("Hello world!");

will write ***Hello world!*** on the screen. We could change the line to

Console.WriteLine("How are you today?");

and see ***How are you today?*** printed on the screen. So, it seems like everything we stuff into the highlighted area gets printed:

Console.WriteLine("Whatever you want printed…");

So, we could maybe do this in order to print the value of a variable:

int age = 24;

Console.WriteLine("age");

Try it! It doesn’t work as we hoped… It prints ***age*** rather than ***24***. The problem seems to be that whatever we put into the highlighted area literally gets printed. Well, that is partly true. The “ and “ symbol on either side of the highlighted area are used to delimit a **string**, while not being part of the string themselves. If we put something between the delimiters, it will always be interpreted as a string.

Can we then just get rid of the string delimiters, like

int age = 24;

Console.WriteLine(age);

Indeed we can! We can now print the value of the **age** variable, and thus print the value of any variable we wish. The **Console.WriteLine** method (we will talk much more about methods pretty soon) is quite flexible with regards to what it can print on the screen. It will print almost everything you put inside the parentheses, or rather; it will try to print the best possible string representation of it. When the value of **age** was printed, it was in fact the string “24” that got printed. For us, that distinction is a bit academic, since the conversion from the number 24 to the string “24” is trivial. We will see later that such conversions can be more complicated.

So far, so good. But what if we want to print something more descriptive, maybe like ***The age now has value …*** followed by the value of **age**. Somewhat surprisingly, you can do this in the following way:

string message = "The value of age is " + age;

Console.WriteLine(message);

What is happening on the right-hand-side here? It looks like we are adding a **string** to an **integer** variable!? Well, the compiler will happily compile and run the code, indeed printing the intended message… What we see here is an example of type conversion.

We saw earlier that you need to be careful when doing integer division, since the result will also be considered an integer. What happens if you try to divide an integer value with a decimal value (i.e. a value of type **double**)? Try to run this code:

int age = 24;

double someNumber = 1.3;

Console.WriteLine("Dividing age by 1.3 is " + age/someNumber);

If an arithmetic operation involves variables of different type, the compiler will choose one of these types and convert all elements to this type. In the example, we use the types **int** and **double**. Which type should be chosen? If **int** was chosen, we would have to convert 1.3 to an integer, and thereby lose the decimal part. If **double** is chosen, the **int** value 24 can simply be converted to 24.0, which is a perfectly valid decimal number. The type **double** is therefore chosen, and the result will thus also be of type **double**. That value can in turn be converted to a **string** type, which is done inside the parentheses of **Console.WriteLine** (note that addition of strings simply means to add the second string to the end of the first string; this is also known as string concatenation).

In general, the compiler allows automatic conversion of types if no information is lost during conversion. We saw above that **double**-to-**int** conversion is problematic, because we lose the decimal part (and thereby some information), but **int**-to-**double** conversion is safe, because any integer value **x** can be converted to **x**.0.

Using string concatenation is one way of printing a longer message – consisting of a mix of text parts and variable values – on the screen, using **Console.WriteLine**. However, there is another way to do this which might be more convenient, by using

a so-called formatting string. Suppose we have two variables like:

string name= “James”;

int age = 23;

and want to print a message like ***James is 23 years old***. Using a formatting string, this will look like:

Console.WriteLine("{0} is {1} years old", name, age);

Notice the use of the curly brackets, like the **{0}** and **{1}**, inside the string. This should be understood as: Take the value from the first variable following the string (in this case, the variable called **name**), and replace **{0}** with that value. Then take the value from the next variable (**age**), and replace **{1}** with that value. In this example, **{0}** is then replaced with “James”, and **{1}** is replaced with “23”, producing the string above. Note that you can use this principle for as many variables as you wish, by adding **{2}**, **{3}** etc. to the formatting string, and adding the variables themselves after the format­ting string (separated by comma).

Finally, you should note that you can even put expressions instead of just variables into a formatting string. The below code

Console.WriteLine("In ten years, {0} is {1} years old", name, age+10);

is perfectly valid as well.

## Logic

The ability to process so-called **logical expressions** is also a key element in almost all programming languages, and indeed also for C#. A logical expression is an expression that evaluates to either **true** or **false**. This type of logic is also known as **Boolean logic**, since it was invented by British mathematician George Boole.

Boolean logic fits very well into the realm of computers, where the **bit** – which can also only have one of two values – is a fundamental concept. Boolean logic is also useful for controlling the **flow of execution** of the code in an application. We will soon see examples of code where certain conditions will decide which part of the code to execute next. Such conditions will be of the kind that are either **true** or **false**.

The most common form of Boolean logic encountered in programming is to evaluate a **relationship** between two elements. A very simple – but quite common – example is to evaluate if two elements are equal to each other. If the elements are e.g. integer numbers, this evaluation is pretty trivial. Other situations are less trivial; consider for instance the strings “Hello” and “hello”. Are they equal? Time will tell…

Starting out with integer numbers, the below code is a simple example of such an evaluation:

int firstNumber = 12;

int secondNumber = 14;

bool areTheyEqual = (firstNumber == secondNumber);

Console.WriteLine("The numbers are equal : {0}", areTheyEqual);

Notice in particular the highlighted area; here we compare the value of **firstNumber** to the value of **secondNumber** (more precisely: we evaluate if **firstNumber** is equal to **secondNumber**). The **==** symbol is a **logical operator**, used to evaluate if two values are equal to each other. Notice that we do not use the single-equal symbol (**=**) for this purpose, even though it might seem natural. Remember that single-equal is used when we assign a new value to a variable! This distinction is very important, but also a bit confusing for those new to programing. As a challenge, try to remove one of the = symbols in the expression, and see what the compiler thinks of that…

Several additional logical operators are available; below is a table of those most com­monly used:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| a == b | a is **equal to** b |
| a != b | a is **not** **equal to** b |
| a > b | a is **strictly greater than** b |
| a >= b | a is **greater than or equal to** b |
| a < b | a is **strictly smaller than** b |
| a <= b | a is **smaller than or equal to** b |

The meaning for all of these operators should be pretty clear, as long as we are dea­ling with numerical values. It is less clear what it means that a string is “smaller than” another string. Shorter? Starts earlier in the alphabet? Depending on the type of the elements you try to compare, it might only be certain of these operators that make sense. The compiler will tell you if you try to perform a meaningless compari­son.

A well-known pitfall in relation to the equal operator occurs when working with deci­mal numbers (of e.g. the type **double**). A decimal number cannot be guaranteed to be represented precisely in memory (how would you represent 1/3 = 0.3333…. ?), so you may experience small so-called **rounding errors** when doing arithmetic with decimal numbers. If you perform a complicated calculation and expect the result to be pre­cise­ly 4, the result might actually be 4.000000001. If you then compare this value to precisely 4, the comparison will evaluate to **false**. A typical workaround is to define a small value (often called **epsilon**) and define that two decimal values are indeed con­sidered equal, if the difference between them is smaller than **epsilon**.

Returning to integer values again, we observe that the operators listed above make it possible to e.g. check if a number is smaller than a certain value (say, 10), like so

int age = 8;

bool isSmaller = (age < 10);

What if we want to check if a value falls within a certain interval? Say we wish to check if somebody is a teenager. The value of **age** should then be:

* Smaller than 20, and
* Larger than 12

So, both of these conditions must be fulfilled. In order to express this in code, we need to introduce the **AND** operator. The AND operator allows us to combine two logical expression into one (more complex) expression.

int age = 14;

bool isTeenager = (age < 20) && (age > 12);

The highlighted symbol **&&** means AND. So, the right-hand-side should be read as “**age** smaller than 20 AND **age** larger than 12”. The somewhat obscure && notation for AND is mostly a matter of tradition.

A close sibling to the AND operator is the OR operator. Suppose we wish to check that somebody is not a teenager. The value of **age** should then be:

* Larger than 19, or
* Smaller than 13

So, just one of these conditions must be fulfilled. In code, this becomes

int age = 14;

bool isNotTeenager = (age > 19) || (age < 13);

The highlighted (and also slightly obscure) symbol **||** means OR. You could say that OR is the more forgiving brother to AND; where AND requires both expressions to be true, OR only requires one of them.

A third member of this small family is the NOT operator. If a NOT operator is used in front of a logical expression, it simply reverses the value of that expression. We could have used that in the previous code example:

int age = 14;

bool isTeenager = (age < 20) && (age > 12);

bool isNotTeenager = !isTeenager;

The highlighted symbol ! means NOT. With these three operators available, we can build up very complex logical expressions, in the same way as we can build up very complex arithmetic expressions. Again, use of parentheses may improve the read­ability of complex logical expressions.

Finally, it can be useful to see how these operators work by means of a **truth table**. Here we list all four possible combinations of two logical expressions A and B, and the result of applying the operators described above:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **A && B** | **A || B** | **!A** |
| true | true | true | true | false |
| true | false | false | true | false |
| false | true | false | true | true |
| false | false | false | false | true |

## The concept of functions

We now have some basic tools available that allow us to write non-trivial pieces of code. For instance, we can calculate the average of three integers:

int average = (value1 + value2 + value3)/3;

This is still rather simple, but if we must do this several places in our code, it becomes tedious to write again and again. This is even more problematic if the logic is more complex. Instead of this, we would like to define the logic in one place, and then refer to that logic instead of writing it again over and over. We can do this by defining a so-called **function**.

A function is a fundamental concept in programming. The syntax may vary a bit from language to language, but you always need to

1. Name the piece of logic, so you can refer to it
2. Define what input the function needs
3. Define the logic of the function
4. Define what the output of the function is

A function written in C# for calculating the average of three numbers – as we just did above – could look something like this:

int CalculateAverage(int val1, int val2, int val3)

{

int average = (val1 + val2 + val3) / 3;

return average;

}

We do not know enough about C# to fully understand this code yet, but the code actually conforms to the four points written above:

1. The name of the function is **CalculateAverage**
2. The function takes the three integers **val1**, **val2** and **val3** as input
3. The logic is to calculate the average of the three values given as input
4. The output is the average value just calculated

We will later on use other words to describe input, output and logic, but the principles are just as described here.

We can now use the function **CalculateAverage** elsewhere in the code, whenever we need to calculate the average of three integers. We can **invoke** or **call** (we will usually use the term **call**) the function by writing code like this:

// Assume that ageJohn, ageJim, ageJack are integer variables

int ageAverage = CalculateAverage(ageJohn, ageJim, ageJack);

Note that we are using an assignment statement here; if we want to use the output value returned by the function for something useful, we need to e.g. assign that value to a variable. Also note that it is possible to use a function as part of an expres­sion, as illustrated in the somewhat silly code below:

int ageAveragePlus10 = CalculateAverage(ageJohn, ageJim, ageJack) + 10;

This is perfectly valid C# code; the compiler will look at the function and think *“Well, the output of calling that function will produce an integer value, so I can just add 10 to that value, no problem!”*. As long as the type of the value produced by the function can be used in the expression the function is part of, everything is fine.

At this point, you may think that we haven’t gained that much by defining a function, since it would be just as easy to simply write the statement that calculates the ave­rage directly. That is true for a simple example like this, but imagine when the logic becomes more complicated. It may then require several lines of code to express the logic in C# code, and it would both be tedious and error-prone to have to write out that collection of statements over and over in the code. Also imagine if you suddenly found an error in your logic! If you had repeated the code over and over in your appli­cation, you would have to correct the error in a lot of places. If you defined a function instead, you only have to fix the error in one place; inside the function.

Another extremely useful property of functions is that you can call a function inside another function. This allows you to define functions at various levels of abstraction in your code. At the lowest level, you may have functions like **CalculateAverage**, that only use simple C# statements. At the next level, you may have functions that call functions like **CalculateAverage**, and they may in turn be called by other function at even higher levels, and so on. This allows you to break down very complex logic – maybe requiring thousands of statements – into manageable parts.

## Pre-OO programming

We have now been introduced to types and variables, basic arithmetic and logic state­ments, and the general concept of functions. In the “old days” of computer program­ming (before ca. 1990) – before so-called **Object-Oriented programming** – this was more or less the level of abstraction applications were written at. Once you master Object-Oriented programming, you may wonder how it was ever possible to create complex software with just these facilities. Still, people did write software to put men on the moon back then…

What we have learned now does indeed relieve us of many considerations that earlier programmers had to handle themselves:

* We can use variables and types, and do not have to worry about details of actual data representation and memory management.
* We can define and use functions, that allows us to divide complex logic into manageable parts.

As indicated above, this alone enables creation of very sophisticated software. How­ever, there was a growing sense in the programming community, that such languages still made it difficult to model real-life concepts, like a “student” or “employee” in a system for school management. It was realised that such “entities” were in a sense a **self-contained unit of both data and functions**, and it would be beneficial to be able to express and use such entities more directly in programming languages. These con­siderations led to the emergence of **Object-Oriented programming**.

# Object-Oriented Programming - fundamentals

Before OO-programming emerged, is was hard to establish a connection between the data and the functions belonging to a specific concept, like a “student” or “employee” in a school management system. You could e.g. use a naming convention that would imply that certain data and certain functions were related, but it was still difficult to create a “unit” of some sort in your code, that would correspond to one “unit” in the domain you were trying to model, like e.g. a “student”. Object-Oriented programming (or just **OO-programming**) introduces concepts to allow just that!

## The Object concept

The central concept in **OO-programming** is the **object**. An object is something that can be created (in the memory of the computer) when your application runs. Objects are created, used for certain purposes, and may also disappear again when they have served their purpose. If your application is a school management system, the appli­cation may start by reading data from an external source, and use that data to create several objects. Each object will be of a certain type; in a school management system, you might have objects of type **Student**, other objects of type **Teacher**, and so on. The objects may be used for many purposes, like being shown on the screen, used for certain calculations, etc.. All that will (of course) be defined by the C# code somebody wrote for creating the application.

This probably sounds radically different than the simple types and variables that we have seen so far – and it is! However, there is nothing magical about it. From the computer’s point of view, things are represented the same way as before. Object-orientation is thus only a tool for making it easier for humans to express (human) logic in terms of code. If there is anything close to magic going on, it is perhaps inside the compiler itself, which has the rather formidable task of translating human-friendly code into (radically different) machine-friendly code…

## State and Behavior

The fundamentally new idea in OO-programming is to join both data and logic into single units (objects). In the OO-world, some different terms are used:

* When talking about the **data** contained in an object, we usually refer to the **state** of an object.
* When talking about the **logic** contained in an object, we usually refer to the **behavior** of an object.

It requires a bit of practice to be comfortable with these terms. As a specific example, suppose we have an object called **john** in our code. For now, we don’t worry about how such an object is created – we’ll learn that soon. We already know that an object should have a type – the type for the object **john** is called **Human** (very soon, we will also learn where such a type as **Human** comes from). So, the object **john** is supposed to represent a human being. More precisely, we can say that the object **john** repre­sents a model of a human being, i.e. that model which is defined by the type **Human**. Exactly what this model contains will be very situation-specific. In some applications, we may only need a very crude model of a human being, that only contains a little bit of state-and-behavior. Other applications may require that we have a much more detailed model.

Suppose that **john** is a fairly simple object (i.e. **Human** is a very simple model of a human being). What “state” is the object **john** in? We defined above that the term “state” is related to data, so a “state” is simply a set of values, that in total describe the state that this particular object is in. In our simple example, we could define that the only data that is relevant here is the name, weight and height. So, if we can obtain these three values, we will know what state the object **john** is in. You can think of the object **john** as a little box containing the described data:

john

Name: John Smith

Weight: 85

Height: 185

During the entire lifetime of the object **john**, we expect that it will always contain these three values, and that each value will always be meaningful. A natural way to use the object would then simply be to get information from the object, if we need it for some purpose. Imagine that there are a lot of objects of the type **Human** present, and we want to calculate the average height for all of them. We would then “ask” each object for that particular information, and expect the object to “respond”.

More generally, we will usually want to be able to “ask” an object for information about its “state”. We may even want to be able to change the state of an object, by providing the new value for a particular piece of the state. Maybe the real-life John has gained a bit of weight, so we would like to update Weight to 90.

## Public and private appearances

When you look at Name, Weight and Height in the blue box above, you may – very naturally – think *“Ah, that’s just three variables! One of type string, and two of type integer”*. The truth is slightly more complicated, however. As a first version of our object, it would indeed be very natural to define it to contain three “variables” (we will soon see that a different term is used), and also that we should be able to – for each variable – get its current value and update its value. Suppose now that we also want to know if an object – or rather; the person represented by the object – is overweight. One definition of overweight is that if your so-called **BMI** (Body Mass Index) is higher than 25, you are considered to be overweight. The BMI is defined as:

BMI = (weight in kilograms) / (height in meters)2

Calculating the BMI with the numbers for **john** given above gives 24,8. Good for John! But the pending question is: should we add a fourth variable to **john**, to hold the value of the BMI? It’s tempting to do this, but consider the definition of BMI. It only uses information that is already present inside the object! So, adding a fourth variable would be a bad idea for three reasons:

* The object would use a bit more memory
* If you update either the weight or the height, you must also update the BMI
* You may risk that the information inside the object becomes inconsistent!

The last two points are closely related; you should definitely avoid having redundant data in an object! Not only simple, duplicated data, but also data that can be calcula­ted from other data. So, no fourth variable inside **john**!

Deciding not to keep the BMI explicitly represented inside the object does however not solve the original problem: we wish to be able to ask the object what its BMI is, and we don’t want to calculate it ourselves. The last point is important; even if we would be able to calculate the BMI ourselves, by extracting the relevant information from the object, we should not have to be burdened with this. We should not need to know the details of BMI calculations, since we are only interested in the result!

This discussion brings us to another powerful feature of objects: the way an object “presents itself” to the outside world (i.e. the state-and-behavior the outside world can obtain from the object) may differ from how state-and-behavior is represented inside the object itself! In our case, the outside world is interested in knowing about four different properties of the object **john**: the Name, Weight, Height and BMI. However, the clever creator of **john** – or more precisely, the type **Human** of which **john** is one instance – has figured out that we only need to store the three first values inside the object:

John (public)

Name: John Smith

Weight: 85

Height: 185

BMI: 24,8

John (private)

name: John Smith

weight: 85

height: 185

The outside world

You can think of an object as having a “public” and a “private” side. The public side is how the object presents itself to the outside world, i.e. what state-and-behavior the object makes available to the outside world. The private side is how the state-and-behavior is actually represented inside the object. In some cases, the relation between public and private is quite trivial: the (public) property **Height** is simply the value of the (private) variable **height** inside john. In other cases, the relation may be more complex, as we just saw for BMI. However, the outside world doesn’t care about this. The BMI is “just another property” and can be treated as such.

This ability seems to solve our problem. We can present a simple set of public proper­ties to the outside world, and hide away the details of implementation inside the object. There is a slight complication, though. We have argued that an external user should be able to obtain – and change – the value of a property. In the example, this makes perfect sense for Name, Weight and Height. But what about BMI? It does not make sense to set the value of BMI directly, since there is no correspon­ding variable inside the object…. Now what? Fortunately, the C# language makes it possible to set restrictions on what an external user can do with a property. For most properties, you can both **get** (i.e. retrieve the current value) and **set** (update the value) the property, but you can also specify that only the get-part is available. That would be a natural restriction to put on the BMI property.

The above concerns the “state” of an object, and how an external user interacts with it. The “behavior” part is similar, and it may sometimes be hard to see exactly where to draw the line between interaction relating to state or to behavior. As a somewhat vague definition, we can say that behavior is something we invoke to make an object “do something”. A behavior will be implemented in terms of a **method** (which is essentially the same as a function) that an external user can call, in a manner very similar to calling a function. A simple example could be a method called **PrintInfo**, that prints something about the object, in our example a method that could print:

*“Hi, my name is John Smith. My height is 185 cm, and my weight is 85 kg”.*

This is a “behavior” – when we invoke this particular behavior on the object **john**, this line is printed on the screen. It does not change the state of the object, however. We will of course see example of more interesting behaviors later. Just as for state, it may also sometimes make sense to define certain behaviors (i.e. functions) to be “private”, for instance if their only purpose is to help in the implementation of more complex, publicly available behaviors.

## The Class concept

We have several times claimed that all objects must have a type, and that the object **john** has the type **Human** in the example above. So, where do these types come from, and how do you define such a type?

In C# - and in OO-programming in general – types are defined by a **class definition**. A class definition is where you define all features that every object of this class will have. This includes:

* Publicly available properties
* Publicly available behaviors (called methods)
* Private data structures and methods needed to implement the public properties and methods.

Once a class definition has been completed, it will be possible to create objects of this particular class (i.e. type – we will also use term “class” from now). In our previous exam­ple, somebody must have written the **Human** class definition, in order to make it possible to create objects – like **john** – of the class **Human**.

It makes sense to distinguish between the creator of a class definition and a user of a class definition (by “user” we here mean a programmer that wants to use objects of a particular class). The creator will of course know about all details – both public and private – of the class, while the user only needs to know about the public parts. From the user’s perspective, the class is a kind of “black box”; the user can create objects of a particular class, and can interact with the objects through the public parts (also called the interface) defined in the class definition. However, the user cannot – and should not – obtain information about the internal structure of the class.

This separation of the public and private side of a class also enables a certain degree of freedom, with respect to changes in code. If a programmer uses objects of a certain class, he only uses the public part. So, as long as the public part stays unchanged, it is possible for the creator of the class to update the code inside the class definition, e.g. to fix errors or improve performance.

## Using objects of an existing class

Before we dive into the details of how to define a class, we take a look at how to use an already existing. The first question would naturally be: what classes are available for use? Class definitions can come from several sources, including:

* **The .NET class library:** The C# language is actually just one part of a larger soft­ware ecosystem called **.NET**, created by Microsoft. A part of this system is a very large collection of ready-to-use classes, called the .NET Framework Class Library. It is well beyond the scope of these notes to detail the content of the library, but once you get to a level where you need to create fairly sophisticated software, you may often find a class in the library that suits your needs.
* **Third-party suppliers:** The classes in the .NET class library are usually quite general, and are not focused on a particular real-world domain (say, finance). Some companies develop smaller, more specialised class libraries, that you (or your company) can license for use in your own projects.
* **Open source:** Just as for other kinds of software, there is also a fair amount of open-source C# code available from various sources.
* **Your company:** Most companies dealing with software development will deve­lop a code base over time, which might also contain useful classes
* **Yourself**: If all of the above fails, you may have to write a class definition yourself ☺.

There are various ways to try to navigate your way through a class library, and the easiest solution is often to use Google (or what ever search engine you prefer) for the job. The mechanics for making a class available for use in your project can vary a bit, and we take a closer look at that later. For now, we will just assume that class defini­tions can be made available.

Let us assume that a class definition for a class called **Student** is available. We should then be able to create an object of class **Student**. In C# code, this is done like:

Student firstStudent = new Student();

This line contains some new elements, but also elements we have seen before. Compare it with a line of code we should be familiar with now:

int age = 18;

The overall structure of these two lines is the same:

* A type name (a class is also just a type)
* A variable name
* An assignment operator
* A right-hand-side

For the second line, the right-hand-side is very simple; just a numerical value. The right-hand-side in the first line is more spectacular:

new Student();

This line reads: “please create an object of the type **Student**”. The keyword **new** is crucial here; this instructs the application that we wish to create a brand new object. The creation process is then set in motion, which essentially involves:

* Allocating a piece of memory, to hold the data that is part of the object
* Running a piece of initialisation code defined in the **Student** class definition, which ensures that the object is in a meaningful state once it has been created.

Note that we as users of the class do not need to know exactly what happens during this initialisation; we just need to know that the object is ready for use, once it has been created.

Once the line has completed, a brand new object of type **Student** has been created, and the variable **firstStudent** refers to that object. Note the term “refers to”, as op­posed “is equal to”. When we deal with objects, the assignment process is a bit more complex than assignment of primitive types like integers. When object creation is set in motion by the **new** keyword, we are in a sense making a method call. The return value of that call is a reference to an object. This also implies that the type of the variable **firstStudent** is not **Student**, but rather “reference to a **Student** object”. We discuss this distinction and its implications in a moment; for now, we will just see what we can do with this variable.

Given the variable **firstStudent** that now refers to a **Student** object, we can now interact with the object through this variable. Suppose that the class creator has decided that the **Student** class should contain a (public) property called **Name**. How do we then retrieve that property from the object? Like so:

string name = firstStudent.Name;

The most important thing to notice is the use of the “.” (dot) just after the variable **firstStudent**. This is how you specify that you want to interact with the object that **firstStudent** refers to. This is an extremely important point to understand! When you wish to interact with an object, you must specify what object you wish to interact with. Suppose we created not just one, but a couple of **Student** objects:

Student firstStudent = new Student();

Student secondStudent = new Student();

firstStudent.Name = "Allan";

secondStudent.Name = "Jane";

Console.WriteLine(firstStudent.Name);

Console.WriteLine(secondStudent.Name);

What will this code print on the screen? First ***Allan***, then ***Jane***. Even though both objects have the same type, they can easily be in different states. Just as most humans have different names, weight, height, etc., but are all of the class **Human**. A common beginner’s mistake is to write code like:

Student.Name = "Carl";

That does not make sense, because **Student** is not an object; it is the name of a class definition (it does, however, turn out that code like the above can make sense some­times, but we will cross that bridge when we get to it…).

Just as you might wonder how we get to know what classes we have available, you may also wonder what properties and methods we have available for (objects of) a particular class. Again, there might be various sources of information available, but the Visual Studio environment can also help you. As soon as you type the (.) dot after a variable that refers to an object, Visual Studio will pop up a list box with all those properties and methods you can make use of. Scrolling to a specific entry will often pop up some additional information about this specific entry. If you add comments formatted in a specific way to your own class definitions, those comments will actually also pop up when using objects of that class.

The example above illustrated how to retrieve a property from an object, i.e. a part of the state of the object. How about behaviors? Behaviors – in the form of methods – are invoked (or **called**, as we will usually say), in a very similar manner. Suppose we have defined a method **PrintInformation**, that does just that; prints out information about the object in a human-friendly way. Calling this method will look like:

firstStudent.PrintInformation();

This is very similar to what we just saw for properties, except for a subtle difference: the parentheses following **PrintInformation**. When a method is defined, the author can choose to define that a number of **parameters** must be included in a method call. A parameter is a piece of information that the method needs in order to do its job. Some methods do not need extra information – the **PrintInformation** methods finds the information it needs inside the object on which it is called, so the caller of the method need not provide any extra information. A method that does not need extra information – i.e. it takes zero parameters – will be called as above, using the method name followed by an empty set of parentheses.

Imagine now that we have a class that provides very simple mathematical methods like **Add**, **Multiply**, and so on. Would it make sense to have an **Add** method taking zero parameters? Not really – we need to tell the method what values to add. An **Add** method taking two parameters would make more sense, so we could make calls like **Add(3,7)**. The value returned by **Add** can be picked up in a variable, like:

int result = myCalculator.Add(3, 7);

Note the comma separating the two values; if a method needs more than one para-meter, the caller must provide them inside the parentheses, separated by commas. We used two specific values in the example, but we can actually put any sort of ex­pres­sion into a parameter list, as long as the type of the result matches the expected type of the parameter! This is also a very important point. We have here assumed that the **Add** method takes two integer values as parameters. A call like the below would therefore be illegal:

int result = myCalculator.Add("3", "7");

However, the below is indeed legal, but maybe a bit mind-bending:

int result = myCalculator.Add(myCalculator.Add(1,2), 3\*4);

Again; we can put any expression we can dream up into a parameter list, as long as it evaluates to a value of the expected type!

## A (very short) detour to Code Quality, part 2

The attentive reader has probably noticed that we use a slightly different standard for naming classes, properties and methods, than we have used for variables so far. For classes, properties and methods, we also use the camelCase typography, but now with a capitalised first letter! The argument is – again – that this is a widespread standard, and we see no reason not to adopt it as well. This standard is often referred to as PascalCase.

We also – again – note that we should strive to give all our classes, properties and methods descriptive names, to increase the clarity of the code. However, the naming of properties and methods should take the class within which they are defined into account. What does that mean? If you are creating a class named **Student**, and this class contains a property that can retrieve the name of the student, it might be temp­ting to name this property **StudentName**. Such a naming is too verbose. If you are dealing with a **Student** object, it should be pretty obvious what a property named **Name** will return.

## Final remarks on object creation

Previously in this chapter, we claimed that you could create a **Student** object in the following way:

Student firstStudent = new Student();

That is probably also correct, but it will depend a bit on what options the creator of the **Student** class has made available for object creation. Recall that we claimed that using the **new** keyword would cause certain actions to happen:

* A piece of memory is allocated, to hold the data that is part of the object
* A piece of initialisation code defined in the **Student** class definition is executed, to ensure that the object is in a meaningful state once it has been created.

The second part doesn’t really hold up, if you think about it. We have not given any details about the definition of the **Student** class, but it would be reasonable to expect it to provide several properties, for instance **Name**, **Address**, **DateOfBirth**, and so on. If you create a **Student** object as defined above, what state will it then be in? What would the **Name** property return? Probably nothing (e.g. an empty string), because we have not provided any information as part of the creation process! That is hardly a realistic modeling of the real world. In a school administration system, you would pro­bably not be able to create a new student in the system, unless some minimal amount of information is available (maybe name, address and social security number). Likewise, it seems reasonable to enforce a similar restriction on creation of **Student** objects. We should not be able to create a **Student** object with no informa­tion in it, since such an object is meaningless. Fortunately, the class creator can enforce such restrictions. With the knowledge we now have, we can see that the code for object creation looks similar to a method call:

Student firstStudent = new Student();

Note the (empty) parentheses. The object creation does in fact involve a method call, to this “initialisation code” we have mentioned before. Just as for any other method, the class creator can define that the method call for object creation must include a number of parameters. A more realistic version of the above code could then be:

Student firstStudent = new Student("Allan Smith", 1988, 2, 21);

In this case, we must provide four parameters (name and date-of-birth) in order to create a **Student** object. Now it seems more plausible that a just-created **Student** object will be in a meaningful state from the moment it is created.

Deciding exactly what information to consider mandatory for object creation will of course be highly situational, and it turns out that the class creator can provide several “versions” of the object initialisation code, each taking different sets of information as parameters. Ultimately, the requirement specification for the application will decide what versions that should be made available. In the terminology of class definitions, such a piece of initialisation code is known as a **constructor** (a method called when an object is “constructed”).

## Value types and Reference types

Before we dive into how to create class definitions ourselves, we need to understand a fundamental difference between objects and so-called **simple types** (sometimes also called primitive types).

Simple types are some of those types we saw early on in these notes, like **int** and **double**. We saw that we can e.g. create variables of a simple type, like:

int age;

We can also assign a value to such a variable, like:

age = 18;

We can even do both in one line, like:

int age = 18;

For objects, things looked a bit different. Object creation involved the use of the keyword **new**, like:

Student firstStudent = new Student("Allan Smith", 1988, 2, 21);

So, the syntax is a bit more complicated on the right-hand-side, when dealing with objects. The left side – where the variable is defined – looks pretty much the same. However, there is a subtle – but quite important – difference.

When you define a variable of a simple type like **int** or **double**, and subsequently assign a value to it, the content (when looking directly into the computer’s memory) of the variable will be that actual value, which is probably what you would expect. Such a variable is therefore known as a **value-type** variable.

However, if the type of the variable is a class (like above, where **firstStudent** is of type **Student**), the content of the variable is not the object itself, but instead a reference to the object. You may recall that we earlier on stressed that a variable like **firstStudent** has the type “reference to an object of type **Student**” rather than just **Student**. The reference is as such just an address specification into the memory of the computer; the important point to grasp is that the variable does not contain the object itself, only a reference or “handle” to it. Such a variable is thus called a **reference-type** variable. These considerations give rise to (at least) two questions:

* Why does this difference exist?
* Should I care?

The first question is hard to answer precisely without getting into rather technical details about computer memory management, but it is to some extent a consequence of the fact that simple types have been around longer than classes and objects. When the concept of classes and objects entered programming languages, it was realised that a more advanced form of memory management was needed, but the existing, simpler memory management was retained for the simple types. The fact that this difference exists is simply something we as programmers must embrace.

Should you care about it? Yes, because this difference has some consequences that will seem quite surprising, if you don’t know a bit about what happens “under the hood”. Buckle up…

Consider first two variables of a simple type, like **int**. We can create them like this:

int age1 = 18;

int age2 = 21;

These two variables will occupy two separate parts of the computer memory, as illustrated below:

Age2

21

Age1

18

We can change the values of the two variables as we wish, even using the value of one variable to set the value of the other:

age1 = age2;

age2 = 23;

Age2

23

Age1

21

For objects, things work in a different way. The statement

Student s1 = new Student();

will create a new **Student** object somewhere in memory, and set **s1** to be a reference to that object (we have momentarily suspended our intention to give descriptive names to all variables…):

s1

(Student object)

This is by itself not really something we need to think much about, since we know how to interact with an object through the reference ( e.g. **s1.Name**). Let’s bring one more variable into play, and update the code to:

Student s1 = new Student();

Student s2 = s1;

How many variables have we created? Two. How many **Student** objects have we created? Only one! But both variables (of reference type) now refer to the same **Student** object, like so:

s2

s1

(Student object)

Is that a problem? Not as such, but try to guess what the code below will print on the screen:

s1.Name = "John";

s2.Name = "Allan";

Console.WriteLine(s1.Name);

If this didn’t surprise you… well, good for you! Some might guess that ***John*** would be printed, since we set the name for **s1** to “John”. However, the next statement will overwrite that name, since **s1** and **s2** both refer to the same object! Saying that we “set the name for **s1** to John” is too simplified. What we actually do is to “set the name for the object referred to by **s1** to John”. As it happens, **s2** also refers to that object, thus overwriting the name we previously assigned. The assignment statement we saw previously:

Student s2 = s1;

will not create a new **Student** object, by only set the references equal to each other!

An additional difference between value-type and reference-type variables is the fact that reference-type variables can be set to refer to… nothing. The special keyword **null** is available for this particular purpose:

Student s1 = null;

This is often used in practice; you may have some sort of complicated object in your code, that is only created if certain circumstances apply. The fact that a reference-type variable can be equal to **null** does however make it more complicated to use such a variable for e.g. calling a method. Suppose that **s1** is indeed equal to **null**. What should happen if you try this:

Console.WriteLine(s1.Name);

**s1** does not refer to any object, so there is no way to retrieve a name… If you try this, you will see that Visual Studio will respond with an error message reporting a “null pointer exception”. Trying to use a null reference is a very common error in program­ming, and is something you will need to anticipate and handle, once you get to know what to do about it…

This chapter should have provided you with enough knowledge to use existing classes, by creating objects and using the available properties and methods. Next, we shall see how to create our own class definitions.

## Class definition elements

Being able to define your own classes provides you with the ultimate ability to create “packages” of functionality, that fit exactly to your needs. It is, however, still worth the effort to explore various sources for existing classes first. The amount of available classes is ever-growing, and one of those may be a close-enough fit.

With that in mind, we will embark on the – somewhat large – topic of creating classes from scratch. Visual Studio will help you get started: If you highlight a project in the Solution Explorer – this could be the **Sandbox** project we have used previously – and right-click to bring up the context menu, you can choose the entry:

***Add | New from template | Class***

Choosing this entry brings up a tiny dialog window, where you simply enter the name of the class you wish to create. To create a class called **Human**, simply type “Human” (without the “”) into the box, and hit the ***OK*** button.

This sets a couple of things in motion in Visual Studio. A new file called **Human.cs** is added to the project, and the file is also opened by Visual Studio in the editor area. The initial content of the file should look like this:

namespace Sandbox

{

public class Human

{

}

}

We need not worry about the first and last line; this is related to a unit of organisation called namespaces (see the chapter on Code Organisation), which is not important now. We will thus zoom in on this bit:

public class Human

{

}

This is an absolutely minimal definition for the class **Human**. With this, we can actually start to create objects of type **Human**, but they will not be very interesting, since they have neither state nor behavior.

What do the parts of this definition mean? Let’s break it down:

* First is the keyword **public**. This is an **access specifier**, which tells us that this class can be used by everyone else. You might wonder if you would ever choose otherwise, but once you create larger projects, it may make perfect sense to keep some classes “private”, i.e. only to be used internally in the application code. In these notes, we will always create public classes.
* Next is the keyword **class**. This simply tells us that a class definition will now follow.
* Then follows a **{**, often referred to as a **curly bracket**. This symbol – along with the counterpart symbol **}** – are the delimiters of the class specification. Every­thing related to the class definition must be placed within these brackets.

Since all class definitions must contain this bare minimum, Visual Studio is kind enough to generate this code automatically.

The specific content of a class definition will vary from class to class. However, the content falls in a few well-defined categories, which we will detail in the following.

### Instance fields

We have seen examples of variables several times, and variables will usually also be part of a class definition. However, variables in a class definition can have different purposes. Some variables will be used inside methods (we get to method definitions very soon), but other variables are used for representing the state of an object of the class. These variables are called **instance fields**. The word “instance” signify that whenever a new object is created, this object will contain its own set of instance fields, that are independent of the instance fields in other objects. If you change the value of an instance field in one object, it will not effect the corresponding instance field in any other object.

If we wish to add an instance field to the **Human** class, to hold the name of an indi­vidual, it will look like:

public class Human

{

private string \_name;

}

This sort of looks like what we have seen before, with a few additions. The keyword **private** indicates that this instance field can not be accessed from the outside. That is, if you create a **Human** object, and try to get hold of the value of the instance field (or try to change it), the compiler will protest – that is an illegal operation. Again, you may wonder why you would create an instance field and then hide it away; the main reason is that we want access to such an instance field to go through a so-called **property**. We have mentioned these properties in the previous chapter, and we shall see in a moment how to create a property.

Also, we have prefixed the name of the instance field with an underscore (\_). This has emerged as a standard for “branding” instance fields in a way that distinguishes them from plain variables. We adopt that standard as well, and encourage you to do so.

If we wish to add more instance fields to our class, we simply add them one after another (it is considered good form to define instance fields on separate lines):

public class Human

{

private string \_name;

private int \_height;

private int \_weight;

}

Note that we always specify instance fields to be “private”, since we wish to restrict access to instance fields to only be possible through **properties**.

### Properties

In the chapter about usage of existing classes, we stated that it is usually possible to retrieve certain “properties” from a given object. One example was an object of the type **Student**, where we assumed that a property called **Name** was available:

string name = firstStudent.Name;

Such a property only becomes available, if the creator of the **Student** class has decided to include such a property in the definition of the class. Including a **Name** property in the **Human** class also seems like a very natural thing. The initial code for creating such a property looks like this (the rest of the class definition is omitted):

public string Name

{

get { }

set { }

}

If you type this code into the **Human** class definition in Visual Studio, you will be in­form­ed that the code is invalid… Don’t be alarmed by this; we will insert the relevant code in a moment. First, we take a step back and see how a property can be used by someone who has created a **Human** object.

Imagine that somebody has created a **Human** object, like so:

Human firstHuman = new Human();

We also assume that the **Name** property is available for use, meaning that the below lines of code should be perfectly valid:

firstHuman.Name = "Adam";

Console.WriteLine(firstHuman.Name);

What goes on here? It seems like the **Name** property is part of the state of any **Human** object. The type of that part of the state is **string**, and we can do certain things with the value of that part of the state. We can

* Set the value to something we specify
* Get the value out again, and e.g. print it.

These are two separate operations, but they involve the same part of the state. If we wish to enable these two operations, we must specify it in the definition of the **Name** property. This bring us back to the code from before:

public string Name

{

get { }

set { }

}

What we now need to figure out is what statements we need to write, in order to enable the two operations. The statements needed to “get” the value should be written between the **{}** in the line with the **get** keyword, and likewise for **set**.

Let us consider the **set** operation. The result of that operation should be to update the state of the object to the given value, such that when somebody wants to **get** that property, that value is returned. So, the provided value must be stored inside the object somehow… That is exactly why we have instance fields! Previously, we defined an instance field of type **string**, with the name **\_name**. That seems like a proper place to store that value ☺.

How do we then “store” the provided value in **\_name**? This is how:

public string Name

{

get { }

set { \_name = value;}

}

One single statement does the trick. Note, however, that a keyword **value** is also used here (you have probably noticed by now that keywords are **blue**). The **set** operation is always called when an assignment statement involving a property is performed, like:

firstHuman.Name = "Adam";

In general, **value** is simply set to the value of the right-hand-side (be it a simple value or an expression). In this case, **value** will be set to “Adam”, meaning that the **set** operation will update the value of **\_name** to “Adam”, and thereby changing the state of the object.

Having all this in place, the **get** operation is relatively simple. We wish to retrieve the current value of the property **Name**, and since we have defined this value to be stored in **\_name**, we should simply return the current value of **\_name**:

public string Name

{

get { return \_name; }

set { \_name = value; }

}

Note the keyword **return**. This is also a quite important part of the C# syntax, meaning *“return the value of the expression following the keyword* ***return****, to the caller of this method”*. In this simple case, the value of **\_name** is returned, as we wished.

If you have never dealt with properties before, it can be difficult to figure out exactly when these **get** and **set** operations are invoked. The general rules are:

* If a property appears on the left-hand-side of an assignment statement (i.e. we are assigning a new value to the property), the **set** operation is invoked.
* In all other scenarios where a property is being used, the **get** operation is invoked.

If you have paid close attention so far, you might see a way to short-circuit all this fuss for retrieving and setting a single value. Recall that the keyword **public** is used to indicate that a certain part of an object is accessible for everyone, which is why all properties are marked as being **public**, as opposed to all the instance fields marked as **private**. So, why not simply mark the instance fields as **public**? We can then just write code like:

firstHuman.\_name = "Adam";

Console.WriteLine(firstHuman.\_name);

No more messing around with properties, that just do the same thing anyway! In this very simple case, there would not be any difference. However, it is one of the pivotal principles in Object-Oriented programming to keep the details of internal implemen­tation private, the motivation being that we can then change this implementation without affecting the external user. If we make the **Human** class slightly more complicated, we can see this idea in action.

In the chapter about public and private appearance of an object, we used a property called BMI (Body Mass Index) as an example. The BMI for a person can be calculated from the weight and height:

BMI = (weight in kilograms) / (height in meters)2

How would we implement a BMI property in the **Human** class? First the skeleton code for a BMI property:

public int BMI

{

get { }

set { }

}

First consider the **get** operation. Since the BMI is calculated as above, we can calcu­late the value to return by using the current values of **\_weight** and **\_height**, as below:

public int BMI

{

get { return (\_weight \* 10000)/(\_height \* \_height); }

set { }

}

The factor 10000 is just to get the units right. The important point is that the value is now calculated; it is not just pulled out of some instance field. We could have chosen to create a **\_bmi** instance field and store the value there (think about why this would be a bad idea), but we didn’t…but the user of the object doesn’t need to know this! All the user knows is that a BMI value can be retrieved from a **Human** object, but he does not know anything about how the value is created. It could be stored in an instance field, it could be calculated…it doesn’t matter.

What about the **set** operation? Does it even make sense to have a **set** operation for the BMI? Not really… Since the BMI is calculated, it should not be allowed to set it “manually”, since that value may contradict the calculated value. Can we then prevent statements like the below?

firstHuman.BMI = 12;

Indeed we can! We simply remove the **set** operation from the property definition:

public int BMI

{

get { return (\_weight \* 10000)/(\_height \* \_height); }

}

This turns the property into a “read-only” property; you can retrieve the value, but not change it yourself. This is a fairly common situation, and this is indeed the correct way to handle it – this is not a “hack”.

The property concept can be a bit mind-bending at first, since it can seem overly complex in many cases. If all your properties are simple one-to-one mappings to single instance fields, there is indeed not that much gained. However, as soon as you are in a situation where

* A property is something that needs to be calculated, or
* It is relevant to be able to execute some code before assigning a new value to an instance field

you will benefit from using properties. We have seen an example of the first situation already; the second situation could be if you wish to perform some sort of “sanity check” before you update the value of an instance field. Consider a **set** operation for a **Height** property:

set { \_height = value; }

What if somebody by mistake tries to set **Height** to -10…? That is hardly a sensible value of a height, so we should somehow detect and handle that error. That can be done by adding some code to the **set** operation before the actual assignment. Precisely what kind of code we should add is a topic for one of the later chapter.

### Methods

By now, you have hopefully recognised that **properties** and **object state** are closely related; if we wish to know – or change – part of the state of an object, we use pro­perties. But objects usually also have **behavior**. We can make the objects perform certain actions, that go beyond a simple state change. If we have an object represen­ting a collection of students (this could be part of a school administration system), a useful action could be to ask the object to add an additional student to the collec­tion (maybe in the form of a **Student** object). In code, it could look like:

Student aStudent = new Student();

SchoolAdministrationSystem admSystem = new SchoolAdministrationSystem();

// aStudent is updated with relevant data

admSystem.AddStudent(aStudent);

You could argue that this is also just an update of the state of the **admSystem** object, but even so, it is definitely an action that would be hard to implement in a meaningful way with properties only. The main point is valid anyhow; if you wish to enable objects of a certain class to perform such actions, you do so by defining **methods** in the class definition.

A method is essentially a collection of statements, that can be invoked by calling the method on a specific object. A method definition will always contain these elements:

* **An access specifier**: We have seen before that certain parts of a class definition can be declared as being **public** or **private**. This goes for methods as well. Some methods should be available for an external user, while other methods may simply be “helper methods”, that only exist for making the implementation of public methods easier, and should thus be marked as private.
* **A return type**: We saw that the **get** operation in a property definition needs to specify the value it returns. Methods may also return values to the caller, and we need to specify what type (e.g. **string**, **int** or maybe a class type) this value will have.
* **A method name**: Just as for e.g. variables, we need to give our method defi­ni­tion a name, so we can refer to it when we wish to call it. As mentioned before, the name should be as descriptive as possible, and be written in CamelCase with a capital first letter.
* **A list of parameters**: We have already seen that some methods may require the caller to specify a list of parameters, since the method may need information from the caller in order to do its job. In the method definition, this parameter list must be specified, including the type for each parameter.
* **A method body**: The term “body” here means the set of statements inside the method definition. When the method is called, the statements will be executed.

Let us see an example of a method definition. At this point, we only know a few kinds of rather simple C# statements, so we can only create some rather bland methods. A useful method on a **Human** class could be a method that returns a string represen­tation of the state of the object:

public string StateAsString()

{

string state = "Name is " + \_name + ", height is " + \_height +

", weight is " + \_weight;

return state;

}

If we dissect the definition according to the elements mentioned above, we get:

* **Access specifier**: Is **public**, so an external user can call this method.
* **Return type**: Is **string** – we can also see that the expression (here just a variable) after the **return** keyword indeed has the type **string**.
* **Method name**: Is **StateAsString**
* **List of parameters**: Is empty, which is perfectly valid. A method may take zero parameters
* **Method body**: The two lines of code between the { and the }.

An additional element is worth mentioning here: The first line of the method body contains a variable declaration, as we have seen many times before. When a variable is declared inside a method body, it is a so-called **local variable**. You may recall that a class definition can contain instance fields, that look almost like local variables; they have a type and a name (instance fields also have an access specifier).

Why do we distinguish between local variables and instance fields? An instance field is created when the object that encapsulates it is created, and only then. Likewise, it is destroyed when the object is destroyed. The lifespan of an instance field is therefore exactly the same as the lifespan of the encapsulating object. If a method call changes the value of an instance field, another method call will be able to access that instance field and retrieve the value. The value thus “endures” between method calls. This is not the case for local variables. A local variable is created when the method it is defined inside is called. Also – and that is the most important point – the local variable is destroyed again when the method call is finished. Local variables thus have a much shorter lifespan that instance fields, since they only exist during a method call; not before, and not after.

The obvious consequence is that if you need to save a value inside an object across method calls, you will need to save it in an instance field… and saving something in an instance field is exactly to change the state of an object! Creating and changing the value of a local variable is not considered a state change, since these actions will not cause any “permanent” change in the state of the object.

For further illustration, consider the below method for adding two given numbers and printing the result on the screen:

public void AddAndPrintResult(int a, int b)

{

int result = a + b;

System.Console.WriteLine(result);

}

Two points are of interest here: First, note that this method takes two parameters, which are specified in the parentheses just after the method name. Each parameter is specified by a type specification and a name, and parameters are separated by comma. Also, note the keyword **void**, that appears where we would expect a type specification. The type specification should be for the return type…but this method does not return anything! The keyword **return** is not present in the method body either. This is fine, but the C# syntax specifies that you must specify a type at this position in a method definition. Therefore, the keyword **void** simply means “no type”. The **void** type is used fairly often, so it is important to know its meaning.

With these final points, we are now capable of defining method as part of a class definition. Once we learn about more advanced C# language constructions, we can create more advanced method with richer functionality. A couple of more general remarks about methods are in place, though:

* When you create methods, you should also strive for clarity. If you are creating a public method with complex functionality, you may quickly end up with a method with many lines of code. Even though the method might work as speci­fied, it will then be a good idea to see if the method can be broken down into additional (probably private) methods, that can then be called from the public method. This should not change the functionality, but make the method easier to understand and maintain.
* When you add methods to a class, the methods should be relevant for that class. There is nothing that prevents you from adding a **Multiply** method to a **Human** class, but that doesn’t seem like a functionality that naturally belongs to that class. Figuring out what methods that should be present in a class is strictly speaking a software design matter.

Finally, you might wonder if properties are not just a special kind of methods. Indeed they are. The syntax is a bit different, and you do not have the same degrees of free­dom with regards to naming, parameters and return type, but you do have the ability to specify a “method body”, that will be executed when the **get** or **set** operation is invoked.

### Constructors

During the discussion about how to use an existing class, we saw an example of how to create an object of a specific class:

Human firstHuman = new Human();

We claimed that the statement on the right-hand-side will cause a new **Human** object will be created, and that a piece of “initialisation code” will be executed as well. The purpose of that code should be to ensure that the object is in a meaningful state from the moment it is created.

This “initialisation code” is also written as part of the class definition, in the form of a **constructor**. A constructor is also a just a method, but with a special syntax and some limitations compared to ordinary methods. For our **Human** class example, a very simple constructor could look like this:

public Human()

{

}

This is as simple as it gets. Compared to ordinary method, we again see an access specifier as the first part. However, there does not seem to be a return type… That is the first important difference; a constructor can never return a value to the caller, so no return type – not even **void** – is specified. Next, we see the word “Human”, i.e. exactly the same name as the class. This is also a defining characteristic for a con­structor; it always has the exact same name as the class it is defined in. Next follows the parameter list. The list is empty in this example, but we are allowed to specify parameters to a constructor, in the same manner as for ordinary methods. Finally follows the method body, on which there are no restrictions either.

The most important point is to understand that once the statement containing the **new Human()** part is executed, the code inside the constructor will be executed. In that sense, you can say that **new Human()** is a method call, that:

1. Creates a new **Human** object
2. Executes the code within the constructor’s body, on the newly created object
3. Returns a reference to the **Human** object to the caller

As said above, the caller would then expect the object to be ready to use, and thus be in a meaningful state from the start. If we assume that the **Human** class contains the instance fields we defined earlier on:

public class Human

{

private string \_name;

private int \_height;

private int \_weight;

}

then it would seem that the simple constructor – which does nothing – is not good enough. What would the values of the three instance fields be? C# does set the initial value for e.g. an **int** to a well-defined value (zero), and for a **string** to the empty string, but that is hardly meaningful either. What then? Should we set the values to some sort of “default” values, like:

public Human()

{

\_name = "Adam";

\_height = 180;

\_weight = 80;

}

That’s not really what we want either. The essence of the problem is that we should not be able to create a **Human** object, until we have enough meaningful information about it. We discussed a similar problem for the **Student** class; you should not be able to create an “empty” **Student** object, since it does not make sense.

We can impose such a restriction by adding parameters to the constructor definition. If we define – and that should probably be a design decision – that you cannot create a new **Human** object without knowing the name, height and weight, you should define the constructor as below:

public Human(string name, int height, int weight)

{

\_name = name;

\_height = height;

\_weight = weight;

}

In this way, the previous simple statement **new Human()** becomes invalid and uncom­pilable. It now becomes mandatory for the caller to provide the information needed to create a meaningful **Human** object:

Human firstHuman = new Human("Adam", 180, 80);

This is a very sound principle: Define your constructor in a way that makes it impos­si­ble to create an object in a meaningless state.

For completeness, it should be noted that you can actually define more than one con­structor for a class. This could e.g. reflect a situation where you would prefer that cer­tain information is available when an object is to be created, but you will also allow creation with less information. For a **Student** class, you could make one constructor that takes all relevant information (name, address, country, phone, CPR number, and so on), but maybe also allow a version that only requires name and CPR number. Again, such requirements should be resolved during design.

You might get the impression from the above that a caller should provide initial values for all instance fields in a class. That is definitely not the case! Consider if a **Student** class also contained a field called **\_numberOfExamsPassed**. What would be a reason­able initial value for this field? Zero, of course! And that would be the case for all new **Student** objects. In general, the caller should only need to provide initial values for those properties that are individual for an object.

## Class collaboration, and a bit about Abstraction

Once we know how to create our own classes, we can start to build more complex models, involving more than one class. We recommended earlier that you should break a complex method into a set of simpler methods, that can “collaborate” to implement complex functionality. Likewise, you should strive to create simple classes, that are closely related to specific aspects of your model. Suppose we wish to create an application for simulating a car – this could perhaps be part of a racing game. We would probably create a **Car** class, and fill in functionality relating to various aspects of a real-life car. A real-life car is a very complex system, and you can perceive it as a set of sub-systems, that collaborate in a well-defined way. You could see the engine, the lighting system, the navigation system, etc.. as exam­ples of such sub-systems. If we cram all the functionality into a single **Car** class, it will end up being very complex. It would be a better approach to create classes corre­sponding to the sub-systems, like an **Engine** class, a **NavigationSystem** class, and so on. The role of the **Car** class would then be to hold the sub-systems together, and coordinate various actions between the subsystems. In C# code, we could imagine that part of the **Car** class could look like this:

public class Car

{

private string \_modelName;

private Engine \_theEngine;

private NavigationSystem \_theNavigationSystem;

// many other fields would probably follow...

public Car(string modelName)

{

\_modelName = modelName;

\_theEngine = new Engine();

\_theNavigationSystem = new NavigationSystem();

// (rest of constructor)

}

public void Start()

{

\_theEngine.Start();

\_theNavigationSystem.Start();

// (rest of Start method)

}

// many other methods would probably follow...

}

First, note that we now have instance fields that are of a reference type. For instance, the field **\_theEngine** is a reference to an **Engine** object. This is perfectly valid, and is a consequence of the notion of seeing a car being “composed” by sub-systems. When a **Car** object is created, we expect it to be in a meaningful state after creation, so it is quite natural that the **Car** constructor should ensure that a new **Engine** object and a new **NavigationSystem** is created. Likewise, the **start** method “relays” the starting command to the subsystems, and the **Car** object thus acts as the coordinating entity. You can take this idea further, and imagine that **Engine** and **NavigationSystem** are themselves composed of sub-systems, until a point where the sub-systems become so simple that further decomposition is unnecessary.

What we see here is also a first example of an extremely important concept in Object-Orientation: **abstraction**. We did intentionally not write *“important concept in Object-Oriented programming”*, since it is strictly speaking a design concept. However, modern software development does not distinguish software design and software development as sharply as it was traditionally done, so we can discuss the concept here as well.

**Abstraction** is the idea that you should be able to work with software development at various levels of “abstraction” or “complexity”. What does that mean? If you investi­gate how various car industry professionals work with the development of a real-life car, you will probably quickly realise that nobody knows all the details about the car… and that is a good thing!

A systems engineer may have “sub-system communication and coordination” as his area of responsibility. He will need to figure out how the various sub-system need to work together, but he will not know – and will not need to know – the details about how e.g. the navigation system works internally. All he is interested in knowing is how that sub-system interacts with the outside world. Of course, that needs to be speci­fied in sufficient detail. Once he knows that, he can use the sub-system as he sees fit, without knowing what goes on inside it. So, the systems engineer works at that parti­cular **level of abstraction**. A navigation system engineer probably works at a lower level of abstraction – he needs to work with all the internal details of the navigation system, but that system may itself rely on other, smaller sub-systems, and so on. As long as you know how to interact with a subsystem at your particular level of abstrac­tion, you don’t need to know about internal details.

You can hopefully see that this way of thinking fits very well to the main features of object-oriented programming; you can specify how an object presents itself to the outside world – i.e. how the outside world should interact with it – and then hide away the details of implementation inside the private sections of the object.

## Static – no object needed

Over the last pages, we have again and again insisted that you define methods in classes, and call methods on objects. This is a very clean distinction, but there are in fact situations where you don’t need objects in order to call methods.

Suppose you have a class called **SimpleMath**, that contains simple methods for addi­tion, subtraction and so on. The “header” of an **Add** method will probably look like:

public int Add(int a, int b)

So, the **Add** method takes two parameters **a** and **b**, and returns the sum. Very simple. So simple, that it is hard to see why we even need an object to call the method on… We said earlier that some methods may need parameters, in order to provide infor­mation needed to do its job. In this case, all the information needed is provided as parameters. If called on an object, the method would not change anything in the state of the object. In total, there are not any good arguments for having to create a **Simple­Math** object. Instead, the method can be declared as a **static** method:

public static int Add(int a, int b)

{

return (a + b);

}

We can now call the method like this:

SimpleMath.Add(2, 6);

We do not create a **SimpleMath** object, but simply call the method “on the class”. This may seem confusing, now that we have been accustomed to calling methods on objects, but it is an important feature to know. In fact, we have been using a static method almost from the beginning:

Console.WriteLine("WriteLine is a static method!");

**Console** is the name of a class, not an object! If you investigate the .NET Framework class library further, you will find that static methods are quite common. The very useful **Math** class is filled with static methods.

You can apply the **static** keyword to all the elements of a class definition: instance fields, proper­ties, and even on the class itself. If you define a class as being static, it becomes impos­sible to create an object of that class. Also, a static class can only contain static elements (how would you access a non-static element, if you cannot create an object…?).

Having static elements in a non-static class is however possible, and quite common. A very common static element is a so-called **constant**. A constant is a variable that cannot change its value. That sounds a bit contradictory, since you would expect a “variable” to be able to change its value… However, you will often need to use some fixed value in your code, the classic example being the value of **π** (pi). That value is actually found in the **Math** class, called **Math.PI**. You can create a constant inside a class definition like this:

public class CardDeck

{

public const int CardsInDeck = 52;

}

Notice the keyword **const** – this defines the instance field to be a constant. We also need to specify the value of the constant as well. You might expect that you should also add the keyword **static**, in order to make the constant static. However, since constants are always considered to be static – since there will never be a reason to create more than one instance of a constant – you don’t need to specify it explicitly.

For someone new to programming, it might be difficult to figure out when to declare a class element to be static. The question you should ask yourself is: *“does this ele­ment depend on the state of individual objects?”*. If the answer to that question is no, you can declare that element as being a static element. It is in fact recommended that you do this! Declaring something as static is not “cheating” or a “hack”; it is a way to inform the user of the element that he can use it without having to create an object first. This saves both time and code.

# Programming – next level

So far, we have only learned about a few types of individual C# statements, which we can combine to sequences of statements. This sequence of statements will always be executed in the same order. That is obviously a severe limitation – it is very easy to imagine some business logic that involves **choices**: If a certain condition is fulfilled, we will perform certain actions; if the condition is not fulfilled, we may perform different actions, or maybe no actions at all. We therefore need additional C# statements that enable us to implement such choices in C# code. Such statements are of course avail­able, and are in general known as **control statements**. These state­ments control the “flow of execution” of the other statements, and allow us to create much more sophisticated code.

The two main categories of control statements are

* Conditional statements
* Repetition statements

## Conditional statements

In the category of conditional statements, we find

* The **if**-statement
* The **if-else** statement
* The multi-**if-else** statement
* The **switch** statement

As you can probably tell from the naming, the first three statement types are essen­tially variations over the same format, while the **switch** statement is fundamentally different.

### The if-statement

As stated above, we will need to be able to implement conditions, if we want to implement non-trivial logic in C#. The **if**-statement is the first tool for this. The syntax for an **if**-statement is quite simple:

if (*condition*)

{

}

An **if**-statement always contains three defining elements:

* The keyword **if**, followed by
* A pair of parentheses, containing a **logical condition** (for completeness, note that the word ***condition*** is not a C# keyword; it just indicates where the con­dition should be written)
* A **code block**, inside the curly brackets **{** and **}**. A code block is essentially the same thing as a **method body**, i.e. a sequence of C# statements.

The most interesting feature is the **condition**. A condition is in this context a **logical** condition, which in plain terms is “something that is either true or false”. We have seen such a thing before: a **logical expression**. That’s what a condition is: it is a logical expression, that will evaluate to either **true** or **false**. The expression itself can be very simple or very complex, but whenever the “flow of execution” reaches the condition, the logical expression will be evaluated.

If the logical expression evaluates to **true**, the statements inside the code block will be executed. We should also note that the statements are executed once (we will later see why this is important to remember). If the logical expression evaluates to **false**, the entire code block is skipped! The condition is thus a sort of “gatekeeper”; it will only allow the “flow of execution” to enter the code block, if the condition is **true**.

What happens after an **if**-statement? When the **if**-statement is done – which may or may not include execution of the statements in the code block – the first statement following the **if**-statement is executed. In that sense, nothing is changed. We still execute statements in the order they are written, but the statements themselves can now be more complex.

A very simple example of how to use an **if**-statement is given below:

int age = 17;

Console.WriteLine("Starting to check age...");

if (age < 18)

{

Console.WriteLine("You are still a child...");

}

Console.WriteLine("Finished checking age");

In a more realistic situation, we could imagine that **age** is read from an external source. If we run the above code, the output will be:

Starting to check age...

You are still a child...

Finished checking age

Why is the code block executed? When the **if**-statement is reached, the value of **age** is 17, so the condition becomes: 17 < 18. That is indeed true, so we enter the code block and execute the statements inside it (in this case only a single statement).

If we run the same code again, but change the value of **age** to 19, the output is:

Starting to check age...

Finished checking age

In this case, the condition evaluated to false, so the code block was skipped.

If the above code was part of a real system, we would probably like the system to tell us the result of the check, no matter what the result is. We could do this by using two **if**-statements:

int age = 17;

Console.WriteLine("Starting to check age...");

if (age < 18)

{

Console.WriteLine("You are still a child...");

}

if (age > 18)

{

Console.WriteLine("You are an adult!");

}

Console.WriteLine("Finished checking age");

That seems to solve the problem, since the application now provides an answer in all cases…or does it? It is probably easy to see that we can never have a situation where both possible answers are given, since both conditions cannot be true at the same time. Is there a situation where no answer is given? Yes, if the value of **age** is exactly 18. In that case, both conditions will be false, so no answer is given.

Even for such a simple situation, it can be a bit difficult to spot the error. What we really want is to be absolutely sure that exactly one answer is given, no matter the value of **age**. Sure, we could modify the second condition to (**age** >= 18), but a better (less error-prone) solution is to use a slightly extended version of the **if**-statement, called the **if-else** statement.

### The if-else statement

The syntax for the **if-else** statement is:

if (*condition*)

{

}

else

{

}

Compared to the **if**-statement, we have added to elements:

* The **else** keyword
* An additional code block

The functionality is probably not too hard to guess: If the condition evaluates to true, we execute the first code block only. If the condition evaluates to false, we execute the second code block only. We are thus guaranteed that exactly one of the two code blocks is executed; never both, never none! This construction is less error-prone than the construction using two **if**-statements, and it is strongly recommended to use the **if-else** statement, if you wish to execute exactly one of two code blocks, depending on the value of a logical expression.

For completeness, it should be mentioned that if – as in the example above – you have code blocks containing just a single statement, you can omit the curly brackets, and write the code like:

if (age < 18)

Console.WriteLine("You are still a child...");

else

Console.WriteLine("You are an adult!");

Even though this is syntactically valid code, it is recommended to always use the **{}** delimiters, since it makes the code easier to read. Also, be aware that the below code is also syntactically valid:

if (age < 18);

Console.WriteLine("You are still a child...");

If you run this code, the code will always print the ***you are still a child…*** message, no matter the value of **age**. Note the semi-colon (;) after the condition… This is allowed, but fairly meaningless. The code now checks the condition…and does nothing. How­ever, the semi-colon indicates that the **if**-statement is now completed, so the next statement is not considered part of the **if**-statement any more… An error like that can be hard to spot, but Visual Studio is clever enough to identify it, and puts up a do-you-really-mean-that warning.

Concerning use of semi-colon, it might also seem strange that **if**-statements and **if-else** statements are not completed with a semi-colon. Whenever you use a code block with the **{}** delimiters, it is not required to use semi-colon to end the statement.

Finally, you may wonder if there are any restrictions on the type of C# statements, we can place inside a code block being part of a conditional statement. There isn’t any. Inside the code block of an **if**-statement, you could for instance put…another **if**-state­ment! Imagine you want to check if somebody is a teenager:

if (age > 12)

{

if (age < 20)

{

Console.WriteLine("You are a teenager");

}

}

In order to enter the outermost code block, **age** must be larger than 12. Once inside that block, **age** must also be smaller than 20 to enter the innermost code block. Only then will the message be printed.

By adding **if**-statements to our repertoire of C# statements, we can suddenly create quite complex methods, by combining statements to whatever “depth” we need. In the above case, it would however be a better solution simply to use a single **if**-state­ment, with a somewhat more complex condition:

if ((age < 20) && (age > 12))

{

Console.WriteLine("You are a teenager");

}

The logic is, of course, the same. Exactly how to “balance” a complex conditional statement between complex conditions and nested statements is mostly a matter of taste and readability.

You often encounter situations where you need to perform a very simple action, depending on whether or not a condition is true or false. This could be assigning a value to a variable, where the value will depend on the condition, like:

int age = 15;

string message = "You are ";

if (age < 18)

{

message = message + "a child.";

}

else

{

message = message + "an adult.";

}

In such cases, it can be convenient to use the so-called **ternary operator**:

*condition* ? *expression1* : *expression2*

This reads: if the condition is true, return the value of **expression1**, else return the value of **expression2**. If we rewrite the above **if**-statement using the ternary operator, we get:

message = message + ((age < 18) ? "a child." : "an adult.");

This is definitely more compact. The logic is exactly the same, so using the ternary operator is mostly a matter of taste.

### The multi if-else statement

The third variant of **if**-statements is useful in situations where you wish to choose between several (i.e. more than two) alternative actions. Suppose we need to find the appropriate mark for a test, where the score for the test is given as a percentage, i.e. a number between 0 and 100. The logic for finding the mark could be like:

|  |  |  |
| --- | --- | --- |
| **Lower limit** | **Upper limit** | **Mark** |
| 0 | 39 | D |
| 40 | 69 | C |
| 70 | 89 | B |
| 90 | 100 | A |

This logic can be written as a somewhat complex nested **if-else**-statement:

if (score >= 90)

{

Console.WriteLine("Mark is: A");

}

else

{

if (score >= 70)

{

Console.WriteLine("Mark is: B");

}

else

{

if (score >= 40)

{

Console.WriteLine("Mark is: C");

}

else

{

Console.WriteLine("Mark is: D");

}

}

}

Feel free to check the logic yourself… This code is perfectly fine, but can be a bit hard to understand due to the deep nesting of statements. The multi-**if-else** statement allows you to formulate the code slightly different:

if (score >= 90)

{

Console.WriteLine("Mark is: A");

}

else if (score >= 70)

{

Console.WriteLine("Mark is: B");

}

else if (score >= 40)

{

Console.WriteLine("Mark is: C");

}

else

{

Console.WriteLine("Mark is: D");

}

The code does exactly the same as the previous example, but is easier to read. It also makes it clearer that the four alternatives are equally “worthy”, which may be harder to see when using nested statements. If you compare this with the simple **if-else** statement, we have actually just added a number of **else-if** blocks between the **if**-part and the **else**-part. An important detail to remember is that a multi-**if-else** statement must end with an **else**-part.

Finally, we note that even though it is possible to formulate “overlapping” conditions (such that more than one condition can be true at the same time) in a multi **if-else** statement, it is still only one code block that will be executed; the first code block where the corresponding condition is true, or alternatively the **else** code block.

### The switch statement

In some situation, the business logic has a nature where there is a distinct outcome for certain specific value of a variable. Maybe the logic for calculating child support could be like this:

|  |  |
| --- | --- |
| **Number of children** | **Child support (kr. per month)** |
| 0 | 0 |
| 1 | 1200 |
| 2 | 2000 |
| 3 | 2600 |
| >3 | 3000 |

There is no simple formula for this dependency, so we would probably implement this logic by using a multi-**if-else**-statement, in a manner similar to the previous example. However, the **switch**-statement allows us to “directly” choose an alternative based on a specific value:

switch (noOfChildren)

{

case 0:

childSupport = 0;

break;

case 1:

childSupport = 1200;

break;

case 2:

childSupport = 2000;

break;

case 3:

childSupport = 2600;

break;

default:

childSupport = 3000;

break;

}

The important features of a **switch**-statement are:

* At the outermost level, we use the keyword **switch**, followed by the expression (typically just a variable) that we “switch on” in parentheses.
* We write a **case**-statement for each of the cases that we wish to handle indivi­dually, using the keyword **case** followed by the actual value, followed by “:” (colon, not semicolon!)
* Each case contains a sequence of statements, concluded by a **break**-statement. The **break**-statement indicates that no more of the code within the **switch**-statement should be executed. It is perfectly legal to include **if**-statements, etc. in the code before the **break** statement, but often you will just put a single line of code. If you need many lines of code for each case, a **switch**-statement may not be the best choice.
* If the value is not handled explicitly by a matching **case**-statement, it is caught in the **default**-statement, and the lines of code specified here are executed.

Again, there is nothing you can do in a **switch**-statement that you cannot do in a **if**- statement; they are logically equivalent.

In general, when dealing with conditional statements, you should choose the type of state­ment that you feel is a best fit for the problem at hand, and makes the code as easy as possible to understand.

## Repetition statements

The various conditional statements allow us some control of the flow of execution, but mostly in the sense that we may or may not execute certain sections of code. A very useful ability would be the ability to repeat execution of a code block, as long as some condition is true. The so-called **repetition statements** enable us to do just that.

Just as for conditional statements, there are a few variants of repetition statements to choose from, but they are essentially all variations over the same theme: repeat exe­cution of a block of code, until a condition becomes true.

The two most common types of repetition statements are:

* The **while**-statement (or **while**-loop)
* The **for**-statement (or **for**-loop)

A third type called **foreach**-loop is also quite useful, but only in relation with so-called **collections**, which we will present later.

### The while-loop

The syntax for the **while**-loop is quite simple, and is almost identical to that of the **if**-statement:

while (*condition*)

{

}

Syntactically, the only difference is the keyword **while** (instead of **if**). Apart from that, the **while**-loop consists of the same three elements:

* The keyword **while**
* A logical condition
* A code block

The functionality is different, though. The **if**-statement will – depending on the value of the logical expression in the condition – execute the code block one or zero times. The **while**-statement will execute the code block several times, more specifically as long as the condition is true. The word “several” is a bit misleading; the code block might only be executed once, or not at all! The exact number of **iterations** – meaning the number of times the code block is executed – depends on the condition.

The exact way a **while**-loop works is probably best described by a number of steps:

1. The “flow of execution” reaches the **while**-loop.
2. The condition is evaluated, giving either **true** or **false**.
3. In case of **true**:
   1. the code block is executed, and
   2. the flow of execution returns to step 2.
4. In case of **false**: the **while**-loop is considered completed, and the flow of execution will continue with the first statement after the **while**-loop.

The entire difference between a **while**-loop and an **if**-statement lies in step 3b. If you remove step 3b from the above, you have a precise description of an **if**-statement! The ability to send the flow of execution back to a point it has already passed, makes all the difference.

A valid concern here is how you ever get out of such a loop again! Won’t the condi­tion keep on being true, if it was true the first time around? That is indeed possible, and you (well, the “flow of execution”) would then be stuck in an **infinite loop**. The only way to break out of such an infinite loop is to terminate the application! To avoid this situation, something must happen in the code block that can affect the value of the condition. This is a very important point about any type of loop statement.

Let’s see an example of a fairly simple **while**-loop:

int number = 1;

while (number < 5)

{

Console.WriteLine("The value of the number is {0}", number);

number = number + 2;

}

If you follow the steps above, you should be able to work out how many times the code block is executed: two times. The first time the condition is evaluated, the con­dition is 1 < 5, which is **true**. The message is printed, and the value of **number** is then increased by 2. Next time around, the condition is 3 < 5, which is also **true**. We do the same steps again. Third time around, the condition is 5 < 5, which is **false**. So, we do not do a third iteration through the loop. The loop has now finished.

This is of course a simple, not very useful example, but we can also use it to illustrate how easy it is to get into trouble. Consider the code below:

int number = 1;

while (number < 5)

{

Console.WriteLine("The value of the number is {0}", number);

}

It looks fairly innocent, but if we try to run it, we will be stuck in the dreaded infinite loop. We took out the statement that increased the value of **number**, so the condi­tion will now remain true forever… Visual Studio is also clever enough to see this and warn us about it, so you might run into fewer infinite loops than the programmers before you have ☺.

Just as for the conditional statements, there are no restrictions on the code you can put into a **while**-loop code block. You can put a **while**-loop inside a **while**-loop, if that is what your logic dictates. Nested loops are quite common in programming.

The example above is a so-called **counter-controlled** **while**-loop; we use the numeric value of some variable to check if we should do another iteration through the loop. This is typically used when the **while**-loop should be repeated a fixed number of times, or at least a numeric value known when the start of the loop is reached.

A different situation arises if you want a **while**-loop to continue until a certain “event” occurs. What does that mean? Suppose we have defined a class **Reader**, which can read a single character from the keyboard (i.e. the key the user presses). We could then imagine code like the below:

Reader myReader = new Reader();

string keyPressed = "";

while (*condition*)

{

keyPressed = myReader.ReadFromKeyboard();

Console.WriteLine("You pressed {0}", keyPressed);

}

The idea is that **ReadFromKeyboard()** will wait until the user has pressed a key; when that happens, the key is read and stored in **keyPressed**. We then keep doing this, until…what? We could perhaps do it, say, ten times, but what we really want is for the user to tell us when he wants to stop. Of course, the user could just terminate the application, but if we want the user to be able to stop the loop, we could do this by defining that a certain value of **keyPressed** means “stop”. We could e.g. choose the character ‘q’ (meaning “quit”). The code would then become:

Reader myReader = new Reader();

string keyPressed = "";

while (keyPressed != "q")

{

keyPressed = myReader.ReadFromKeyboard();

Console.WriteLine("You pressed {0}", keyPressed);

}

This means: as long as **keyPressed** is not equal to ‘q’, we keep on iterating (and there­by obtaining a new value from the user). Of course, we have no way of knowing if this will happen after one, four or 427 iterations, so we cannot use a counter-controller **while**-loop here. The above construction is called a **sentinel-controlled** **while**-loop. You can think of the condition as a “sentinel” that will only allow further iteration if a variable has (or does not have) a specific value. Also, the loop itself does not have control over the variable, but retrieves it from some outside source.

When you are new to loop statements, it can be easy to forget something. All loops do however have the below elements in common:

* **Initialisation**: Before the loop itself is entered, we usually – but not always – initialise some variable that is also used as part of the condition.
* **Condition**: The logical condition itself, which is evaluated before performing another iteration of the loop.
* **Change**: Since the condition itself is fixed, at least one of the values of the variables in the condition must have the chance to change during an iteration. Otherwise, we have an infinite loop. Formally put, the probability that some-thing changes must be larger than 0 (zero).
* **Code block**: The sets of statements that are executed during an iteration. Some of those statements must cause the change mentioned above.

You can use this as a “checklist” when creating a loop statement, to be sure that you have considered all the relevant elements.

### The for-loop

Let’s take a look at a very common, counter-controlled **while**-loop. Some comments have been added to the code, to identify the four elements we listed above:

int number = 1; // Initialisation

while (number < 5) // Condition

{

Console.WriteLine(number); // Code block

number = number + 1; // Change

}

This type of **while**-loop is very generic: do something a certain number of times, using an integer variable to track how many iterations we have performed. The loop state­ment known as the **for**-loop is tailored to this scenario. A **for**-loop implementing the same logic as the above **while**-loop looks like:

for (int number = 1; number < 5; number = number + 1)

{

Console.WriteLine(number);

}

If you compare this code to the previous code, you can hopefully see that we have just rearranged the four elements; the elements themselves are the same. In general, a **for**-loop has this structure:

for (*initialisation* ; *condition* ; *change*)

{

*Code block*

}

A **for**-loop and a **while**-loop are logically equivalent; there is nothing you can do with a **for**-loop that you can’t do with a **while**-loop. So why choose a **for**-loop over a **while**-loop? It is mostly a matter of taste. The structure of the **for**-loop does per­haps make it harder to forget one of the elements, since the **for**-loop looks a bit “odd” if you remove one of the elements. Still, this is a subjective criterion.

A slight drawback of the **for**-loop could be, that it is less obvious what order the operations are done in, and how often. Both are the same as for the **while**-loop: the initialisation is done once, as the first operation when the **for**-loop is reached. The condition is then checked, and – if the condition is true – the code block is then executed. After that, the change operation is done. The condition is then checked again, and so forth.

At this point, it is also relevant to introduce an alternative way to change the value of an integer variable. A very common format for the **for**-loop is this:

for (int number = 1; number < 5; number = number + 1)

{

// Code block

}

Having a “counter variable” that simply counts the number of iterations performed – and a corresponding condition that remains true as long as the desired number of iterations has not been performed yet – is very common, and you will often see the above written as:

for (int number = 1; number < 5; number++)

{

// Code block

}

This is exactly the same logic as before, except that the new notation **number++** is used. This notation simply means: increase **number** by one. It may look confusing at first, but since we very often need to increase an integer variable by one (also called to **increment** the variable), you will soon appreciate it. Similarly, you can decrease the value by one using the notation **number--**.

Finally, you should know that none of the elements in the **for**-loop are mandatory. You could in principle write a **for**-loop like:

for (;;)

{

// Code block

}

This is a legal **for**-loop, that will iterate forever…

Are there any general guidelines for choosing between a **while**-loop and a **for**-loop? Not really – as said before, it is mostly a matter of taste. Many prefer to use a **for**-loop when the iteration is controlled by a simple counter variable, and to use a **while**-loop when you have a sentinel-controlled scenario. Choose the loop type you are most com­fortable with, but try to choose in a consistent manner.

## Debugging

As soon as we start to use conditional and repetition statements, we can start to create much more interesting – and complex – code. This makes it much more fun to code, but it also makes it harder to get the code right the first time. You will now start to find yourself in that somewhat frustrating situation where:

* Your code is syntactically correct (and can compile)
* The code seems correct when you read it
* The code does not produce the results you expect!

No matter how much effort you put into the design, and no matter how brilliant a pro­grammer you become, you will inevitably spend a large part of your program­ming life in this situation. We therefore need to know about tools and techniques to help us find errors (or **bugs**, as they are often called) in our code. This process of finding and fixing bugs is called **debugging**.

A first – and quite obvious – technique is simply to read the code closely once again. In many cases, the bug is quite simple (like e.g. using a minus instead of a plus some­where), and surprisingly many bugs can be caught this way. Even better is to try to explain the code to someone else; the process of stating the intention of the code aloud will often reveal bugs. In lack of somebody to explain it to, just explain it to your dog or teddy-bear – it’s the act of explaining aloud that matters ☺.

If such manual techniques are not successful, you can add some statements to the code, that print out useful information to the screen, e.g. inside a loop. It could be the value of a variable that seems to cause the problem, like:

for (int divideBy = 10; divideBy >= 0; divideBy--)

{

Console.WriteLine("Trying to divide by {0}",divideBy);

Console.WriteLine(100 / divideBy);

}

This technique can be helpful if the logic is not too complex, but it quickly becomes a bit of a mess to add all these extra lines of code, and it is tedious to remove them again. A slightly better technique is to use the built-in statement **Debug.WriteLine()** instead; this prints to the *Output Window* in Visual Studio instead, and only prints when running the application in *Debug* mode. Still, adding and managing such printing statements is a somewhat old-fashioned approach to debugging, and we will gene­rally try to avoid it.

A more powerful and elegant technique is to use the so-called **integrated debugger**, which is part of Visual Studio. Almost all modern programming tools contain an inte­grated debugger.

The integrated debugger can help with debugging in a lot of ways, and getting familiar with the debugger is definitely a worthwhile investment if you are serious about pro­gramming. Still, you can get quite far with knowing just a few things.

A very useful concept is a **breakpoint**. A breakpoint is a position in the code (a specific statement) that you choose. When you then run the program, the program will pause (not terminate!) at the breakpoint. You can think of it a pausing the “flow of execu­tion” at this specific point in the code. You choose the location of a break­point simply by clicking in the leftmost part of the code editing window. A red dot will appear (you can remove the breakpoint simply by clicking on it again):



If you run the application now, the flow of execution will pause at the breakpoint, more specifically just before that line of code gets executed.

What now? Now you can inspect the current values of all variables simply by hovering the mouse cursor over the variable in the code editor window. This is in itself quite useful, since you may often find that the values are different from what you expected. If you then want the application to continue the execution from the breakpoint, you simply click the green triangle button, just as when starting the application (note that the text at the button now says *Continue*). Note that you can place multiple break­points in the code; hitting *Continue* will then resume execution until the next break­point is reached.

The above features – being able to pause the flow of execution at a break­point, and inspect variable values by hovering over them – are by themselves extremely useful for debugging. Getting used to using this feature will save you a lot of time. The next useful feature to know is the ability to “step” through the execution of the applica­tion. When you reach a break­point, a small set of buttons appear (they can also be found under the *Debug* menu):



There are a few more options available, but these three are the most important for now. These three options (and corresponding buttons) are called:

* *Step into*
* *Step over*
* *Step out*

These “step” buttons enable you to advance the flow of execution by a single state­ment at a time, just as if all lines of code contained breakpoints. The interes­ting question is then: what is the next statement to be executed?

The obvious answer would be the next line of code in the method where we placed the breakpoint. If you click the *Step over* button, the arrow marking the current posi­tion in the code will indeed advance to that statement (note that this need not be the line that “physically” follows in the code, since we might have conditional state­ments, loops etc. in play here).

But what if the statement we are currently at is a method call? If we want to investi­gate what happens inside that method call, we can use the *Step into* option. This will take us into the code for the called method. In this way, you can step infinitely deep into methods being called by other methods (cue music from ***Inception***...).

If you step through all the statements in the called method, you will eventually be returned to the calling method – if you want to be taken immediately back to the calling method, you can use the *Step out* option, which then finishes the method call. Using these three ways of stepping through the code is also extremely useful, since you can follow the flow of execution in every detail. In this way, you will often find that the code doesn’t behave as you expected.

In total: as soon as your code grows beyond trivial, it is a really good investment to familiarise yourself with the most fundamental features of the integrated debugger. The best way to learn this is – as always – to practice! There are more advanced features in the debugger than mentioned here, but get some practice in using the features described here, before diving deeper into the capabilities of the debugger.

## Data structures - fundamentals

Learning about control statements enable us to go far beyond simple, sequential bits of code. Likewise, we now need to go beyond simple variables, that can only contain a single value. In many situations, we need to handle multiple values that have some­thing in common. Examples could be:

* Names of students in a class
* Temperature measurements over a long period
* Information about cars that can be rented at a Car Rental Service

…and so on, and so on. We could in principle just declare a lot of individual variables for e.g. holding the names of students, but a such an approach quickly becomes very clumsy to implement. Therefore, we need ways to handle such **collections** of data more elegantly. Fortunately, all modern programming languages – C# included – contain a number of ways to handle collections of data, by using so-called **data structures**.

### Arrays (and why you should not use them…)

One such data structure is the **array**. The array is a classic data structure, and it has been around for much longer than Object-Oriented programming. Therefore, it doesn’t quite “fit” into such a language as C#, since the syntax for using it has a non-Object-Oriented flavor to it. So why learn about it at all? Even if better alternatives do exist, you may still encounter use of arrays in existing code, and certain parts of the array syntax has leaked into more modern data structures. So, knowing about arrays is still useful background knowledge to have as a programmer.

Before diving into the syntax for arrays, we describe them on a conceptual level first. In essence, they are just a construction for handling a set of values of the same type (e.g. **int** or **string**). We can think of an array as a line of boxes, into which we can put a single value. Below is an array of integers:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **index** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| **value** | 34 | -233 | 9801 | 67 | 2 | -9582 | 770 |

Note that the “line of boxes” mentioned above is only the shaded part of the drawing, in the row labeled *value*. So, we have a total of 7 integer values in this particular array. What is the *index* then? The *index* is the “address” of a specific element in the array. When we put something into the array, we have to specify which “address” it should have. The *index* is exactly that. We can then use the index again, when we wish to retrieve an element from the array. The index itself is just an integer number; the only peculiarity is the fact that an array index always starts from 0 (zero). This also means that in this particular array, the last element has index 6 (not 7)!

Let us now see how to use arrays in C#. More specifically, we need to be able to:

* Create an array
* Enter a value into an array
* Retrieve a value from an array

The syntax for array creation is the first place where the pre-OO nature of arrays start showing. First, we need to declare a variable that can refer to an array:

int[] myFirstArray;

We deliberately used the phrase “refer to”, since an array is indeed an object, and a variable of an array type is thus by-reference. This particular variable refers to an array of **int** values. The important thing to notice here is the use of **[]**, called **square brackets**. A proper declaration of an array variable is thus the type for the values it should contain, followed by [].

Just declaring the variable does not create the array itself. Creating the array looks like:

myFirstArray = new int[7];

This will create an array with 7 elements, that can contain an **int** value. Again, notice the somewhat odd syntax. We are indeed creating a new object here, by using the **new** keyword. However, there is not as such a class corresponding to the array…

With this, we can start to use the array. If we wish to put a value into the array, we must specify the index of the element into which we put the value:

myFirstArray[3] = 23;

This puts the integer value 23 into the element with index 3 (i.e. the fourth element in the array). There is not more to it than that. If you wish to retrieve an element from the array, you must again use the index:

int singleValue = myFirstArray[5];

The important thing to realise is this: once you specify an element in an array by its index, that element can be used in exactly the same manner as a simple variable of that type. You can use it in expressions, you can assign a value to it, and so on. If you want to increase the value of an element by one, you could just write:

myFirstArray[4]++;

This is no different than writing e.g. **age++**.

In the above example, we just created an array of **int** values, with 7 elements. We did not really specify the content of the array. Is the array then empty initially? No, since the concept “empty” doesn’t quite make sense for an array. Once the array is created, it immediately has the specified size, and the content of each element is then inter­preted as an **int** value. In C#, the elements in a array of numerical values are set to 0 (zero) by default, so that will be the initial content of this array. If we already know the initial values when we want to create the array, we can use a handy syntax to put those elements into the array:

int[] myFirstArray = new int[] { 34, -233, 9801, 67, 2, -9582, 770 };

This will create an array with 7 elements and put the specified values into the array. Neat!

So far, we have not gained that much as compared to just using simple variables. Arrays do however go hand-in-hand with repetition statements. Suppose we want to print all the elements in an array. That is quite easy to do with a **for**-loop:

for (int index = 0; index < 7; index++)

{

Console.WriteLine(myFirstArray[index]);

}

That’s it! The variable **index** starts at 0, and keeps increasing until it reaches 7, where the condition becomes false. Also, this code (almost) doesn’t change if we have an array with 10000 elements instead of 7. The only small problem with this code is the explicit use of the size of the array. If we do change the size of the array, we must also change the condition in the loop. Can we avoid that? Yes, since you can retrieve the length of the array from the array itself:

for (int index = 0; index < myFirstArray.Length; index++)

{

Console.WriteLine(myFirstArray[index]);

}

This is a more robust style, since it is no longer necessary to change the condition, no matter the size of the array.

The above is a simple – but still quite typical – example of how arrays are used: Do some operation for each of the elements in the array (in this case: print them on the screen). Since this usage pattern occurs so often, a variant of repetition loops have been created just for this purpose, called a **foreach**-loop:

foreach (var value in myFirstArray)

{

Console.WriteLine(value);

}

Here you don’t even need to worry about array sizes, loop variables and so on: just specify the name of the array, and what you want to do with each element. The gene­ric version of the **foreach**-loop looks like this:

foreach (var *variableName* in *arrayName*)

{

// Whatever you want to do with the value

}

You can choose whatever variable name you prefer; it is just a “placeholder”, that will hold the actual values from the array during the iterations. You may also have noticed the use of the keyword **var**. Strictly speaking, you should also specify the type of the placeholder variable. However, the compiler can figure out what the type should be (it must be the same as the type of the elements in the array), so it “allows” you to be a bit lazy. By using **var**, you are saying “you figure it out!”. The compiler will protest if this is not possible ☺.

With arrays and loop statements (including the **foreach** loop), we can start to handle collections of values instead of just single values. However, in a modern language like C#, arrays will rarely be the best choice for such a task. Even though arrays are rather easy to work with, they have several drawbacks:

**The size of an array is fixed**: When you create an array, you must specify its size. This means that already at creation time, you must anticipate how many elements you may need to store in the array. That is often impossible, or at least very uncertain. What if the array is used in a school administration system? How many students should it be possible to handle? 100? 10000? Who knows… You could then argue that you should just create a “sufficiently large” array, maybe with one million elements. Is that a problem? It could be, since you will then use a fixed-size chunk of memory for this array, even though it will often be next-to-empty. Even though RAM is cheap these days, we should still be wary of excessive memory consumption.

**You can use array indices that are invalid:** These is nothing stopping you (even though Visual Studio will raise an eyebrow…) from writing a statement like:

myFirstArray[-2]++;

If you try to run this code, you will get an error (more specifically an **exception**, that we will learn about later).

**There is no clear definition of an “empty” element:** We mentioned above that an array of a numerical type will – unless told otherwise – be initialised with the value 0 (zero) in each element. Can we then assume that if an element contains the value 0, it is considered empty? You could, but it’s a fragile strategy. What if 0 is also a valid value? Maybe the array contains temperature measurements, and 0 may then be a perfectly valid temperature. We could then choose another value as indicating empty, but that just shifts the problem. The core of the problem is that as soon as the array is created, any content in an element is interpreted as being of the specified type.

**There is no help with common tasks:** Since the array is not really a class, there are no methods available for various common tasks, like e.g. sorting the elements. It should be mentioned that C# does contain an **Array** class, but this is more a collec­tion of assorted static methods that can be used on arrays, not part of the array as such.

In total, there are a variety of more modern classes available in C#, that are better choices for handling collections of values. Certain very specific situations – e.g. where the size of the collection will never change, and no sophisticated processing of the elements is needed – may still justify the use of arrays, but they are mostly a leftover from the earlier days of programming. Still, some syntactical elements from arrays are also found in more modern classes.

### Lists

The .NET Framework Class Library contains several classes for handling collections of values. The **List** class is probably the class that has most in common with the classic array structure.

The **List** class does not suffer from the drawbacks described above. The pur­pose is similar to arrays: insert and retrieve values (or object references) of a specific type. The syntax for creating a **List** object is:

List<int> myFirstList = new List<int>();

There are a couple of things to notice here:

* The statement creates a **List** object, that can hold a collection of **int** values. The type specification goes between the **pointy brackets** **<>**, that follow right after the **List** class name. The **List** class is a **generic** class; we will discuss gene­ric classes later, but for now we just note that this is the syntax we must use for specifying the type of the elements.
* If you type in the statement in Visual Studio, you will notice that an additional line is generated by Visual Studio, at the top of the file:

using System.Collections.Generic;

The **List** class is part of the .NET Framework class library, and the above line instructs Visual Studio that we would like to use the part of the class library containing this class. This part also contains other useful collection classes.

* We do not need to specify an initial size of the list! Upon creation, the list is indeed empty. Once we start adding elements to the list, the size is increased accordingly, and also decreased if elements are deleted.

Having created the (initially empty) list, we can add elements to it:

myFirstList.Add(982);

This is also quite different from the array style; here we use a method call in order to insert a value. Note that there is no specification of the index. The **Add** method will simply add the new element to the end of the list. If you at some point wish to insert an element at a specific position, you can use the **Insert** method:

myFirstList.Insert(2,980);

This will insert the value 980 into the element at index 2. A very important point here is that this may cause other elements to be moved! Suppose the list looks like this before the insertion:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **index** | 0 | 1 | 2 | 3 |
| **value** | 34 | -233 | 9801 | 67 |

After the insertion, the list will look like:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **index** | 0 | 1 | 2 | 3 | 4 |
| **value** | 34 | -233 | 980 | 9801 | 67 |

This is intentional, but is a feature that might surprise you if you are used to wor­king with arrays. Likewise, you can remove an element specified by index with the **RemoveAt** method:

myFirstList.RemoveAt(1);

This will shrink the list, again with the consequence that elements will be moved. The list will change from:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **index** | 0 | 1 | 2 | 3 | 4 |
| **value** | 34 | -233 | 980 | 9801 | 67 |

to

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **index** | 0 | 1 | 2 | 3 |
| **value** | 34 | 980 | 9801 | 67 |

The list is thus much more “dynamic” than an array, and you cannot expect an ele­ment to retain its original position. With regards to removing an element, you might wonder why the method isn’t just named **Remove**. The **List** class does indeed contain a **Remove** method – also taking a single argument – but this method will remove the first occurrence of the specified value. In this case, the method argu­ment is thus inter­preted as a value, not an index! This can be somewhat confusing, and you should be sure to understand the meaning of the method arguments before using the methods.

The **List** class contains a lot of additional methods, and we will only present a few of them here:

|  |  |
| --- | --- |
| **Clear** | Removes all elements from the list |
| **Contains** | Returns **true** or **false**, depending on whether the specified value was found in the list |
| **IndexOf** | Returns the index of the first occurrence of the specified value (or -1 if no occurrence is found). |
| **Sort** | Sorts the elements in the list |

Feel free to dig deeper into the C# documentation yourself, if you want to know more about the available methods. Some of them are quite sophisticated.

All these methods enter elements into the list, and process them in various ways. How do you retrieve a single element by index? This is done by using the old-school array syntax:

int aValue = myFirstList[3];

This is convenient, but the responsibility for specifying a valid index rests with the caller. Nothing prevents you from specifying an invalid index, which will provoke an error (just as for arrays). You can actually also use the array-style syntax to change the value of an element, like:

myFirstList[3] = 111;

This statement does not add a new element to the list, it only changes the value of an existing element (assuming the index is valid…).

Lists are thus used through a mix of explicit, named methods like **Add** and **Insert**, and array-style indexing. This may seem a bit confusing. You could argue that all inter­action with a list ought to be through methods, to keep things consistent. Still, the array-like indexing using [] is a very well-established standard in programming (you will find it in a lot of programming languages), and if you look a bit further under the hood, you will find that using [] is actually also a method call (called an **indexer**). Indexers are similar to properties; in code, they do not really look like method calls, but they are in fact method calls with a special syntax. We will not discuss indexers further in this note, but if you define your own class to handle collections of values, you can add indexers to your class definition, just as you can add properties.

Processing the elements in a list is just as easy as for an array, if we use the **foreach**-statement:

foreach (var value in myFirstList)

{

Console.WriteLine(value);

}

This loop is almost identical to the loop we used for printing the elements in an array, and that is in fact an important point. This loop doesn’t really care if we give it an array or a list, since these structures are “similar enough” to be processed in this way.

What do we mean by “similar enough”? If you think about it, all that seems to be necessary is:

* The structure must have a well-defined starting point (i.e. first element)
* The structure must have a well-defined ending point (i.e. last element)
* It must be well-defined how we proceed from one element to the next element

If these requirements are fulfilled, we can find the first element, proceed from the first element to the second element, to the third, and so on. We can also detect when we have reached the end. Arrays and lists both have these features, along with many other collection classes. The exact details of how to achieve this are not so interesting, and they will be hidden inside the classes themselves. Later on in these notes, we will learn the proper terminology to express such “similarities” in a more precise way.

### Dictionary

The **List** class is a significant improvement over the array structure, and make many tasks much easier. There are however a number of scenarios that lists cannot handle very well. One such scenario – that often occurs in practice – is when we need to handle data that has a **key-value relationship**.

Suppose we have a system for school management, and we need to create and mana­ge a lot of **Student** objects (we assume the system contains a **Student** class). We may thus need to use a collection class that can contain (references to) **Student** objects. A **List** could be a possible choice, like:

List<Student> allStudents = new List<Student>();

We can then go ahead and create **Student** objects, and insert them into the list. So far, no problems. A useful feature of such a system is probably the ability to look up a specific student. What information would we need in order to do that? Something that uniquely identifies a student. The name is not really enough, since that may not be unique. The Social Security Number (SSN, in Denmark known as CPR) is however unique. So, given an SSN, we must find the student with that SNN. If we were to do this with a list, it would probably look like:

bool found = false;

int index = 0;

Student theStudent = null;

while (!found && index < allStudents.Count)

{

if (allStudents[index].SSN == givenSSN)

{

found = true;

theStudent = allStudents[index];

}

}

In plain language: Examine the elements one by one, until the correct student is found, or all elements have been examined.

There are several problems with this code. First, it is somewhat complex, so there is definitely a risk of not getting it right the first time. Second, it is also very inefficient. By “inefficient” we mean that it will use much more computing power than needed. Suppose we have a school with 1,000 students. How many **Student** objects will we on average have to examine before finding the right one? Probably about 500. Consider how nice it would be if we could just use the SSN as an index! All of the code above would boil down to:

Student theStudent = allStudents[givenSSN];

If a student with the specified SSN exists, **theStudent** will then refer to that object; if not, **theStudent** will be **null**. The collection class called **Dictionary** enables you to do just that!

When declaring a **Dictionary** object, you must specify a type for the key, and a type for the value. In our example, the SSN is the key, and a **string** could be a proper type for this. The value is then (a reference to) a **Student** object:

Dictionary<string, Student> allStudents = new Dictionary<string, Student>();

The syntax is a bit intimidating, but it is just the same style as for the **List**, with an extra type parameter. We can now start adding elements to the dictionary:

allStudents.Add(aStudent.SSN, aStudent);

For the **Dictionary**, the **Add** method takes two parameters: the key (here the SNN), and the value (the **Student** object). You can also use the array-like syntax:

allStudents[aStudent.SSN] = aStudent;

The two statements do the same thing, but with a subtle difference. If you try to add an element with a key that doesn’t already exist, the element is just inserted, no matter which style you use. However, if the key already exist, then:

* The **Add** method will provoke an error (more specifically an exception)
* The index method will just overwrite the existing value with the new value.

You should thus choose the alternative that matches the behavior you want.

In order to remove an element, you only need to specify the key:

allStudents.Remove(givenSSN);

The method will return a **bool** value: **true** if an element was found and removed, **false** otherwise.

To retrieve an element, you use the array-style index syntax:

aStudent = allStudents[givenSSN];

If an element with the specified key exist, the method will return that element. If no such element exist, an error will occur! If this is not the desired behavior, you can use the method **ContainsKey** to check if a given key exist in the dictionary, like

if (allStudents.ContainsKey(givenSSN))

{

aStudent = allStudents[givenSSN];

}

This would in fact be a perfect opportunity to use the ternary operator:

aStudent = allStudents.ContainsKey(givenSSN) ? allStudents[givenSSN] : null;

The code needed for retrieving values associated with keys is thus much simpler for a dictionary, compared to a list. As an extra bonus, the dictionary can retrieve such a value much more efficiently (i.e. faster) than a list. We saw in the above, that using a list would require sequential search through the entire list. This implies that if the size of the list doubles, the average time needed to find a specific element also doubles. For a dictionary, the time needed to find a value is (almost) constant, no matter how many values we store in the dictionary! This makes the dictionary a very attractive choice, if you have an obvious key-value relationship in your data. Just as for the **List**, there are a lot of additional methods available for the **Dictionary**, and you should take some time to get an overview of the methods.

There are several other collection classes available in the .NET Framework class library (we will discuss some of these a bit later in the notes), but knowing about the **List** and **Dictionary** will take you a long way. When should you then choose one class over the other? Some general guidelines follow below:

Choose a **List** if:

* There is no obvious key-value relationship in your data
* You don’t need to retrieve data very often
* The typical operations will involve doing the same operation to all elements

Choose a **Dictionary** if:

* There is a clear key-value relationship in your data
* You often need to retrieve a specific value, given a key.

As always, using your common sense is also a good starting point ☺.

### Enumerations

A final, somewhat small, topic in relation to data structures are **enumerations**. Enumerations are not related to collection classes; they solve a different problem.

The primitive types available in C# are often not exactly what we want. Suppose we wish to make an application that deals with fruit, say these five kinds of fruit: *Apple*, *Pear*, *Cherry*, *Banana* and *Kiwi*. How should we then represent a fruit? Let’s examine some possible strategies:

* As a **bool**: That is obviously not a good strategy, since we only have two pos­sible values for a **bool**
* As an **int**: Possible (say, use the values 1 to 5, where 1 = *Apple*, and so on), but not very convenient.
  + Since an **int** can obviously have a value smaller than 1 or larger than 5, we may assign a value that does not correspond to a fruit
  + Code can be hard to understand
* As a **string**: Possible, but with problems similar to **int**

What we would really like is a custom-made type that can only hold exactly the five values given above, making it impossible to specify a wrong value. This can be done by defining an **enumerated type**. In code, it could look like:

public enum FruitType { Apple, Pear, Cherry, Banana, Kiwi }

An enumerated type will typically be part of a class definition, so we can imagine a **Fruit** class in which **FruitType** is declared.

If we now want to declare a variable elsewhere in the program, and assign a value to it, it will look like:

Fruit.FruitType aFruit = Fruit.FruitType.Apple;

The point is: We can never assign a value to the variable that does not represent a valid fruit, since the type itself specifies the legal values. Errors are caught at compile-time, not at run-time. Also, the intention of the code is probably easier to understand, even though the syntax for using the enumerated type is a bit verbose.

A typical rule-of-thumb is to use enumerations for types where there are from 3 to around 10 legal values. This doesn’t mean that it is forbidden to have an enumeration with 12 valid values, but your code will inevitably become more complex when the number of valid values increases.

## Data structures – beyond List and Dictionary

We claimed above that knowledge about the **List** and **Dictionary** class will go a long way with regards to managing sets of identical elements, be they of a simple type or of a class type. That is indeed true, but there are additional collection classes in the .NET class library worth knowing about. Also, we should be a bit more precise about the advantages and drawbacks of the various collection classes. These advantages and drawbacks are often closely related to the usage pattern of the collection, so we need to know more details about this.

### Run-time complexity

When we need to execute a certain piece of code, we may often be interested in the time it takes for the code to be executed. The specific time may depend on a lot of factors, including

* The computer hardware
* Other programs runnings simultaneously
* The code itself, i.e. has the code been written in an effective way

Run-time complexity deals with the last factor. We will never be able to determine exactly how long execution will take in “absolute time” (measured in e.g. micro­seconds), but we can analyse how long execution will take relatively to the size of the data the code needs to process (we will call such a piece of code for an algorithm).

Let’s see an example. Suppose we have a **List** of **int** values, and we want to determine if a specific value is in the list (the **List** class actually has such a method, but we will analyse our own implementation here). This is a classic case of the so-called **Linear Search** algorithm:

List<int> numbers = new List<int>();

// Insert some values into the list…

int valueToLookFor = 37;

bool valueWasFound = false;

for (int index = 0; index < numbers.Count; index++)

{

if (numbers[index] == valueToLookFor)

{

valueWasFound = true;

}

}

The highlighted part is particularly interesting, since this part decides how many loop itera­tions we will perform. Not surprisingly, the number of iterations will increase, if we put more elements into the list. We can probably expect that if we double the number of elements, the time it takes to complete the loop will also roughly double. If we increase the number ten times, the time it takes to complete the loop will also increase roughly ten times, and so on.

We can express this relation between list size and running time a bit more formally. If there are **n** elements in the list, the time **T** it takes to complete the loop is then:

**T** = c \* **n**

The letter **c** denotes a constant; if the left- and right-hand side should truly be equal to each other, then c must be equal to **T**/**n** (**T** divided by **n**). We stated above that the absolute running is not so interesting, since it may depend on e.g. specific computer hard­ware. Therefore, we usually throw away the constant, and express the relation between running time and data size in this form:

**T** = **O**(**n**)

The “O”-notation (often called the “Big-Oh” notation) is a standard notation in Com­puter Science, and should here be read as “the running time **T** is proportional to **n**”. If **n** doubles, we expect **T** to double, and so on.

This is a fairly simple example. Suppose we want to calculate the product of all combi­nations of value pairs in the list:

for (int i = 0; i < numbers.Count; i++)

{

for (int j = 0; j < numbers.Count; j++)

{

Console.WriteLine(numbers[i] \* numbers[j]);

}

}

This is a bit harder to analyse, but you can probably figure out that if we double the number of values in the list (i.e the value of **n**), the number of products printed will now quadruple, i.e. **T** will quadruple. An increase of **n** by a factor of 10 will increase the value of **T** by a factor of 100. In general, the relationship is now:

**T** = **O**(**n2**)

This should be read as “the running time **T** is proportional to the square of **n**”. Since this loop does something different than the first loop, we cannot really use this information to say whether or not one of these two algorithms is “better” than the other. However, if we did have two algorithms solving the same problem, we would deem the algorithm having time complexity O(**n**) as better than the algorithm having time complexity O(**n2**). Even though the O(**n**)-algorithm might actually be slower than the O(**n2**)-algorithm for small values of **n**, the O(**n**)-algorithm will at some point beat the O(**n2**)-algorithm.

In this way, we consider an O(**n2**)-algorithm to be “slower” than an O(**n**)-algorithm. Can an algorithm be faster than O(**n**)? Indeed it can! Consider this problem: Retrieve the first element in a list. This sounds trivial, and we can indeed solve it with a single line of code:

Console.WriteLine(numbers[0]);

We should however be a bit cautious here; the statement **numbers[0]** will cause some code inside the **List** class to execute, so we don’t know exactly what happens, or how long it takes. It does however turn out that for a **List** object, this operation is a so-called **constant-time operation**, i.e. the time needed to complete the operation does not depend on the size of data. This also seems intuitively correct; the time need­ed to find the first element in a list should not depend on the number of ele­ments in the list.

The run-time complexity of a constant-time operation is expressed as:

**T** = **O**(**1**)

This may look a bit weird the first time you see´it, but it makes perfect sense. This is indeed a run-time that does not depend on **n**. The **List** class actually has the property that any element can be retrieved in constant time.

What about the range from O(1) to O(**n**)? Are there algorithms that are not constant-time, but have a run-time complexity “lower” that O(**n**)? Indeed there are! Quite a lot, actually. Consider for instance the problem of checking if a given number is present in a list of sorted numbers. We will not write up code for this problem, but just try to think it through.

If the list is sorted, we could maybe look at the element in the middle of the list. We know we can look up this element in constant time. This can give us three outcomes:

1. The element is equal to the value we are looking for
2. The element is smaller than the value we are looking for
3. The element is greater than the value we are looking for

What is our next action in each case? Case 1 is easy: we are done. Case 2 is slightly more tricky, but if the middle element is smaller than the value we are looking for, we can draw the following conclusion: Either the element is in the “higher” half of the list (since the list is sorted), or it is not there at all. We can do a similar analysis for Case 3. We can thus repeat the logic above once more, but – and this is the crucial point – only for the “higher” half of the list! In this single step, which we know only takes constant time, we have effectively reduced the size of the problem to half the original size! We can keep doing this over and over, until we have a list of length 1, i.e. a single element. If this element is indeed the value we are looking for, we can answer the ori­ginal question with **true**, otherwise answer it with **false**.

It is hopefully obvious that it is much faster to find a given value in a sorted list, as compared to an unsorted list. But how much faster? Suppose we started with a list of 64 elements. The first step will reduce the list size to 32, then 16, then 8, 4, 2 and finally 1. A total of 6 steps. How many more steps are needed, if the list contained 128 elements? Just one, i.e. a total of 7 steps. In general, we need **p** constant-time steps in order to find an element in a sorted list with 2**p** elements. This gives us a relation between the problem size and the running time:

**n** = 2**T**

This is somewhat backwards, since we want the running time **T** as a function of **n**. If you remember your high-school mathematics, you can solve this by using **logarithms**, and end up with this run-time complexity:

**T** = **O**(log(**n**))

Where “log” means “logarithm”. If you don’t remember (or know) what logarithms are, the important property to know here is that the logarithm function grows very slowly. A couple of examples of logarithm values are given below:

|  |  |
| --- | --- |
| **n** | **log(n)** |
| 1.000 ≈ 210 | ≈ 10 |
| 1.000.000 ≈ 220 | ≈ 20 |
| 1.000.000.000 ≈ 230 | ≈ 30 |

An O(log(**n**)) algorithm is thus much, much faster than an O(**n**) algorithm. If you are familiar with an old-fashioned paper phonebook – where the person-number entries are ordered alphabetically by person name – the (simple) task of looking up a number for a given person is an O(log(**n**)) algorithm, while the much harder task of looking up a person for a given number is an O(**n**) algorithm.

Armed with this knowledge about run-time complexities, we can perform a more thorough analysis about when to choose one collection class over another. Before that, we will have a look at some additional collection classes.

### Additional collection classes

Choosing either the **List** or **Dictionary** class will often be quite adequate for managing a collection of elements. Still, there are some additional collection classes available, that may be relevant to consider if certain circumstances apply. We will give an over­view of a few of these; if you need more detailed information, there are plenty of courses to be found online.

The **LinkedList** class

We have earlier mentioned that the **List** is very efficient with regards to retrieving an element. Retrieving an element by index takes constant time, and can obviously not be done faster. Insertion and deletion are a bit different, however.

The **Add** method inserts a given element at the end of the list. This can also be done in constant time, since we do not need to move any other elements in the list. Inser­tion at the front of the list – by using the **InsertAt** method – will however cause all of the existing elements in the list to be moved up one position, to make room for the new element. Insertion at a “random” position in a **List** containing **n** elements is there­fore considered an O(**n**) operation.

The **LinkedList** offers better performance for certain types of insertion. Where a **List** can be thought of as one chunck of memory allocated to contain the elements in the list, the **LinkedList** can be thought of as several small chunks of memory, where each chunk contain one element, plus a reference to the previous and next element in the data structure. The **LinkedList** class provides properties **First** and **Last**, which returns references to the first and last element in the linked list, respectively.

Insertion into a linked list can be done in constant time at any position in the list, once you have a reference to the position where the element should be inserted. Suppose the new element is called E, and you wish to insert it just after an element called X. Before insertion, element Y follows after element X:

Y

E

X

Insertion of E requires just two steps:

1. Set X to refer to E.
2. Set E to refer to Y.

E

Y

X

These properties have several consequences with regards to peformance:

* Looking up an element by index is **inefficient** (takes O(**n**)), since you will have to start at the first element of the linked list, and step through the chain of elements one by one.
* Inserting an element at the end of the linked list is **efficient** (takes O(1)), since the property **Last** returns the last element, and insertion as such only takes constant time.
* Inserting an element at the front of the linked list is **efficient** (takes O(1)), since the property **First** returns the first element, and insertion as such only takes constant time.
* Inserting an element in a random position is **inefficient** (takes O(n)), since you will have to look up the position first before inserting.

It is fairly easy to make a similar analysis for deletion, which is efficient when done at both ends of the list, otherwise inefficient.

All this leads us to the big question: When should you use a **LinkedList** instead of a **List**? This will require knowledge about the typical usage pattern for the collection. If you e.g. often need to look up an element by index, it will be a poor choice to use a **LinkedList**, since a **List** does this much more efficiently. However, if you:

* Often need to do insertions (or deletions) at the front of the collection.
* Often need to apply some operation to all elements, using e.g. a foreach loop.
* Rarely need to look up specific elements by index

it could be worth using a **LinkedList** for your collection. A problem related hereto is the fact that the **List** class and the **LinkedList** class do not offer the same set of properties and methods. If you have written code using a **List** for your collection, you will need to change that code if you decide to switch to a **LinkedList**. A way to encap­sulate this problem could be to define an application-specific collection interface, containing exactly those collection-oriented properties and methods you need for your application. You can then create two implementations of this interface; oine using a **List** class, and one using a **LinkedList** class. It will then be much simpler to conduct experiments with both classes.

**The Queue and Stack class**

The collection classes **Queue** and **Stack** are examples of collection classes that are not as such tuned for efficiency, but rather provide an easy-to-use interface for collections with some special properties. The terms “queue” and “stack” here denotes some pro­per­ties about the order in which elements enter and leave the collection.

A “queue” denotes the situation where elements leave the collection in the same order as they were entered. This resembles the real-life concept of a queue in e.g. a supermarket: If customer A enters a queue at a cash register before customer B, we also expect that customer A will be served – and thus leave the queue – before customer B. This ordering is usually denoted FIFO (First-In First-Out). If you need to maintain such an ordering of elements, you can use the **Queue<T>** class. The **Queue** class has three essential methods:

|  |  |
| --- | --- |
| **Enqueue(T element)** | Inserts the element at the back of the queue |
| **T Dequeue()** | Returns and removes the element at the front of the queue |
| **Peek()** | Returns the element at the front of the queue |

We leave it as a small exercise to think about whether a **List** or a **LinkedList** will pro-vide the most efficient implementation of a **Queue**…

A “stack” denotes the situation where elements leave the collection in the opposite order as they were entered. You can imagine a stack of papers on a table; the bottom paper was entered first into the stack, but will be the last paper to leave the stack, since papers can only be removed from the top of the stack (at least we assume so). This ordering is usually denoted LIFO (Last-In First-Out). If you need to maintain such an ordering of elements, you can use the **Stack<T>** class. The **Stack** class has three essential methods:

|  |  |
| --- | --- |
| **Push(T element)** | Inserts the element at the top of the stack |
| **T Pop()** | Returns and removes the element at the top of the stack |
| **Peek()** | Returns the element at the top of the stack |

The reason for the diversity of method naming between **Queue** and **Stack** is mostly historical. The **Push** and **Pop** method names for the **Stack** class are common for all Object-Oriented languages, while e.g. Java uses the names **Add** and **Remove** for **Queue** methods.

The **HashSet** class

Whenever we have data with an obvious key/value relation, we prefer to use the Dictionary class, since it provies very efficient operations for insertion, deletion and lookup by key. We may however encounter situations where data has key-like proper­ties, but do not point to any data. With “key-like” properties, we mean:

* Each element must unique
* It must be efficient to check if an element is already present in the collection
* It must be efficient to insert an element into the collection

Furthermore, it may also be required that more advanced so-called **set-oriented operations** can be performed. A **set** is the mathematical term for a collection of elements, and certain operations are well-defined for sets, like:

|  |  |
| --- | --- |
| **Union** | Given two sets A and B, the union of A and B is the set containing all elements that are a member of A or B (or both) |
| **Intersection** | Given two sets A and B, the intersection of A and B is the set containing all elements that are a member of A and B |
| **Complement** | Given two sets A and B, the complement of B is the set containing all elements that are a member of B and not a member of A |
| **Subset** | Given two sets A and B, A is a subset of B if all elements that are a member of A are also a member of B |
| **Superset** | Given two sets A and B, A is a superset of B if all elements that are a member of B are also a member of A |

If you need to store data which has such key-like properties, and/or need to perform set-oriented operations on the data, the **HashSet** class offers support for this. In addi­tion to **Add** and **Remove** methods, the class contains methods corresponding to the operations described above. See the class documentation for further details.

The “hash” part of the **HashSet** class name relates to the internal representation of the data. A so-called **hash table** is used to store data. This representation makes it possible to lookup data in “almost” constant time. By “almost” is meant that even though it is theoretically possible for the lookup to take O(**n**), it turns out that we in practice can look up elements in constant time, at the expense of using a bit more memory than by using e.g. a **List**. A **Dictionary** also uses a hash table for internal storage. If you are interested in more knowledge about hash tables, there are several sources online.

## Recursion – iteration without loops

We have previously discussed loop statements (**while**- and **for**-loops) as a construc­tion for executing the same block of code multiple times. Another way of achieving this is to use **recursion**. Recursion is not a specific type of statement; it denotes the technique of letting a method call itself repeatedly, until some condition is no longer true. In that sense, recursion is definitely strongly related to tradition iteration, but certain problems can be solved very elegantly using recursion.

A first example of recursion is the method below:

public void PrintHello()

{

Console.WriteLine("Hello");

PrintHello();

}

This is not a particularly useful method, since it will keep calling itself over and over, and it is therefore effectively an infinite loop. We can improve the method by adding a way of breaking out of the infinite loop:

public void PrintHello(int numberOfCallsLeft)

{

if (numberOfCallsLeft > 0)

{

Console.WriteLine("Hello");

PrintHello(numberOfCallsLeft - 1);

}

}

This will cause the the method to terminate at some point. However, this somewhat pointless functionality could have been written as e.g. a **for**-loop instead, so we have not really gained anything. A more interesting example is the so-called **Factorial** func­tion. The Factorial function takes an integer **n** as input (**n** must be a positive integer), and returns the product **n** x (**n** – 1) x (**n** – 2) x … x 2 x 1. If e.g. **n** = 5, the **Factorial** will be 5 x 4 x 3 x 2 x 1 = 120. The Factorial of **n** is usually written as **n!**

This definition of **n!** makes it fairly to calculate **n!** using a traditional loop statement. You can however also think of the definition of **n!** as:

* **Factorial**(1) = 1, and
* **Factorial**(**n**) = **n** x **Factorial**(**n** – 1)

This is a **recursive** definition of **n**!, which we can dissect into several useful parts:

* **A trivial case**: a case for which we have a simple solution, that does not require any calculation.
* **A division strategy**: a way of splitting the problem into smaller parts, which can themselves be solved trivially or by recursion
* **A combination strategy**: a way of combining the solutions for the simpler problems into a solution for the original problem

For the **Factorial** function, these parts become:

* **A trivial case**: **Factorial**(1) = 1.
* **A division strategy**: Split **Factorial**(**n**) into **n** (trivial) and **Factorial**(**n** - 1), which can be solved by recursion
* **A combination strategy**: Multiply **n** and **Factorial**(**n** - 1).

With these definitions, we can write actual code for a **Factorial** method:

public int Factorial(int n)

{

return (n <= 1) ? 1 : (n \* Factorial(n - 1));

}

We have extended the “trivial case” a bit, to avoid an infinite loop if the method is called with a number smaller than 1. An alternative could be to throw an exception.

Even though the above looks fairly elegant, it is still not a significant improvement over the iterative version of **Factorial**. It serves more as an illustration of how to apply the steps defined above to a specific problem. A problem of a quite different nature – which turns out to be very elegantly solved by recursion – is the **Towers of Hanoi** game. The problem to be solved in this game is a follows:



1. Given three pegs A, B and C and a set of disks 1, 2, 3, …, n. The disks have increasing diameter, such that disk 1 is smallest, then disk 2, etc.
2. The starting point is as illustrated above, i.e. all disks are on peg A, with the largest disk at the bottom.
3. The goal is to end up with all disks on peg C, in the same order as they were initially on peg A. You can move disks by obeying these rules:
   1. Only one disk can be moved at a time
   2. A disk can only be placed on a larger disk
   3. All disks (except the disc being moved) must be on a peg

One way to solve this puzzle is simply to apply the breakdown rules from above:

* **A trivial case**: **n** = 0, no disks to move.
* **A division strategy**: This will consist of three steps:
  1. Move (n – 1) disks from A to B
  2. Move disk n from A to C
  3. Move (n – 1) disks from B from C
* **A combination strategy**: The three steps in combination solves the original problem for **n** disks.

It might be a bit hard to spot the recursion here, but it actually occurs in steps 1 and 3. What we do in these steps is to solve a smaller Towers of Hanoi problem. In step 1, a problem with (**n** – 1) disks is solved, where A is the “source” peg and B is the “target” peg. Step 3 is similar, except that peg B is now “source” and peg C is “target”. If we denote the last peg as “extra”, we can see that the only difference between the origi­nal problem and the smaller problem is that the pegs A, B and C play different roles. In the original problem, peg A is “source”, B is “extra” and C is “target”, while the problem in step 1 has peg A as “source”, C is “extra” and B is “target”, and so forth. We can then write up this quite compact code for solving the puzzle:

public void TowersOfHanoi(string pegA, string pegB, string pegC, int n)

{

if (n > 0)

{

TowersOfHanoi(pegA, pegC, pegB, n - 1);

Console.WriteLine("Move disk " + n + ": " + pegA + "->" + pegC);

TowersOfHanoi(pegB, pegA, pegC, n - 1);

}

}

It is possible to write a non-recursive version of Towers of Hanoi, but it is considerably harder and not as intuitive as the above solution.

Even though several problems can be elegantly solved by recursion , it is by no means guaranteed that recursion **efficiently** solves the problem! A famous number sequence known as the **Fibonacci** sequence, is defined as:

* **Fibonacci**(**n**) = 1 for n = 1, 2 else
* **Fibonacci**(**n**) = **Fibonacci**(**n** – 1) + **Fibonacci**(**n** – 2)

This translates very easily to a recursive method:

public int Fibonacci(int n)

{

return (n < 3) ? 1 : (Fibonacci(n - 1) + Fibonacci(n - 2));

}

This looks very similar to the **Factorial** method, but with one extremely important difference: Each call of **Factorial** generates a single recursive call, while each call of **Fibonacci** generates two recursive calls! That may seem insignifcant, but each of those calls will in turn generate two recursive calls, an so on. In terms of run-time complexity, this has dramatic consequences. While the run-time complexity of **Factorial** is O(**n**), the run-time complexity of **Fibonacci** is O(2**n**)!

You should thus see recursion as an additional tool-in-the-toolbox, that certainly offers very elegant and compact solutions to certain problems, but can also be somewhat deceptive with regards to efficiency. Consider using recursion if

* The non-recursive solution to the problem is substantially more complex.
* The recursive solution has an efficiency comparable to the non-recursive solution.

## Keeping your code DRY

A long way back in these notes, we discussed a situation where a **Human** class con­tained three properties: **Weight**, **Height** and **BMI**. It turned out that we could imple­ment these three properties using only two instance fields: one for weight, and one for height. The BMI property could then be calculated from the weight and the height. Compared to having an extra instance field for BMI, this gave us two advantages:

* We use less memory, if **Human** only has two instance fields instead of three
* We eliminate the risk of data inconsistency, since the value of the BMI is not stored explicitly

This is a simple example of a very important principle in programming (and computer science in general): the **DRY** principle:

**D**on’t

**R**epeat

**Y**ourself

There are other ways to phrase this principle (for instance **SSoT**: Single Source of Truth), but the essence is the same. Applied to programming, the principle dictates that we should avoid duplicating any kind of information in our code. This goes both for data and algorithms! More specifically, we will try to avoid duplication at four levels in our code:

* Values
* Instance fields
* Methods
* Classes

### DRY and values

Imagine that you create an application for some sort of “world simulation”. To keep things simple, you define your world to be a 2-dimensional grid, where something can happen in each cell of the grid. To see if your model works properly, you start out with a small world, say 10x10 cells. You will probably write several loops like:

for (int x = 0; x < 10; x++)

{

for (int y = 0; y < 10; y++)

{

// Do something for each cell

}

}

Once everything seems to work, you might want to crank up the world size to a larger grid, say 100x100. In order to do this, you must find all occurrences of such loops, and replace 10 with 100. Visual Studio can in fact do this for you, by doing a project-wide search-and-replace operation. That is, however, a risky operation. What if the value 10 is used for other purposes in the code? Replacing those values with 100 is probably asking for trouble…

The way forward here is to remove these explicitly stated values from the code (such values are often called **magic numbers**), and replace them with something else. An obvious suggestion is **constants**. In the example, we could simply introduce two con­stants **DimensionX** and **DimensionY**, and use them in the relevant loops instead. If a value is only used within a single class, you can just declare the constant as part of that class definition. If the value is used in multiple classes, things get a bit more com­pli­cated. One strategy could be to define a public constant, as part of a “setup” class. In the world simulation example, we could imagine a **WorldSettings** class, from which the dimension constants can then be retrieved, like:

public static class WorldSettings

{

public const int DimensionX = 100;

public const int DimensionY = 100;

}

Using the constants will then look like:

for (int x = 0; x < WorldSettings.DimensionX; x++)

{

for (int y = 0; y < WorldSettings.DimensionY; y++)

{

// Do something for each cell

}

}

This has two advantages: First, the specific values of the constants are now defined once, and only need to be updated there. Second, it improves the readability of your code. If you need to understand what the code is actually doing, it is probably easier to understand **WorldSettings.DimensionX** than just the value 10.

### DRY and instance fields

We have already seen the DRY principle in action in the BMI example, so there is not that much to add here. The general rule-of-thumb is to look out for data that:

* An external user is interested in (and we therefore wish to expose as a proper­ty of a class)
* Can be calculated from other data

A valid concern here is the effort (i.e. computing power) needed to do the calculation. Returning a value directly from an instance field is of course a very fast operation, while a calculation will require more effort. Still, this should be seen in the correct perspective. Suppose the primary purpose of the BMI in the example is to show the value in a user dialog. In that case, it doesn’t really matter if it takes one micro-second or 50 micro-seconds to retrieve the value, since it is negligible compared to the time it takes to open the dialog, and for the user to read the content. If there is a vast diffe­rence (say, the calculation takes several seconds), you may need to recon­­sider. Still, you should not break the principle just for efficiency reasons, if efficiency is not the primary concern.

### DRY and methods

Once we apply the DRY principle to source code, things can get a bit more complex. Consider the below addition to the **Human** class (don’t worry if you don’t understand exactly what happens in the code – it’s just about printing information on the screen with a bit of visual decoration):

public void PrintNameNicely()

{

string nameLine = "Name is : " +\_name;

nameLine = nameLine.PadLeft(20);

nameLine = nameLine.PadRight(40);

nameLine = "|" + nameLine + "|";

Console.WriteLine("------------------------------------------");

Console.WriteLine(nameLine);

Console.WriteLine("------------------------------------------");

}

So, our customer sees this, and says *“Hey, that looks nice! Can’t you also make it possible to print the height like that?”*. Well, of course we can! Here we go:

public void PrintHeightNicely()

{

string nameLine = "Height is : " + \_height;

nameLine = nameLine.PadLeft(20);

nameLine = nameLine.PadRight(40);

nameLine = "|" + nameLine + "|";

Console.WriteLine("------------------------------------------");

Console.WriteLine(nameLine);

Console.WriteLine("------------------------------------------");

}

That didn’t take long – we just copy-pasted the code from **PrintNameNicely**, and made a few changes. Maybe we would even make a **PrintWeightNicely** method in the same manner. Later on, the customer comes back and says *“Well... I would rather have stars (\*) printed instead of hyphens (-)…can you do that?”*. Yes you can…but how many places will you have to make that change? Three places, since you just copy-pasted code from the original method. More work, and also the risk of missing an update! The “fixed” code might now print name and height with stars, but weight with hyphens… Now the customer is not happy!

How should we then approach this problem? If you apply the DRY principle here, you should look at the code above and think *“what do these two methods have in com­mon?”*. Quite a lot, since the only difference is the specific data that is printed in the middle line, plus the leading text (***Height is :*** vs ***Name is :***). Everything else is the same. So, if we create a new method, where these two pieces of information are converted into parameters, we have a single method that can handle both situations! Let’s create such a method:

private void PrintNicely(string data, string leadText)

{

string dataLine = leadText + data;

dataLine = dataLine.PadLeft(20);

dataLine = dataLine.PadRight(40);

dataLine = "|" + dataLine + "|";

Console.WriteLine("------------------------------------------");

Console.WriteLine(dataLine);

Console.WriteLine("------------------------------------------");

}

This method does not contain references to specific fields, but relies on two para­meters instead. This method can now be called from the original methods:

public void PrintNameNicely()

{

PrintNicely(\_name, "Name is :");

}

public void PrintHeightNicely()

{

PrintNicely(\_height.ToString(), "Height is :");

}

We do introduce a very small complication, since the caller must now convert the data to a string. Converting an **int** to a **string** is however a very simple operation. The gain is absolutely worth the price, since we now only have to do the hyphen-to-star update in one single place! Our code is now much DRYer ☺.

The attentive reader will perhaps wonder why the new method is private – why not make it public, and simply discard the old methods? Remember that public methods are used by the outside world, so deleting them may cause problems for others. Even changing them by e.g. changing the name and/or the required parameters can cause problems. However, we are free to change the implementation, which we have done.

Are we done? Well, take a closer look at the **PrintNicely** method. Does it contain any repetition? The statements that print the top and bottom line are indeed identical. We could create yet another method:

private void PrintSeparator()

{

Console.WriteLine("------------------------------------------");

}

and update the **PrintNicely** method accordingly:

private void PrintNicely(string data, string leadText)

{

string dataLine = leadText + data;

dataLine = dataLine.PadLeft(20);

dataLine = dataLine.PadRight(40);

dataLine = "|" + dataLine + "|";

PrintSeparator();

Console.WriteLine(dataLine);

PrintSeparator();

}

Here the gain seems a bit more marginal, but we have in fact isolated the use of hyphens even further.

Could we go even further? There are still some spots of repetition left, for instance the use of the vertical separator line (**|**). Instead of writing it explicitly twice in the code, we could perhaps add a constant **verticalSeparator** to the class:

private const string verticalSeparator = "|";

So once again, we have isolated the specification of the vertical separator to one single place in the code. If you feel up to the challenge, you can probably find even more ways to improve the method.

A natural question here is of course: Is it worth the effort? Giving a definitive answer to that question is impossible. It will always be a matter of risk versus reward, and depend on specific circumstances. If you don’t care about (or don’t want to pay for) such improvements to your code, you can in principle use that saved effort for some­thing else. However, if you anticipate that the code will indeed need to be modified later on, that modification may require more effort, due to the less-than-optimal state of the code.

Code modifications of this kind are another example of code **refactoring**. We saw a simpler refactoring much earlier in these notes (simple renaming of variables), and we are now stepping up to some not-so-trivial refactorings here. Remember that any refactoring is supposed to:

* Improve the structure of the code, making it easier to maintain and extend
* Keep the functionality of the code unchanged.

With regards to when it is “worth it” to do such a refactoring, Martin Fowler suggests a “rule of three” in his **Refactoring** book: When you write the same code the third time, you should definitely refactor. This could be a useful rule-of-thumb to start with.

### DRY and classes (and a brief introduction to Inheritance)

We have now seen examples of how to eliminate code duplication inside a class, both with respect to instance fields and methods. Duplication may however also occur across several classes. Suppose we are creating a banking system, where we need to model different types of bank accounts. We might have a standard bank account, with the below features:

1. You can deposit money into the account
2. You can withdraw money freely from the account, as long as the resulting balance is positive
3. You get an interest rate assigned once a year

We might also have a savings bank account, with the below features

1. You can deposit money into the account
2. You can withdraw money from the account three times per month, as long as the resulting balance is positive
3. You get an improved interest rate assigned once a year

If we just go straight ahead and create code for these two types of account, we will probably create two classes **StandardAccount** and **SavingsAccount**. What will these classes look like, i.e. what instance fields and methods might they have? A reasonable suggestion could be that:

* Both classes have an instance field **\_balance**, and a read-only property **Balance**
* Both classes have a **Deposit** method, and the implementation will be identical
* Both classes have a **Withdraw** method, but the implementation will be different, since different business rules apply
* Both classes have an **AssignInterest** method, but the implementation will be different, since different business rules apply

So, even though the individual classes might not contain any duplicate code, we will still have code duplication, in the sense that e.g. the **Deposit** method is implemented identically in two classes. Can we eliminate such cross-class code duplication as well? Indeed we can, by using a mechanism called **inheritance**. The concept of inheritance is one of the pillars of Object-Oriented programming. The ability to “share” common features between classes is extremely useful, and is a very powerful tool for solving the cross-class code duplication problem.

Consider again the two classes **StandardAccount** and **SavingsAccount**. We saw above that they have certain features in common, and certain features that are individual. We could illustrate this relationship like so:



Each ellipsis represent the code for one class: the yellow and blue areas will thus be code specific for one particular class, while the green area will be the code common to both classes. We would like to lift the green area out of the individual classes, and make the individual classes “refer” to the green area:



The way to express this in terms of object-orientation is as follows:

* We create a **base class** called **Account**. This class is supposed to contain those elements (instance fields, properties, methods, etc.) that are common to all bank accounts. This is the green area.
* We create two **derived classes** called **StandardAccount** and **SavingsAccount**, that only contains those elements that are specific for that particular bank account type. This is the yellow and blue area, respectively.
* We let **StandardAccount** and **SavingsAccount** inherit from **Account**. The inheritance mechanism has the effect that e.g. a **SavingsAccount** object will now contain both the common elements defined in **Account** and the specific elements defined in **SavingsAccount**.

We chose the phrase “a **SavingsAccount** object will now contain…” deliberately. Inheritance does not mean that the source code from the base class is somehow sucked into the derived class before compilation. The source code for **Account** will only exist in the **Account** class definition, which was the whole point. The effect – in terms of functionality – is however just as if we had duplicated the code. So once again, we improved the structure of the code, without changing the functionality.

How do you then specify inheritance in C#? In the example above, we would not be able to see any signs of inheritance, if we just looked at the **Account** class. That class would look like any other class, with some instance fields, methods, etc.. For the derived classes, we explicitly specify inheritance like this:

public class StandardAccount : Account

{

// Rest of class definition follows here...

A colon, followed by the name of the base class. That’s it. We have now specified that **StandardAccount** inherits from **Account**.

There is quite a lot more to know about inheritance than what we described here, but we will return to this shortly. For now, we just note that inheritance enables us to eliminate cross-class code duplication, and is yet another tool to DRY up our code ☺.

### When does code become DRY?

As you are hopefully aware, these notes are about **Object-Oriented programming**. Another discipline in Object-Oriented Software Development is **Object-Oriented Design**. These two disciplines are obviously related, but exactly how they are related and what activities they involve – and how these activities may overlap – is a matter of debate. The traditional standpoint can – very simplified – be stated like:

*If we put a lot of effort into developing a sound and detailed Object-Oriented Design, the subsequent programming is almost trivial.*

In other words: If you think hard and long about the domain you want to model with your software, you can probably figure out what classes you need, how they will be related in terms of base classes and derived classes, what instance fields and methods you will need, and so on and so forth…without writing a single line of code! Once you are done with this activity – which is traditionally called Object-Oriented design – you will get the code right the first time!

Is that how it works in the real world? Rarely… In real life, you will typically create a design that looks “reasonable”, start to create some code, discover some flaws in the design, rethink the design, rework and extend the code, and so on; a much more iterative approach, where the line between design an code is blurred. This state-of-affairs has been embraced by the so-called Agile movement. Their standpoint – again very simplified – is:

*It does not make sense to separate design and programming – design happens and evolves hand-in-hand with code. We need to focus on structured ways to change the design of existing code.*

Hopefully, you can see that an activity like refactoring is exactly such a way to change the design of existing code. Refactoring is indeed considered one of the pillars of agile software development.

So, why all this high-level talk at this point? Think about the previous example, where we used inheritance to eliminate code duplication. If you subscribe to the traditional standpoint on design-vs-code, you would argue that code duplication should never have happened in the first place! You shouldn’t market inheritance as a remedy for code duplication, since it is a tool for design purposes! We don’t really want to take sides in this discussion, but either way, you may indeed find yourself in a situation one day, where you can eliminate code duplication by introducing inheritance. Also, the specific “mechanics” of inheritance are definitely in the realm of programming, and should be mastered by any object-oriented programmer, no matter if inheritance emerges as a result of up-front design or of code reorganisation. As promised, we will now investigate these mechanics in more detail.

# Object-oriented programming – Next level

Once we get to the point where our models consist of several classes, there will inevitably be some kind of relationship between the classes. We have already seen examples of classes that themselves refer to other classes: A **Car** class could have a relation to a **Wheel** class, an **Employee** class could have a relation to a **Teacher** class, and so forth. We usually make a distinction between two fundamental kinds of class relationships: the ***has-a*** relationship, and the ***is-a*** relationship.

## The has-a relationship (Composition)

The **Car**-**Wheel** example above is a classic example of a ***has-a*** relationship. A **Car** is probably a (model of a) quite complex system, so it will make perfect sense to divide the system into a number of smaller sub-systems, each represented by a **class**. The **Car** class will thus be “composed” by several sub-classes; this sort of relationship is therefore denoted **composition**. In a **Car** class definition, the relation to the other classes will be implemented by a number of instance fields, which will have the type of the classes representing the smaller systems, like

class Car

{

private Car[] wheels;

public Car()

{

wheels = new Car[4];

// (rest of Car constructor)

}

}

We have seen similar examples before, and there is as such not that much more to explain about this relationship. The higher-level classes will use the lower level-classes to implement their own functionality, by using properties, calling methods, etc.. Still, the fact that classes can make use of other classes enables us to construct very com­plex systems of collaborating classes.

## The is-a relationship (Inheritance)

A different kind of relationship can emerge as a result of discovering similarities between classes. Imagine we are working with a system for school administration. At some point, we see that we have a class **Teacher**, and another class **Secretary**. Upon further examination, we also see that the two classes have a lot in common. They probably both have properties like **Name**, **Address**, **Salary**, etc.. In accordance with the DRY principle, this is a situation we should do something about. But what? Should we try to merge the two classes into one larger class, that can accommodate both teachers and secretaries? That is not a good solution, since classes should be focused on a single responsibility. A different approach is to try to move the common parts of the two classes into a new class, and only retain the truly teacher-specific and secre­tary-specific in the original classes. The common class could be called **Employee** – it is supposed to contain only those parts that are common for all types of employees.

This “decomposition” of the original classes has at least brought us in line with the DRY principle, since no code is present twice anymore. Still, we need to have classes that can fully represent a teacher and a secretary, respectively. Could we then make these classes refer to the new **Employee** class through composition? That would indeed be a possible solution, but it would be a somewhat convoluted version of a ***has-a*** relationship: A **Teacher** has-a **Employee**… This sounds much more like an ***is-a*** relationship: A **Teacher** is-a **Employee** An ***is-a*** relationship a usually implemented by so-called **inheritance**.

## The inheritance mechanism

The two classes that are part of an **is-a** relationship each play a different role. One class is more general, while the other is more specific. In the example, the **Employee** class is general, while the **Teacher** class is more specific. By “more specific”, we mean that the **Teacher** class should be all that the **Employee** class is, plus a bit extra (the parts that are specific for a teacher). We achieve this by using **inheritance**. In C#, inheritance is syntactically quite simple to express. If we want the **Teacher** class to “inherit” from the **Employee** class, it will look like:

class Teacher : Employee

{

// Teacher-specific parts

}

This reads “class **Teacher** inherits from class **Employee**”. In this way, a **Teacher** object will expose all the public properties and methods that are defined in the **Employee** class, plus its own public properties and methods. To the outside world, a **Teacher** object thus appears exactly like before the “decomposition”, while we have achieved our goal of not repeating code in classes.

In more general terms, we refer to the class from which someone inherits as the **base class**, while the class that inherits (from the base class) is called the **derived class**. Another terminology is **superclass** (the base class) and **subclass** (the derived class).

### The protected access level

We mentioned above that the derived class will expose all public parts of the base class, which implies that the derived class can also make use of these properties and methods inside its own properties and methods. But what about elements marked **private** in the base class? A derived class can not access these elements, so private really means private! This may seem very strict, since we could imagine that a derived class would need access to certain parts in the base class, that are not available to the outside world. For this reason, there is a third access specifier called **protected**. An element in the base class marked with **protected** can indeed be accessed by a derived class, but not by an outside user.

Should we then preferably mark all elements in a class as **protected** rather than **pri­vate**, to accommodate any class that wants to inherit from the class? No. We should still be careful about what we expose, even to a derived class. If we allow a derived class direct access to all elements in the base class, it may short-circuit all sorts of validations, etc. that have been put in place in the base class. Also, the derived class might become too dependent on the internal structure in the base class, meaning that it can become difficult to change this structure if needed. We are not saying that you should never use the **protected** access level, just that you should consider the poten­tial con­sequences very carefully first.

### Constructors in derived classes

Even though inheritance in itself is very simple with regards to syntax, there are a number of non-trivial aspects you need to be able to handle. The first arises with regards to constructors. Imagine that we found that our **Employee** class will need a constructor with two parameters:

class Employee

{

public Employee(string name, string address)

{

// (rest of Employee constructor)

}

}

Also, we found that the derived class **Teacher** (derived from **Employee**) only needs one parameter in its own constructor:

class Teacher : Employee

{

public Teacher(string mainSubject)

{

// (rest of Teacher constructor)

}

}

This all looks nice here in print, but in Visual Studio, the code above will be marked in red. Visual Studio will complain that *“Base class* ***Employee*** *does not contain a para­meter­less constructor..”*. The main point is: If you derive from a class with a para­meter­ised constructor, you need to explicitly call this constructor when calling the constructor for the derived class, providing it with the parameter it needs. Syntac­ti­cally, it will look like this:

class Teacher : Employee

{

public Teacher(string mainSubject)

: base(name, address)

{

// (rest of Teacher constructor)

}

}

Note in particular the keyword **base** (preceeded by colon); this is the call to the base class constructor. In the code above, we are still not done. The parameters **name** and **address** are in red, because they are not defined anywhere… Where should they come from? The most obvious solution is that they must also be provided as part of the parameter list for the constructor of the derived class:

class Teacher : Employee

{

public string MainSubject { get; set; }

public Teacher(string name, string address, string mainSubject)

: base(name, address)

{

MainSubject = mainSubject;

}

}

We have here assumed that **Teacher** contains a property **MainSubject**. This is a very common derived class constructor: the parts belonging in the base class are used for calling the base class constructor, while the parts belonging in the derived class are saved in the properties (instance fields) of the derived class itself.

### Overriding methods

The inheritance mechanism described so far is actually sufficient in many situations. The parts defined in the base class and derived class will co-exist peacefully in objects of the derived class type, and an outside user will not really notice that inheritance is in play. It is however possible to use inheritance for more refined purposes, in parti­cular to achieve so-called **polymorphic behavior**, which we describe in some detail later. A prerequisite for polymorphic behavior is the ability to “override” methods defined in the base class.

Suppose that the **Employee** class from before has a **CalculateSalary** method, which has a generic implementation sufficient for most employees. However, it turns out that salary calculation for a teacher is more complex. We must therefore implement a different way of calculating salary in the **Teacher** class. The straightforward way to do this would be to define a new method in **Teacher** called **CalculateTeacherSalary**, since we cannot call it **CalculateSalary** (it would “collide” with the name for the method in the base class). This would be an acceptable solution in many respects, but it is in con­flict with the polymorphic behavior mentioned before.

For this reason – which will become clearer soon – we need to be able to implement a method called **CalculateSalary** in the **Teacher** class, containing the teacher-specific logic for calculating salary. This method should “override” the implementation of **CalculateSalary** in the base class, such that if you call **CalculateSalary** on an object of type **Teacher**, the teacher-specific salary calculation is executed. Is this different from before? Again, if you know you are dealing with a **Teacher** object, you might as well call a method called **CalculateTeacherSalary**? One of the main points of polymorhic behavior is however that you can call **CalculateSalary** on an object that seems like an **Employee** object, but really is a **Teacher** object, and still have the teacher-specific version of **CalculateSalary** executed! That is essentially what polymorphic behavior is.

In order to achieve this, we must however state our intention very explicitly in C#. We need to state two things:

1. In the base class, we must explicitly mark any method that may be overrided in a derived class
2. In the derived class, we must explicitly mark any method that is overriding a corre­sponding method in the base class

By “corresponding” we mean a method with exactly the same signature, i.e. same name, return type and parameter list. If just one of these don’t match, we are not overriding a base class method.

How does look this look with regards to C# syntax? In the base class, we state that a method may be overrided by adding the **virtual** keyword:

class Employee

{

// // (rest of Employee class definition)

public virtual int CalculateSalary()

{

// Generic salary calculation

}

}

Note that we only state that the method may be overrided; the derived class has no obligation to do so. If the derived class chooses to do so, it states this intention by using the **override** keyword in its own definition of the method:

class Teacher : Employee

{

// (rest of Teacher class definition)

public override int CalculateSalary()

{

// Teacher-specific salary calculation

}

}

With this setup in place, we can now achieve this enigmatic “polymorphic behavior”. So, what is it?

### Polymorphic behavior

When we are using classes related by inheritance, we can suddenly loosen up one of our most fundamental assumptions: When you create a variable and assign a value to it, the variable and the value should have the same type. We have seen this almost from the beginning:

int age = 23;

Teacher theTeacher = new Teacher("Per", "Home", "Programming");

However, if **Teacher** inherits from **Employee**, the below code is also valid

Employee theEmployee = new Teacher("Ole", "Away", "Design");

The variable has type **Employee**, but the value has type **Teacher**... but since **Teacher** is-a **Employee** (that’s what inheritance expresses), the above is also valid. But is it also useful? On its own, not so much. We have in fact restricted ourselves a bit in this way. Suppose the **Teacher** class has a property **MainSubject**. Consider then the two lines of code below:

Console.WriteLine(theTeacher.MainSubject); // OK

Console.WriteLine(theEmployee.MainSubject); // ERROR!

We can only use the **MainSubject** property on the variable of type **Teacher**, not on the variable of type **Employee**. This makes good sense, since that property is indeed teacher-specific. Now consider the below code:

Console.WriteLine(theTeacher.CalculateSalary()); // OK

Console.WriteLine(theEmployee.CalculateSalary()); // OK

This also makes sense, since both classes now have an implementation of **Calculate­Salary**. The big question is now:

What implementation of **CalculateSalary** will be called in each case?

The first case is probably most obvious; the variable has type **Teacher**, so the imple­men­­tation in **Teacher** should be called. That is indeed true. In the second case, it is still the implementation in **Teacher** that gets called! This seems surprising, since we make the call on a variable of type **Employee**, and **Employee** has its own implemen­tation of **CalculateSalary**. However, the C# compiler has noticed our intention of overriding the method in the derived class, and will therefore call the implementation in the derived class on any object of that type, even if the object is referred to by a variable of the base type! That is **polymorphic behavior**.

It is understandable if you still cannot appreciate why this is such a useful construct. Suppose we have a more complex system with many types of employees – all inherit­ing from **Employee** – where some choose to override **CalculateSalary**, while other just go with the generic implementa­tion in the base class. Suppose also that part of the system deals with processing salaries for all employees. We could then imagine func­tionality like “for all employees, calculate the salary and print a salary specifica­tion”. In other words, we need to iterate through all employee objects, and make calls to **CalculateSalary** on each object. If we had to do this without using inhe­ritance and polymorphic behavior, we would have to maintain a list for each type of employee, in order to call the correct implementation of **CalculateSalary**, like

List<Teacher> allTeachers= new List<Teacher>();

allTeachers.Add(new Teacher("Hans", "Home", "English"));

List<Secretary> allSecretaries = new List<Secretary>();

allSecretaries.Add(new Secretary("Leon", "Office", "Law"));

// ..and so on

foreach (Teacher t in allTeachers)

{

t.CalculateSalary(); // Teacher-specific salary calculation

}

foreach (Secretary s in allSecretaries)

{

s.CalculateSalary(); // Secretary-specific salary calculation

}

// ..and so on

This is definitely an implementation and maintenance nightmare. With inheritance and polymorphic behavior, we can however achieve our goal with just one list:

List<Employee> allEmployees = new List<Employee>();

allEmployees.Add(new Teacher("Per", "Home", "Programming"));

allEmployees.Add(new Secretary("James", "Office", "Marketing"));

foreach (Employee e in allEmployees)

{

e.CalculateSalary(); // Calls the correct implementation!

}

Through polymorphic behavior, we will always call the correct implementation of **CalculateSalary**, be it the generic or specific version. The above loop will not even need to be updated, if we later on add additional employee types, as long as they inherit from **Employee**. This enables much more clean and generic programming, and is definitely yet another tool for adhering to the DRY principle.

### Calling base class mehods

When we override methods, we often wish to replace the base class method imple­men­tation completely. However, there are also scenarios where we wish to “extend” the base class implementation. That is, we still want the code in the base class method to be executed, but we want to do something additional in the derived class. This could very well be the case for salary calculation: some generic parts of the cal­culation are done in the base class, while some employee-specific parts are done in the derived class. The two parts are then added up in the derived method. In the derived class, we can achieve this using the following syntax:

public override int CalculateSalary()

{

return base.CalculateSalary() + payGrade\*500;

}

Again, the keyword **base** is used to refer to a base class implementation.

### Abstract methods (and classes)

One of our assumptions in the above example was that some sort of gene­ric salary calculation logic exists, that can be used if no extra salary calculation logic applies for a specific kind of employee. What if the salary calculation logic is so diverse that no generic logic exists? What will the implementation of **CalculateSalary** in **Employee** then look like? Maybe this:

public virtual int CalculateSalary()

{

return 0;

}

This could be the case, maybe with the argument *“Well, we have to put something, right?”*. That is however not a valid argument. In general, we often face this situation:

* All classes inheriting from a base class **B** should implement a method **M**
* The is no sensible implementation of **M** in **B** itself

First of all, what is the problem with the not-so-smart implementation of **Calculate­Salary** above? We will override the method in all derived classes anyway, yes? True, if we remember to do it! There will be nothing alerting us that we have forgotten to override it for a new derived class, except that some employee might see a zero on his salary specification… It would be much better if we could make it mandatory for all classes inheriting from **Employee** to implement **CalculateSalary**, while not having a meaningless default implementation in **Employee**. We can achieve this by making the **CalculateSalary** method **abstract**.

An abstract method is a method without a body… that is, we only specifiy the method signature in the class definition:

abstract class Employee

{

public abstract int CalculateSalary();

// (rest of Employee class)

}

Notice the semi-colon at the end. We are really done with what we have to say about this method in the **Employee** class. We only specify its signature, nothing more. If a class inherits from **Employee**, it will now be required to implement **CalculateSalary**. Declaring an abstract method in a class has an additional consequence. Consider the below (invalid) code

Employee e = new Employee("Vivian","Home"); // ERROR!

e.CalculateSalary(); // What should happen here?

What should indeed happen in the second line? It’s meaningless, since there is no implementation of **CalculateSalary** to call. In general, any class that contains an abstract method will itself need to be marked as **abstract** (as **Employee** is above), meaning that you can not create an object of that type! However, you can still have a variable of that type, so the ability for polymorphic behavior is preserved.

With this in place, we can sum up the difference between a **virtual** method and an **abstract** method:

* **Virtual** method: Has an implementation in the base class, can be overriden in a derived class. Use when a meaningful implementation of the method can be done in the base class
* **Abstract** method: Does not have an implementation in the base class, must be overriden in a derived class. Use when no meaningful implementation of the method can be done in the base class

### Interfaces

The concept of defining abstract methods in a class can be taken to the extreme; a class that only contains abstract methods. This is actually a very useful idea, and has even been given its own name in Object-Oriented programming: an **interface**.

An interface can be seen as the absolutely minimal specification of a class. Imagine somebody interested in some particular functionality, for instance a class capable of drawing geometric shapes. How can he state his requirements in a very precise way, but without any assumption about specific implementation details? In terms of an interface definition! An interface definition for a (very simple) geometry drawing system could be specified in C# as:

interface IGeometryDraw

{

void DrawCircle(double x, double y, double radius);

void DrawLine(double x1, double y1, double x2, double y2);

void DrawRectangle(double x1, double y1, double x2, double y2);

}

There are several things to take note of here:

* The keyword **interface** is used instead of **class**
* The interface name starts with an I – this is a naming standard
* There is no access specifier – all methods are per definition **public**
* All methods are **abstract**

Somebody interested in obtaining this functionality could simply state it in terms of this interface definition, alongside a specification of what an implementation of the interface is supposed to do, from an external perspective. It will of course not make sense to implement the **DrawCircle** method in a way that draws a square, so some sort of requirement specification is needed. But apart from that, there is no need for additional information. A programmer could then go back and create a class that implements the interface:

class GeometryDrawV10 : IGeometryDraw

{

public void DrawCircle(double x, double y, double radius)

{

// (rest of DrawCircle)

}

// (rest of GeometryDrawV10)

}

The syntax for implementing an interface is identical to the syntax for inheritance in general. One important difference is however that one class can “inherit” from (i.e. imple­ment) multiple interface, but can only inherit from a single non-interface class. The reasons for this limitation are a bit technical, and beyond the scope of this text.

If you take a tour through parts of the .NET Framework class library, you will see that interface are used quite heavily. Use of interfaces is a very strong mechanism for making couplings between classes as weak as possible, since they are a specification of the absolute minimum you need to know about a class in order to use it.

## The Object class

We conclude this section on inheritance by a small revelation – we have been using inheritance all along. All C# classes tacitly inherit from a “universal” base class named **Object**. This base class has seven methods:

**Equals**

**Finalize**

**GetHashCode**

**GetType**

**MemberwiseClone**

**ReferenceEquals**

**ToString**

Some of these methods can be overrided in a class definition, if you want your class to have certain abilities. A particularly interesting method is the **ToString** method. As we have seen many times before, we can put anything into a **Console.WriteLine** method call; the method will then try to print out the parameter. This works quite well for simple types like e.g. **int**, where the value is printed as expected. However, if we try to do something like

Console.WriteLine(theTeacher);

we will get something like **MyNamespace.Teacher** printed on the screen. What hap­pens is that the method tries to print the string representation of the parameter, more specifically by calling the **ToString** method – which all classes implement due to the inheritance from **Object** – and print the return value. What we see is the base class implementation of **ToString**. If we want a more useful result, we can override the **ToString** method in the **Teacher** class:

public override string ToString()

{

return Name + " teaches " + MainSubject;

}

Now we will see a printout like e.g. “John teaches Design”, whenever the program tries to print a **Teacher** object.

Some of the remaining **Object** methods can also be overrided for more or less exotic purposes; seek up additional information online about this, if you find that you need to do so.

## Exceptions

So far, we have given very little consideration to how to handle error situations. Handling error situations is usually a pretty significant issue in programming, so we cannot ignore it; we need to know about tools and strategies for managing it.

So, what can go wrong? Below are just a few of the error situations we can imagine for a simple value:

* Value is correct type-wise, but is outside the range of meaningful values (example: A test score is supposed to be between 0 and 100, but an **int** can represent many other values, like e.g. -27, 22987).
* Value is used for indexing an array – only values from 0 (zero) up to (**Length** - 1) are meaningful. Other values will produce an error.
* A string does not follow a given syntax (e.g. for a license plate).
* A variable that is supposed to refer to an object has the value **null** instead.

The proper action of the application in the above cases will be situation-dependent. The application may halt, show an error message, silently handle the error, fall back to a default value, etc. In any case, we should be prepared for handling all possible error situations in a graceful manner. Simply shutting down the application is usually not an acceptable option.

Management of error situations can in general be divided into four phases:

1. **Detection** – realising that an error situation has occurred
2. **Signaling** – making the surrounding code aware that an error has been detected
3. **Capturing** – taking responsibility for handling of the error
4. **Handling** – actually performing the error handling actions

The actions corresponding to these phases can be distributed in the code. This may imply that the part of the code detecting the error does not know how to handle the error! Information about the error must then somehow be propagated to the error handling code. One way of doing this could be to use return values. A method could return some sort of error object as its return value, which the caller could then act upon. This strategy does however quickly turn out to become very complicated, so we usually resort to a different mechanism: **exceptions**.

### Throwing and catching

Exceptions are by themselves just a set of classes, that all inherit from the .NET library class **Exception**. If you need to use an exception object, you create it using **new**, just as for any other class. The distinctive feature for exceptions is that you can “throw” and “catch” excep­tion objects.

If an error situation – or “exceptional” situation, to use a broader term which includes errors – occurs, the code which **detects** the situation can “throw” an exception. In C# code, this will look like:

public void Deposit(int amount)

{

if (amount < 0) // Error detected

{

NegativeAmountException ex =

new NegativeAmountException("Deposit");

throw ex;

}

\_balance = \_balance + amount;

}

Here **NegativeAmountException** is a class we have defined ourselves. It inherits from **Exception**, which makes a **NegativeAmountException** object “throwable”. Note the keyword **throw** – this is where the object gets thrown. This is the **signaling** phase of the error handling process. When an exception object is thrown, no more code in the method is executed, just as if we had used the **return** keyword.

What does it mean more specifically to “throw” an object? Throwing an object is diffe­rent from returning an object. An object which is thrown is passed up through the method calling chain just as return values are, but – and this is a significant difference – a method that has no interest in exceptions doesn’t need to do anything at all in terms of handling it. A thrown object will just silently pass up through the method calling chain, until someone decides to “catch” the exception object.

Catching an exception object is something a caller deliberately chooses to do. If you are calling a method that might throw an exception, and you are intent on handling the exception if it occurs, you will encapsulate the call in a **try-catch** statement:

BankAccount theAccount = new BankAccount();

try

{

theAccount.Deposit(-1000);

}

catch (NegativeAmountException ex)

{

Console.WriteLine("{0}: Negative amount not allowed", ex.Message);

}

This may look somewhat tedious with regards to syntax, but remember that it is only if you have the intent of actually handling the error, that you need to use this con­struction.

How do we read the above code? The caller is aware that **Deposit** might throw an exception, so he places the call in the **try**-part of the statement. If the call goes well, nothing else happens – the **catch**-part does not come into play. However, if an excep­tion is thrown, the exception is caught by the **catch**-part (this is the **capturing** phase). Now the code in the **catch**-part is executed. In this simplified case, the code does a very simplistic error handling (this is the **handling** phase), by printing out a message. In a more realistic setting, the code might make a call to some code dedicated to error handling, error recovery and error presentation. Once this code finishes, the state­ments following the entire **try-catch** statement will be executed.

We said above that the **catch**-part was executed if the **Deposit**-statement threw an exception. That is not entirely accurate. The code will only be executed if an excep­tion object of the type **NegativeAmountException** is thrown. A **catch**-statement will only catch exceptions of that type we have specified in the parentheses following the **catch** keyword. If a different exception had been thrown, it would not have been catched here, but possibly futher up the method calling chain. If we want to catch more than one type of exceptions, we can simply write additional **catch**-blocks after the first one, like:

BankAccount theAccount = new BankAccount();

try

{

theAccount.Deposit(-1000);

}

catch (NegativeAmountException ex)

{

Console.WriteLine("{0}: Negative amount not allowed", ex.Message);

}

catch (LargeAmountException ex)

{

Console.WriteLine("{0}: Amount must not exceed …", ex.Message);

}

It can be tempting just to write a single **catch**-block, where you catch exceptions of the type **Exception**, since you will then effectively catch all exceptions. However, that also means that you assume responsibility for handling all types of exceptions, which might not be a good idea…

### The finally block

Once an exception is thrown, the flow-of-execution will not return to the code that threw the exception. This can be problematic if some sort of resource has been claimed by that code (e.g. a file connection), since it does not get a chance to release that resource again. For such purposes, you can add a **finally**-block to a **try-catch** statement:

System.IO.StreamReader file = new System.IO.StreamReader(path);

char[] buffer = new char[10];

try

{

file.ReadBlock(buffer, index, buffer.Length);

// file.Close(); Moved to finally-block

}

catch (System.IO.IOException e)

{

Console.WriteLine("Error reading from {0}. Message = {1}",

path, e.Message);

}

finally

{

file.Close();

}

It would be natural to place the **file.Close** call just after the **file.ReadBlock** call, but if that call throws an exception, the file is never closed. Therefore, it is moved to the **finally**-block. Code inside the **finally**-block is guaranteed to be executed after either the code in the **try**-block or the code in the **catch**-block. So, no matter the outcome of the **file.ReadBlock** call, the file is now always closed properly.

### Rethrowing an exception

What if you want to do something if an exception occurs, but also want others to have a chance of handling the exception? You can then “re-throw” the exception. A com­mon scenario could be that you wish to do some sort of logging of the exception, but also want to do more specific exception handling further up the call chain. You can then do a re-throw in the style illustrated below:

try

{

// Code that may throw exceptions

}

catch (Exception ex)

{

Logger.Log(ex.Message); // Log exception

throw; // Rethrow exception

}

This is a quite useful construction, where someone having a stake in error handling can perform a specific kind of handling, but also pass on the exception to others.

### Exceptions summary

We now know the basics of dealing with exceptional situations using exceptions. The essential points are:

* The .NET Framework class library contains a lot of exception classes, including the class **Exception**, which is the base class for all exception classes. If you need a specialised exception class, explore the library first. There might be a class that fits your needs.
* You can define your own exception classes, but they must inherit from **Excep­tion**, or one of the other existing exception classes (including exception classes you have defined yourself).
* You should **throw early**: as soon as you have discovered an error situation that you don’t want to handle yourself, throw an appropriate exception.
* You should **catch late**: do not catch an exception unless you are sure what to do with it. Do not catch an exception and then ignore it by doing nothing.
* Consider rethrowing the exception, when you have dealt with it. Others might want a chance to handle the exception as well.

Exceptions is a construction that is a bit contrary to the ordinary flow-of-execution, but once you get a grasp of it, it is a quite elegant and powerful way of dealing with errors in a non-intrusive way. Remember, you only need to deal explicitly with (i.e. write code for) exceptions, if you have an interest in handling them.

## Generics – types as parameters

We discussed the DRY (Don’t Repeat Yourself) principle a while ago, and considered the principle at various levels (instance field, method and class level). The goal was always the same: avoid writing the same code over and over. At the class level, we saw that **inheritance** is a useful mechanism for avoiding code ducplication, since you can place code shared by several classes in a base class, which can then be inherited from. Still, some situations are not easily solved by inheritance.

### Shortcomings of inheritance

Suppose we have defined a simple domain class **Dog**, and also wish to have a way of representing family relations between dogs. Instead of defining family relations as a part of the **Dog** class itself, we decide to create a separate class **FamilyRelation**, which will represent family relations between **Dog** objects. Such a class could look like this:

public class FamilyRelation

{

private Dog \_self;

private Dog \_father;

private Dog \_mother;

private List<Dog> \_children;

public FamilyRelation(Dog self, Dog father, Dog mother)

{

\_self = self;

\_father = father;

\_mother = mother;

\_children = new List<Dog>();

}

public Dog Self { get { return \_self; } }

public Dog Father { get { return \_father; } }

public Dog Mother { get { return \_mother; } }

public List<Dog> Children { get { return \_children; } }

public void AddChild(Dog child)

{

\_children.Add(child);

}

}

This is pretty straightforward, and we can easily start to use this class:

Dog self = new Dog("King);

Dog mom = new Dog("Spot");

Dog dad = new Dog("Rufus");

FamilyRelation relations = new FamilyRelation(self, mom, dad);

relations.AddChild(new Dog("Lajka"));

So far, so good. Now we also define a domain class **Cat**, and also wish to be able to represent family relations between **Cat** objects. We cannot use the **FamilyRelation** class as-is, since it operates on **Dog** objects. We can do a copy-paste of the class, and create a (very similar) class **FamilyRelationCats**, where we simply replace **Dog** with **Cat**. However, this is exactly the situation we wish to avoid…

**Cat** and **Dog** are probably strongly related, and it will probably make sense to define a base class **Animal**, from which both **Dog** and **Cat** can inherit. If we do that, we could also update the **FamilyRelation** class:

public class FamilyRelation

{

private Animal \_self;

private Animal \_father;

private Animal \_mother;

private List<Animal> \_children;

// ...and so on

}

Now we can use the class for both **Dog** and **Cat** objects. There are however several problems with the class:

1. Nothing will prevent us from mixing **Cat** and **Dog** objects, so a cat could e.g. be a father of a dog…
2. The return type of the properties will be **Animal**, so we will need to try to cast the returned object to a derived class, if we need to do something **Cat**- or **Dog**-specific with the object.
3. We can only use the **FamilyRelation** class for classes which inherit from **Animal**, even though the **FamilyRelation** class itself does not use any **Animal**-specific methods.

Using inheritance cannot really solve these problems for us. What we really want is to turn the **type** of the objects used in **FamilyRelation** into parameters. That is, we wish to define the **FamilyRelation** class with a general type parameter, and wish to use the the **FamilyRelation** class with a specific type. This is essentially the same strategy we use when defining a simple method; we define the method in a general way:

public int ReturnLargest(int a, int b)

{

return (a < b ? b : a);

}

and we use the method with specific values:

ReturnLargest(7, 12);

### Using types as parameters

The feature in C# called **Generics** is exactly the ability to use types as parameters to class definitions (and to methods, which we will also see examples of). We have already used this ability without calling it Generics, when we saw examples of **data structures**. If we needed a list of integers, we could declare it like this:

List<int> myNumbers = new List<int>();

This will create a **List** object, into which we can only insert **int** values. Also, any me­thod that returns an element in the list will have the return type **int**. In that sense, the **List** class – or more correctly, the **List<T>** class – is **type-safe**. We cannot accidentally insert elements of different types into the list, and the elements return from the list have the correct type, i.e. no need for casting. The **<T>** following the **List** class name indicates that the **List** class takes one type parameter – just as a method can take one parameter – which must be specified when using the class, as above. Type parameters can be called whatever you like – just as a parameter to a method – but are usually called **T** (T for Type, maybe…).

With this knowledge, we can create a new and more generally applicable version of the **FamilyRelation** class:

public class FamilyRelation<T>

{

private T \_self;

private T \_father;

private T \_mother;

private List<T> \_children;

public FamilyRelation(T self, T father, T mother)

{

\_self = self;

\_father = father;

\_mother = mother;

\_children = new List<T>();

}

public T Self { get { return \_self; } }

public T Father { get { return \_father; } }

public T Mother { get { return \_mother; } }

public List<T> Children { get { return \_children; } }

public void AddChild(T child)

{

\_children.Add(child);

}

}

We have simply substituted **Dog** with **T**, and added the **<T>** declaration after the **FamilyRelation** class name. At first sight, this may look confusing. What is **T** actually? Is it a new class? No, it is simply a (type) parameter to the class. Again, think of it as very similar to a parameter to a method. You can give it any name you like, and when you use the method, you must provide a specific value for the variable. If we now wish to use the **FamilyRelation** class, we must provide a specific type:

FamilyRelation<Dog> relations = new FamilyRelation<Dog>(self, mom, dad);

relations.AddChild(new Dog("Lajka", 45, 20));

The variable **relations** now refers to a **FamilyRelation** object, specialised to the **Dog** type. This solves the problems listed before:

1. We can only insert **Dog** objects into the object
2. All properties have the return type **Dog**
3. **Dog** does not need to inherit from a certain base class in order to use the **FamilyRelation** class.

Using the **FamilyRelation** class to define relations between **Cat** objects is now as easy as it gets:

Cat a = new Cat("Stripe");

Cat b = new Cat("Socks");

Cat c = new Cat("Abby");

FamilyRelation<Cat> catRelations = new FamilyRelation<Cat>(a, b, c);

It doesn’t take much consideration to conclude that any type – even simple types like **int** – can be used with **FamilyRelation**. Using a class called **FamilyRelation** to repre­sent relations between numbers is perhaps a bit warped, but the point is still valid. This could also be seen as an argument for choosing the non-descriptive name **T** for the type parameter. We could have chosen to call the type parameter **Animal** (not to be confused with a base class called **Animal**) to try to indicate the intended use of the class, but that would not prevent anyone from using it with a completely unrelated type like **int**. The name **T** has no implicit meaning in itself, and thus indicates that “anything goes” with regards to choice of type.

### Type constraints

The above considerations about what types to use with **FamilyRelation** leads directly to an important Generics sub-topic: **type constraints**.

The **FamilyRelation** class could be type-parameterised very easily, since the class does little more than store and return elements of type **T**. The term “elements” is chosen deliberately, since these elements may not even be objects, if a simple type like **int** is used. If we begin to add functionality involving the stored elements, we may however run into problem quite soon. Suppose we add the modest requirement that we can create a new **FamilyRelation** object, when only the “self” is known. We suppose that the mother and/or father can be added later. This could be done by adding an extra constructor, like this:

public FamilyRelation(T self)

{

\_self = self;

\_father = null;

\_mother = null;

}

Trying to compile this code will produce an error *“Cannot convert* ***null*** *to type para­meter* ***T****, because it could be a value type”*. This seems reasonable; if the chosen type is **int**, we are trying to assign **null** to an instance field of type **int**, which does not make much sense.

This could be an opportune moment to consider, if we really want to allow use of the **FamilyRelation** class with any type. If we come to the conclusion that the used types should at least be class types (i.e. not simple types like **int** or **bool**), we can specify this by adding a **type constraint** to the class definition:

public class FamilyRelation<T> where T : class

{

// Rest of class definition omitted

}

This expresses that **T** must at least be of a class type. Adding this to the class defini­tion has two consequences in relation to the previous code:

1. The extra constructor is now valid and can compile.
2. Using **FamilyRelation** with type **int** now causes a compilation error.

This is exactly as intended. We can take this principle further, and e.g. constrain **T** to be a class that inherits from a base class **Animal**:

public class FamilyRelation<T> where T : Animal

{

// Rest of class definition omitted

}

Placing this constraint on **T** also allows us to start using the instance fields, i.e. call methods on the objects referred to by the instance fields. If the **Animal** class contains a property **Name**, the below will be legal code inside the **FamilyRelation** class:

public void PrintNamesOfParents()

{

Console.WriteLine(\_father.Name + " and " + \_mother.Name);

}

Due to the constraint on **T**, we are now guaranteed that the **Name** property will always be available.

Deciding exactly how to constrain a type parameter can be a bit trickly. It will always be a balance between keeping the class as general as possible (i.e. keeping the con­straints minimal) and being able to perform type-specific operations within the class (i.e. having enough constraints). You should generally strive to have “just enough con­straints”. If a constraint can be removed without causing compilation errors, it should obviously be removed. You should also consider carefully what operations you really need to perform with type-parameterised objects. Finally, Visual Studio (along with ReSharper) is often able to provide useful suggest­ions for adding (or removing) constraints on type parameters.

### Type parameter variance

A very natural question relating to type parameters concerns how they relate to inhe­ri­tance. More specifically: If we have defined a class **A**, and also define a class **B** which inherits from **A**, we know from Polymorphism whether or not the below lines of code are valid:

A a = new B(); // OK

B b = new A(); // Error

Suppose now we have another class **C<T>**, which takes one type parameter **T**. This could be the **FamilyRelation** class defined above. What would we except about these lines of code?:

C<A> ca = new C<B>(); // ??

C<B> cb = new C<A>(); // ??

Based on intuition, many will probably guess that the same validity applies here, such that the first line is valid while the second is not. It turns out that none of these lines of code are valid… The reasons for this are a bit hairy, but let’s have a look at it.

First of all, we will use the **Animal** class as a base class, and the **Dog** class as subclass. We thus have this as our starting point:

Animal a = new Dog("Spot"); // OK

C<Animal> ca = new C<Dog>(); // Error

The class **C** is now defined as having two very simple methods:

public class C<T>

{

private T \_t;

public T Get()

{

return \_t;

}

public void Set(T t)

{

\_t = t;

}

}

Let us now assume that the below line is in fact valid:

C<Animal> ca = new C<Dog>();

This will create an object of type **C**, where **T** is set to **Dog**. So, what type of object can be returned by the call **ca.Get()**? Only an object of type **Dog**, and since **Dog** is a sub­class of **Animal**, this is a perfectly valid object to return in response to calling the **Get** method on **ca**, since this method has the return type **Animal**. Remember that **C<Dog>** does not inherit from **C<Animal>**, so this is not a matter of polymorphism! The conclu­sion is thus that with respect to the **Get** method, the above statement would be safe.

What about the **Set** method? If we call **ca.Set(…)**, what type of objects will then be valid parameters. Since **ca** has the type **C<Animal>**, the **Set** method will accept any object of type **Animal**, including – but not limited to – **Dog**. The last part is what breaks our assumption. If e.g. **Cat** inherits from **Animal**, we can call **ca.Set(…)** with a **Cat** object, which cannot be inserted into an object of type **C<Dog>**…

Let’s also consider the second line, which does look very contra-intuitive:

C<Dog> cd = new C<Animal>();

We immediately run into problems when considering the **Get** method. This method can only return objects of type **Dog**, but we may easily have an object of a different type (e.g. **Cat**) in the referred-to object, which breaks our assumption. However, the **Set** method does not break the assumption! In this case, the **Set** method only accepts objects of type **Dog**, which we can safely insert into an object of type **C<Animal>**, since **Dog** inherits from **Animal**.

The conclusion so far is thus: Given our current implementation of the class **C<T>**, it makes perfect sense that the two assigment statements from above are invalid. There does however seem to be a pattern as to how the validity is broken. Methods that return a value of type **T** induce one kind of “breakage”, while methods taking a para­meter of type T induce a different kind. It turns out that this difference can be exploi­ted further.

Let’s now add two **interfaces** to the mix. We define two interfaces **ICGet** and **ICSet**, respectively:

public interface ICGet<T>

{

T Get();

}

public interface ICSet<T>

{

void Set(T t);

}

The original class **C<T>** nows implements these two interfaces:

public class C<T> : ICGet<T>, ICSet<T>

{

private T \_t;

public T Get()

{

return \_t;

}

public void Set(T t)

{

\_t = t;

}

}

We can now try out some slightly different statements:

ICGet<Animal> icsa = new C<Dog>();

ICSet<Dog> icgd = new C<Animal>();

With the above class definitions, both lines of code are deemed invalid by the com­pi­ler. However, if you perform an analysis similar to the previous analysis, you will come to the conclusion that both lines are actually safe, and thus in principle valid. So why does the compiler not agree? It turns out that you have to specify your “intention” with a type parameter explicity in the interface definitions:

public interface ICGet<out T>

{

T Get();

}

public interface ICSet<in T>

{

void Set(T t);

}

The keywords **in** and **out** make our intention explicit: A type parameter marked with **in** will only be used as a type for “input” parameters to methods, while a type para­meter marked with **out** will only be used as a type for return values for methods and properties. The compiler should in principle be able to deduce this, but it turns out to be a quite hard problem in practice, which is why it was decided that the programmer must state this explicitly. In C#, it is also only allowed to add these keywords to type parameters for interfaces, not for classes in general.

The keywords **in** and **out** are pretty closely related to the stated intentions; to use the type parameter only for input or output, respectively. More formally, in an interface like **ICGet<out T>**, **T** is said to be declared as being **co-variant**, while in an interface like **ICSet<in T>**, **T** is said to be declared as being **contra-variant**. These terms originate from so-called “category theory” in Mathematics. If a type parameter is not marked with either **in** or **out**, it is said to be **invariant**. The keywords **in** and **out** are probably easier to remember… As with type constraints, the development environment is capa­ble of suggesting when to declare a type parameter to be co- or contra-variant.

The final – and most important – question in relation to this topic is of course: *Should I care about this?* Is it just an academic observation, or does it have any practical con­sequences? Whether or not you will ever be in a situation where type parameter vari­ance will be a crucial matter, if of course very hard to predict. Still, you need not look further than the .NET class library for some very concrete examples. We will take a closer look at two commonly used interfaces in the library, called **IComparable<T>** and **IComparer<T>**.

### The IComparable<T> and IComparer<T> interfaces

Continuing our example with the **Animal** class and the **Cat** and **Dog** subclasses, we can imagine that it at some point becomes necessary to sort a list of e.g. **Dog** objects, for instance according to weight. Since weight is a property all animals have, it makes sense to implement a **Weight** property in the **Animal** base class. Having done this, we can go ahead and try to sort a list of **Animal** objects:

List<Animal> animals = new List<Animal>();

animals.Add(new Dog("King", 70));

animals.Add(new Dog("Spot", 30));

animals.Add(new Dog("Rufus", 80));

animals.Sort();

This naïve attempt will not succeed; in fact, the code will cause an exception to be thrown when the **Sort** method is called (an **InvalidOperationException**). This is a reasonable reaction, since the **Sort** method has no way of knowing that we wish to sort according to weight. How can we state this intention to the **Sort** method? In order to be able to sort a set of objects, we must be able to compare them to each other. One object must be “greater than”, “equal to” or “smaller than” another object, in order to make a meaningful sorting of the objects. One way of achieving this is to let the objects implement the **IComparable<T>** interface. This interface contains just one method **CompareTo**:

int CompareTo(T other);

The specification of the behavior of the **CompareTo** method is as follows:

|  |  |
| --- | --- |
| **if** | **then return** |
| The object on which the method is called is **smaller than** the parameter object | A value **less than 0** (zero). |
| The object on which the method is called is **equal to** the parameter object | **0** (zero). |
| The object on which the method is called is **greater than** the parameter object | A value **greater than 0** (zero). |

A valid implementation of **CompareTo** in the **Animal** class is therefore:

public int CompareTo(Animal other)

{

if (Weight < other.Weight) { return -1; }

if (Weight > other.Weight) { return 1; }

return 0;

}

If we now attempt to run the code containing the call of **Sort** on the list of **Animal** objects, the code will execute without errors, and the list will be sorted as intended. How does this relate to type parameter variance? Only by the fact that the type para­meter **T** in the **IComparable<T>** interface is in fact declared as being contra-variant, since T is only used as the type for the method parameter (i.e. input).

Letting your class implement the **IComparable<T>** interface is one way of making the objects “comparable”, and thereby sortable. However, there are two drawbacks:

1. You may not always be able to let a class implement **IComparable<T>**. The class may be a third-party class, or may due to other circumstance be “closed for modification”.
2. You lock the comparison to one specific implementation. You might need to sort the objects according to a different criterion, which would then require modification of the **CompareTo** method.

An alternative is to use the **IComparer<T>** interface. This interface is not an interface that the class itself should implement, but rather an interface that a class dedicated to comparing objects of type **T** should implement. The interface is as such very similar to the **IComparable<T>** interface:

int Compare(T x, T c2);

The specification of the behavior of the **Compare** method is as follows:

|  |  |
| --- | --- |
| **if** | **then return** |
| Object **x** is **smaller than** object **y** | A value **less than 0** (zero). |
| Object **x** is **equal to** object **y** | **0** (zero). |
| Object **x** is **greater than** object **y** | A value **greater than 0** (zero). |

We can then create a new class **AnimalComparerByWeight**, like this:

public class AnimalComparerByWeight : IComparer<Animal>

{

public int Compare(Animal x, Animal y)

{

if (x.Weight < y.Weight) { return -1;}

if (x.Weight > y.Weight) { return 1;}

return 0;

}

}

Again, note that this is a brand new class, that is not part of the **Animal** class itself. With this class available, we can sort our list of **Animal** objects in a slightly different way:

IComparer<Animal> comparer = new AnimalComparerByWeight();

animals.Sort(comparer);

We have now separated the comparison functionality from the **Animal** class itself, and can now provide a specific implementation of **IComparer<Animal>** as a para­meter to the **Sort** method. If we later wish to compare (and sort) Animal **objects** by a different criterion, we simply create a new class containing a different implementa­tion of **IComparer<Animal>**, and use an instance of that class as a parameter to **Sort**.

Is this then an illustration of contra-variance? Yes, in the sense that **T** is also declared as contra-variant for this interface. A more tangible advantage does however reveal itself if we make a small modification to our code:

List<Dog> animals = new List<Dog>();

animals.Add(new Dog("King", 70));

animals.Add(new Dog("Spot", 30));

animals.Add(new Dog("Rufus", 80));

IComparer<Animal> comparer = new AnimalComparerByWeight();

animals.Sort(comparer);

The **animals** list is now a **List** of **Dog** objects, not a **List** of **Animal** objects. That is, the type parameter **T** is now **Dog**. If you read the specification of the **Sort** method (the version taking an **IComparer** implementation as a parameter), you will see that the type of the parameter is **IComparer<T>**, meaning that the actual type should now be **IComparer<Dog>**, where it previously was **IComparer<Animal>**. However, the variable **comparer** has type **IComparer<Animal>**… Still, the method does accept the parameter of type **IComparer<Dog>**, because of contra-variance! What effectively happens when using **comparer** as parameter is something like this:

IComparer<Dog> parameter = comparer; // of type IComparer<Animal> !!

This is exactly the kind of contra-intuitive assignment we discussed earlier, which is only feasible due to the contra-variance of **T** in **IComparer<T>**. If this was not possible, what would the consequence be? You would then need to create a separate class implementing **IComparer<Dog>** – so it could be used as a parameter for **Sort** – but also a class implementing **IComparer<Cat>**, and a similar class for all subclasses to **Animal**! This would be a quite significant drawback as compared to having a single implementation in the **Animal** base class. So even though co- and contra-variance does appear a bit academic, it does provide tangible and significant advantages in practice.

### Generic methods

A final point in relation to Generics is the fact that you can also use Generics at the method level. The classic example is a method for swapping two values (the **ref** key­word means that the parameters are “by-reference”, which means that the values of **a** and **b** will indeed by swapped, also after the method has completed. If we omitted the **ref** keyword, it would be copies of **a** and **b** that would be swapped):

public void Swap<T>(ref T a, ref T b)

{

T temp = a;

a = b;

b = temp;

}

Just as for classes, you need to specify a concrete type when calling the **Swap** method, like this (assuming that **swapper** is a variable of a class containing the **Swap** method):

int x = 12;

int y = 21;

swapper.Swap<int>(ref x, ref y);

In practice, it turns out that the compiler can often figure out the correct type by examining the type of the parameters, so you can actually omit the type specification:

int x = 12;

int y = 21;

swapper.Swap(ref x, ref y);

You can specify a generic method in any class (and interface), and the class itself does not need to have any type parameters. Finally, you can also impose constraints on the type parameter, using the same syntax as for classes:

public void Swap<T>(ref T a, ref T b) where T : class

{

T temp = a;

a = b;

b = temp;

}

## Functions as parameters

The previous chapter has (hopefully) illustrated that the parameter concept goes beyond simple data, since we can perceive types as parameters as well. This ability helps us define code that is as general as possible, postponing the specific choices for values and types to invocation rather than definition. The next step down this road is to perceive **functions** (i.e. methods) as potential parameters as well.

### A first attempt at parameterising code

Consider for instance the problem of finding an object matching certain conditions in a collection. Suppose we have defined a simple class **Car**, like this:

public class Car

{

private string \_licensePlate;

public string LicensePlate { get { return \_licensePlate; }

// Rest of class definition omitted

}

We could then imagine storing **Car** objects in a **Dictionary<string, Car>**, since the **License­Plate** property is a good candidate for acting as a key for **Car** objects. This makes it very easy to retrieve a **Car** object with a specific license plate:

string licensePlate = "CJ 32 802";

if (carsDict.ContainsKey(licensePlate))

{

Console.WriteLine(carsDict[licensePlate]);

}

Now suppose that we for some reason – maybe due to the usage pattern for our col­lection of **Car** objects – have decided to store the **Car** objects in a **List<Car>** instead. This makes it a bit harder to find a **Car** object with a specific license plate, since we would have to explicitly iterate through the collection:

Car theCar = null;

foreach (Car aCar in carsList)

{

if (aCar.LicensePlate == licensePlate)

{

theCar = aCar;

}

}

This is a very generic piece of code. If we wish to find a **Car** object matching a diffe­rent criterion, we only have to change the condition in the **if**-statement:

Car theCar = null;

foreach (Car aCar in carsList)

{

if (aCar.Price == price)

{

theCar = aCar;

}

}

Since we like DRY code (Don’t Repeat Yourself), it would be nice if we could just write this code once, and then turn the part that varies into a parameter. This is a principle we generally apply when making methods as general as possible. However, the part that varies is now not just a simple data value or a type; it is a small piece of logic.

What is the nature of the code in the condition part of the **if**-statement? It definitely produces a **bool** value as “output”, since it is indeed used as a condition. It also seems reasonable to assume that the condition always involves a **Car** object, since the whole purpose of the code is to select a specific **Car** object. We can then think of that piece of code as being a function itself, with at least two properties:

* It takes a **Car** object as input
* It returns a **bool** value

We can compare this with the four elements we always require in order to define a proper function:

* A name
* A parameter list
* A return type
* A body

So far, we only have two of these. In order to create a proper function for e.g. the case where we want a match on the **Price** property, we could create a method called **PriceMatch**:

private bool PriceMatch(Car aCar, int price)

{

return aCar.Price == price;

}

We can then rewrite the loop from above as:

Car theCar = null;

foreach (Car aCar in carsList)

{

if (PriceMatch(aCar, price))

{

theCar = aCar;

}

}

However, this is not really enough. If we want to match on e.g. the license plate, we will have to alter the code once. What we really want is to turn the condition into a parameter, like this (NB: the below code does NOT compile):

private Car FindCar(List<Car> carsList, Condition matchCondition)

{

Car theCar = null;

foreach (Car aCar in carsList)

{

if (matchCondition(aCar, ...))

{

theCar = aCar;

}

}

return Car;

}

We should then be able to call this code like so:

FindCar(carsList, PriceMatch);

We’re closer, but this doesn’t work either. The problem is that the condition methods will require a different set of parameters, depending on their specific implementation. The **Price­Match** method requires a price (of type **int**), the **LicensePlateMatch** method requires a license plate (of type **string**), and so on. Are we then at a dead end? Fortu­nate­ly not! We can solve this problem by introducing **lambda expressions**.

### Lambda expressions

The short definition of a lambda expression is *“an expression that returns a function”*. This may sound very abstract, but it is actually not so different from what we have seen before. We have definitely seen both arithmetic and logical expressions before; such expressions take a number of parameters as input, and returns a value:

int a = 7;

int b = 12;

int resultOfArithmeticExp = a \* b;

bool resultOfLogicExp = a < b;

We can write these two expressions in a more formal way:

* + 1. (int x, int y) => int
    2. (int x, int y) => bool

This should be read as:

* Expression A takes two **int** parameters **x**, **y** as input, and returns an **int** value
* Expression B takes two **int** parameters **x**, **y** as input, and returns a **bool** value

So far, so good. Let us now write two slightly more complex expressions:

1. (int x, int y) => { return x \* y; }
2. (int x, int y) => { return x < y; }

This should be read as:

* Expression A takes two **int** parameters **x**, **y** as input, and returns a function that calculates (x \* y) and returns the result
* Expression B takes two **int** parameters **x**, **y** as input, and returns a function that calculates (x < y) and returns the result

The mind-bending part about this definition, is to realise that this is not describing invocation of the code in the expressions; it is describing a parameterised function, which we can later on “invoke” with specific parameters. These are examples of lambda expressions.

It is probably still hard to see how this helps us, with regards to the previous problem of matching a **Car** object to specific values. Let’s try to formulate a lambda expression closer to what we need:

(Car c) => { return (*code for specific matching*); }

This is also a lambda expression. When will we write such an expression? Typically just when we need it. Let’s see it in the context of an actual **Car** collection:

List<Car> carsList = new List<Car>();

// ...some Car objects are added to the list

string licensePlate = "CJ 32 802";

int price = 45000;

Predicate<Car> carMatchFunc = (Car c) => { return c.Price == price; };

Car matchCar = carsList.Find(carMatchFunc);

carMatchFunc = (Car c) => { return c.LicensePlate == licensePlate; };

matchCar = carsList.Find(carMatchFunc);

There are several things to take note of in this code. In the highlighted line, we have specified a lambda expression (the yellow part), which follows the syntax we intro­duced above. A particularly interesting feature is that the function part (after the => symbol) uses the local variable **price**. This has the very important consequence that the function only needs a **Car** object as parameter. So, this particular lambda expres­sion takes a **Car** object as input, and returns a function that compares the **Price** pro­perty on the **Car** object with the value of the local variable **price**.

What about the blue part? The lambda expression in the yellow part returns a func­tion taking a **Car** as input, and returning a **bool**. This kind of function is considered a type, just as any other type in C#. The type **Predicate<T>** - which is part of the .NET class library – is defined as being a function taking a parameter of type **T**, and retur­ning a **bool** value. The variable **carMatchFunc** thus has the type *“a function taking a* ***Car*** *object as input, and returning a* ***bool*** *value”*. This is exactly what our lambda expression does, so the code is indeed valid.

In the next line, we make the call **carsList.Find**(**carMatchFunc**). If you study the docu­mentation for the **List<T>** class, you will see that the **Find** method precisely takes a parameter of type **Predicate<T>**. The method will then return the (first) object for which the function return **true**. The call **carsList.Find**(**carMatchFunc**) will thus return the **Car** object for which the **Price** property equals the value of **price**. This is exactly the functionality we wanted!

The last two lines illustrate that **carMatchFunc** is indeed just a variable, to which we can assign different values. In this case, we assign a different lambda expression to the variable (now matching on license plate), and call **Find** again. We have now truly parameterised the logic for finding a **Car** object matching certain conditions.

Wrapping your mind around this functions-as-parameters idea is challenging, and it requires you to think about functionality in a quite abstract way. Still, it is just another incarnation of principles we have seen before. Let’s try to compare the code from above to some simpler code:

Predicate<Car> carMatchFunc = (Car c) => { return c.Price == price; };

Car matchCar = carsList.Find(carMatchFunc);

// ...is just as

int index = 12;

Car someCar = carsList[index];

carMatchFunc = (Car c) => { return c.LicensePlate == licensePlate; };

matchCar = carsList.Find(carMatchFunc);

// ...is just as

index = 16;

someCar = carsList[index];

In the first half, we declare some local variables; two of type **Car**, one of type **int** named **index,** and one of type **Predicate<Car>** named **carMatchFunc**. We assign a specific value 12 to **index**, and a specific “value” (being a lambda expression) to **car­MatchFunc**. We then use these variables – with the values currently assigned to them – to look up some **Car** objects. In the second half, we assign new values to **index** and **carMatchFunc** – the value 16 and a new lambda expression, respectively – and once again use them to look up **Car** objects. So, we declare variables and assign values in the code shown above; those variables – and thus their current values – are then used inside the **List** methods.

For completeness, it should be mentioned that the **List** class contains several variants of the **Find** method. **Find** finds the first object for which the given predi­cate returns **true**, while the **FindLast** method finds the last object. We can easily imagine that the predicate will return **true** for more than one object, so a **FindAll** method is also availa­ble, which returns all objects for which the predicate returns **true**.

The syntax for lambda statements described above can be considered the ”fully dres­sed” version of the syntax. The compiler can often deduce the type of the parameters from the context, and you can also omit parentheses if the expression only takes a single parameter. A more hard-boiled version of the code in our example is thus:

Car matchCar = carsList.Find(c => c.Price == price);

matchCar = carsList.Find(c => c.LicensePlate == licensePlate);

### Delegates

The ability to parameterise functions with other functions is a powerful tool, when it comes to write functions which are as general as possible. The **Find** method is a very good illustration of this idea; we can write a general method that performs an itera­tion through a collection of objects, picking out objects that match a certain criterion, while making it possible for the caller to specify the exact criterion. We also saw that the “signature” of a method (input parameters and return value) can be considered a type – the **Predicate<Car>** was an example of this – and we can therefore declare vari­ables of such a type, and let them refer to e.g. a lambda expression. This was just a single example of a so-called **delegate**.

A **delegate** is essentially just a variable that can refer to a function. We claimed in the previous section that using a variable of such a “function-reference” type was quite similar to using e.g. an **int** variable, and that is almost true. The “almost” is added due to the fact that a delegate can in fact refer to a collection of functions! That is, a dele­gate can refer to zero, one or many functions. When a delegate is invoked, it will in turn invoke all of the functions to which it refers; it “delegates” the actual work to these functions.

The original syntax for creating and using delegates is a bit peculiar, so we will just show it briefly for completeness, and then proceed quickly to a more modern style. The **Predicate<T>** is an example of a this modern style.

Creating and using a delegate formally involves first creating a delegate type, and then a declaring a variable of that type:

delegate bool CarCheckDelegate(Car c);

private CarCheckDelegate theCarCheckDelegate = null;

The first line declares the type **CarCheckDelegate**, which returns a **bool** value, and takes a **Car** object as parameter. The next line then declares a variable **theCarCheck­Delegate** of type **CarCheckDelegate,** and sets its initial value to **null**. With this in place, we can then assign a function reference to the variable:

theCarCheckDelegate = c => c.LicensePlate == licensePlate;

We have still not executed any code; in order to do this, we “call” the delegate:

Car aCar = new Car("CJ 32 802", 5, 50000);

bool result = theCarCheckDelegate(matchCar);

The last line will invoke all of the functions to which the delegate currently refers (in this example just one function).

The above code is valid and will work fine, but the somewhat lengthy syntax can be avoided by using some of the built-in, type-parameterised delegate types, like e.g. **Predicate<T>**. A handful of these delegate types exist:

|  |  |
| --- | --- |
| **Action**  **Action<T1>**  **Action<T1, T2>**  …  **Action<T1,…,T16>** | An **Action** delegate has no (i.e. **void**) return type. All type parameters are thus the types of the input para­meters. You can specify up to 16 input parameter types. |
| **Func<TRes>**  **Func<T1, TRes>**  **Func<T1, T2, TRes>**  …  **Action<T1,…,T16, TRes>** | A **Func** delegate has return type **TRes**. All type parame­ters except the last one are thus the types of the input para­meters. You can specify up to 16 input parameter types. |
| **Predicate<T>** | A **Predicate** delegate always returns a **bool**, and takes one input parameter of type **T**. |
| **Converter<TIn, TOut>** | A **Converter** delegate always returns a value of type **TOut**, and takes one input parameter of type **TIn**. |
| **Comparison<T>** | A **Comparison** delegate takes two input parameters of type T, and should return an int value, following the same rules as specified for the **IComparer** interface. |

It is fairly easy to see that the last three types of delegates are just special cases of the **Func** delegate. So why do they exist? Partly for historic reasons, but perhaps also to make the intention of a delegate clearer. Knowing about **Action** and **Func** usually suf­fi­cient to work with delegates.

We have already seen that use of the built-in delegate types makes the syntax a bit shorter. Declaring a delegate is typically a one-line operation now:

Predicate<Car> theCarCheckDelegate = null;

Assignment to – and invocation of – the delegate follows the same syntax as before.

We claimed above that a delegate can refer to a collection of functions. The syntax for this is fairly straightforward. When the delegate has been declared as above, it refers to zero functions. Adding a reference to a function is done using the **+=** operator:

theCarCheckDelegate += c => c.LicensePlate == licensePlate;

In general, you should use **+=** when adding a function reference to a delegate, since using **=** will remove the existing references! You can subsequently add more function refenreces to the delegate, and even remove them again using the **-=** operator. When the delegate is invoked at some point, all functions to which the delegate refers are invoked, using the same parameters as specified in the delegate invocation.

### Events

The idea of having variables referring to methods, and to invoke several methods “indirectly” through a delegate, does seem to be in contrast with the usual way of structuring code, where method calls are written explicitly. We have already seen that the delegate concept is a useful tool for turning code into a parameter, but that does not imply that delegates in general are a superior way to structure code. For applica­tions where many “clients” (a “client” is here defined as a specific part of the applica­tion code) are interested in being informed about changes in other parts of the code, delegates can be suitable solution. In fact, a specialised version of delegates called **events** are specifically designed for such scenarios.

The only feature that distinguishes an event from an ordinary delegate is the use of the keyword **event** used in the declaration:

event Action<double> TemperatureChanged;

This declares a delegate of type **Action<double>** (no return value, takes one para­meter of type **double**), which is also an event.

How could this event be used in practice? Suppose we have defined a class **Tempera­tureMonitor**, which monitors a temperature of some sort. The event declared above could then be part of this class. It would be declared either as a public instance field, or hidden behind methods/properties. In the code below, the first solution has been chosen:

public class TemperatureMonitor

{

private double \_temperature;

public event Action<double> TemperatureChanged;

public TemperatureMonitor()

{

TemperatureChanged = null;

MonitorDevice();

}

private void MonitorDevice()

{

// We assume GetTemperatureFromDevice retrieves

// an actual temperature from some device, and

// keep doing so at regular intervals

double newTemperature = GetTemperatureFromDevice();

if (Math.Abs(newTemperature - \_temperature) > 0.1)

{

\_temperature = newTemperature;

OnTemperatureChanged();

}

}

private void OnTemperatureChanged()

{

TemperatureChanged?.Invoke(\_temperature);

}

}

The single line in **OnTemperatureChanged** is a standard code “idiom” (a short piece of code used very often), which reads “if **TemperatureChanged** is not **null**, call **Invoke** on **TemperatureChanged**, otherwise do nothing”. Given the code in **MonitorDevice**, the net effect is thus that whenever the temperature changes (we have included a thres­hold of 0.1 degree, to avoid calling **OnTemperatureChanged** on very small changes), the method **OnTemperatureChanged** is called, which in turn invokes the delegate **TemperatureChanged**. When an event delegate is invoked, is it often called to raise the event.

The idea is now than any client interested in knowing about temperature changes can “attach” a function to the event. The only requirement for the function is that it must match the type of the delegate, in this case **Action<double>**. We can imagine that e.g. a class responsible for displying the temperature in a GUI would like to be notified about temperature changes:

public class GUIClient

{

// Rest of class omitted by brevity

public void TemperatureHasChanged(double temperature)

{

Console.WriteLine("Current temperature : " + temperature);

}

}

The **TemperatureHasChanged** method can now be attached to the event delegate:

TemperatureMonitor monitor = new TemperatureMonitor();

GUIClient client = new GUIClient();

monitor.TemperatureChanged += client.TemperatureHasChanged;

Note that the attachment is done for objects; if we for some reason need several **GUI­Client** objects, we must attach the **TemperatureHasChanged** method to the event for each of those objects. However, it is indeed possible to declare an event as **static**, just as for any other instance field.

It is not obvious why we need the **event** keyword at all, since an event dele­gate seems to be just like any other delegate. That is true, except for a subtle diffe­rence, relating to a remark made earlier in this section. Note that in the highlighted line of code, we use the **+=** operator to attach a method to the delegate. This is the correct way to do this, since using **=** would remove all previously attached methods. However, for an event delegate, we cannot use **=** at all! If we try to substitute **+=** with **=** in the above code, Visual Studio reports an error: *“The event* ***TemperatureChanged*** *can only appear on the left-hand side of* ***+=*** *and* ***-=****, except when used within the class* ***Tempera­ture­Monitor****”*. Removing the **event** keyword from the declaration of **Tempe­rature­Changed** in **TemperatureMonitor** will “fix” the error, i.e. make the code compi­lable. Adding the **event** keyword is thus a fail-safe mechanism, preventing impro­per use of the delegate.

The concept of “event-driven applications” is not easy to graps initially, since it is very different from the classic, sequential execution model. However, whenever you have an application where one part of the application needs to know immediately if some­thing specific happens in another part, you will probably need to use events for mana­ging this. The typical example is a GUI-rich application, where the GUI needs to invoke an action when a user performs a GUI operation. Events can also be used if changes in data (as in the example) need to be relayed immediately.

# GUI Development

So far, we have only used some very primitive facilities for interacting with the users of our small applications. For more sophisticated applications with richer user inter­action, this is clearly not enough. We will therefore now embark on the – quite large – topic of GUI (Graphical User Interface) development.

Modern GUI development is a rather complicated activity. There is a large number of GUI “components” or **controls** (like buttons, drop-down menus, list boxes, etc.) to choose from, and the layout of the GUI can be specified in many different ways. In this chapter, we only discuss the more general challenges related to GUI develop­ment, and subse­quently focus on so-called **data binding**, which relates to the problem of keeping the GUI and the underlying data model consistent at all times.

## GUI and Object-Orientation

We have so far mostly used Object-Orientation as a way of modeling aspects of a cer­tain **domain**, like a bank, a role-playing game, etc.. Object-Orientation is however also a useful tool in the realm of GUI development. We can imagine classes that represent specific GUI elements – like a **Button** class, a **Window** class, etc. – and we can also imagine relations between such classes, in terms of inheritance and composition. Not surprisingly, such a system of classes has already been created, and is part of the .NET Framework class library. In fact, this part of the class library is quite large, and it is easy to get lost in the vast jungle of controls, available properties, and so on. We will only scratch the surface here.

## Event-driven applications

Applications equipped with a “real” GUI usually have a different model of execution than we have seen so far. Until now, most applications have had a very well-defined flow-of-execution, beginning execution of the logic immediately when the application is launched, and usually executing the entire logic without user interaction. A GUI-rich application can be characterised as an **event-driven** application, meaning that the application will usually wait for the user to initiate some specific action, perform that action on the users initiative, and then wait for the user to initiate the next action. The user may initiate an action by e.g. clicking on a button or making a selection in a list box; such an **event** will in turn launch a specific part of the business logic defined in the application. As we will see later on in this chapter, this model of execution makes it a bit more complicated to activate specific parts of the code, in response to the users actions.

## The duality of a GUI components

All GUI components share a few characteristics, that influence how we in general define and use GUI components. Most importantly, the full specification of a GUI compo­nent falls into two fundamentally different categories:

* The **visual appearance** of the GUI component: How does the component manifest itself on the device on which the user interacts with the application?
* The **behavior** of the GUI component: What happens inside the application when the user interacts with the GUI component?

Why is this distinction important? Primarily due to the role of time. The visual appear­ance of a GUI component does not depend on the “flow of execution”, but has a more timeless nature (this is a bit simplified, but we will elaborate a bit on this shortly). Conversely, behavior is per definition something that starts (with respect to time) and ends. This difference is important in relation to how we should specify appearance and behavior, respectively.

Considering behavior first, it should not be surprising that C# itself is a very suitable language for specifying behavior – that’s what we have been doing over and over in the previous chapters. However, C# may not be the best choice for specifying appear­ance. We have characterised C# as an Object-Oriented language; it can also be charac­terised as a language with a **procedural** nature. In less academic terms, this means a language where we describe how things are done. We use sequences of state­ments to describe this, with the underlying assumption that statements are executed in a certain order with respect to time. Other languages can be characterised as being **declarative**. These languages only describe relations between certain elements, and time is not as such a factor. One example of such a language is HTML (HyperText Markup Language). HTML has traditionally been used for specifying the layout of a website, i.e. its visual appearance. The point we’re trying to make is: since the visual appearance is not as such dependent of time or the “flow of execution” of the appli­cation, it can be advantageous to use a declarative language to describe the visual appearance, rather than try to describe it in e.g. C#. Or even shorter: Use the right tools for the right job!

So, the conclusion of the above discussion is:

* Use C# for defining behavior for a GUI component
* Use a declarative langauge for defining the visual appearance of a GUI component.

This conclusion leads to two new problems:

* What language should we then use for defining appearance?
* How can we define a single C# class that contains definitions of both the behavior and the appearance?

Microsoft has resolved this by introducing the language **XAML** (eXtensible Application Markup Language), which is a specialised version of **XML** (eXtensible Markup Langu­age). We will talk a bit more about XAML/XML in a moment; for now, just note that XAML is indeed a declarative language.

If XAML is used for defining appearance for a GUI component, while C# is still used for defining behavior, how is it then possible to “merge” this into a single C# class? C# does – very conveniently – allow a class definition to be split across more than one file! The keyword **partial** can be added to a class definition, which indicates to the compiler that the class definition is spread across a number of files.

With this in place, only one more piece of the puzzle remains: transformation of a XAML-based definition to a (partial) C# class. This is indeed possible to do, and is an integral part of Visual Studio. We don’t need to know how this is done, only that it is possible to do it ☺.

The diagram below should illustrate how the pieces are put together. The behavior part is written in C# “by hand”, and is stored in a file with the extension **.xaml.cs**. The appearance part is written (or imported, see later) in XAML, and is stored in a file with the extension **.xaml**. This file is then processed by Visual Studio, generating a C# file with the extension **.xaml.g.cs**. This file – which now contains a partial class definition – is then combined with the hand-written C# file, which also contain a partial class definition. The end result is thus a complete C# class, containing definitions for both behavior and appearance.

MyControl

C# - generated

MyControl (behavior)

C# - written

MyControl.xaml.cs

MyControl (appearance)

C# - generated

MyControl.xaml.g.cs

MyControl (appearance)

XAML - written/imported

MyControl.xaml

This may look like a lot of trouble for creating a C# class, and it is indeed possible to write a GUI component class directly in C# without all this fuss. However, the division of definition of behavior and appearance does make it possible to define a GUI using an external tool. If the external tool is capable of exporting a GUI definition to XAML, we can simply import that definition into Visual Studio! Microsoft has made such a tool, called **Microsoft Blend**. The point is that GUI (appearance) definition is an activity that doesn’t really require programming skills, so it should be possible for a design professional to work with GUI design using a tool specialised for this purpose.

## What is XML (and XAML)

We claimed above that XAML is a specialisation of XML, which in turn is a declarative language. So, what is XML?

XML (eXtensible Markup Langu­age) is a language for specifying structural relations between data. What does that mean? Suppose I have some data about a small book collection, like:

“War and Peace”, Tolstoy, 539 pages

“Huckleberry Finn”, Twain, 341 pages

As human beings, we can fairly easily understand this data. If a computer appli­cation had to process this data, it would need some extra data to make sense of the data. Suppose now we write the data as:

<BookCollection>

<Book>

<Title>War and Peace</Title>

<Author>Tolstoy</Author>

<Pages>539</Pages>

</Book>

<Book>

<Title> Huckleberry Finn</Title>

<Author>Twain</Author>

<Pages>341</Pages>

</Book>

</BookCollection>

Here we have added “data about the data”, so-called **meta-data**. This meta-data can be used by an application to e.g. search for specific types of data, or to display the data in a certain manner, depending on the type of data. Also, the meta-data defines the relation between the actual data. More specifically, we express the meta-data in terms of **tags**. A tag is a sort of keyword, that we have decided has a certain meaning. In the example, the word **Book** has a certain – hopefully obvious – meaning. We can then insert a tag into our description like this:

<Book>

</Book>

The **<Book>** tag means *“now will follow data about a Book”*, and the **</Book>** tag means *“now ends the data about a Book”*. Between these so-called **opening** and **closing** tags, we can then add data about a specific book:

<Book>

<Title>Huckleberry Finn</Title>

<Author>Twain</Author>

<Pages>341</Pages>

</Book>

So, it seems that there are three pieces of data about a book: title, author and (the number of) pages. All of the above (the opening tag, the details and the closing tag) defines an **element**, here of type **Book**. Note that the details themselves follow the same structure: opening tag, details, closing tag. We can thus have elements within elements, and this is precisely what enables us to express relations between ele­ments. In the XML above, the **Title** element is a “child” of the **Book** element, and **Book** elements are themselves children of the **BookCollection** element.

This is essentially what XML is; data, and meta-data specifying the structure of the data itself. You can hopefully see that such a language has a very different nature than C#, since there is no concept of “flow of execution” here. We only declare certain relations between data. A file containing XML code – plus a little bit of information about the XML version being used – is formally called an **XML document**.

Even though we have just stated that XML is very different from C# (which is true), it doesn’t mean that it operates in a realm that is completely different from that of C#. If you think a bit about it, you can interpret the XML data as describing the structure between specific objects: A **BookCollection** object is just a collection-type object, that contains two **Book** objects. A **Book** object in turn has some properties **Title**, **Author** and **Pages**, that have some specific values. If we had defined similar classes in C#, we could easily create a similar structure by creating C# objects, and use composition to relate them to each other. Some describe XML as an “object instantiation language”, which is fairly accurate. In that light, it may be easier to understand why a transfor­mation from XML to C# isn’t that complicated to perform.

We said earlier that by adding this meta-data, a computer application could make better sense of the data. This is a bit too simplified. No computer applications know as such what a “book” is… When you wish to use data on XML format, the “producer” (which could be a human being or perhaps another application) and the “consumer” (the application receiving the XML) of the XML must agree on the XML “language” used for communication. That “language” is essentially just a specification of

* Names that are considered valid tag names (i.e. data types)
* What relation elements must have to each other

Such a specification is called a **schema**. We don’t need to know much more about schemas at this point, but it is the schema that defines the type names we may use and how the types are related. For the above example, the schema may define that:

* **BookCollection**, **Book**, **Title**, **Author** and **Pages** are legal types
* A **BookCollection** element may contain any number of **Book** elements
* A **Book** element must contain exactly one **Title** element
* …and so on

In this way, it is the schema that defines a “specialisation” of XML. Our example is related to books, so we could name this specialisation XBML (eXtensible Book Markup Langu­age). Likewise, Microsoft has defined a schema related to specifi­cation of graphical elements in an application, and named it XAML (eXtensible Application Markup Langu­age). Describing the structure of a GUI fits pretty well with this way of structuring information. A GUI usually starts with a window, inside which is a number of pages, within which there are a number of controls, and so on.

XAML makes heavy use of another XML feature called **attributes**. An attribute is just another way of representing data. In the example above, we could have chosen to use attri­butes instead:

<Book Title="War and Peace" Author="Tolstoy" Pages="539">

</Book>

The meaning is exactly the same, but expressed in terms of attributes. Attributes are written inside the **<** and **>** of the opening tag. Each attribute should be understood as a key-value pair; the value of **Title** is *“War and Peace”*. Note that all attribute values must be specified as strings, even if they are numeric (such as the page number). It should also be mentioned that you can use a shorthand for tags, if the element only contains attributes:

<Book Title="War and Peace" Author="Tolstoy" Pages="539" />

The **/** symbol from the closing tag has thus been moved to the end of the opening tag. As before, the meaning is exactly the same.

When should you use attributes in favor of child elements? There is no clear-cut rule for this, and it is strictly speaking not a relevant question here, since we will have to follow the style that XAML “imposes” on us. The important point here is to under-stand what an attribute is.

There are of course many tutorials about XML on the web, if you want to learn more. The W3Schools tutorial <http://www.w3schools.com/xml/> is a good starting point.

## XAML and Visual Studio – getting started

Let’s recap the story so far:

* There is a fundamental difference between the visual appearance and the behavior of a GUI component
* The behavior is defined by using C#
* The visual appearance is defined by using XAML (eXtensible Application Markup Language)
* You can create the (visual appearance of a) GUI directly in Visual Studio (using XAML), or by using a third-party tool

The next step is to investigate in more detail how we go about creating an application with a real GUI. In Visual Studio, we have usually created and worked with so-called “console applications”, where we use **Console.WriteLine** to print simple texts on the screen. However, we can also choose to create an application of the type Universal Windows Platform (UWP). In Visual Studio, we can choose this application type in the **New Project** dialog:



Here, we choose to create a “Blank App (Universal Windows)”, and have chosen the name **GUIExample**. Once Visual Studio has created the project (which may take a little while, if this is the first project of this kind you are creating), we are met with some­thing which looks radically different from the projects we have worked with so far. The **Solution Explorer** window will look something like this:



There is no Program.cs file, and a couple of files with the extension .xaml. The file called MainPage.xaml is the most interesting one right now. If you double-click this file, an view looking similar to the below will appear:



This is the **main view** of the application right now. Or more specifically; if we run the application as it is right now, this is what it will look like, if we run it on a Windows phone with a 5” screen with resolution 1920x1080. One of the fundamental ideas behind UWP is that a UWP application should be able to run on any hardware device that runs Windows 10, be it an ordinary computer, a smartphone, an Xbox, a Smart-TV or whatever. Therefore, it is possible to preview what an application will look like on a certain device. If you look closely in the upper-left corner of the previous image, you will see a list-box where **“ 5” phone (1920 x 1080) ”** has been chosen. We can choose other devices, for instance a 23” desktop with 1920x1080 resolution. We will in general work with this choice, and will not in this text dive into the details related to how to ensure that a UWP application looks reasonable on all possible devices. Assuring this involves setting up the GUI elements in a certain manner, which is not a feature we will focus on here.

Below the top part (usually called the **Design View**) is a window containing XML code; more specifically XAML code, as described earlier. This may initially look quite com­plex, but the good news is that the XAML code present from the beginning is usually not something we need to think much about, let alone change. This is actually the definitions of the “schemas” we also mentioned earlier, so this just ensures the we “speak XAML” in this file.

The only part that is directly related to the GUI is the **<Grid>** element. Visual Studio inserts this per default. To keep things simple, we will simply delete it, so we start completely from scratch. Be sure, however, that you only delete the **<Grid>** tag (from **<Grid>** to **</Grid>**, both included).

With this out of the way, we have a blank canvas to work with. Right now, the GUI does not contain any elements at all. How do we then add a GUI element? You can do this in a couple of ways.

The first way is the graphical way. Just next to the **Solution Explorer** window, there should be a **Toolbox** pane to click on (alternatively, choose **View | Toolbox** in the menu). If you then expand the **Common XAML Controls** element, you will see a long list of GUI elements. One of those elements is called **Button**. If you click on the entry in the list, you can then simply drag an element onto the view. If you do so, you will probably end up with an enormous button, filling up the entire view! Don’t worry, we will fix that later on. For now, just note that some text (probably one long line of text) has been added to the XAML part of the window. The text (with a bit of formatting) will probably look like:

<Button x:Name="button" Content="Button" HorizontalAlignment="Left"

Height="1080" VerticalAlignment="Top" Width="1920"/>

How do we read this? The tag type specifies what kind of GUI control the tag defines, in this case a **Button**. Next follows a “programmatic name” for the button, which we don’t need to worry about now. Then follows the “content” of the button, which for now is simply the text **Button**. The last four elements are related to the graphical layout of the button; if you try to change 1080 to 100, and change 1920 to 200, you will see that the button changes size accordingly (remember that the values must be enclosed in **“”**, even though they are numeric!).

You have probably by now figured out that the visual view and the XAML code are just two sides of the same thing; it is simply two ways of looking at the same data. This also means that you can work with the GUI in both ways, choosing the way that you are most comfortable with. You can thus add GUI elements simply by writing proper XAML in the XAML window. There is even a third way to work with the GUI: If you select the button in the view and then press F4, a **Properties** window will open:



Here the **Layout** part has been expanded, and you can again see the values that we have already seen in the XAML code and in the Design view. So, the Properties view is just a third view into the same data. If you close the Properties view, you can always open it again by pressing F4.

All these ways to view data, the number of GUI controls and the myriad of properties you can set for a control can feel overwhelming, and there are indeed many little knobs you can turn. However, you only need to know about a few control and a few properties for controls in order to get started.

Before diving into the specific of certain, a couple of thing about working with these types of applications should be mentioned. First, it is as mentioned before possible to get a “preview” of how the application will look on various devices, by choosing such a device in the Design view. It is even possible to try to run the application in a device simulator, where it is not only the visual appearance that is simulated, but also things like memory limitations, etc.. You choose between such simulators by expanding the lis box attached to the well-known **Start** button. However, note that you will need to download device-specific simulators to Visual Studio, before they can be used. In or­der to keep things simple, we will in general just go with the “Local Machine” option, that just launches the application directly on your computer.

Another – somewhat more obscure – issue is related to setting access rights to the file system. If you create a UWP application – or e.g. download an exercise project – build it, and try to launch it, you may very well experience an “Access denied” error. Access to the file system is a non-trivial issue for UWP applications[[8]](#footnote-8), but the easiest way to fix theis problem is as follows.

1. Using the **File Explorer**, go to the folder where you intend to store your UWP applications.
2. Right-click the folder, and go to **Properties** in the menu.
3. Go to the **Security** tab.
4. Click on the **Edit** button.
5. Click on the **Add** button.
6. Add **SYSTEM** as a user name.
7. Give the **SYSTEM** user full access rights to the folder.

This should get rid of any problems with denied access. With that out of the way, we return to the description of the GUI controls.

## Simple controls

The term “simple controls” is not a formal one – we just use it to describe a handful of very fundamental controls, that will almost always be part of a typical GUI. You may of course encounter more specialised situations, where you will need to rely on other types of controls. If so, seek information about such controls online[[9]](#footnote-9).

### Button

We have already seen an example on a **Button** control, where the content of the button was a simple text. The content of a button can be more complex – for instance an image combined with text – but the overall purpose of a button is of course to enable a user to invoke a certain action. We will later see how you associate such an action with the button, such that the action is invoked when the user clicks/taps on the button.

### TextBlock

A **TextBlock** is just a piece of text to be displayed somewhere in the GUI, without being part of another control (e.g. a **Button**). The user cannot as such interact with a **TextBlock**, but it is possible to change the text in a **TextBlock** dynamically, as we will see examples of later. A very simple **TextBlock** will in XAML look like:

<TextBlock Text="Hello there" />

The **Text** property is thus the holder of the content itself. In addition to this, there are of course a multitude of attributes related to the visual appearance of the text. These attributes are probably easiest to explore in the Properties window.

### TextBox

A **TextBox** is to some extent similar to a **TextBlock**, the major difference being that a user can usually enter text into a **TextBox** control (if the control is enabled for recei­ving user input). A simple **TextBox** in XAML could be:

<TextBox Text="Hello there" Width="100" Height="50" />

Just as for the **TextBlock**, the text itself can be formatted in various ways. It is quite common to use a **TextBox** together with a **TextBlock**, where the **TextBlock** can con­tain a “lead text”, describing the data you should enter into the following **TextBox**.

### Image

An **Image** control will – not surprisingly – contain an image, and it will usually be a passive element in the GUI. The most important property of an **Image** control is the **Source** property, where you specify the source for the image to display. The source is a URL; it can point to a local file, but also to a destination on the web:

<Image Source="http://shortly.be/content/auto\_site\_logo.png" />

Note that an **Image** control can be used as part of the content of other controls, for instance a **Button**. It is also possible to make an **Image** control responsive, i.e. make it react to being clicked or tapped by the user.

### Slider

The **Slider** control is maybe on the borderline of being a “simple” control, and does not occur that often in GUIs compared to the previous controls. A slider control can be used to set a value between a specified minimum and maximum value. This can be utilised to allow the user to choose a numeric value inside a certain interval, without requiring the user to enter the number into e.g. a **TextBox**, which would require that the application does some validation of the entered data. This can be avoided using a **Slider** control. A simple **Slider** control can look like.

<Slider Maximum="800" Minimum="100" Value="400"/>

Whenever the user slides the marker in the **Slider** control, the **Value** property will change accordingly. We will later see how that value can then be used to control the value of properties in other controls.

## Layout controls

The small examples above have not been placed in some sort of context. When a GUI becomes just a bit more complex, we need ways to organise the GUI elements into larger groups, in order to manage the visual layout. A number of **layout controls** are available for this purpose. You may recall that Visual Studio put a **Grid** element into the XAML code per default; the **Grid** control is an example of such a layout control.

### Grid

The **Grid** layout control is used if the overall layout of a window follows a regular row/ column structure, i.e. where all columns have the same number of rows, and vice versa. Placing a **Grid** tag inside the XAML code will create a 1-by-1 grid; if you wish to create a grid with several rows and columns, you will need to add a number of so-called row- and column-definitions to the **Grid**. The below XAML defines a 3-by-2 grid:

<Grid>

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

</Grid>

In the Design view, you will see something like:



There are a number of ways to control how the columns and rows are sized with respect to each other; we will get back to this in the chapter on control properties. Once a suitable structure has been defined, you can add controls inside specific grid cells. You do this using the below syntax:

<Grid>

<Grid.RowDefinitions>

<RowDefinition/>

<RowDefinition/>

<RowDefinition/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<Image Grid.Row="0" Grid.Column="1"

Source="http://shortly.be/content/auto\_site\_logo.png"/>

<Slider Grid.Row="1" Grid.Column="1"

Maximum="800" Minimum="100" Value="400" Width="400" />

</Grid>

Since the controls are defined within the **Grid** tag, they are per definition assumed to be positioned within the grid. All specifications of relative positioning of a control – like e.g. alignment – will now be done relative to the grid cell containing the control.

### StackPanel

The **StackPanel** control is in a sense more primitive than the **Grid** control, but also more flexible. Inside a **StackPanel**, you can specify a sequence (or “stack”) of controls, e.g. like this:

<StackPanel>

<Image Source="http://shortly.be/content/auto\_site\_logo.png"/>

<Slider Maximum="800" Minimum="100" Value="400" Width="400" />

</StackPanel>

This will “stack” the controls visually on top of each other. This is in itself rarely what you want. However, two additional features make the **StackPanel** really useful:

* You can orient the **StackPanel** to stack the controls either vertically (default) or horizontally
* You can put **StackPanels** inside **StackPanels**

The latter property is as such not something special for the **StackPanel**; if you wish, you can also put a **Grid** control within a **Grid** control. Still, these two features alone make it possible to create quite sophisticated layouts. Also, there is no need to specify the position of an embedded control explicitly (as was the case for the **Grid** control), since the order of the controls themselves specify the position within the **StackPanel**. Only drawback is perhaps that the nesting level of the **Stackpanel** controls tends to become quite deep.

## Control properties

We have already used some of the (many) properties that are available on controls, and we will not try to give a comprehensive description of all properties here. A good way to explore the available properties for various control types is to select a control in the Design view, and then open the Properties window (press F4). Also, remember that even though there might be dozens of properties to fiddle with, you will usually only need to use a few of them. In the below, we will give an overview of some impor­tant categories, and describe a few specific properties in more detail.

### Default properties

All controls have a so-called default property. The default property can be set simply by writing its value between the opening and closing tag of the control, like:

<TextBlock>Hello all</TextBlock>

The default property for a specific control is usually chosen such that it is a commonly used property. For a **Page** control, the default property is **Content**, which is the reason for the commonly seen error *“The property ‘Content’ is set more than once”*, if you just add a number of controls directly to a page. The error is usually fixed by wrapping the controls into a **StackPanel** tag.

### Complex properties

The term complex is not a formal one here; it is just a common denominator for properties that cannot be set by a single simple value. Consider for instance this definition of the background for a **Button** control:

<Button.Background>

<LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">

<GradientStop Color="Black" Offset="0" />

<GradientStop Color="White" Offset="1" />

</LinearGradientBrush>

</Button.Background>

Here we have a complex definition of the background, so we need to write out the entire hierarchy of “sub-properties” we need to set.

### Attached properties

We have already used attached properties, when we defined the positioning of a control inside a **Grid** control:

<Image Grid.Row="0" Grid.Column="1"

Source="http://shortly.be/content/auto\_site\_logo.png"/>

<Slider Grid.Row="1" Grid.Column="1" Maximum="800" Minimum="100"

Value="400" Width="400" />

The **Grid.Row** property is an attached property for e.g. the **Image** control, since it refers back to enclosing **Grid** control.

### Layout properties

The layout of most controls can be controlled in quite detailed ways, and it can be somewhat confusing to figure out why a certain combination of layout property values result in a certain graphical layout. There are various ways to e.g. specify the size of a control; some specify the size in an absolute measure (e.g. pixels), while others specify a relative size. The commonly used properties **Height** and **Width** can be set in (at least) four ways:

|  |  |
| --- | --- |
| “Auto” | Adjusts according to the controls inside the control in question |
| “\*” | A certain part of the controls total size |
| “60” | 60 physical pixels |
| “3\*” | A certain part of the controls total size (three times larger than “\*” |

On top of this comes the fact that sizes are sometimes relative to the size of the en­clo­sing control, which may again be relative, and so on… The best advice is probably to go for a fairly simple layout initially, which can then be adjusted later.

A first attempt at organising a set of controls often results in the controls being visu­ally mashed up against each other, which is not very visually pleasing. The **Margin** property comes in handy for this problem. If you wish to create a **Button** control with a 10-pixel margin to all sides, it will look like:

<Button Content="OK" Margin="10,10,10,10"/>

The margins are specified in the order left-top-right-bottom. Adding just a fex pixels of space between controls will usually make a significant difference.

### Using styles

Once your GUI grows beyond a few controls, you will often be in a situation where you want several controls of the same type – say, a set of **TextBox** controls – to follow the same layout. This can of course be achieved just by explicitly setting all relevant properties for each control, but this can quickly become cumbersome to maintain, if you decide to change the layout. One way around this is to define a **Style**.

A **Style** is a set of settings for certain properties, that you wish to apply to several controls of the same type. A **Style** is specified as a so-called **Resource**, typically apply­ing to an entire **Page**, like:

<Page.Resources>

<Style x:Key="TextBoxStyle" TargetType="TextBox">

<Setter Property="FontSize" Value="24"/>

<Setter Property="Width" Value="300"/>

<Setter Property="Margin" Value="5,5,5,5"/>

</Style>

</Page.Resources>

Apart from the slightly peculiar **Setter/Value**-syntax, it is pretty straightforward. You can then apply such a style to a control of the specified type, like:

<TextBox Style="{StaticResource TextBoxStyle}" Text="(Name)"/>

There are of course more in-depth descriptions of layout properties – and XAML layout considerations in general – available elsewhere, for instance here[[10]](#footnote-10).

## Data Binding

As the previous chapter illustrates, it is by no means a simple task to accurately speci­fy the graphical layout of an application GUI…and we haven’t even begun to use the GUI for anything yet! The GUI is rarely an end-goal in itself; it is just a tool for helping the user to interact with the application. The ultimate goal for most applica­tions is to do some sort of data manipulation, and the GUI is the surface through which the user can manipulate the data. This in turn implies that the GUI elements must somehow be in contact with the data. So how is this achieved?

A central concept in achieving this goal is so-called **data binding**. Data binding covers the idea that data – both in simple and complex forms – can be “bound” to GUI con­trols, such that changes in the data are directly reflected in the GUI controls, and – very importantly – vice versa. If the actual data inside the application comes “out of sync” with the data presented to the user by the GUI, the results can be cata­stro­phic. Imagine a banking application where the balance of a bank account suddenly is diffe­rent from what the user sees on the screen. Or perhaps a military control system… This problem is almost as old as computer programming itself, and has been handled in various ways. Most of these ways are variants of data binding.

### Simple binding between GUI elements

The simplest form of data binding can be between two GUI controls defined on the same page. In the previous example with an **Image** and a **Slider** control, we can add data binding in the following way (image source omitted for brevity):

<Image x:Name="theImage" Source="…"

Height="{Binding ElementName=theSlider, Path=Value}"

Width="{Binding Height}" />

<Slider x:Name="theSlider" Maximum="800" Minimum="100" Value="400"

Width="400" Height="400"/>

How should this be read? The two interesting entries are the **“{Binding …}”** entries. The first entry should be read as *“bind the* ***Height*** *of the* ***Image*** *control to the GUI element named* ***theSlider*** *(which is a* ***Slider****), specifically to the* ***Value*** *property”*. With this binding, the height of the image will actually change if we slide the marker in the **Slider** control back and forth. The second entry reads *“Bind the* ***Width*** *property of the* ***Image*** *to the* ***Height*** *property of the (same)* ***Image****”*. This just ensures than the width and height of the image are both changed according to the value of the slider.

Data bindings like these are fairly simple, since they only involve the GUI controls them­­selves. Things get a bit more complex when GUI elements must be bound to data that is not part of the GUI itself, but rather part of the data “model” contained in the application; that is, the model of whatever domain the application concerns.

### Data binding between GUI elements and model objects

We mentioned in the start of this chapter, that one of the fundamental ideas in this approach to GUI development is the division of specification of appearance and beha­vior. That is, we specify the visual appearance of a GUI element in XAML, and the beha­­vior (including interaction with domain model objects) in C#. However, since the XAML code is transformed into C# and compiled along with the “native” C# code, the GUI controls will effectively be nothing more than C# objects, living in the context defined by the class definition they are part of.

A consequence of this fact is that it is fairly straightforward to bind a GUI control to a non-GUI property of the class in which it is defined. Still, this is not the recommended approach. We will in general aim to keep the code-behind files associated with XAML files as small as possible – preferably empty – since there are several advantages of keeping the binding specifi­cations in the XAML code. One such advantage is **portabili­ty**. XAML code can be used in other contexts than C# programming, but any logic placed in the code-behind files cannot be ported as easily.

The preferred approach is therefore to specifiy the data binding in the XAML files alone. Fortunately, this is also relatively easy to do. A key concept related to this is the **data context**. We said before that GUI controls are just C# objects living inside a class definition. This also means that it is possible to refer to objects of other classes, e.g. domain model classes. In XAML, this is done by specifying a **data context** for a GUI control. Data contexts are “hierarchical”, in the sense that if you e.g. set a data con­text for a **Page** control, all GUI controls defined within that **Page** will also be set to use that data context (unless they themselves explicitly set a different data context). A common approach is therefore to set the data context once, for the “outermost” control in a window, typically a **Page** control.

How does this look in practice? Suppose we define a simple domain class called **Car**, that for now only has one very simple property called **Brand**:

class Car

{

public string Brand { get { return "Toyota"; } }

public Car() { }

}

The class is just defined in the same C# project as the XAML code is defined in, and there is for now nothing special about it. In the XAML code, we add a simple **TextBox** control as well:

<TextBox Text="(not set)"/>

The **TextBox** control is defined inside a **StackPanel**, which again is defined inside a **Page**. We now set the data context property for the **Page** control (this must be done within the **Page** start- and end-tag):

<Page.DataContext>

<local:Car/>

</Page.DataContext>

The validity of this code rests on the fact that a namespace declaration for **local** was added to the XAML code at creation:

xmlns:local="using:GUIExample"

The **GUIExample** term is simply the name of the C# project we are working with here; in your own project, it will be substituted with the name of your own project. With this in place, we can now change the **TextBox** declaration to:

<TextBox Text="{Binding Brand}"/>

This should be read as “the value of the **Text** property in the **TextBox** control is now bound to the value of the **Brand** property of the **Car** object.” With the data con­text in place, this binding is not more complex w.r.t. syntax than binding to a value from another GUI control. Running the application now will indeed update the text in the **TextBox** control as desired:



There are however a couple of issues to consider. First, it was stated above that the **Text** property is bound to the **Brand** property on the **Car** object…. But which **Car** object is that? With a data context specification like this, the application will create a new **Car** object whenever the **Page** object containing the data context specification is created, e.g. when the user navigates to that page in the application. In this simple example, the behavior is always the same, since the **Brand** property always returns the same value (**Toyota**). In a more realistic setup, it will require a more elaborate appro­ach to cause this behavior to create the desired binding. We return to this in the chapter on the MVVM Architecture.

Second, we have now seen that the text in the **TextBox** control is actually updated with the text returned by the **Brand** property on the **Car** object. So, the binding going from **Car/Brand** to **TextBox/Text** (we use this informal **Class/Property** notation as a short way of saying “the **Property** property on the object of type **Class**”) seems to work. But what about the other way around? It would be natural to expect that we could type something into the text box, and expect that **Car/Brand** is updated to this new value. Achieving that does however require a few additions to the code.

First of all, the **Car/Brand** property must be changed, since it doesn’t have a **set**-part yet. The **Car** class then becomes a bit more realistic:

class Car

{

private string \_brand;

public string Brand

{

get { return \_brand; }

set { \_brand = value; }

}

public Car()

{

\_brand = "Toyota";

}

}

Next issue is to have a way of seeing if the value is actually changed in the **Car** object itself, if we type something into the text box. In order to do this, we create a simple **TextBlock** control, and bind it to the same property:

<TextBox Text="{Binding Brand}"/>

<TextBox Text=""/>

<TextBlock Text="{Binding Brand}"/>

An empty text box has also been added here; this is just for being able to leave the first **TextBox** control when having completed the typing of the new value. The GUI will as such not react to the typed value until the **TextBox** control “loses focus”, i.e. the screen cursor has been moved to a different control. Since you cannot move the cursor to a **TextBlock** control, an extra **TextBox** has been added to enable this. It has no other function.

If we now run the application, and type a new value into the text box, we will not see the desired behavior. This is partly due to an incomplete data binding specification. Data binding can be specified to have a certain “mode”:

|  |  |
| --- | --- |
| **One-time** | Data is only retrieved from the source once. Subsequent changes are not reflected. |
| **One-way** | Data flows from the source to the target, so changes in the source value are reflected in the target, but not vice versa. |
| **Two-way** | Data flows from the source to the target, and vice versa. Changes in both source and target values are reflected in the counterpart as well. |

We obviously want a two-way binding for the **TextBox** control, but that is not the default value (default is one-way). We must therefore explicitly set the data binding mode for the **TextBox** control:

<TextBox Text="{Binding Brand, Mode=TwoWay}"/>

All seems to be in place now, but if the application is run now, we still don’t see the expected behavior… If you place a breakpoint in the **set**-part of the **Brand** property, you will indeed find that the **set**-part is called, and the instance field is updated as it should. What is missing?

The last part of the puzzle is to enable the source object (here the **Car** object) to signal to the outside world that a property has been changed. Even though the **TextBlock** con­trol has set up a binding to the **Brand** property, it is not automatically notifed about changes in the property value! This does happen automatically when you bind to other GUI controls directly in the XAML, but not for “pure” C# classes.

The mechanism commonly used for enabling such notifications is the **INotifyProperty­Changed** interface, which is part of the .NET class library. If you want a class to be able to signal changes to its properties to those interested in knowing about such changes, the class should implement this inter­face. Doing this for the **Car** class adds some extra code to the class:

class Car : INotifyPropertyChanged

{

private string \_brand;

public string Brand

{

get { return \_brand; }

set { \_brand = value; }

}

public Car()

{

\_brand = "Toyota";

}

public event PropertyChangedEventHandler PropertyChanged;

protected virtual void OnPropertyChanged

([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this,

new PropertyChangedEventArgs(propertyName));

}

}

The highlighted code is the code added for implementing the interface. This code is a bit complex, and contains a couple of language elements we have not seen before, for instance an **event**. The good news is that this code is completely generic, and you do not need to worry about it as such, let alone change it.

The only code we need to explicitly add is one line of code to the **set**-part of the **Brand** property, like this:

public string Brand

{

get { return \_brand; }

set

{

\_brand = value;

OnPropertyChanged();

}

}

That’s it! If we now run the application, we will indeed see that as soon as we leave the **TextBox** control (by hitting Tab or clicking in the empty text box), the text block below is updated with the new value. The **Car/Brand** property now “broadcasts” changes to its value to those interested, i.e. those elements that have set up a binding to that specific property.

We can now recap all the steps needed in order to achieve a working two-way binding between a GUI control and an ordinary C# class:

1. Make sure that an appropriate **namespace** definition is part of your XAML page. It will be part of the top lines in the XAML file, and typically look like:

xmlns:local="using:YourProjectName"

1. Set the **data context** at an appropriate level in the XAML page, for instance for the **Page** element itself. The code will typically look like:

<Page.DataContext>

<local:YourDomainClass/>

</Page.DataContext>

1. Create a **data binding** for the relevant control. The types of the control element and the source property should be compatible, i.e. a **Text** element will usually bind to a property of type **string**, and so on. A typical binding will look like:

<TextBox Text="{Binding YourProperty}"/>

1. Remember to set the **binding mode** as well. If a GUI element and an object property must be in sync at all times, the binding type should be two-way:

<TextBox Text="{Binding YourProperty, Mode=TwoWay}"/>

1. Your domain class must implement the **INotifyPropertyChanged** inteface in order to be able to notify others about changes in its property values:

class YourDomainClass : INotifyPropertyChanged

{

// Rest of class definition

}

1. Finally, the **set**-part of each relevant property must call **OnPropertyChanged** when the value of the property has been updated:

public string Brand

{

get { return \_brand; }

set { \_brand = value;

OnPropertyChanged();

}

}

## Collection Views and Data Binding

Creating bindings to simple elements like e.g. the **Text** element in a **TextBox** control is thus fairly straightforward. These bindings are usually relevant in situations where the data context is suitably represented by a single domain object. If we need to manage and interact with a **collection** of domain objects, we need to use other control types suitable for such collections. Such control types do of course exist, but require a bit more elaboration in order to specify appearance and data bindings properly.

### The ListView control – getting started

The **ListView** control is one of those controls that can can handle a collection of items. In its simplest form, it can be used completely without data binding:

<ListView>

<ListViewItem>Toyota</ListViewItem>

<ListViewItem>BMW</ListViewItem>

<ListViewItem>Opel</ListViewItem>

<ListViewItem>Volvo</ListViewItem>

</ListView>

This form is of course only relevant when the items in the list are “constant”, i.e. will not change during the lifetime of the application. This is rarely the situation in prac­tice, so we need to figure out how to create data bindings for a **ListView**.

Compared with the recap for single-element bindings above, the first and second steps regarding namespace and data context are still needed, in order to make the classes in the project visible to the control. The third step (data binding) will look a bit different. First, we update the **Car** class with a new property **BrandNames**:

public List<string> BrandNames

{

get { return new List<string>() {"Toyota","BMW","Opel","Volvo"}; }

}

This list is of course also “constant”, but could in principle have been populated by reading the values from a file or database; the data binding does not care about where the values originally come from. The binding itself looks like:

<ListView ItemsSource="{Binding BrandNames}" />

Not much more complicated than binding to a single element, except for the use of the **ListView** property **ItemsSource**. As the name indicates, we can bind this property to a collection of values, here more specifically a list of **string** values.

This is much more useful than the first example, since the population of the list can be done in any way we wish, as long as the list is ready-to-use when the **ListView** asks for it. A natural next step would be to enable the user to add new elements to the list. This will require some additions to the code. First, we add a **TextBox** control to the GUI, so the user can type in a new brand name (this is just the **TextBox** control from the previous example):

<TextBox Text="{Binding Brand, Mode=TwoWay}"/>

<ListView ItemsSource="{Binding BrandNames}"/>

Next, we make the **Car** class a bit more general, by adding an instance field to hold the list of brand names:

private string \_brand;

private List<string> \_brandNames;

public string Brand

{

get { return \_brand; }

set

{

\_brand = value;

OnPropertyChanged();

}

}

public List<string> BrandNames

{

get { return \_brandNames; }

}

public Car()

{

\_brand = "Toyota";

\_brandNames = new List<string>() {"Toyota", "BMW", "Opel", "Volvo"};

}

Is that enough? No, since we do not at any point add any new elements into the list of brand names. However, a small addition to the **set**-part of the **Brand** property can fix this (please note that this is not a very pretty nor user-friendly solution; it only serves to explore what it takes to get the data binding up-and-running…) :

public string Brand

{

get { return \_brand; }

set

{

\_brand = value;

\_brandNames.Add(\_brand);

OnPropertyChanged();

}

}

If you run this code, you will find that the situation is similar to what we saw at some stage in the previous example: it seems like the list should be updated, and running the code through the debugger reveals that the list does indeed get updated, but this is not reflected in the GUI… The underlying problem is actually also very similar: the list does not notify the outside world when it gets updated. Here the solution is less straightforward, since it does not make much sense to implement a **set**-part of the **BrandNames** property. It seems that the **Add** method on the **List** class ought to some­how call the **OnPropertyChanged** method.

The solution is to substitute the use of the **List** class with another class from the .NET class library: the class **Observable­Collection**. This class does what we want; when an element is added or removed from the collection, a notification is broadcasted. Once the substitution is made in the code, the list in the GUI does get updated whenever we leave the text box.

The **Observable­Collection** class is quite convenient, but it does have one major draw­back. If you add or remove an element, notifications are indeed triggered. However, if you just update a value in an existing element, no notifications are triggered! If you need this sort of functionality, a more sophisticated solution must be used.

### The ListView control – displaying items

In the above example, we did not really worry about how each item in the collection (i.e. the **ObservableCollection** object) was displayed. Since each item has the type **string**, it is easy for the **ListView** control to display the items. What if we wish to dis­play a collec­tion of **Car** objects instead? First, we need to create a property in the code where such a collection can be bound to. In order to keep things simple rather than pretty, we just add an additional property to the **Car** class:

public ObservableCollection<Car> Cars

{

get

{

return new ObservableCollection<Car>()

{

new Car() { \_brand = "Volvo" },

new Car() { \_brand = "BMW" },

new Car() { \_brand = "Opel" },

new Car() { \_brand = "Toyota" }

};

}

}

The data binding for the **ListView** control also needs to be updated:

<ListView ItemsSource="{Binding Cars}"/>

Running the code with these updates does produce a **ListView** control containing four items, but they are not displayed in a useful way:



What has happened is that the **ListView** has called the **ToString** method on each ob­ject, and displays the result of that call. Since we have not overrided **ToString**, this is the result. If we override **ToString** in the **Car** class, we get a better result:

public override string ToString()

{

return "This is a " + \_brand;

}



If the data you wish to display in a **ListView** control can be appropriately displayed by a simple string, you probably don’t need something much more advanced than this. It is however possible to specify more elaborate ways to display elements in a **ListView**, by defining a so-called **data template**.

### Defining a data template

The data template is a way of specifying the visual appearance of a single item in the **ListView**. Formally, the data template is set as the value of the **ItemTemplate** proper­ty of a **ListView** control, like this:

<ListView ItemsSource="{Binding Cars}">

<ListView.ItemTemplate>

<DataTemplate>

(your item layout specification)

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

Inside the **DataTemplate** tag, you specify the layout of an item. You can use all of the available control types without restriction. If you e.g. want to display each **Car** item as a small picture followed by the brand name, it could look like this:

<DataTemplate>

<StackPanel Orientation="Horizontal">

<Image Source="{Binding ImageSource}" Height="80" Width="80"/>

<TextBlock Text="{Binding Brand}"/>

</StackPanel>

</DataTemplate>

Assuming that the **ImageSource** property has been added to the **Car** class, and that some suitable images have been made available, the **ListView** will now look like:



Since we have chosen to jam everything into the **Car** class, the data template does not need to be told explicitly that the bindings refer to properties on the **Car** class. If the class structure is more elaborate, it is possible to state explicitly what the type of the items in the **ListView** is:

<DataTemplate x:DataType="local:Car">

<StackPanel Orientation="Horizontal">

<Image Source="{Binding ImageSource}" Height="80" Width="80"/>

<TextBlock Text="{Binding Brand}"/>

</StackPanel>

</DataTemplate>

The **x:** syntax just implies that we are referring to a specific part of the namespaces included at the top of the XAML file.

### The ListView control – binding to SelectedItem

A final (in this text) useful feature of the **ListView** control is the ability to bind to the currently selected item in the control. The **SelectedItem** property on the control can be targeted in a data binding:

<ListView ItemsSource="{Binding Cars}"

SelectedItem="{Binding SelectedCar}">

<ListView.ItemTemplate>

<DataTemplate>

(your item layout specification)

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

A **ListView** control is often used in a so-called **Master/Details view**. In this type of view, the user can select an item in the **ListView** control, and see further details of the selected item in a different part of the view. If we imagine that the **Car** class contains some additional properties, we could add some additional controls to the page that until now only contains the **ListView** control:

<StackPanel Orientation="Horizontal">

<ListView ItemsSource="{Binding Cars}"

SelectedItem="{Binding SelectedCar, Mode=TwoWay}">

<ListView.ItemTemplate>

<DataTemplate x:DataType="local:Car">

<StackPanel Orientation="Horizontal">

<Image Source="{Binding ImageSource}"

Height="50" Width="50"/>

<TextBlock Text="{Binding Brand}"/>

</StackPanel>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

<StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text = "Brand"/>

<TextBlock Text = "{Binding SelectedCar.Brand}"/>

</StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text = "Color "/>

<TextBlock Text = "{Binding SelectedCar.Color}"/>

</StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text = "Seats "/>

<TextBlock Text = "{Binding SelectedCar.Seats}"/>

</StackPanel>

<StackPanel Orientation="Horizontal">

<TextBlock Text = "Price "/>

<TextBlock Text = "{Binding SelectedCar.Price}"/>

</StackPanel>

</StackPanel>

</StackPanel>

With this setup, we get something that is at least a rough draft of a Master/Details view for a **Car** domain object:



There is obviously some work to do with regards to the purely visual presentation of the data, but the essential wiring with respect to data binding is largely in place now.

### The GridView

Once we know how to deal with a **ListView** control, there is actually not that much more to say about a **GridView** control. The considerations that applied to **ListView** are also valid here, so it is mostly a matter of the visual presentation. With a little bit of adjustment, a **GridView**-version of the previous example could look like:

<GridView ItemsSource="{Binding Cars}">

<GridView.ItemTemplate>

<DataTemplate>

<StackPanel HorizontalAlignment="Center">

<Image Source="{Binding ImageSource}"

Height="200" Width="200"/>

<TextBlock FontSize="48" Text="{Binding Brand}"/>

</StackPanel>

</DataTemplate>

</GridView.ItemTemplate>

</GridView>

The visual result is:



Again, a bit of visual polishing would probably be in order, but the fundamental setup is almost identical to the setup we developed for the **ListView** control example.

## Commands

So far, we have mostly concentrated on how to bind GUI controls to existing data, in the form of domain model objects. We did have a single example of how to add an element to a list, but it was done in a haphazard way inside the **set**-part of a property. That is definitely not a recommendable approach for a real system. We will therefore now look closer at how to perform modifications to a domain data model, i.e. how to add, delete and update domain objects through a GUI.

### Deleting a domain object

As such, the actions we wish to perform are not particularly complicated. Suppose we now have a fuller **Car** domain class, like:

public class Car : INotifyPropertyChanged

{

private string \_licensePlate;

private string \_brand;

private string \_model;

private string \_imageSource;

private string \_color;

private int \_seats;

private int \_price;

// ...and so on (Properties, etc.)

}

In this class, we assume that the **LicensePlate** property (associated with the **\_license­Plate** instance field) defines a unique key for **Car** objects, i.e. no two **Car** objects will have the same value for **LicensePlate**. Also, we have created a class **CarCatalog**, which holds a collection of **Car** objects (in an **ObservableCollection**), like:

public class CarCatalog : INotifyPropertyChanged

{

private ObservableCollection<Car> \_cars;

private Car \_selectedCar;

public ObservableCollection<Car> Cars

{

get { return \_cars; }

}

// ...and so on

}

How would we go about implementing functionality for deleting a **Car** object from our model? Since we assume that the collection of car objects is indeed that entity which represents the cars we have in our model, deletion of a specific car will amout to dele­ting the corresponding **Car** object from the **CarCatalog** object. We thus assume that there will always be exactly one **CarCatalog** object present, and we only need to dele­te the **Car** object from that object. Since we have defined that the **LicensePlate** field uniquely maps to a **Car** object, we can add a **Delete** method to our **CarCatalog** class:

public bool Delete(string licensePlate)

{

// Implementation of deletion functionality

}

We imagine the method will return **true** if a **Car** object with the given value for **Licen­se­Plate** is indeed found and deleted, otherwise **false**. The implementation details are as such not interesting.

With this functionality available on the **CarCatalog** class, the remaining issues are:

* How do we select the value for **LicensePlate** to be a used in a call of **Delete**?
* How is the deletion functionality activated from the GUI?

If we assume that we still have some sort of Master/Details view set up for cars, a natural way of selecting the car targeted for deletion could be to select a car in the Master view (i.e. a list view), and then have a button labeled **Delete** available for the user to tap. That is, when the **Delete** button is tapped, the **Car** object corresponding to the selected item in the list view should be deleted:



We have already seen that it is fairly easy to bind the selection in the list view to e.g. a **SelectedCar** property on the **CarCatalog**, so once we tap on the **Delete** button, we can easily retrieve a value for **LicensePlate** to use in the call to **Delete**:

public bool DoDelete()

{

return (SelectedCar != null && Delete(SelectedCar.LicensePlate));

}

The last remaining issue is how to invoke this **DoDelete** method, i.e. how do we bind the action of tap­ping the **Delete** button to the invocation of the method? The prefer­red way to achieve this is to encapsulate the deletion code into a so-called **Command** object.

### The ICommand interface

A **Command** object is in this context an object that implements the **ICommand** inter­face. This interface is part of the .NET class library, and contains two methods and an event. A simple class **Command** that inherits from **ICommand** could look like:

class Command : ICommand

{

public bool CanExecute(object parameter)

{

// Should return whether or not the command

// can currently be executed.

}

public void Execute(object parameter)

{

// The code to execute

}

public event EventHandler CanExecuteChanged;

}

This is in itself not very useful, but illustrates the elements contained in the interface. The more important point is that we can create properties on a domain class like **CarCatalog**, that have the type **ICommand** and return a **Command** object:

public ICommand DeletionCommand

{

get { return new Command(); }

}

We will implement a more proper deletion command in a moment, but the point is that we can now bind the **DeletionCommand** property to a **Delete** button, like this:

<Button Content="Delete" Command="{Binding DeletionCommand}"/>

A **Button** control has a **Command** property, which can be bound to a corresponding property on a non-GUI object, if the property is of type **ICommand**. The consequence of this binding is that whenever the user taps the **Delete** button, the application will retrieve the object returned by the bound-to property, and call its **Execute** method. Furthermore, the GUI control will only be enabled if the **CanExecute** method returns **true**. With this in mind, we can now implement a proper deletion command:

class DeleteCommand : ICommand

{

private CarCatalog \_carCatalog;

public DeleteCommand(CarCatalog carCatalog)

{

\_carCatalog = carCatalog;

}

public bool CanExecute(object parameter)

{

return true; // For now...

}

public void Execute(object parameter)

{

\_carCatalog.DoDelete();

}

public event EventHandler CanExecuteChanged;

}

Note that we will return to the implementation of **CanExecute** in a moment. The **DeleteCommand** class takes a reference to the **CarCatalog** object in its constructor, such that it can call the **Do­Delete** method on the object. With this in place, we can update the implementation of **DeletionCommand**:

public ICommand DeletionCommand

{

get { return new DeleteCommand(this); }

}

Running the application now will demonstrate that we can indeed select a car in the list view, click on the **Delete** button, and see the car being deleted from the list view. A small problem remains, however; the **Delete** button is always enabled, even when no car is selected. The situation is handled in **DoDelete** (the not-**null** check), but it would be better if the button was only enabled when a car is selected. The method **CanExecute** is the key to this. A straightforward – and even correct – implementation of the method is:

public bool CanExecute(object parameter)

{

return \_carCatalog.SelectedCar != null;

}

However, if we run the application, the **Delete** button is now always disabled… The reason for this is somewhat subtle. When the application is launched, the view containing the list view is created. This also causes a **DeleteCommand** object to be created – due to the binding to the **Button/Command** binding – in a state where no car has been selected. The method thus returns **false**, and the object is never asked to “refresh” this value. This is what the **CanExecuteChanged** event is used for. If the state of the **CarCatalog** changes in a way that makes it relevant to refresh the state of the button – i.e. call **CanExecute** again – the event must be “raised”. This happens exactly when the selection in the list view changes. Handling this requires a couple of changes in the code.

First, we add a new method **RaiseCanExecuteChanged** to the **DeleteCommand** class:

public void RaiseCanExecuteChanged()

{

CanExecuteChanged?.Invoke(this, EventArgs.Empty);

}

We can then call this method when the selection in the list view changes (see later). We also change **CarCollection** a bit. First, we add an instance field for holding a **Delete­Command** object, and create that object in the constructor:

public class CarCatalog : INotifyPropertyChanged

{

private DeleteCommand \_deleteCommand;

// Rest of instance fields…

public CarCatalog()

{

\_deleteCommand = new DeleteCommand(this);

// Rest of constructor…

}

// Rest of class…

}

We then return that object in the **DeletionCommand** property:

public ICommand DeletionCommand

{

get { return \_deleteCommand; }

}

Finally, we update the **set**-part of the **SelectedCar** property with the call to the new **DeleteCommand** method:

public Car SelectedCar

{

get { return \_selectedCar; }

set

{

\_selectedCar = value;

OnPropertyChanged();

\_deleteCommand.RaiseCanExecuteChanged();

}

}

A test of the application shows that the state of the **Delete** button now reflects whe­ther or not a car is selected in the list view. When a car is seleted and deleted by tapping **Delete**, the car is indeed deleted from the list view and the **Delete** button becomes disabled, until a new car is selected.

The addition of **RaiseCanExecuteChanged** to **DeleteCommand** is very generic, and it is therefore quite common to create a command base class that implements **ICommand** and contains the **RaiseCanExecuteChanged** method. One such example is the **Relay-Command** class, that can be found in various incarnations online. We list one example of a **RelayCommand** implementation below:

public class RelayCommand : ICommand

{

private readonly Action \_execute;

private readonly Func<bool> \_canExecute;

public event EventHandler CanExecuteChanged;

public RelayCommand(Action execute, Func<bool> canExecute)

{

\_execute = execute;

\_canExecute = canExecute;

}

public bool CanExecute(object parameter)

{

return ((\_canExecute == null) || \_canExecute());

}

public void Execute(object parameter)

{

\_execute();

}

public void RaiseCanExecuteChanged()

{

CanExecuteChanged?.Invoke(this, EventArgs.Empty);

}

}

This implementation uses some constructs that have not been mentioned in these notes. The types **Action** and **Func<bool>** are in fact **function types**. The two types should be read as:

* **Action**: Any function that does not take any parameters and returns **void**
* **Func<bool>**: Any function that does not take any parameters and returns **bool**

If we had used **RelayCommand** in the example, we could have created a command object for deletion in this very compact form:

\_deleteCommand = new RelayCommand( () => { DoDelete(); },

() => SelectedCar != null);

If you have never seen this style of wrapping a code segment up as a parameter, it pro­bably looks quite strange. You can achieve the same effect by creating two small methods in **CarCollection**, and use those methods as parameters instead:

public void DoDeleteRelay() { DoDelete(); }

public bool CarIsSelected() { return SelectedCar != null; }

// ...

\_deleteCommand = new RelayCommand(DoDeleteRelay, CarIsSelected);

No style is better than the other; the compact form avoids creating new methods just for using them as parameters, while the latter form is probably easier to understand. Exactly how you choose to organise your command classes in a real project is of course up to you.

You may at this point wonder, if this is not an overly complex way of invoking a func­tion­ality that only amounts to a couple of lines of code. There’s not really any way around it. Properties on GUI controls can bind to properties – not methods – on other objects, so the functionality-wrapped-into-command-objects style is hard to avoid. Furthermore, the use of command object does actually provide several advantages on its own. Since a command object encapsulates a piece of functionality without know­ledge about the context in which it is used, it is pretty easy to set up tests for a com­mand in an artificial test environment.

Another advantage of command objects is that they can be handled like any other kind of object; we can e.g. set a command object into a queue of commands waiting for execution, if we are in a situation where commands cannot always be executed immediately. We could also imagine that facilities for e.g. executing commands asyn­chronously (meaning that we don’t wait for the execution of a command to complete before proceeding with other parts of the code) can be implemented in a general way, if all relevant functionality is wrapped into command objects. We could even imagine saving a set of waiting commands to a file, and later on read them again for execution, when the needed resources are available.

Yet another – maybe not so obvious – advantage comes from the fact that we can bind as many GUI controls to a command property as we wish. Often the GUI will allow you to invoke a functionality in several ways; we could imagine that a deletion functionality could be invoked by tapping a button, but also by choosing a menu item. The availabi­lity of such GUI elements should of course be consistent, such that if a Delete button is disabled, the corresponding menu item also becomes disabled. This feature comes for free if the relevant GUI controls all bind to the same command object.

## The MVVM application architecture

The previous chapter has covered a fair bit of ground, going from a simple data bind­ing between two GUI elements, to a collection-based data binding to a Master/Details view, using command objects for invoking functionality. Still, our approach is not very robust with regards to separation of visual presen­tation and domain logic. We have not really stated anything about the roles of the **Car** and **CarCatalog** classes in the example, but it seems a fair assumption that the **Car** class is a domain class, and that other domain-oriented classes will use **Car** objects to perform some sort of business logic. The **CarCatalog** class could be perceived as such a domain class as well. In other words: the **Car** and **CarCatalog** classes should not be involved with how data relating to cars are presen­ted to the user.

In our example, we had fairly modest requirements with regards to visual presenta­tion of cars, so we could get by with using the simple properties available on the **Car** class. We will now pursue an architecture for our application that clearly separates the domain classes from the presentation-oriented part of the application. One such architecture is the **MVVM** architecture.

### MVVM architecture – fundamental features

The acronym **MVVM** is short for **Model-View-ViewModel**, and is as stated above found­ed on the principle of clear separation between presentation and domain logic. We did have some degree of separation in the previous example, in the sense that the presentation was defined in one class, while the domain logic – tiny as it was – was confined to the **Car** class. This can be described as a Model-View architecture:

Model

View

In this setup, the View classes will have explicit knowledge about the Model classes, and will thus be vunerable to changes in the Model classes. Likewise, the Model class­es may have to contain properties that are only present for making life easier for the View classes, and may even need to be up­dated if the view-oriented requirements for the application change.

The remedy for this situation is to insert a layer between the View and the Model, aptly named the ViewModel. This layer is meant to “mediate” between the two original layers, such that each can change independently of the other:

View-Model

View

Model

### MVVM – single domain object

How would this idea manifest itself in the previous example? Let’s unwind all the way back to the example where we had a simple binding between a **Car** (domain) class and a simple view in place. At that point, the **Car** class looked like this:

class Car : INotifyPropertyChanged

{

private string \_brand;

public string Brand

{

get { return \_brand; }

set

{

\_brand = value;

OnPropertyChanged();

}

}

public Car()

{

\_brand = "Toyota";

}

// … and code for OnPropertyChanged

}

The highlighted code is the code we added in order to make the binding to the view work properly. Essentially, this implies implementing the **INotifyPropertyChanged** interface, plus calling **OnPropertyChanged** in the **set**-part of properties. Now that we are used to this sort of code, it may look quite innocent. Still, this code is only added in order to service views wishing to present car data. This means that the **Car** class now has (at least) two main responsibilities:

* Acting as a domain class
* Acting as a point-of-contact for views wishing to present car data

This is not an ideal situation, since classes should in general only have one main re­spon­sibility. Following the MVVM architecture outlined above, we should therefore insert a new class between the **Car** class and the views. We could name this class **CarViewModel**.

What resposibilities should this new class have? It should act as the point-of-contact for views interested in presenting car data, and service the needs of those views. Also, it should be in contact with a **Car** object, since it will need to access car data in order to relay these data – perhaps in a modified form – to the views. More specifically, we can then conclude that the class **CarViewModel** should

* Implement **INotifyPropertyChanged**, in order to provide views with up-to-date car data (we can imagine that more than one view binds to **CarViewModel**)
* Refer to a **Car** object, and keep this object up-to-date in accordance with actions from the views (via data bindings and commands).

With this, we can begin to outline an example of a **CarViewModel** class:

public class CarViewModel : INotifyPropertyChanged

{

private Car \_domainObject;

public CarViewModel()

{

\_domainObject = new Car();

}

public string Brand

{

get { return \_domainObject.Brand; }

set

{

\_domainObject.Brand = value;

OnPropertyChanged();

}

}

public string BrandText

{

get { return "The car is a " + \_domainObject.Brand; }

}

// ... plus OnPropertyChanged code

}

As we will see later, this class does have some issues, but the general concepts are in place. We refer to a domain object (the **Car** object), and manage requests regarding data for this object. That is, we return relevant data from the object when requested, either “unprocessed” as in the **get**-part of the **Brand** property, or “processed” as in the **get**-part of the **BrandText** property, where a bit of additional text is prepended.

The term “relevant” here means that we should not just make up properties that we imagine could be of use; the properties added to the **CarViewModel** class should be driven by requirements for the views presenting car data. These requirements have ideally nothing to do with the domain-driven requirements for a **Car** class, and they should therefore be completely separated. A **Car** class should not have to change because a view-oriented requirement changes, and a view should not have to change because a detail in the **Car** implementation changes. However, the **CarViewModel** is indeed allowed to change if either the views or the domain class changes, since this is exactly its main responsibility; to mediate between a domain class and the views presenting it.

The introduction of the **CarViewModel** class enables us to do two things:

* We can “clean up” the **Car** class, so it no longer needs to contain any elements relating to presentation
* We can create bindings using the **CarViewModel** class instead of the **Car** class

The cleaned-up **Car** class now looks like a “pure” domain class:

public class Car

{

private string \_brand;

public string Brand

{

get { return \_brand; }

set { \_brand = value; }

}

public Car()

{

\_brand = "Toyota";

}

}

Bindings to **CarViewModel** are now done like (the **Button** has no functionality yet):

<Page.DataContext>

<local:CarViewModel/>

</Page.DataContext>

<StackPanel>

<TextBox Text="{Binding Brand, Mode=TwoWay}"/>

<TextBlock Text="{Binding Brand}"/>

<TextBlock Text="{Binding BrandText}"/>

<Button Content="OK"/>

</StackPanel>

If we run this example, we will indeed see that the first **TextBlock** control (bound to the **Brand** property in the **CarViewModel** class) is updated whenever we type a new value into the **TextBox** control. Placing a breakpoint in the **set**-part of the **Brand** property on the **Car** class will also reveal that the value does indeed get set in the enclosed **Car** object. All is thus in order, and we have obtained the desired separation while maintaining the functionality. Except for one problem…

Note that we added an additional **TextBlock** control to the example, binding to the **BrandText** property on the **CarViewModel** class. This text did not get updated when we typed in a new brand value. Why not…? In order for that text to get updated, someone must call **OnPropertyChanged** with **BrandText** as parameter! So far, we have always called **OnPropertyChanged** without any parameter. Let’s have a second look on the implementation of **OnPropertyChanged**:

protected virtual void OnPropertyChanged

([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this,

new PropertyChangedEventArgs(propertyName));

}

We said earlier that we need not understand the details of this implementation, and this is still true. However, we note that the implementation is such that if no para­meter is given, the implementation sees – through some rather advanced C# magic –the name of the calling property as the actual parameter. The consequence is that if we call **OnPropertyChanged** from a property named **Brand**, the implementation sees that name as the parameter, and does what it should; it notifes all properties bound to **Brand** that the value of the property has changed.

With regards to the **BrandText** property, there are two problems:

* The property doesn’t have a **set**-part.
* Nobody calls **OnPropertyChanged** for this property

It is pretty obvious that **BrandText** cannot have a **set**-part, since its value is a so-called **aggregated value**, where the value does not have a direct counterpart in the domain object. **BrandText** is of course a simple case, where the value is composed of a single domain object value plus some fixed text, but we could easily imagine e.g. a longer text pieced together from several values from the domain object. The main point is in any case valid: **BrandText** cannot have a **set**-part, so we cannot call **OnProperty­Changed** from there.

Instead, we must look for places in the code where something happens that might affect the value of **BrandText**. Again, this case it pretty obvious, since it is only in the **set**-part of the **Brand** property such a change happens. We must therefore add an extra call to **OnPropertyChanged** here:

public string Brand

{

get { return \_domainObject.Brand; }

set

{

\_domainObject.Brand = value;

OnPropertyChanged();

OnPropertyChanged(nameof(BrandText));

}

}

The parameter to **OnPropertyChanged** is of type **string**, so we could also have written **“BrandText”** directly. However, the **nameof(BrandText)** style – which returns the name of the property as a string – is more robust, since it will be affected at compile-time, if we decide to rename **BrandText** to something else. If we had just written a string directly, we would not see any errors before running the application.

With this addition, the application behaves as expected. For a simple setup like this, it is relatively easy to manage such dependencies, and add extra calls to **OnProperty­Changed** where needed. However, when you have a more complex setup with many depencies, you may need some sort of framework to manage the dependencies. You could e.g. use a map of property dependencies, as outlined below:

public class PropertyDependencyManager

{

private Dictionary<string, List<string>> \_dependencyMap;

public PropertyDependencyManager()

{

\_dependencyMap = new Dictionary<string, List<string>>();

}

public void AddDependency(string property, string dependentProperty)

{

// Add a new dependency

}

public List<string> GetDependencies(string property)

{

// Return properties depending on this property

}

}

With this in place, the **CarViewModel** class could be extended like this:

public CarViewModel()

{

\_domainObject = new Car();

\_dependencyManager = new PropertyDependencyManager();

\_dependencyManager.AddDependency(nameof(Brand), nameof(BrandText));

}

public string Brand

{

get { return \_domainObject.Brand; }

set

{

\_domainObject.Brand = value;

OnPropertyChanged();

foreach (var dependentProperty in

\_dependencyManager.GetDependencies(nameof(Brand)))

{

OnPropertyChanged(dependentProperty);

}

}

}

It will now be easy to manage addition of other properties based on aggregated values, since all you need to do is to:

* Add the dependencies to the dependency manager
* Add the **foreach**-loop to all **set**-parts of properties

### MVVM – collection of domain objects

The next natural step is to extend this principle to a setup where a collection of domain objects need to be handled. The guiding principle should again be that the domain classes should not contain presentation-oriented elements. Thererfore, a collection class **CarCollection** for **Car** objects will be pretty simple (the **Car** class has been extended with a few properties:

public class CarCollection

{

private List<Car> \_cars;

public List<Car> All

{

get { return \_cars; }

}

public CarCollection()

{

\_cars = new List<Car>();

\_cars.Add(new Car("AX 32 501", "BMW", "318i","Assets\\BMW.jpg"));

\_cars.Add(new Car("CP 73 001", "Volvo", "X40", "Assets\\Volvo.jpg"));

\_cars.Add(new Car("BK 55 734", "Opel", "Astra", "Assets\\Opel.jpg"));

\_cars.Add(new Car("AZ 60 922", "Toyota", "Auris", "Assets\\Toyota.jpg"));

}

}

Our goal is – just as in the View-Model setup – to work our way towards a working Master/Details view.

### An Item view model class

So, we again need to consider how car data is presented in a **ListView** control, i.e. the Master part of the view. We deliberately didn’t say “how a **Car** object is presented”, since the **ListView** will not bind to a collection of **Car** objects, but rather to a collection of view model objects relating to cars. We will therefore need to create such a view model class. However, the purpose of this particular view model class is not neces­sa­rily to give a detailed presentation of car data. The view model class is supposed to provide the presentation data needed for presenting car data in the Master view, and since the purpose of the Master view is to provide the user with enough data to make a well-defined selection in the view – such that further details can be presented in the Details view – we may only need to provide a small subset of the available data. This could e.g. be the license plate and a small image of the car. A view model class for this specific purpose – which we in general will categorise as an **Item** view model class – would therefore be quite simple:

public class CarItemViewModel

{

private Car \_domainObject;

public CarItemViewModel(Car domainObject)

{

\_domainObject = domainObject;

}

public string ImageSource

{

get { return \_domainObject.ImageSource; }

}

public string Description

{

get { return \_domainObject.Brand + " (" +

\_domainObject.LicensePlate + ")"; }

}

public Car DomainObject

{

get { return \_domainObject; }

}

}

Since you will not be able to change data through the elements in the Master view, we don’t need any **set**-parts of any properties here, and thus no management of dependencies. Note that we have also included a property that returns a reference to the enclosed domain object; this will come in handy later ☺.

This item view model class can then be used in a collection-oriented view model class, which will be the class behind the Master part of the view. We will cate­gorise such a class as a **Master** view model class.

### A Master view model class

A class in this category is therefore responsible for converting a collection of domain objects into a collection of item view model objects. These objects can then be bound to by a collection-oriented GUI control, e.g. a **ListView** control. In our example, we now create such a class named **CarMasterViewModel**:

public class CarMasterViewModel

{

private List<CarItemViewModel> \_itemViewModelCollection;

private CarCollection \_carCollection;

public List<CarItemViewModel> ItemViewModelCollection

{

get

{

\_itemViewModelCollection = new List <CarItemViewModel>();

foreach (var car in \_carCollection.All)

{

\_itemViewModelCollection.Add(new CarItemViewModel(car));

}

return \_itemViewModelCollection;

}

}

public CarMasterViewModel(CarCollection carCollection)

{

\_carCollection = carCollection;

}

}

The class is handed a reference to a collection of **Car** objects at creation time, and will when requested return a corresponding collection of **CarItemViewModel** objects. In this setup, the **CarMasterViewModel** object will not hold any state as such, since the collection of view model objects is constructed whenever the **get**-part of the property is accessed.

These objects can then be bound to through the **ItemViewModelCollection** property:

<Page.DataContext>

<local:CarMasterViewModel/>

</Page.DataContext>

<ListView ItemsSource="{Binding ItemViewModelCollection}">

<ListView.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal">

<Image Source="{Binding ImageSource}"/>

<TextBlock Text="{Binding Description}"/>

</StackPanel>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

We now have the Master part of the view in place.

### A Details view model class

For the Details part, we create yet another view model class for this specific purpose. This class will need to be able to present more car data, plus have the ability to enable editing of parts of the data. Exactly how and what data to present – and what data the user should be allowed to edit – will of course depend of the specific situation. We categorise such a class as a **Details** view model class. The main features of such a class will be:

* It refers to a single domain object
* It exposes properties that the Details view can bind to
* Some properties (simple properties) may directly correspond to properties on the domain object
* Other properties (aggregated properties) may be based on a number of domain object properties

For the car example, a Details view model class could look like (property dependency code omitted for brevity):

public class CarDetailsViewModel : INotifyPropertyChanged

{

private Car \_domainObject;

public CarDetailsViewModel(Car domainObject)

{

\_domainObject = domainObject;

}

public string LicensePlate { get { return \_domainObject.LicensePlate; } }

public string Brand { get { return \_domainObject.Brand; } }

public string Model { get { return \_domainObject.Model; } }

public string Heading

{

get { return "A " + Brand + " " + Model +

", priced at " + Price + " kr."; }

}

public int Price

{

get { return \_domainObject.Price; }

set { \_domainObject.Price = value;

OnPropertyChanged(); }

}

}

Here we have assumed that it only makes sense to change the **Price** property of the car, once it has been created. It is therefore only the **Price** property that has a **set**-part. The **Heading** property is an aggregated property, and it should be “refreshed” (by a call to **OnPropertyChanged**) whenever the **Price** property changes.

We can now use this class for establishing bindings in the Details part of the view. In the below example, we assume that the selection made in the Master view will be reflected in a property called **DetailsViewModelSelected**, which has the type **CarDetailsView­Model**. Exactly where this property is defined – and how the binding to the property is established – is discussed later.

<StackPanel>

<TextBlock Style="{...}"

Text="{Binding DetailsViewModelSelected.Heading}"/>

<StackPanel Orientation="Horizontal">

<TextBlock Style="{...}" Text="License plate"/>

<TextBox Style="{...}" IsReadOnly="True"

Text="{Binding DetailsViewModelSelected.LicensePlate}"/>

</StackPanel>

...

<StackPanel Orientation="Horizontal">

<TextBlock Style="{...}" Text="Price"/>

<TextBox Style="{...}"

Text="{Binding DetailsViewModelSelected.Price, Mode=TwoWay}"/>

</StackPanel>

</StackPanel>

With these bindings, we can now display the relevant properties of a car in the Details view. We can even edit the properties, if it makes sense. We earlier assumed that it only makes sense to change the price of a car, and it is therefore only the binding to the **Price** property that is a two-way binding. Also, the **TextBox** controls showing read-only data have been disabled for user input, by setting the **IsReadOnly** property to true.

### A MasterDetailsViewModel class

We are now pretty close to having all the pieces we need in order to build a working Master/Details view, with view model classes backing each element of the view. Let us recap what we have in place now:

|  |  |
| --- | --- |
| **DomainModel** | The **CarCollection** class plays the role of the domain model, since it contains and maintains a collection of domain objects. |
| **ItemViewModel** | This class defines how car data is presented in e.g. a **ListView**, to provide the user with just enough infor­mation about the item. The user can then make the selection with certainty about the identity of the selected item. |
| **MasterViewModel** | This class just handles the transformation of domain objects into a collection of item view model objects, to be displayed in e.g. a **ListView** control. |
| **DetailsViewModel** | This class exposes relevant properties about the details of a **Car** object, such that the user can view and possibly edit that data. |

The only piece missing is a class that ties these elements together. In order to see what this class should contain, we will recap how the Master/Details view is supposed to work:

* In the Master part, a collection of **ItemViewModel** objects are displayed, and the user can select one of these objects. The items should correspond to the domain objects in the domain model.
* In the Details part, details corresponding to the item selected in the Master part should be displayed. The controls displaying these details are bound to a **DetailsView­Model** object.

The class must then be able to do (at least) the following:

* Keep a reference to the domain model, to enable the creation of items to display in the Master view
* Provide a property to which the selected item in the Master view can bind. This must be a two-way binding, since changes in the selection should update the Details part of the view.
* Establish the link between the item selected in the Master view (of type **ItemView­Model**) and the object to which the Details view bind (of type **DetailsViewModel**)

These requirements alone implies that the class should contains these elements:

public class CarMasterDetailsViewModel : INotifyPropertyChanged

{

private CarCollection \_domainModel;

private CarMasterViewModel \_masterViewModel;

private CarItemViewModel \_itemViewModelSelected;

private CarDetailsViewModel \_detailsViewModelSelected;

public List<CarItemViewModel> ItemViewModelCollection {...}

public CarItemViewModel ItemViewModelSelected {...}

public CarDetailsViewModel DetailsViewModelSelected {...}

public CarMasterDetailsViewModel()

{

\_domainModel = new CarCollection();

\_masterViewModel = new CarMasterViewModel(\_domainModel);

\_detailsViewModelSelected = null;

\_itemViewModelSelected = null;

}

}

We have chosen to name the class **MasterDetailsViewModel**, since it is the top-level class to which the view itself will bind. The three properties are central for the class:

* **ItemViewModelCollection**: Will refer to a **ItemViewModelCollection** property in a **Car­Master­ViewModel** object. Will be bound to the **ItemsSource** property in the Master view
* **ItemViewModelSelected**: Will be bound to the **SelectedItem** property in the Master view with a two-way binding.
* **DetailsViewModelSelected**: Will refer to a **CarDetailsViewModel** object, speci­fically that object which corresponds to the selected item in the Master view. The Details view will then bind to this property.

We can then establish the bindings for the View:

<Page.DataContext>

<local:CarMasterDetailsViewModel/>

</Page.DataContext>

<!--Master view-->

<ListView

ItemsSource="{Binding ItemViewModelCollection }"

SelectedItem="{Binding ItemViewModelSelected, Mode=TwoWay}">

<ListView.ItemTemplate> ...

</ListView>

<!--Details view-->

<StackPanel>

<TextBlock Style="{...}"

Text="{Binding DetailsViewModelSelected.Heading}"/>

<StackPanel Orientation="Horizontal">

<TextBlock Style="{...}"

Text="License plate"/>

<TextBox Style="{...}"

Text="{Binding DetailsViewModelSelected.LicensePlate}"/>

</StackPanel>

...

<StackPanel Orientation="Horizontal">

<TextBlock Style="{...}"

Text="Price"/>

<TextBox Style="{...}"

Text="{Binding DetailsViewModelSelected.Price,

Mode=TwoWay}"/>

</StackPanel>

</StackPanel>

The only remaining issue is the implementation of the three properties. Two of them are trivial:

public List<CarItemViewModel> ItemViewModelCollection

{

get { return \_masterViewModel.ItemViewModelCollection; }

}

public CarDetailsViewModel DetailsViewModelSelected

{

get { return \_detailsViewModelSelected; }

}

Note that only the **get**-part is needed, since you cannot change these properties through the view. The last property is a bit more complex:

public CarItemViewModel ItemViewModelSelected

{

get { return \_itemViewModelSelected; }

set

{

\_itemViewModelSelected= value;

\_detailsViewModelSelected= \_itemViewModelSelected == null ? null

: new CarDetailsViewModel(\_itemViewModelSelected.DomainObject);

OnPropertyChanged();

OnPropertyChanged(nameof(DetailsViewModelSelected));

}

}

The two plain lines of code are completely standard for a property used for a two-way binding. The highlighted lines are more interesting.

The first highlighted line establishes the link between the selected item in the Master view, and the **CarDetailsViewModel** object the Details part should now bind to. When the selection changes, a new **CarDetailsViewModel** object is created, and is simply given a reference to that domain object which **\_itemViewModelSelected** refers to. Remem­ber that these are just references being passed around, so no new domain objects are created in the process. If no object is selected – i.e. the new value is null – no new **CarDetailsViewModel** object is created.

The second highlighted line is the standard “trick” for getting the View to read the **DetailsViewModelSelected** property again. We have just updated **\_detailsView­Model­Selected** to refer to the new **CarDetailsViewModel** object, so the view needs to show that data now. The call to **OnPropertyChanged(nameof(DetailsViewModel­Selected))** will cause all of the properties bound through **DetailsViewModelSelected** (see the XAML above) to be re-read by the view.

With this in place, the pieces have come together. We now have a functional, MVVM-based Master/Details view for car data, which responds to user selection as excepted, and allows updating of the price field. One significant omission so far is that we have not added functionality for e.g. deleting an item. We will now add this functionality by using Commands, just as we did for the View-Model setup.

### Adding a Delete command to an MVVM-based view

Since the view binds to **CarMasterDetailsViewModel**, all properties for binding to commands should be implemented in this class. This is not particularly difficult, and follows the same pattern as for the Model-View setup. The complications arise when we need to implement a new version of **DeleteCommand**. Let’s review the implemen­tation in the Model-View setup:

class DeleteCommand : ICommand

{

private CarCatalog \_carCatalog;

public DeleteCommand(CarCatalog carCatalog)

{

\_carCatalog = carCatalog;

}

public bool CanExecute(object parameter)

{

return true;

}

public void Execute(object parameter)

{

\_carCatalog.DoDelete();

}

public event EventHandler CanExecuteChanged;

}

The problem is that even though we can still perform a deletion in the **CarCatalog** object, it is not enough. Since the domain model is not involved in the presen­tation anymore, it does not implement **INotifyPropertyChanged** or use an **Observable­Collection**. In other words; just deleting in the domain model will not cause the view to update accordingly.

We solution is to implement a **Delete** metod in the **CarMasterDetailsViewModel** class. This is also appropriate, since this class should keep the view(s) and the model consistent with each other. The implementation is fairly simple:

public void Delete(string licensePlate)

{

\_domainModel.Delete(licensePlate);

ItemViewModelSelected= null;

OnPropertyChanged(nameof(ItemViewModelCollection));

}

Let’s detail what happens here:

1. The domain object identified by the license plate (which acts as a key here) is deleted from the domain model.
2. The selection is set to **null**. This is a design choice; you could also have chosen to set the selection to e.g. the first object left in the list. Setting the selection to **null** will also cause the Details “selection” to become **null**.
3. Since we have deleted a domain object, we poke the **ItemViewModelCollection** property, so a fresh list of **ItemViewModel** objects is created and retrieved by the view.

The final issue is then the revised implementation of the **DeleteCommand** class:

public class DeleteCommand : ICommand

{

private CarMasterDetailsViewModel \_viewModel;

public DeleteCommand(CarMasterDetailsViewModel viewModel)

{

\_viewModel = viewModel;

}

public bool CanExecute(object parameter)

{

return \_viewModel.ItemViewModelSelected != null;

}

public void Execute(object parameter)

{

CarItemViewModel selected = \_viewModel.ItemViewModelSelected;

\_viewModel.Delete(selected.DomainObject.LicensePlate);

}

}

We note that the command is only allowed to execute if a selection has been made in the view (the **CanExecute** method), and that the **Execute** metod “digs out” the license plate for the currently selected item, and uses that value when calling **Delete** on the view model.

### Generalising the classes

These final additions brings us to a major milestone – we now have a Master/Details view including command-based functionality, playing entirely by the MVVM rulebook. The obvious question is now: what will it take to implement the same functionality for another domain class? On an architecture level, the entire class structure should be reusable, since nothing in it relies on specific details for the domain class. The only assumption has been that the domain class is “simple”, i.e. that all properties have a simple type.

If this assumption is met, it should almost be a matter of copy-paste plus search-and-replace for adapting the structure for a different domain class. Some details will of course be domain-specific, like the properties in the **CarDetailsViewModel**, but quite a lot of code has a very generic nature. Consider the **CarMaster­View­Model** class:

public class CarMasterViewModel

{

private List<CarItemViewModel> \_itemViewModelCollection;

private CarCollection \_carCollection;

public List<CarItemViewModel> ItemViewModelCollection

{

get

{

\_itemViewModelCollection = new List<CarItemViewModel>();

foreach (var car in \_carCollection.All)

{

\_itemViewModelCollection.Add(new CarItemViewModel(car));

}

return \_itemViewModelCollection;

}

}

public CarMasterViewModel(CarCollection carCollection)

{

\_carCollection = carCollection;

}

}

We have not discussed the topic of **Generics** in C# yet, but we have actually used it a lot. We have often written e.g. **List<Car>**, meaning “a **List** object that can hold a col­lec­tion of **Car** objects”. **Car** is here used as a specifc kind of parameter; a **type** para­meter. List-based functionality is very generic, so it would be nice if you could write that functionality once, and then use it no matter the type of objects you put into the list. Generics enable you to do just that.

Consider now the generalised **Master­View­Model** class below:

public class MasterViewModel<TDomainClass, TItemViewModel>

{

public static List<TItemViewModel> GetItemViewModelCollection(

Model<TDomainClass> model, Factory factory)

{

List<TItemViewModel> itemViewModelCollection =

new List<TItemViewModel>();

foreach (var obj in model.All)

{

itemViewModelCollection.Add(new TItemViewModel(obj));

}

return itemViewModelCollection;

}

}

We are not fully equipped to understand such code, but the point is just to illustrate that we can parameterise the code to such an extent that it can be reused for all domain classes we wish to include. With a class like the above available, we could change e.g. the type of an instance field in the **CarMasterDetailsViewModel** class:

// private CarMasterViewModel \_masterViewModel; (Original)

private MasterViewModel<Car, CarItemViewModel> \_masterViewModel;

Instead of having to create a new master view model class for each domain class, we could then simply reuse the generic version.

We will not go into further details with this topic here, but revisit it when we have learned more about Generics, since Generics is a crucial tool to use for this purpose.

## Using files as persistent storage

We have until this point not cared about what happens to our data, once the appli­cation containing the data terminates. Real-life applications will usually contain facilities for saving and/or loading certain data into the application from so-called **persistent storage**. The term persistent storage just covers all those media for data storage, where data is retained even after the power for the media is shut off. Exam­ples of locations for persistent storage could be your own local hard drive, or simply “the cloud”, where the exact location of the data is not known.

Management of persistent storage is a large topic, and more advanced applications will often use some sort of **database** for structured storage of data. For smaller appli­cations producing small amounts of data, simpler media may suffice. Storing data in a text file on your own hard drive is a fairly easy solution to use for such situations.

We will therefore here give a short and pretty specific introduction to how to use text files for persistent storage, with the specfic goal of being able to store a collection of domain data objects in a file. Our approach will also rely on using the so-called **JSON** (**JavaScript Object Notation**) format. JSON can be considered a sort of alternative to XML, since it is just a structured way of storing data on text form. We will not explain JSON further here, but there are plenty of sources about JSON online[[11]](#footnote-11).

The reason for using JSON at all is because it provides a general and convenient way of transform­ing C# objects to a text format, which can then easily be written to a file. Likewise, the text (on JSON format) can be read from the file again, and transformed back into C# objects. There are several third-party class libraries available for JSON conversion, the most popular (at the time of writing) being the **NewtonSoft.JSON** package. This library can be installed as a NuGet package: In Visual Studio, go to **Tools | NuGet Package Manager | Manage NuGet Packages for Solution**, and choose **Browse**. The **NewtonSoft.JSON** package is usually found on the first page; if that is not the case, simply search for it using the search field. Once the package is found, simply click **Install**. At the time of writing, package version 8.0.3 was used.

With the package in place, it is possible to create a small C# helper class targeted for our needs. We need to be able to save a collection of domain objects into a file, and load it back into the application. An example of such a class called **FilePersistency** follows below:

public class FilePersistency<T> where T : class

{

private const string FileName = "data.json";

private CreationCollisionOption \_options;

private StorageFolder \_folder;

public FilePersistency()

{

\_options = CreationCollisionOption.OpenIfExists;

\_folder = ApplicationData.Current.LocalFolder;

}

public async Task Save(List<T> data)

{

var dataFile = await \_folder.CreateFileAsync(FileName, \_options);

string dataJSON = JsonConvert.SerializeObject(data);

await FileIO.WriteTextAsync(dataFile, dataJSON);

}

public async Task<List<T>> Load()

{

try

{

StorageFile dataFile = await \_folder.GetFileAsync(FileName);

string dataJSON = await FileIO.ReadTextAsync(dataFile);

return (dataJSON != null) ?

JsonConvert.DeserializeObject<List<T>>(dataJSON)

: new List<T>();

}

catch (FileNotFoundException)

{

await Save(new List<T>());

return new List<T>();

}

}

}

The file operation methods **Create**- and **GetFileAsync** are both part of the .NET class library. They are as such not particularly remarkable, but they both end with the suffix **Async**. Also – and perhaps more intriguing – is the keyword **await** seen just before calling the methods. The .NET class library contains a set of classes used for so-called **asynchronous programming**. This is also an important and rather advanced topic[[12]](#footnote-12), but for now we will just note that methods ending with **Async** offers the opportunity to continue execution of the application, even though the method has not returned a result yet! However, the coding style used above essentially cancels out this effect, so the flow-of-execution will be as we are used to: we wait for a method to return a result, before proceeding to the next line of code.

The code above does a two-step conversion between the file **json.data** and the list of C# objects of type **T** (**T** is a **type parameter**, so it can be any class when the **FilePer­sistency** class is actually used). Consider first the **Save** method. The first line creates (or opens, if the file already exists) a file, and returns a variable **dataFile** which now refers to the file. The next line converts the incoming data (in the form of a **List** of objects of type **T**) to JSON format. Since this is a text format, the result of the conver­sion is a **string**. This kind of conversion between an in-memory object and a sequence of cha­racters is also known as **serializing** the object, which is why the called method is named **SerializeObject**. Finally, the **JSON** string is written to the file.

JSON string

List of C# objects

File

The **Load** method is essentially the same deal, just in reversed order. A **string** is read from the file, and then converted (also known as **deserialized**) into a **List** of objects of type **T**

List of C# objects

File

JSON string

The **try-catch** block in the **Load** method is added to handle the case where the caller attempt­s to load data from a file that has not been created yet, e.g. the first time the appli­ca­tion is executed. In that case, the **Load** method simply calls the **Save** method with an empty list, to invoke creation of the file.

How can we then use **FilePersistency** in our MVVM setup? Sticking to the **Car** domain example, we could imagine a fairly simple class e.g. called **CarSource**, containing two methods **Load** and **Save**:

public class CarSource

{

private FilePersistency<Car> \_fileSource;

public NoteSource()

{

\_fileSource = new FilePersistency<Car>();

}

public async Task Load(CarModel model)

{

model.Clear();

List<Car> carList = await \_fileSource.Load();

foreach (Car c in carList)

{

model.Add(c);

}

}

public async Task Save(CarModel model)

{

await \_fileSource.Save(model.All);

}

}

We have here assumed that the **CarModel** class is a class that can provide all of the contained **Car** objects as a **List**, through a property called **All**. Also, the **CarModel** class has an **Add** method for adding a single **Car** object. The **CarSource** class effectively adds another element to the chain of conversions shown before:

JSON string

List<Car>

CarModel

File

The syntax **async Task** in the method declarations may look a bit strange, but you should not be too intimidated by it. It is just a bit of lingo needed when working with these asynchronous methods. With the **CarSource** class in place, we can now add load- and save-functionality to our MVVM application. In the **MasterDetailsView­Model** class, we can add parameterless **Load** and **Save** methods:

private async void Load()

{

await \_source.Load(\_model);

...

}

private async void Save()

{

await \_source.Save(\_model);

...

}

The (...) just indicate that we might need to do a bit more in each method; we could imagine that we would need to refresh the **ListView** after having loaded data from the file, by calling **OnPropertyChanged** for a collection property bound to by the **ListView**. Wrapping these methods up as commands is also pretty simple:

private RelayCommand \_loadCommand;

private RelayCommand \_saveCommand;

...

\_loadCommand = new RelayCommand(Load, CanLoad);

\_saveCommand = new RelayCommand(Save, CanSave);

...

public ICommand LoadCommand

{

get { return \_loadCommand; }

}

public ICommand SaveCommand

{

get { return \_saveCommand; }

}

...

The **async** syntax has ebbed out at this point, so the command-handling code looks just as before. These commands could then be bound to e.g. buttons or maybe menu items in the application. The exact conditions for when to allow **Load** and **Save** – i.e. the exact code for **CanLoad** and **CanSave** – will depend on the requirements for the application.

1. We will in general refer to a computer program as an App (short for ”Application”), without assuming anything specific about what platform the App runs on. This could be an ordinary PC, a smartphone, a tablet, or some other device. [↑](#footnote-ref-1)
2. Details on how to do this may vary from course to course, and is thus found elsewhere [↑](#footnote-ref-2)
3. Again, the specific details of how to obtain and install ReSharper will be part of the course-specific materials. [↑](#footnote-ref-3)
4. Again, details on how to do this may vary from course to course, and is thus found elsewhere. [↑](#footnote-ref-4)
5. Draconian: To apply severe punishments to small offenses [↑](#footnote-ref-5)
6. One could, however, also claim that if you feel the need to add comments to your code, it might be an indication that your code should be restructured to make it clearer… [↑](#footnote-ref-6)
7. https://martinfowler.com/books/refactoring.html [↑](#footnote-ref-7)
8. https://docs.microsoft.com/en-us/windows/uwp/files/file-access-permissions [↑](#footnote-ref-8)
9. For instance at https://msdn.microsoft.com/en-us/library/windows/apps/mt185406.aspx [↑](#footnote-ref-9)
10. https://docs.microsoft.com/en-us/windows/uwp/layout/layouts-with-xaml [↑](#footnote-ref-10)
11. https://en.wikipedia.org/wiki/JSON [↑](#footnote-ref-11)
12. https://msdn.microsoft.com/en-us/library/mt674882.aspx [↑](#footnote-ref-12)