

SOFT 153 – Minimal C# Coding Requirements

Thomas Wennekers

School of Computing, Engineering and Mathematics

Plymouth University, Drake Circus, PSQ 8AA Plymouth, UK

Email: thomas.wennekers@plymouth.ac.uk

Abstract

This document describes the programming knowledge required for SOFT153. This knowledge should be present already at the start of the module, for example, acquired in SOFT 151 or 152 during semester 1. Students that miss the relevant programming skills have an increased likelihood to fail SOFT153 and are therefore strongly encouraged to learn basic programming skills as soon as possible. These basic skills are not a part of SOFT153, but a precondition.

1 Introduction

The algorithmic part of module SOFT153 is about how data can be organised and processed efficiently on computers. It is not about basic programming. It assumes that the students are familiar with the most basic programming constructs like basic data types (int, float, char), arithmetic and logical expressions, loops, conditional statements, and a little more. These constructs are common in almost all programming languages. If they are not learnt early throughout a course in Computing, students are likely to struggle with later modules.

In earlier years I used an exercise that contains 15 simple coding tasks like printing to the screen, writing a loop that counts up to ten, or writing a function/method that swaps two numbers. An experienced programmer can do these in about 10 minutes. It turned out in previous years that some students starting on SOFT153 do not as easily solve these problems – perhaps around 25 students per year have little or no previous experience with writing any code. These students have major problems following the SOFT153 module and indeed have a high likelihood to fail the module.

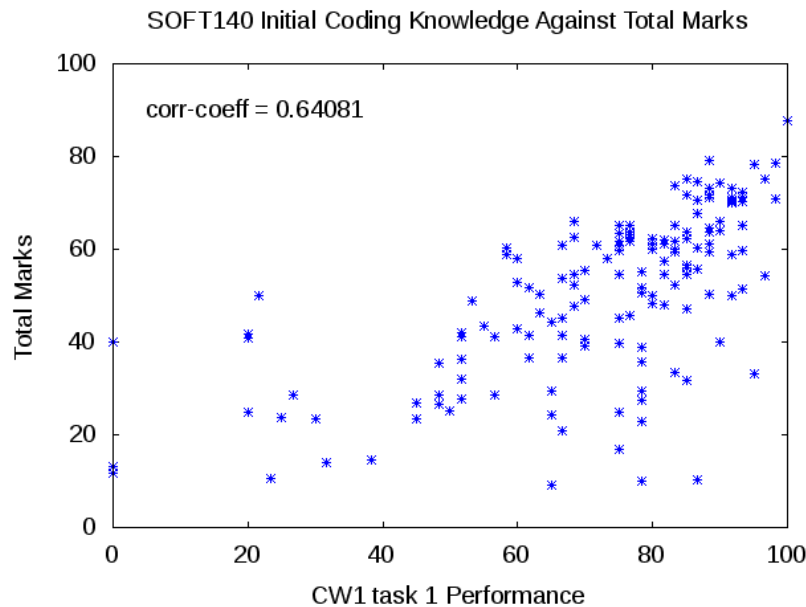


Figure 1: Final marks plotted against marks in task 1 of assignment 1 for the year 2011/12.

Figure 1 displays the final module marks students achieved in the academic year 2011/12 against their performance on the simple programming tasks mentioned. Apparently a large amount of students received marks larger than 50 percent on these tasks. Given that the tasks are indeed very simple and the concepts supposed to have been learnt in semester 1, these results are not great. Only few students showed a high proficiency in coding.

An even more striking feature of the figure is that students that were not able to do the simple coding tasks in assignment one often failed the whole module and never reached more than a mark of 50. This indicates that it is absolutely crucial in order to succeed on SOFT153 to understand and get used to basic programming constructs as taught in SOFT151 and SOFT152 in the first semester. Any deficits should be dealt with immediately.

It is also visible from the figure that quite a number of students that performed well on assignment 1 task 1 still failed the module overall. There are various possible reasons for this, one being code sharing among friends. Students that use code from others in their assignments and are not absolutely confident that they could have done the tasks easily themselves are apparently under the threat of failure, too. So, please, fairly investigate your skills, and in case of suspected deficits, improve!

On that background the following document is provided, which summarises the most basic constructs in programming using the programming language C#. These constructs appear in the same or similar form in many other programming languages; understanding them thoroughly once makes learning other languages much easier.

The constructs described are practically all that are required to successfully follow SOFT153, do the practicals and assignments, and the exam at the end of term two. Most programming languages have many more and very advanced features like all kinds of in-built data types and functionality, GUIs, interfaces to data-bases and web-servers, and much more. This is all entirely irrelevant for SOFT153. You will not need any of this.

SOFT153 is about how data-structures work: students are asked to implement their own lists, queues, or search and sort algorithms. Programming languages do provide these functionalities, but on SOFT153 students are not allowed to use these tools. The goal of SOFT153 is to provide an understanding of how these data-structures and algorithms work internally. Other modules deal with how to use them. The understanding of how data-structures and algorithms work will later allow to choose optimal structures for specific applications and to develop efficient code.

In a similar vein, in SOFT153 students are only asked to write console programs; GUIs are prohibited. In the past some students found that pointless, but it is not. GUI-programming is content of other modules and distracts students from what matters on SOFT153. A few input/output statements in a console-program can be written in minutes; a GUI needs much more time, which would be better spent focussing on the main aspects of the code. GUI-based code is also often badly structured by mixing graphical design with actual functionality. In SOFT153 we are exclusively interested in the functionality; how do things work internally?

While reading this document it is suggested to have Visual Studio or MonoDevelop open in order to try things out immediately.

This document does not intend to replace a good book or tutorial in programming. It summarises what is needed on SOFT153.

2 Code Layout and Comments

2.1 Basic Code Layout

Figure 2 shows a most simple C# program. It does nothing useful, but your own code will later be embedded within similar lines. Visual Studio can automatically generate files that look similar when a new project is started.

First note that the red words are “keywords”, special words that have a meaning to the compiler (Visual Studio or MonoDevelop) you are using. You need to write them exactly as they appear. It is usually considered very bad practice to use them in any other context than the one they are defined for. For example using keywords as variable or function names is not a good idea (and some compilers prohibit that).

The black words are names or “identifiers”. They can name different things like variables, functions, and other things (e.g., modules like “System”). You choose the

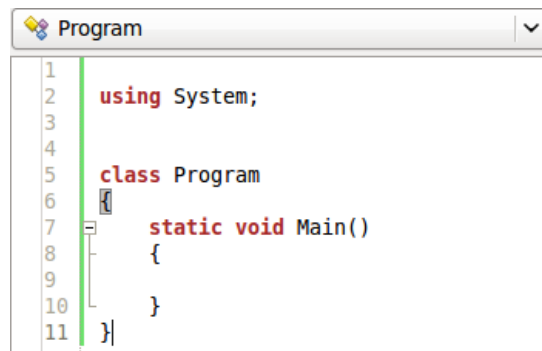


Figure 2: Basic Code Layout

names for your variables and functions.

Line 2, `using System` advises the development environment to import some functionality that every C# system provides by default. “System” is the name of a module of code provided by the compiler you use (like Visual Studio C# or MonoDevelop). In SOFT153 we will mostly use “System” to read from the keyboard and write to the console. We will come to this later. When a new project is started in Visual Studio a certain number of modules are included automatically in the same way. We will not use them in SOFT153 and the respective lines can be safely removed. As a matter of fact there is only a single other module beside System that we will sometimes use. That module is for reading and writing data from files on the hard drive. If you believe you need something else, chances are you are not doing something like it was asked for.

Line 5 starts a new “class”. This is a term used heavily in object-oriented programming. In SOFT153 we do not deal with that, but we still need to define a class, here called “Program”, that holds all the code and functionality we want to implement. C# expects us to define a class. The name “Program” is quite arbitrary, another name can be chosen as long as it does not already exist (for example in the “System”-module that was imported). In general, names should be informative and say something about the thing that is named. Since we are going to write many programs a name “Program” is apparently not an optimal choice. On the other hand “Main” is the default name for the entry-point into an executable code element.

The opening parenthesis in line 6 opens what is called a “block”. Note that there is a matching closing parenthesis in line 11. Everything between these two parentheses belongs to the class “Program”. In the case shown this is not much, just one function, called “Main”. We ignore the keywords “static” and “void” for the moment. The Main function is the “entry-point” of the class. If it is compiled and executed the run-time environment will enter the Main-function to see what has to be done. In our case there is nothing to do, because the Main-routine does not contain any actual code. Therefore the runtime-environment will immediately exit the code and terminate the program. So, if we want to have our code do something we will have to put it into the Main-routine.

Later more about that.

2.2 Commenting

It is good practice to comment code in a meaningful way; not too much, not too little. Figure 3 shows an example, the same code as shown in the previous figure, together with some comments. The amount of comments may already be a bit heavy given that the code does nothing at all.

```
1
2  /* This program has been written to demonstrate basic code layout
3   * and commenting practice for SOFT148
4   *
5   * Author: Thomas Wennekers
6   * Date  : 26 June 2012
7   */
8
9
10 // We include (or 'use') "System" mostly for reading the keyboard
11 // and writing to the console
12
13 using System;
14
15
16 // "Program" is just a name for our main class
17
18 class Program
19 {
20
21     // ... other variable and function definitions could go here ...
22
23     static void Main() // the Main routine is a function that
24                       // is entered when code is executed
25     {
26
27         // ... code to do something goes here ...
28
29     }
30 }
```

Figure 3: Code with comments

First note that there are two types of comments, inline-comments and multi-line comments. Inline comments start with “//”; everything after that until the end of the line is the commenting text and ignored when the program is executed. If the comment proceeds onto the next line you need another “//”.

Multiline comments span several lines. They are started by “/*” and closed by “*/”. Everything in between is the comment and ignored when the program runs. Note that MonoDevelop, the programming environment for C# on Macs and Linux machines colours comments in blue. That makes the code and comment structure more transparent. Visual Studio does it similar.

Observe that the program shown in Figure 3 has a block at the top that explains where the code is about, who wrote it, and some more information. This is commonly considered good practice. You can also put licensing information on the top of the file

that usually would be the first thing to put before anything else. In the assignments you would probably also add your student ID.

Furthermore, note that I have scattered a couple of inline comments over the code; these are usually supposed to be quite short and informative. The amount I give in the example is maybe already a bit too extensive. Informative means that you don't have to comment obvious things. But be aware of any tricks or crucial lines in the code; they typically deserve a comment. It is also common to point out possible pit-falls, i.e., locations that you suspect may cause problems in certain situations. Many programmers also write a few comments for every function they define before the function or after their header; I have added a comment after the Main function, which however can't say much, as the function does nothing.

It is not only good practice to comment code reasonably, but also to lay it out properly. You can use white-space, empty lines, indenting and comments to make your code easier to understand. The code in Figure 3 looks quite good, however, the one in Figure 4 is really bad.

```
1      using System; // we use system
2
3
4      class      knurtz234
5      {
6
7
8          static void
9          Mian() // the Main routine is a function that contains stuffs and computes the main thigns in my code; I haven't checked it much but hope it os correct
10         {
11
12
13         }
14     }
```

Figure 4: Bad layout and commenting

First note the comments. The comment in line 2 is entirely useless. Every programmer should know what that line means. Don't comment the obvious; comment locations where it makes sense. The comment in line 9 is also practically useless beside being full of typos. It is obvious that the main-function would compute something; it would be better to say what it computes. The line is also much too long. Not everybody has a super-broad screen, don't go wider than 100 or 120 characters. Some people also don't like an "I" and "me" style in comments, "I did", "I thought", better stay neutral and to the facts.

A second thing that's bad in the code in Figure 4 are the names. The names in your code should be meaningful – "knurtz234" means nothing, "Mian" is misspelled. In general, avoid unspecific names, for example "function1", "variable3", "counter", "wesfdf", "int1", "var2", "x1", "x2", "x3", "x25", and so on. There are exceptions, counter variables in loops are often named "i", "j", "k"; the amount of elements to process is often called "n" or "m"; "result" is not uncommon for a return value of a function, neither are "x", "y", "z", "u", "v" for function arguments.

Another bad aspect of the code in 4 is the indentation. In many places a compiler ignores blanks and empty lines (with the exception of strings, keywords and identifiers). This can (and should) be used to make code more readable. In the figure it has been used

to the opposite effect: There is an unnecessarily wide space between the “class” and the classname “knurtz234”. The opening and closing brackets of the blocks are differently indented; it would be better if matching brackets are indented by the same amount of blanks or tabs. It is also useful to indent each new block that is opened by the same amount of space. In the example the class opens a block and everything inside that block should be inserted by the same amount of space. However, if another block is opened inside this block the content inside the new block should be indented by another so-and-so many spaces. The amount of indentation is a matter of taste, some use 2 blanks, some use 4, some use tabs; but it should be consistent. In Figure 3 indentation is consistent, everything on the same level has been indented by the same amount of space; in Figure 4 it is not. Note also that the “Using System” has been indented, but should not have been. It is located in the outer most block, outside of any parentheses. Everything there, like the “class knurtz234” definition should not be indented at all.

3 Basic Data Types and Expressions on Them

Computing is about processing data. So, one needs ways to describe data. Data can be just everything, from simple things like the current temperature (just a number) to complex things like the structured content on a DVD (the index, hundreds of thousands of frames containing thousand of pixels each, the sound stream(s), any additional information). Computer Scientists have developed efficient ways to describe and process simple and structured data.

Typically data when processed on a computer is hold in main memory or on a mass storage device. It is usually accessed in a program by a name (an “identifier”) given to it. The compiler and runtime environment automatically resolve the name to the location where the data actually sits in your system. There is nowadays not too much need to worry about that.

The above means there is a difference between the data and the “variable” that actually refers to it in your code. “Data” is something somewhere in memory, a “variable” is a name that references data in some way.

3.1 Basic Types

There are some simple (in the sense of “most elementary”) data types available in most programming languages - “integers”, “floating point numbers”, “characters”, “strings”, and “boolean values”. Many programming languages are what is called “strongly typed”, meaning that you always have to specify precisely what type a variable or data item has. “Weakly typed” and so-called “dynamically typed” languages are much less restrictive, but this can cause very nasty programming mistakes. C# is strongly typed. You will have to keep track of types.

```

1
2  class Programm
3  {
4      static void Main() // define some basic types
5      {
6          int    int32      = 123;
7          float  float_number = 1.524f;
8          double double_number = 1.524;
9          char   char_variable = 'a';
10         string string_var   = "some text";
11         bool    bool_var    = false;
12
13
14         // .. the main program still does nothing
15     }
16 }
17

```

Figure 5: Declaring variables of basic types

Figure 5 displays an example that declares a number of variables of different types. The black text are the names of the variables, you can choose them as you want but they have to be unique. They also typically should reflect what the variable is about; and you may want to add a comment after variables that play a central role in the program; saying very briefly what the variable is for (unless that's clear from the name already). Note that the program in the figure doesn't do anything with the variables allocated. Most likely if you compile the code your compiler would complain about that. The variables are unnecessary and would only waste space in memory, because they are not used at all.

The green words in Figure 5 are the actual types of the variables. Each variable has to be given a name and a type. The type specifies what values it can take and what you can possibly do with a variable.

A variable can be assigned an initial or default value, but does not have to. In Figure 5 all variables get assigned some value, which are shown in pink. The equal-signs mean "set the value of the variable to whatever data is given on the right". If you don't want to assign a value when declaring a variable leave out everything after the name in pink as well as the black equal sign. You can later assign a value to the variable as you wish. However, should you later use the variable unassigned your compiler will probably complain. An unassigned variable can have just any value (whatever is contained in the memory allocated for the variable at the time of allocation); that hardly ever makes sense.

Depending on what computer language you use your system will be more or less restrictive regarding what you can assign to a variable. In general you should only try to assign a value that matches the type of the variable. For example, if you try to assign a string to an integer or floating point variable your compiler will likely complain (but some languages can deal with that). Likewise a char can only hold a single byte; assigning it a longer string or an integer value will lead to undefined results. C# will usually complain about such cases (but not all other computer languages do). Note also that there is a difference in initialisers for floats and doubles (and similarly for short and long integers). A data item "1.234" is a double by default and can be assigned to a double variable.

However, in order to assign the same value to a float variable you have to explicitly tag it as float using an 'f'-character "1.234f". The difference is that a double variable holds the value up to a higher precision. Compilers in the past didn't care much about that and implicitly converted data according to the target data type without complaining. This can lead to nasty and hard to trace bugs. Therefore, C# will often complain (although it still does some implicit conversions).

Some elementary data types are:

int : roughly equivalent to an integer number in Mathematics, i.e., a positive or negative integer number including zero. However, an "int" variable typically only holds 32 bits of the number, that is can only distinguish 2^{32} possible different values. Therefore not all integer numbers can be represented; there are upper and lower bounds. For completeness, it should also be said that there are "short" and "long" versions of integers on computers that can store less or more than 2^{32} different values.

double : these numbers roughly correspond with the real numbers, like 2.4, or "pi" or 7.123445437372727. Again on the computer numbers can only be represented with a finite amount of digits, such that not all real numbers can really be represented by the data type double. This can in seldom cases lead to trouble if very many or very large numbers have to be processed.

float : this type is similar to double but uses less memory space to represent numbers. This means compared with doubles even less of the real numbers can be faithfully represented. However, for the purpose of SOFT153 this distinction is unimportant. It can become important if you ever do precise simulations of physical processes or in finance, when calculations have to be exact given very large numbers to be processed. In these cases "double" variables should be preferred over "float".

char : This is a single character represented by a number between 0 and 255. There is a table, the so-called ASCII table (for American Standard Code of Information Interchange) that specifies which number stands for which character. A more modern approach to specify characters would be to use "Unicode", but this is outside the range of the SOFT153 module. Unicode can represent characters in many languages including non-latin ones. Please consult google for more information.

string : A "string" classically was an array (see section about arrays below) of characters, i.e., a certain number of characters stored consecutively in memory and accessed by one single name. However, with the advent of object-oriented programming the situation has become more complicated. Strings can now be specific data-types quite different from character-sequences. You will learn about this in other modules.

bool : A boolean variable is one that can hold only one of two values, either "True" or "False". It expresses the truth-value of "predicates", i.e., logical assertions. We will

cover boolean variables in section 4. Note that the "false" in line 11 of Figure 5 is printed in red – "true" and "false" are keywords that the compiler recognises.

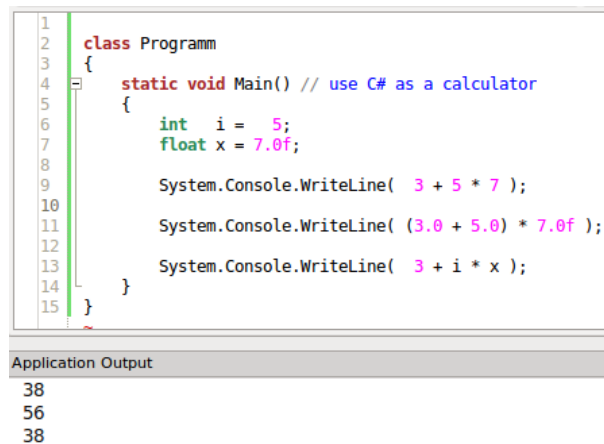
3.2 Operations and expressions on basic types

Data types are useless without operations on them. "Operators" take one or more data items or variables, do something with them, and return a value depending on the result.

Arithmetic Operations

A sort of operators that you are very familiar with are the arithmetic operations: plus, minus, multiplication, division, exponentiation. These and more operators are available by default in practically all programming languages. You can combine them into what is called "expressions" in Computer Science. This means, you can use a programming language as a calculator. The form of expressions is actually very similar to what one learns at school to put into a pocket calculator.

To make practical use of this you will need to know how to print something in C#. For that purpose functions provided by the "System" - libraries can be used, e.g. in the way shown in Figure 6.



```
1
2 class Programm
3 {
4     static void Main() // use C# as a calculator
5     {
6         int i = 5;
7         float x = 7.0f;
8
9         System.Console.WriteLine( 3 + 5 * 7 );
10        System.Console.WriteLine( (3.0 + 5.0) * 7.0f );
11        System.Console.WriteLine( 3 + i * x );
12    }
13 }
14
15
```

Application Output

```
38
56
38
```

Figure 6: Code used as a calculator

The code in Figure 6 first declares an integer and a float variable and assigns some values to them. It then computes three arithmetic expressions and prints the results. The print statements use a function "System.Console.WriteLine", the name of which indicates that we are "using System" and in particular a sub-part of it, "Console", that deals with writing to a console (also called terminal or "DOS-prompt" on Windows systems). "WriteLine" is a function that prints the value of its argument (i.e., what is contained in the brackets of the function argument), and adds a Newline to start a new line. The lower three lines in Figure 6 show the output when the program is run.

The first line, 38, is apparently the result of $3 + 5 * 7$ as learnt in school. Evaluation of this expression makes use of what is called “operator precedence”, multiplication comes before addition. If it were the other way round, addition before multiplication, the result of $3 + 5 * 7$ would be $3 + 5 = 8$ times $7 = 56$. This is apparently what the second line in the Application Output shows. The respective calculation has been forced to first do the addition by adding parentheses in the statement, $(3 + 5) * 7$. This can be done on many calculators as well.

Note also that the numbers in the second calculation are actually floating point numbers of single or double precision. The value printed as a result, however, is an integer. C# implicitly converts between different data types here. This is called (implicit) type casting.

The third expression in Figure 6 makes use of previously declared and assigned variables, i and x . Again, types are implicitly converted and multiplications are done before additions as is usual in mathematics.

Question: What would be the result of the third expression, if x would not be an integer, but let's say $x = 7.1$?¹

Hint: The modulo operator, `%`, is a special operator on two integers that divides one integer x by a second integer y and returns the remainder, e.g., $10 \% 3 = 1$ because ten divided by three is three plus a remainder of **one**.

Hint two: If you divide two integers the result will be “integer-division”, i.e., $5/2 = 2$, because 2 fits twice into 5 and the remainder is discarded in integer divisions. If you want the floating point division you have to type-cast, e.g., $5.0/2 = 2.5$ or $(float)5/2 = 2.5$ or $1.0 * 5/2 = 2.5$ (try these three options in C#; the “(float)” keyword explicitly converts a variable to float (if that is possible)).

Operations on characters and strings

Characters in many programming languages are a special type of integers, i.e., integers that are just one byte or 8 bit long and therefore can hold just $2^8 = 256$ different values. There is a table, the so-called ASCII table (for American Standard of Information Interchange) that defines which value refers to which character of the alphabet (plus some additional symbols and control keys, most notably TAB (`'\t'`), NEWLINE (`'\n'`) or CARRIAGE RETURN (`'\r'`). The blank (`' '`) for example maps to the numerical value 32, try this in C#: `System.Console.WriteLine((int)' ');`. The `(int)` casts the character `' '`, i.e., the blank, to an integer, which is then printed (32). Alternatively you could try `System.Console.WriteLine(' ' + 0);` this does an implicit type-cast of the blank to an integer, adds zero, and prints the result (32). Other characters have other codes (`'A'`=65, `'z'`=122, etc. Check google for 'ASCII code'). Doing arithmetics with characters, like `9*'A'`, is possible but not considered good practice.

¹Answer: 38.5, all numbers are cast to floating point numbers now, $3 + i * x = 3 + 5 * 7.1 = 38.5$.

Classical strings are nothing but “arrays” (sequences) of characters. Arrays are treated in section 7.1. Access to different characters in an array is possible using similar methods as described there. However, numerous special operators for arrays exist, for example to retrieve the length of a string, or substrings starting and ending at certain positions. Object-oriented programming languages also have special data types called “Strings”, which provide even more functionality. In SOFT153 we will not make use of these.

Logical Operations

Boolean (`'bool'`) variables can hold logical values `'true'` and `'false'`. A number of logical operators can be applied to boolean variables in a similar way as arithmetic operations are applied to numbers. The most common logical operators are AND, OR and NOT, expressed in many programming languages as `'&&'`, `'||'` and `'!'`.

NOT : The NOT operation negates a logical value $!true = false$ and $!false = true$. This also holds for variables: If `a=true` is a boolean variable set to `true`, the value of `!a` would be `false` (because `a` is true). Check this in C#:

```
bool a = true;
System.Console.WriteLine( !a );
```

AND: The AND operation combines two logical values into a new one. If a and b are two logical variables the expression `a && b` is true if and only if both a and b are true. If any of them is false ANDing them also results in false.

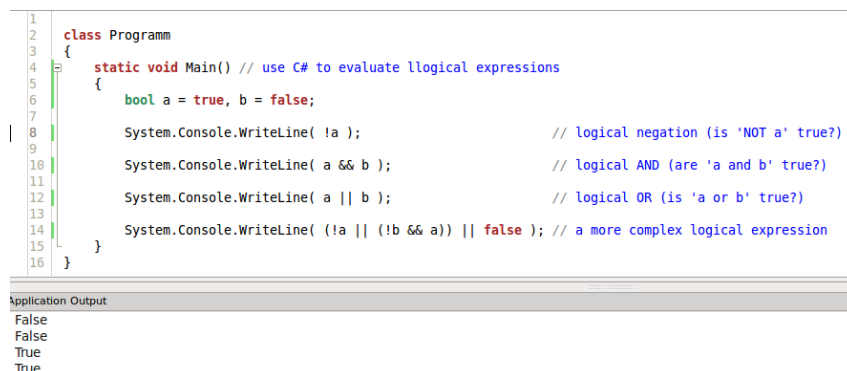
OR: The OR operation also combines two logical variables, but this time in order to result in “true” only one or the other (or both) have to be true. Only if both of a and b are false is the expression `a || b` also false.

Instead of variables a and b , logical expression can also contain explicit values “true” and “false” or more complex logical expressions, see the examples below.

There are more logical operators than AND, OR and NOT which are not covered here. Also, as for arithmetic rules, there is precedence among logical operations. NOT is first evaluated, then AND, and finally OR. “AND comes before OR” in logic as “multiplication comes before addition” in arithmetics.

Note also that there is a second set of so-called “bitwise” logical operators. A character can hold 8 bits, an integer (typically) 32. These bits can be set individually and bit-wise logical operations act bit-by-bit. The bit-wise NOT (“`~`”) would negate all 8 bits of a character at once, for example, changing `10011000` into `01100111` by flipping all bits. The bitwise AND (“`&`”) and bitwise OR (“`|`”) follow the same principle, they act on pairs of bits assuming 1 means true and 0 false. Bitwise operations are often used to define “flags” for certain properties that objects have or have not; these need only one bit

to store. Using bitwise operations can be more space and time efficient than using one boolean variable per property. We will not use bitwise operations on SOFT153.



```
1 class Programm
2 {
3     static void Main() // use C# to evaluate logical expressions
4     {
5         bool a = true, b = false;
6
7         System.Console.WriteLine( !a );           // logical negation (is 'NOT a' true?)
8         System.Console.WriteLine( a && b );        // logical AND (are 'a and b' true?)
9         System.Console.WriteLine( a || b );        // logical OR (is 'a or b' true?)
10        System.Console.WriteLine( (!a || (!b && a)) || false ); // a more complex logical expression
11    }
12 }
13
14 Application Output
15 False
16 False
17 True
18 True
```

Figure 7: Boolean logic in C#

Figure 7 displays examples of logical expressions. First two boolean variables a and b are defined and set to true and false, respectively. The next three lines compute NOT, AND and OR and print the results. Results are shown at the bottom of the figure. Because, a is set initially to true, $!a$ evaluates to false. $a \&\& b$ is also false, because b is false. $a || b$ evaluates to true because one of its components (a) is true. The last line uses a more complex boolean expression. Notice that unless elements are not grouped by parentheses, NOT is evaluated before AND which in turn is evaluate before OR. Therefore, everything in the first set of parentheses $!a || (!b \&\& a)$ is evaluated first and ORed with the explicitly given **false**. The **false** can not make the OR expression **true**, so we can ignore it. The expression in parentheses is also an ORed expression of two sub-expressions, $!a$ and $(!b \&\& a)$. The first of these is **false** because a is initially set to true. The second is **true** because b is false but a is true. Because it is true ORing it with $!a$ also is true and ORing the whole expression $!a || (!b \&\& a)$ with false can not change the result. Therefore the somewhat complex expression in the last line of the figure evaluates to True.

Logic can by times be a bit mind-twisting. However, it is often used in conditional statements to test properties of objects and control the flow of computations based on the results. This is covered in the next few sections.

4 Conditional Statements

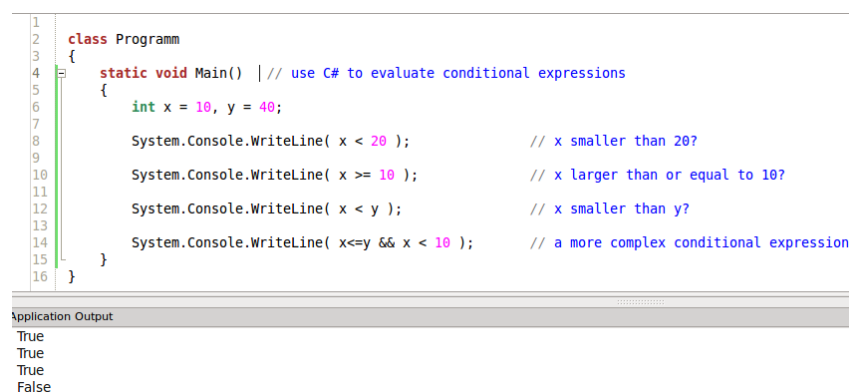
Conditional expressions are expressions that evaluate to boolean “truth” values and not numbers like arithmetic expressions. They can be pure boolean expressions that combine logical variables using AND, OR, NOT and perhaps other logical operators (see previous section 3.2) or combine these and arithmetic expressions with comparisons.

Conditional expressions appear in *conditional statements* which guide the flow of processing through a program. It may for example be that at some moment in time a condition may be either satisfied (true) or not (false) and that depending on the result different elements of code have to be executed. A conditional statement (the “if-else-statement”) would be used to implement this.

4.1 Conditional expressions

Conditional expressions are expressions that evaluate to boolean “truth” values. There are different possibilities. Any boolean expression is a conditional expression; these have been described in section 3.2. Comparisons done on arithmetic, character or string variables also result in truth values. C# has further operators that result in truth values and that may apply to objects; since we are not using object oriented programming on SOFT153 the latter are left out here.

Comparisons between basic types, however, are very important. There are several comparison operators for number types, e.g., tests for equality (==), smaller than (<), smaller or equal (<=), larger than (>), larger or equal (>=). These can be used in conjunction with numerical variables and values to form conditional expressions. Figure 8 shows some examples.



```
1 class Programm
2 {
3     static void Main() // use C# to evaluate conditional expressions
4     {
5         int x = 10, y = 40;
6
7         System.Console.WriteLine( x < 20 );           // x smaller than 20?
8
9         System.Console.WriteLine( x >= 10 );          // x larger than or equal to 10?
10
11        System.Console.WriteLine( x < y );            // x smaller than y?
12
13        System.Console.WriteLine( x <= y && x < 10 );  // a more complex conditional expression
14    }
15 }
16 }
```

Application Output

True
True
True
False

Figure 8: Conditional Expressions in C#

In Figure 8, first two int-variables, x and y, are declared and assigned values of 10 and 40, respectively. The next three lines show quite simple conditional expressions and the output at the bottom of the figure shows the logical values of the expressions. Apparently, if $x = 10$ it is “True” that x is smaller than 20 and also “True” that it is larger or equal than 10 (because 10 is equal to 10). $x < y$ also evaluates to “True” because x is 10 and y is 40, and 10 is apparently smaller than 40. If the numbers were differently chosen the conditional expressions could obviously evaluate to different results.

The last example in Figure 8 shows that conditional expressions can be combined using logical operators. This is a very common situation. In words the example reads

“Is x smaller or equal y AND is x smaller than 10?”. In the present example x is smaller or equal than y, because 10 is smaller than 40; this part of the AND is therefore true. However, x is not smaller than 10 (it is equal to it). Therefore the second clause in the AND expression is false, and that implies that also the AND-expression itself is false. This is confirmed as printed in the last line in the “Application Output” box.

4.2 The if-statement

Conditional expressions are most commonly used in conditional statements. These control the flow of computations in code. Depending on the values of certain variables it may be required to do different things.

The most basic conditional statement is the if-statement. It checks a condition and executes a block of code if that condition evaluates to true. The syntax is

```
if ( <conditional expression> )
{
    // do something
}
```

`<conditional expression>` is any valid conditional expression as explained above (see, e.g., Figure 8). Code to be conditionally executed has to be placed inside the curly brackets. At the moment the block only contains a comment. In principle it can contain more or less complex code including further conditional statements. This code would only be executed if the conditional expression in the if-clause evaluates to true; otherwise it is skipped. If the block contains only a single statement the curly brackets can be omitted. Figure 9 shows examples.

The first example uses a single boolean variable to decide whether or not the first string is printed; in a more complicated program that variable could depend on user input or other events in the code. Because the body of the if-statement consists of a single statement curly-brackets are not needed to define the block of statements that are governed by the if-clause.

The second example uses a mix of conditional and logical expressions. It also uses three print-statements such that curly brackets are needed to delineate where the if-block starts and where it ends.

4.3 The if-else-statement

It often happens that depending on some condition one has to either execute some lines of code if the condition is true, or other lines if it evaluates to false. The if-else statement does this. The syntax is

```

2  class Programm
3  {
4      static void Main()    // use C# to evaluate conditional expressions
5      {
6          bool x = true;
7          int y = 10;
8
9          if ( x )           // every boolean expression is a conditional expression
10             System.Console.WriteLine( "The first condition is True" );
11
12             if ( ( x || !x ) && ( y > 0 ) ) // a more complicated conditional
13             {
14                 System.Console.WriteLine( "The second condition is also True" );
15                 System.Console.Write( "because y is positive and " );
16                 System.Console.WriteLine( "( x || !x ) is " + ( x || !x ) );
17             }
18         }
19     }

```

Application Output

The first condition is True
The second condition is also True
because y is positive and (x || !x) is True

Figure 9: Two if-statements

```

2  class Programm
3  {
4      static void Main()    // use of an if-else statement
5      {
6          bool x = true;
7
8          x = x || !x && x && x; // guess the result!
9
10         if ( x )           // test the result
11             System.Console.WriteLine( "The condition is True" );
12         else
13             System.Console.WriteLine( "The condition is False" );
14     }
15 }
16
~
~

```

Application Output

The condition is True

Figure 10: Example of an if-else-statement


```

if ( <conditional expression> )
{
    \\ code if the expression is true
}
else
{
    \\ code if the expression is false
}

```

The first part is the same as in the if-statement: if the conditional expression is true the first block of code will be executed. However, if it is false, that block will be discarded and the second block starting with “else” will be executed instead. Figure 10 shows an example. Note that the curly brackets can again be omitted if the if- or the else-block have only a single statement.

4.4 The else-if-statement

It can happen that more than one or two code alternatives are possible at a certain location in a program. In that case else-if-blocks can be added to the if- and if-else-statements. The syntax is

```

if ( <conditional expression 1> )
{
    // block 1
}
else if ( <conditional expression 2> )
{
    // block 2
}
else if ( <conditional expression 3> )
{
    // block 3
}
else
{
    // else - block
}

```

There can be an arbitrary number of else-if-blocks; the example shows two of them. The final else-block can be missing. During execution of the code the conditional expressions are consecutively checked until a condition evaluates to true. The corresponding code block is then executed, and all later else-if- or else-blocks are discarded. Figure 11 displays

```
1  class Program
2  {
3      static void Main()
4      {
5          int x = 15;
6
7          if (x<10)
8              System.Console.WriteLine( "x is smaller than 10");
9          else if ( x < 20 )
10             {
11                 System.Console.WriteLine( "x is larger than or equal to 10");
12                 System.Console.WriteLine( " but smaller than 20");
13             }
14          else if ( x < 30 )
15             {
16                 System.Console.WriteLine( "x is larger than or equal to 20");
17                 System.Console.WriteLine( " but smaller than 30");
18             }
19          else
20              System.Console.WriteLine( "x is larger than or equal to 30");
21      }
22  }
23  }
```

Application Output

```
x is larger than or equal to 10
but smaller than 20
```

Figure 11: Code using else-if clauses

an example where an integer variable x is set to 15. Subsequently the variable is tested if it falls into various number ranges and a message is printed accordingly. Observe that some of the blocks contain only one statement such that curly brackets are not required (but they could be used, for example, if that increased readability of the code).

Observe also that after the first if-clause is tested it is clear that x is not smaller than 10. This information does not have to be checked again in the else-if clause following the if-clause. The condition of that clause is true, such that we know that x is larger than or equal to 10 and smaller than 20. The conditional expressions in the if- and else-if-clauses can be more complicated than shown in the example; every valid conditional expression is possible. In practice one would usually try to make the conditions not too complex in order to keep code easily understandable.

4.5 The switch-statement

It often happens that code has to be conditionally executed based on some integer valued decision variable, for example if a user selects an item from a menu, and different items require different responses. This is best done using the so-called switch-statement. The syntax is shown in the example below where an integer “caseSwitch” is set to some value; afterwards a switch-statement executes different code depending on the value of the caseSwitch variable.

```
int caseSwitch = 1;

switch ( caseSwitch )
```

```

{
    case 1:
        System.Console.WriteLine("Case 1");
        break;

    case 5:
        System.Console.WriteLine("Case 5");
        System.Console.WriteLine("Case 5 has two statements");
        break;

    default:
        System.Console.WriteLine("Default case");
        break;
}

```

During execution of the code the values of `caseSwitch` is successively checked against the values of the case-statements. If a match is found the respective code is entered and executed. Note that even if there is more than one statement to execute as in case 5, curly brackets are not required to outline where a block of code starts and where it ends. It always starts at a case-line and it always ends at the next “break”. This implies that if a case block misses a break, the control flow will “fall through” to the next case and also execute this code. It executes all code down to the next break-statement.

In the example there are two cases “1” and “5” that can cause possible matches. If `caseSwitch` does not have either of these values the “default”-block at the bottom is executed. This default case can be missing in which case no code is executed at all if none of the cases matches.

Finally note that the same code can have several case-clauses. Something like the following is perfectly legal. It would be equivalent to an if-statement that prints the text if `caseSwitch` is either 1 or 2.

```

int caseSwitch = 2;

switch ( caseSwitch )
{
    case 1:
    case 2:
        System.Console.WriteLine("Case 1 or 2");
        break;

    // ... more cases ...
}

```

5 Loops

Beside conditionals, loops are a major programming construct. They are used to do things repetitively either for a fixed number of time or as long as a certain condition holds.

5.1 The for-loop

This is a loop that is typically used to execute some code for a fixed number of time. In its simplest case it needs a counter variable that keeps track of how often the loop has already been executed and a condition when to break the loop. It may look like this:

```
for ( int i=0; i<10; i++ )
{
    // do something
}
```

The word “for” is a keyword that starts the for-loop just as “if” starts an if-statement. Also as for if-statements the for-loop has a body of code to be executed. Here the body contains a single comment only. If it contains only a single statement the curly brackets are not required (but they can’t hurt either).

In the head of the loop, an integer variable i is defined and initialised to 0. This is a counter-variable that keeps track of how often the loop has been executed. Counter variables are often names i , j , or k , but more meaningful names can sometimes make code better understandable. The next statement in the header $i<10$ states that the loop is only to be executed as long as i is smaller than 10. This is tested before the loop is entered and each time it is re-entered. The final statement in the header $i++$; increments i by one; this is a common shortcut for $i=i+1$;. This statement is executed after all code in the code-block of the loop and before the exit-condition is checked the next time.

So, if the code as shown above is executed, first a counter variable i is allocated and set to 0. Then the exit condition is checked and if it is true the code block is entered and executed. In our case there is no code only a comment, so nothing is done. Then i is incremented, and the exit-condition is checked again. If it is still true the code block is executed again, but now with i equal to 1. After the block has been executed i is further incremented by one and the process goes on, until i is finally set to 10 in which case the exit condition signals false, the code block is not entered again and the loop terminates.

Again note that i in each iteration has a different value. The code to be executed may or may not make use of this. For example, i could be used as an index into an array of objects such that each loop-iteration operates on a different array element. More about this later in the section about arrays, section 7.1.

The example above displays the most common use of a for-loop. The for-loop is more powerful than the example shows. The initialisation, exit condition, and update of the

```

1  class Program
2  {
3      static void Main()
4      {
5          int n=10, i;
6          for ( i=2, i*=3;           // init
7              i<2*n && i>5;         // exit
8              System.Console.WriteLine( i ), i+=2 ) // update
9              // empty body
10             ;
11     }
12 }

```

Application Output

```

6
8
10
12
14
16
18

```

Figure 12: Bad and confusing use of a for-loop

counter variables can be (much) more complicated. The code in Figure 12 shows a perhaps confusing example that should be considered bad practice:

At first the init-clause shows that the counter can be declared outside the for-loop. This is ok. The clause also shows that the counter does not need to start at zero. This is also ok. It should however be set in a transparent way. In the example we use a compound statement (separated by a comma) that first sets i to 2 and then multiplies it by three. The initial value achieved this way is therefore 6. One should usually avoid complicated initialisations of this kind.

The exit condition in Figure 12 is also more complex than comparison for a simple upper bound. This can be ok, but the condition should not be made too difficult.

The update clause in the figure is entirely bad practise because it calls a print function that prints out the current value of the counter variable. Afterward the variable is incremented by 2. The latter can make sense, but the former should be avoided. The print statement should rather be put inside the body of the for-loop, which is empty in the example shown. Still something is printed because the update clause (and the exit clause) are computed during each iteration.

This is an example of how to write bad code. Best use for a for-loop is simple counting between fixed boundaries given by some variables or explicit numbers. It can make sense to use other than increments by one including negative ones in which case the loop counts downward and the exit-condition has to be chosen accordingly, see Figure 13.

5.2 The while-loop

The best use of the for-loop is when the number of iterations required is known or can be computed easily. This is not always the case. It can happen that only a condition is given that has to be repeatedly checked in order to decide whether the loop has to be executed again.

```
1  class Program
2  {
3      static void Main()
4      {
5          int start = 20, end = 10, increment = -2;
6          for ( int j = start; j >= end; j = j + increment )
7              System.Console.WriteLine( j );
8      }
9  }
```

Application Output

```
20
18
16
14
12
10
```

Figure 13: Downward counting for-loop

A loop that does this is the ‘while-loop’. Its syntax is

```
while ( <conditional expression> )
{
    // do something
}
```

The word “while” is again a keyword that starts the loop and the curly brackets include a block of code that is to be executed in each iteration of the code. Conditional expressions we know already from earlier sections. As long as the expression evaluates to true the loop is re-entered. This can actually quite easily lead to infinite loops that never exit, like this:

```
while ( true )
    System.Console.WriteLine( " .. and again ..." );
```

This code endlessly prints “... and again ..” to the console because the condition that decides if the while-loop is (re-)entered is always true. In this case the infinite loop is easy to spot but in more complicated cases it can be quite difficult to check whether a while-loop ever terminates.

Another example could be to print all square numbers between 1 and 100. Because we don’t know how many numbers this are we may want to use a while-loop like this:

In the example in Figure 14 the condition to check in the while-clause comes in naturally. Closer inspection of the code reveals that k is indeed used as a counter variable. The while-loop could therefore be easily transformed into a for-loop as shown in Figure 15. Such a conversion, however, is not always simply. It is often better to use while-loops, especially if it is not known how many iterations are required but this number is only implicitly given by some conditional expression.

```
1  class Program
2  {
3      static void Main()
4      {
5          int k = 1;
6          while ( k*k <= 100 )
7          {
8              System.Console.WriteLine( k*k ) ;
9              k = k+1;
10         }
11     }
12 }
```

Application Output

```
1
4
9
16
25
36
49
64
81
100
```

Figure 14: Code to compute square numbers using a while-loop

```
1  class Program
2  {
3      static void Main()
4      {
5          for ( int k=1; k*k <= 100; k++ )
6              System.Console.WriteLine( k*k ) ;
7      }
8  }
```

Figure 15: Square numbers computed using a for-loop

For completeness we should mention that C# also has a do-while-loop. This has the form

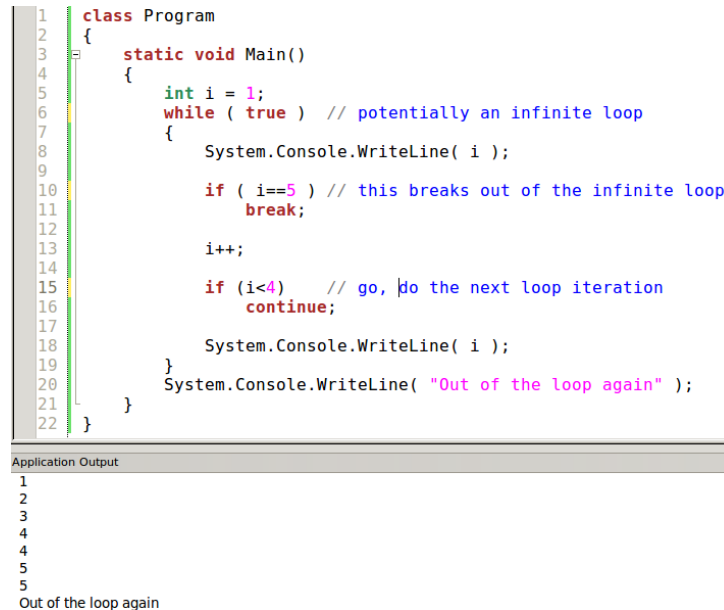
```
do
{
    \\ do something
} while ( <conditional expression> );
```

Whereas the condition of the while-loop is checked at the beginning of the code-block, it is checked in a do-while-loop at the end. As long as the conditional expression is true the code is repeated. This has the major difference that the code-block in the do-while-loop is always at least executed once, whereas in the while-loop the code-block may never be entered if the entry condition is always false.

5.3 Breaking loops – ‘break’ and ‘continue’

It is possible to break loops while executing their code block. This holds for all loops mentioned, for-, while-, and do-while-loop. We only demonstrate it using while-loops.

The “break” statement breaks out of the innermost loop currently executed and proceeds after that loop. The “continue” statement also breaks the loop but in contrast proceeds at the top of the loop testing the enter condition again.



```
1 class Program
2 {
3     static void Main()
4     {
5         int i = 1;
6         while ( true ) // potentially an infinite loop
7         {
8             System.Console.WriteLine( i );
9
10            if ( i==5 ) // this breaks out of the infinite loop
11                break;
12
13            i++;
14
15            if ( i<4 ) // go, do the next loop iteration
16                continue;
17
18            System.Console.WriteLine( i );
19        }
20        System.Console.WriteLine( "Out of the loop again" );
21    }
22 }
```

Application Output

```
1
2
3
4
4
5
5
Out of the loop again
```

Figure 16: The break- and continue-statements in action

Figure 16 shows an example that uses the break- and continue-statements. The example has a possibly infinite while-loop (because the condition is always satisfied). However, it also has a counter variable i that controls the flow of execution. Each time the loop is entered the value of i is printed. When i reaches a value of 5 the control flow breaks out of the loop (see lines 10/11). Otherwise i is incremented by one (line 13). As long as i stays smaller than 4 the continue statement in line 16 throws the control flow back to the head of the while loop where the loop is entered again because the while-clause is “true”. If i grows larger than 3 the continue statement is no longer executed, because it is blocked by the if-clause. Instead the value of i is printed again. Therefore in iterations 4 and 5 i is printed twice, whereas it is only printed once in the earlier iterations. If the loop is broken when i reaches the value 5 the break-statement in line 5 forces the control flow to resume after the block of the while loop. Then the print statement in line 20 is executed after which the program terminates.

Figure 17 shows an example that uses two nested loops. This example demonstrates that “break” (and also “continue”) only break out of the innermost loop.

The Main function contains a while-loop starting at line 6 and ending at line 20 with


```

1  class Program
2  {
3      static void Main()
4      {
5          int i = 1;
6          while ( i < 4 ) // outer while-loop
7          {
8              System.Console.WriteLine( "i = " + i );
9
10             int j=1;
11             while ( true ) // inner while-loop
12             {
13                 System.Console.WriteLine( " j = " + j );
14
15                 if (j==2)
16                     break;
17                 j++; // inner loop counter
18             }
19             i++; // outer loop counter
20         }
21         System.Console.WriteLine( "Out of the loop again" );
22     }
23 }

```

Application Output

```

i = 1
j = 1
j = 2
i = 2
j = 1
j = 2
i = 3
j = 1
j = 2
Out of the loop again

```

Figure 17: Breaking a nested loop

another while-loop inside (i.e., “nested”) that starts at line 11 and ends at line 18. Both loops have a counter variable, *i* and *j*, respectively. The second loop is potentially infinite because the loop-head contains just “true” as the condition for re-entrance. However, in line 15/16 *j* is checked if it has reached a value of two, and if yes, the (inner) loop is broken (exited). It can be seen in the Application Output that the inner loop exits after 2 iterations, but not the outer loop. This still iterates as many times as the loop header specifies. If the outer loop needs to be broken another break-statement is required within that code block as well.

```

1  class Program
2  {
3      static void Main()
4      {
5          int[] array = new int[] { 5, 1, 0, 3, 5 };
6          foreach ( int i in array )
7          {
8              System.Console.WriteLine(i);
9          }
10     }
11 }

```

Application Output

```

5
1
0
3
5

```

Figure 18: Example of a foreach-loop

5.4 The foreach-loop

C# has another loop construct which can also be found in a number of other programming languages, the "foreach-loop". This takes an array or data collection and executes a code block on all its elements. An example is shown in Figure 18.

In line 5 of 18 an array of integers is defined (see section 7.1) and the foreach-loop goes from line 6 to 9 initiated by the keyword "foreach". Because the array is an integer array a variable *i* of type int is declared to iteratively take on all values in the array. The values are just printed in the code block of the loop.

As other loops the foreach-loop allows to use the break- and continue-keywords. The break keyword stops iterating through the collection of data and the continue keyword proceeds straight away to the next element.

In SOFT153 foreach loops should be avoided because they hide mechanisms of how the looping is actually performed behind the scene. It may even be dangerous to use a foreach-loop in cases where the order of the elements in the collection matters. In SOFT153 for- and while-loops are entirely sufficient.

6 Functions and Procedures

Simple programs can be coded in one simple Main-routine. However, as problems become more complicated it will become more and more clumsy to code an entire project in a single Main-routine. It is then common to break down the functionality required to solve the problem into functions and procedures for sub-problems. Functions and procedures are also called "methods" in the context of object-oriented programming. Because we don't cover Oop in SOFT153 we prefer to speak of functions and procedures.

Functions and procedures encapsulate certain functionality in a program. They take some values or data and compute something with the data. They can either change the data in that course, or they can return a value. The main difference between functions and procedures is that the latter don't return values, but the former do. Some people do not make the distinction at all but use the terms interchangeably.

Functions and procedures need to be defined inside a class before they can be used. In the previous example we always had already one function - the Main()-routine. This is actually a procedure, because it does not return a value. We can define as many functions and procedures as we like.

Lets again assume we want to print square numbers. Instead of a single Main-routine as earlier we can define a function that takes a number and computes its square. This is how the function definition could look.

The red "static" keyword relates to how the function is stored and processed on the computer. SOFT153 does not deal with these issues. The black string "square" is the

```

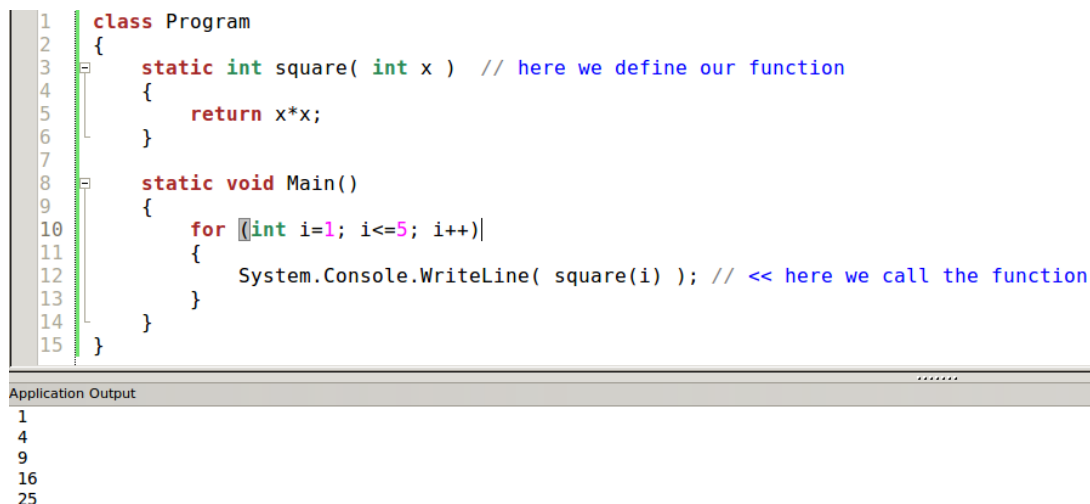
static int square( int x )
{
    return x*x;
}

```

Figure 19: Definition of a function that squares numbers

name of the function which has to be unique and should be informative about what the function is supposed to do. The "int x" inside brackets declares that the function takes an argument called x of type int. This argument is available as a variable inside the function body, check the `x*x` in line 3. Arguments need a type (here int) and a name (*x*). `x*x` in line 3 means the square of *x* when understood as an arithmetic expression. This is what we wanted to have the function compute. The "return"-keyword means that the function returns a value (to the code that calls it, see below). This value is of type int and this is further indicated by the green "int" in front of the function name "square".

After having defined the function we can use it in the Main-routine or elsewhere. Lets compute the squares of the integer numbers 1 to 5. Here is the code; it is obvious to use a for-loop for this. Lines 3 to 6 in Figure 20 repeat the function definition. Note that this



```

1  class Program
2  {
3      static int square( int x ) // here we define our function
4      {
5          return x*x;
6      }
7
8      static void Main()
9      {
10         for (int i=1; i<=5; i++)
11         {
12             System.Console.WriteLine( square(i) ); // << here we call the function
13         }
14     }
15 }

```

Application Output

```

1
4
9
16
25

```

Figure 20: Definition and repeated call of a function that squares numbers

is just a second function beside the Main-routine. We can define more if we need them. Lines 8 to 13 is the Main-routine. Notice the "void" return type of that routine. This indicates that the Main-routine does not return a value. We can define other procedures that do not return anything, which, however, could still modify their arguments or do other work. The Main routine counts from 1 to 5 and calls our square-function using the counter variable as the function argument, `square(i)`. This means the square-function gets successively called with different values for its argument (called *x* in the function definition, but the names do not need to match). Because the function returns a result, the square of its argument, that result will be available to the WriteLine-routine which

```
static int multiply ( int x, int y )
{
    return x*y;
}
```

Figure 21: A function that takes two arguments x and y of type integer.

then prints it out to the console. We could alternatively assign the result to another variable or use it immediately in an arithmetic or conditional expression. The function can be used everywhere, where an integer or an integer variable could be used.

Functions can have more than one argument, but only one return value (only few programming languages allow to have more than one return value). The return value can, however, be an array or a class (see later); it does not have to be a simple data type. If more than one argument is needed they have to be defined sequentially with type and names separated by commas, for example as shown in Figure 21. The arguments do not have to be of the same types.

6.1 Call-by-value

In the discussion of Figure 20 you will have noticed that the argument name x of the square-function and the variable it is actually called on, the counter variable i in line 12, are not the same. This points at the problem what actually happens, when a function is called. In the situation shown where the argument is a simple integer the call uses a mechanism named "call-by-value". This means (without going into too much detail) that the value of the variable used in the call is copied into the argument of the function. The name inside the function (x) refers to a completely different location in memory than the variable the functions was called on (i). Therefore if we would change the value of x inside the function, the value of i would not be touched at all. In fact all changes to x are lost when we leave the function again. This is further illustrated in Figure 22.

The code defines a function `set_x_to_five` that prints the value of its argument x , sets x to 5 and prints the value again (lines 3 to 8). In the Main-routine a variable x is defined and set to 2. The value is printed and the previously defined function called. After the call the value of x is printed again.

What can be seen in the Application Output is that the call to the function does not change the value of x in the Main-routine at all. However, inside the `set_x_to_five()`-function the value x is changed. This, perhaps confusing behaviour is due to the call-by-value principle. The variables x in the function and the Main-routine are completely different entities. They have the same name, but refer to different locations in memory; we could give them different names, but that would not change anything. When the program is executed, the code is entered in the Main-routine and the x -variable there is assigned a value of 2 which is then printed. When the function `set_x_to_five(x)` is called only the value of x (which is 2) is copied into the location of the variable x in the

```
1  class Program
2  {
3      static void set_x_to_five( int x )
4      {
5          System.Console.WriteLine( "Inside function: " + x );
6          x = 5;
7          System.Console.WriteLine( "Inside function: " + x );
8      }
9
10     static void Main()
11     {
12         int x = 2;
13         System.Console.WriteLine( "In Main : " + x );
14         set_x_to_five( x );
15         System.Console.WriteLine( "In Main : " + x );
16     }
17 }
```

Application Output

```
In Main : 2
Inside function: 2
Inside function: 5
In Main : 2
```

Figure 22: Illustration of call-by-value

function, x afterwards contains a copy and prints it as 2. It then changes the value of the new x variable inside the function call to 5 and prints it correctly. Then the function returns. It does not return a value, it just gives the control flow back to the Main-routine. All variables defined locally in the function are invalidated. The transient change of the x -variable inside the function is not transferred back to the x -variable defined in the Main-routine. The two are completely different things. Therefore the x -variable in the Main-routine still has a value of 2 after the function call.

To summarise: arguments of functions that are of simple types (int, float, double and similar) are copied "by-value" into a function's arguments when it is called and these new copies are available only locally inside the function. Any changes get lost when the control returns to the code that called the function.

6.2 Call-by-reference

It often happens that one wants to keep changes done to arguments in a function. In this case one has to use another mechanism to transfer arguments into (and out) of functions. This mechanism is named "call-by-reference".

Each variable, simple or complex, occupies space somewhere in memory. The numbers actually stored make out the value of the variable (or values, if the data type is not simple, but a collection or array). The space in memory also has a "reference" that the runtime environment knows and can use to actually find it. Instead of making copies of values into arguments when a function is called one could also copy the references over. The called function then can find the original variables and can operate on them directly. This would update the original variables and the changes would persist when the function terminates.

```

1  class Program
2  {
3      static void set_x_to_five( ref int x ) // <<< note the ref keyword
4      {
5          System.Console.WriteLine( "Inside function: " + x );
6          x = 5;
7          System.Console.WriteLine( "Inside function: " + x );
8      }
9
10     static void Main()
11     {
12         int x = 2;
13         System.Console.WriteLine( "In Main : " + x );
14         set_x_to_five( ref x ); // <<< note the ref keyword
15         System.Console.WriteLine( "In Main : " + x );
16     }
17 }

```

Application Output

```

In Main : 2
Inside function: 2
Inside function: 5
In Main : 5

```

Figure 23: Illustration of call-by-reference; compare with Figure 22.

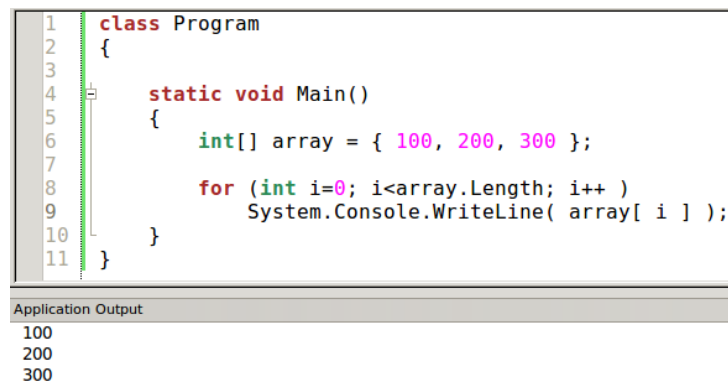
It is necessary to explicitly tell C# if a simple argument is supposed to be call-by-reference. This makes use of the "ref"-keyword. An example is shown in Figure 23. The code in Figure 23 is almost the same as that in Figure 22 with the exception that the argument to the function is now declared as reference-type. Likewise it must be called using a ref-keyword, cf., line 14. The Application Output shows that the code now does what we would expect. The value of x in the Main-routine is first printed as 2. The reference is passed to the `set_x_to_five` function and increased properly to 5. After returning from that function the x -variable in the Main-routine has also changed to 5. This is because the `set_x_to_five`-function did not deal with a copy of the value of that variable but with the memory space the variable was actually stored in. In other words the x variables in the function and the Main-routine referred to the same object in memory and not different ones as before.

Note that the situation is more complicated for arrays, records and classes. Some of these are by default called by value (records) and others by reference (arrays and classes). Even more confusingly, the rules are different in different programming languages (like Java). More about this later.

7 Compound Data

Integers, floating point numbers, boolean variables and characters are simple data types that hold a single value. Much more common are more complex types that consist of collections of data. These can be homogeneous, i.e., all elements in the collection are of the same type, or heterogeneous in which case a data type can comprise various individual entries of different types.

"Arrays" are typically used to implement fixed-size homogeneous data collections, whereas "records" or "classes" implement heterogeneous compound data types.



```
1  class Program
2  {
3
4      static void Main()
5      {
6          int[] array = { 100, 200, 300 };
7
8          for (int i=0; i<array.Length; i++ )
9              System.Console.WriteLine( array[ i ] );
10     }
11 }
```

Application Output

100
200
300

Figure 24: Integer array with inline-initialisation

7.1 Arrays

Arrays come in different formats, they can be one- or two-dimensional, or of even higher dimension. A monaural sound file, for example, could contain a one-dimensional array of sound samples of a certain size (e.g., 16 bit). A picture, in contrast, would more typically be stored in a two-dimensional array of "pixels". Other data may have a different natural dimension. C# supports arrays of arbitrary dimension.

The Elements in an array have all to be of the same type. That does not necessarily has to be a simple type like `int` or `double`. Arrays can also be build from other arrays or from records and classes. More about this in section 7.6. In the present section we consider integer and string arrays only.

7.2 Number Arrays

Figure 24 shows how an integer array can be defined and initialised in C#. Line 6 declares an array of name "array" (not a good name in general, but informative here because we are learning about arrays). The type of the array is "int" and the square brackets after the int indicate that the variable is supposed to hold an array. The array is initialised on the same line by a couple of numbers. The array is one-dimensional and contains three entries.

Array elements can be accessed by the array name followed by an index in square brackets. Indexes start at zero. Therefore, `array[0]` accesses the first element in the array and `array[2]` the last one (one less than the length of the array, because we start counting at zero). References to array elements can be used like any other variable. Values can be assigned or read out and printed by using this form of referencing by indexing. Figure 24, for example, iterates through the array using a for-loop accessing its elements using the counter variable as an index into the array, and printing out the values stored in the individual locations.

```
1  class Program
2  {
3
4      static void Main()
5      {
6          int[] values = new int[3];
7
8          values[0] = 5;
9          values[1] = 2*values[0];
10         values[2] = 4*values[1];
11
12         for (int i=0; i<values.Length; i++ )
13             System.Console.WriteLine( values[ i ] );
14     }
15 }
```

Application Output

```
5
10
40
```

Figure 25: Integer array with explicit initialisation.

Figure 25 shows that values can be assigned to array elements in the same way. It also demonstrates a different way of allocating space for an array: In line 6 the expression `new int[3]` creates a new array large enough to hold three integers somewhere in memory, which is assigned to the array-variable using the equal-sign. We typically do not need to know where precisely the memory space is located, because we can access its elements by indexed referencing. This is shown in lines 8-10 in the figure where the elements successively get values assigned. Note again that expressions like `array[1]` or `array[i]` can be used to assign values as well as to retrieve them. In Figures 24 and 25 a for loop is used to iterate through the elements in the array and print their values. Note that this makes use of a method defined for arrays called “Length”; `array.Length()` returns the number of elements in the array.

If an array of a different type than “int” is required the type specifiers in the code has to be adapted accordingly, e.g., from “int” to “double”. This may also require to use different initialisers. The compiler will complain about possible mistakes.

7.3 String Arrays

Figure 26 shows an example of a one-dimensional array of strings. This example is quite similar to the one using integers in Figure 25.

In some languages strings are implemented as arrays of characters, such that string arrays may appear as arrays of arrays. This however, is not entirely correct. The string array usually only contains references to the strings and not the memory space where the data is actually stored. Figure 27 shows that the strings can nonetheless be indexed like arrays in order to access the individual characters in them. Character 3 (counting from 0 onwards) in the zeroth word is an ‘n’ as correctly printed in the Application Output.


```
1  class Program
2  {
3
4      static void Main()
5      {
6          string[] values = new string[3];
7
8          values[0] = "do";
9          values[1] = "not";
10         values[2] = "lie";
11
12         for (int i=0; i<values.Length; i++ )
13             System.Console.WriteLine( values[ i ] );
14     }
15 }
```

Application Output

```
do
not
lie
```

Figure 26: Use of a string array

```
1  class Program
2  {
3
4      static void Main()
5      {
6          string[] a = {"twenty", "little", "ducks" };|
7
8          System.Console.WriteLine( a[0][3] );
9      }
10 }
```

Application Output

```
n
```

Figure 27: Access to individual characters in strings in an array

7.4 Two- and Higher-dimensional Arrays

It is straightforward to define two- and higher-dimensional arrays. Examples are shown in Figure 28 where lines 6, 7, and 8 define 2, 3 and 4-dimensional arrays, respectively. The number of places in the type modifier after the base-type “int” specifies the dimension, i.e., `int []` defines a one-dimensional array type, `int[,]` a three-dimensional array type and so on. Arrays are to be given names which declares them, but they do not yet exist in memory. For example, the array *d3* in line 7 is declared but never assigned any memory. This can be done inline (similar as in Figure 24), but more commonly for higher-dimensional arrays using the `new` keyword. The arrays *d* and *d4* get memory assigned in lines 6 and 8. After the `new`-keyword the base type is repeated and then a list of requested elements in each dimension provided. The number of dimensions must match that of the initially requested array-type. The number of elements in each dimension is usually set by some requirements of the code.

The array *d4* in Figure 28 is actually never used again. C# will complain about this. In lines 10 to 12 of the code the array *d2* is initialised; its values are set to some numbers; if an array is allocated for the first time they are all zero. Note that access to an array

```

1  class Program
2  {
3
4      static void Main()
5      {
6          int[,] d = new int [ 3, 4 ];
7          int[,] d3;
8          int[,,,] d4 = new int [ 1, 4, 5, 3 ];
9
10         for ( int i=0; i<3; i++ )
11             for ( int j=0; j<4; j++ )
12                 d[i,j] = 10*i+j;
13     }
14 }

```

Figure 28: Two and higher-dimensional arrays.

element is of the form $d2[i,j]$ where i and j are indexes of rows and columns in the two-dimensional array. They uniquely address a single location in the array. In higher dimensions the number of indexes required would accordingly be larger.

Arrays with dimensions larger than two will not be required in SOFT153.

7.5 Arrays as Function Arguments

Arrays can be used as function arguments. This is demonstrated in Figure 29.

```

1  class Program
2  {
3      static int array_function( int[] array )
4      {
5          array[2] = 10;
6          return array[0] + array[1];
7      }
8
9      static void Main()
10     {
11         int[] a = { 3, 6, 9 };
12         System.Console.WriteLine( array_function( a ) );
13         System.Console.WriteLine( a[2] );
14     }
15 }

```

Application Output

```

9
10

```

Figure 29: Definition and use of a function with an array argument

The type of the function argument in the function definition (line 3) needs to reflect the basetype (int in Figure 29) and the proper dimension (1 in the example; expressed as `[]`). A function can of course have more than one array argument. The example creates a one-dimensional array in the Main-routine holding 3 values (3, 6, 9) in line 11 and calls the function in line 12. Note that the function returns a value (line 6) which can therefore be printed when the call to the function in line 12 returns. This prints a '9' in

the Application Output at the bottom of the figure, because `a[0]` is 3 and `a[1]` is 6 and the function returns their sum.

It is important to know that array arguments are by default call-by-reference, it is not a whole copy of an array that is given to a function, but only a reference to the memory space that holds the original data. Therefore changes made inside the function using the array variable will persist when the function returns. In line 11 the element `a[2]` is set to 9, but inside the function (line 5) it is reset to 10. Because this change is made to the original array (and not a copy of it) the change persists after the function exits. This can be verified by printing the value of `a[2]` in line 13. The Application Output shows that this is 10 as expected.

7.6 Records and classes

Arrays hold homogeneous data, each element of an array has the same data type. It is often desired to define data types with elements of different types. For, example a student record could contain a first and last name, a student ID, age, height, course taken, and the like. These are partly numbers, partly strings. Using an array to hold the data would be inappropriate.

A solution to this problem are records and classes. Their implementation and use differs quite a bit between various programming languages. We will not be going into the details; these are taught in modules about object-oriented programming.

```
class student
{
    public string firstname;
    public string lastname;
    public int studentID;
    public int age;
}
```

Figure 30: A student record defined as a class

Instead we will only make use of the most simple constructs of classes in order to define records with heterogeneous entries. Figure 30 gives an example how a student record could be defined as a class. “class” is a keyword that initiates the definition and “student” is the name of the new class defined. The four lines inside the block of the definition each specify a “field” of the class; there can be an arbitrary number of fields. The “public” keyword refers to how a field can be accessed; this is not of much concern in SOFT153; we define all fields as public. The green strings in the definition are the types of the fields and the black strings their names. Note that the types are different, such that the record is heterogeneous. The field names, as usual, should be meaningful.

After it has been defined a class can be used just as a new type. Figure 31 shows how an instance of student can be created in a program. Lines 1 to 7 define the class

```

1  class student
2  {
3      public string firstname;
4      public string lastname;
5      public int    studentID;
6      public int    age;
7  }
8
9  class Program
10 {
11     static void Main()
12     {
13         student s = new student();
14
15         s.firstname = "John";
16         s.lastname  = "Dow";
17         s.studentID = 12345;
18         s.age       = 20;
19     }
20 }

```

Figure 31: Access to fields in a class using the dot-operator (‘.’)

named “student” and line 13 creates a new instance of it. To access the fields of a class the dot-operator (‘.’) is used, as in `s.firstname` where ‘s’ is the name of the previously created student record, the dot ‘.’ means “access a field of the class”, and “firstname” is the name of the class to access. In the example we assign the four fields of the class reasonable values; these must be of the type given to a field in the class definition.

It is furthermore possible to use arrays of classes. This is demonstrated in Figure 32.

```

1  class student
2  {
3      public string firstname;
4      public string lastname;
5      public int    studentID;
6      public int    age;
7  }
8
9  class Program
10 {
11     static void Main()
12     {
13         student[] s = new student[10];
14
15         s[3].firstname = "John";
16         s[3].lastname  = "Dow";
17         s[3].studentID = 12345;
18         s[3].age       = 20;
19     }
20 }

```

Figure 32: Creation and use of an array of a class type

The student class is defined in the example in Figure 32 as before, but now line 13 creates a whole array of 10 student records. The syntax is the same as for other arrays, remember that an integer array is created this way: `int[] n = new int[10]`. It is also

possible to create two- and higher dimensional arrays as explained earlier in section 7.1.

Access to fields in an array of records first accesses the array element using brackets and indexes, and then the field-element using the dot-operator. If *s* is an array of student records as in Figure 32, then *s*[3] would access the third instance of student in the array (starting at 0), and *s*[3].*firstname* would access its field-element called “firstname”.

Classes can be used as function arguments. This is shown in Figure 33.

```
9  class Program
10 {
11     static void print_student_data( student s )
12     {
13         System.Console.WriteLine( "First Name : " + s.firstname );
14         System.Console.WriteLine( "Last Name : " + s.lastname );
15         System.Console.WriteLine( "Student ID : " + s.studentID );
16         System.Console.WriteLine( "Age : " + s.age );
17     }
18
19     static void Main()
20     {
21         student s = new student();
22
23         s.firstname = "John";
24         s.lastname = "Dow";
25         s.studentID = 12345;
26         s.age = 20;
27
28         print_student_data( s );
29     }
30 }
```

Application Output

```
First Name : John
Last Name : Dow
Student ID : 12345
Age : 20
```

Figure 33: A class as a function argument (see lines 11 and 28)

Beside the Main-routine in Figure 33 which creates an instance of the student class (line 21) and initialises it (lines 23-26), we also define a function called *print_student_data* in lines 11 to 17 that is supposed to take a student instance and print its data to the console. The argument of the function definition *student s* states that there is one argument of type “student” and name “s”. This variable is available inside the function and can be accessed in the ways described above. The function is called in line 28 of the example and the Application Output at the bottom shows that it works as it should.

One important aspect when using classes as arguments of functions is that they are by default call-by-reference (see section 6.2). When the function is called on an instance of the class the data is *not* copied to the argument variable, but a reference to the space in memory that contains the original data is passed. This implies that changes to any of the fields inside the function will persist when the function finishes. In the example in Figure 33 field elements are only accessed for printing, but not changed. This could be done for example as *s.age*=22 which would assign a new age to the record *s*.

It should finally be mentioned that it is possible to define arrays of classes and also to define classes that contain other classes. The latter is indeed most fundamental to object-oriented programming, but we will not make use of it in SOFT153. We will, however, use arrays of classes. Students that want to go for high marks need to understand how to use them and demonstrate this in the assignments.

```

1  class student
2  {
3      public string firstname;
4      public string lastname;
5      public int studentID;
6      public int age;
7  }
8
9  class module
10 {
11     public string name;
12     public int ID;
13     public student[] students = new student[50];
14 }

```

Figure 34: Classes can contain other classes and even arrays of them

Figure 34 first defines the already known student record (lines 1 to 7) and afterwards another class “module” that contains a field in line 13 which is an array of instances of the student class. If m is an instance of the module class a field of the 12th student instance could be accessed in the following way: `m.students[12].firstname`; this uses the indexing brackets and the dot-operator to successively select the wanted fields and elements in the two nested classes. In words the expression reads: of the module instance m select the students array (using the first ‘.’ and the fieldname ‘students’); of that array select the 12th element (using brackets and the index); and of that element select the field named ‘firstname’ (using the second dot-operator and the fieldname): `m.students[12].firstname`

8 Miscellaneous Left Overs

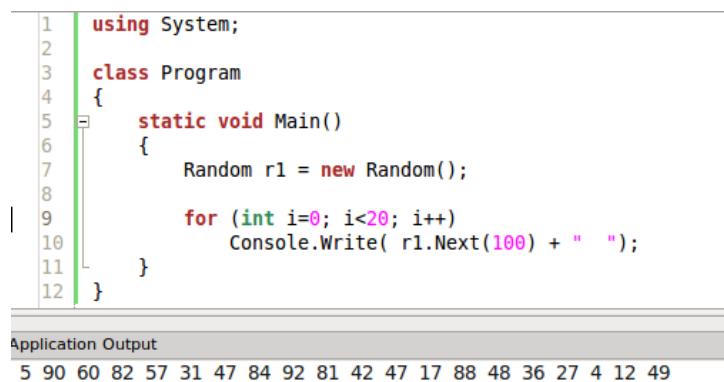
The programming constructs described in the previous sections are almost all that is required in order to follow the content of SOFT153 and to do the assignments. As explained in the introduction the module is about how to implement advanced data structures that make use of the basic concepts described in this document. Students are expected to know most or all of the covered basic constructs already, but not the advanced data structures and algorithms covered in the module. All newer programming languages do provide such advanced structures (like Lists, ArrayLists, or Dictionaries), but in SOFT153 students are not allowed to use these C# in-built structures and functions. The main point of the module is to understand how they work. For that purpose we will implement them from scratch.

A few features of C# will occasionally have to be used in the lectures, practicals, or assignments. Some of them are dealt with in the next few sections. These are, for example, the generation of random numbers, or ways to read from files on the hard drive.

8.1 Random numbers

Many algorithms use “random numbers”, for example encryption algorithms, or code in games that randomly decides about actions to take in order to develop the game in an unexpected way.

C# and other programming languages provide so-called “random numbers generators” that we will sometimes use in SOFT153. This makes use of a class “Random” defined in “System”. To generate random numbers one creates one (or more) instances of the class and uses the Next()-function defined in the System-library to obtain random values. This is shown in Figure 35.



```
1 using System;
2
3 class Program
4 {
5     static void Main()
6     {
7         Random r1 = new Random();
8
9         for (int i=0; i<20; i++)
10             Console.Write( r1.Next(100) + " ");
11     }
12 }
```

Application Output

5 90 60 82 57 31 47 84 92 81 42 47 17 88 48 36 27 4 12 49

Figure 35: Use of a random number generator

In line 7 of the example in Figure 35 an instance of the class “Random” is created. This creates a random number generator, but not yet a single random number. To get numbers a function called “Next()” has to be called as shown in line 10; Next returns a random integer value that can be printed. Because line 9 defines a for-loop, 20 values are shown in the Application Output. These do apparently look random.

Note that the Next() function takes an argument, here 100. This tells the System only to generate numbers smaller than 100, i.e., between 0 and 99. It is also possible to set the lower bound. If no bound is given the random numbers span the whole range of integers, i.e., from 0 up to System.Int32.MaxValue - 1.

When a random number generator is created the constructor can also be given an argument, for example as in `Random r = new Random(1);`. This argument is a so-called seed. All random number generators seeded with the same value generate precisely the same sequence of random numbers. This shows that random numbers are not random

at all; they are deterministically constructed by smart algorithms; the sequences only look random. If no seed is provided the random number generator is seeded from the current system time. This can cause problems if several random number generators are created in quick succession, because they will all get the same seed and therefore create the same sequences. That may not be desired.

It also happens that floating point random numbers are needed. Uniformly distributed values in the range $[0,1[$ can be created from integer numbers by dividing them by the maximum value possible, e.g., `x = 1.0 * r1.Next() / Int32.MaxValue;`. The multiplication by 1.0 is necessary to avoid integer division, which would result in all zeros, because the random numbers generated are always smaller than `Int32.MaxValue`.

8.2 Reading Data From Files

On a few occasions we need to read data from files like the one shown in Figure 36. The

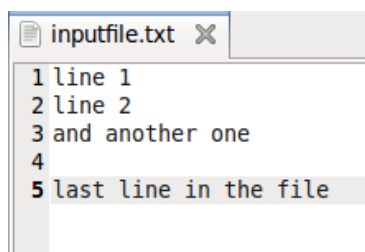


Figure 36: Short file with a few lines

file in this example has the name “inputfile.txt” and it contains 5 lines with text, one of which is empty.

There are different file formats and various ways to read files. For the purpose of SOFT153 it suffices to know how to open a text file for reading and reading it line by line. Other methods should not be used.

The way to do this is to use a so-called `StreamReader` provided by the System-library “System.IO”. The example in Figure 37 shows in line 2 that we have to tell C# that we want to use functionality provided by System.IO.

Line 8 creates an instance of the class `StreamReader`. The argument of the call specifies the path to the file to read. This can either be an absolute path name, or a relative one. If it is a relative one, as in the example, one has to take into account that Visual Studio and MonoDevelop, by default, place executable code two subdirectories below the directory that actually contains the source code. If the source code and the file to read are in the same directory, the executable will be in a subdirectory “bin/Debug” of that directory. Because the executable code is executed in this directory it can only find the file to read if it would be placed in the same directory as the executable, or if we correct for the two levels of subdirectories. This is what the string “../..” in line 8 does. The file to


```
1 using System;
2 using System.IO;
3
4 class Program
5 {
6     static void Main()
7     {
8         StreamReader inputFile = new StreamReader("../.." + "inputfile.txt");
9         while (true)
10         {
11             String line = inputFile.ReadLine();
12             if (line == null)
13                 break;
14             Console.WriteLine( line );
15         }
16     }
17 }
```

Application Output

line 1
line 2
and another one

last line in the file

Figure 37: Reading from a file

read is searched for under “.././inputfile.txt” relative to the directory that contains the executable (i.e., bin/Debug). “..” means “go up one level ins the directory hierarchy.

Lines 9 to 15 implement a loop that iteratively reads lines from the file and prints them out. If no line can be read the loop breaks, typically after all lines have been read. The method `ReadLine()` of the class `StreamReader` actually reads a single line form a file and returns it as a string. If no line can be read, for example, at the end of the file, the string will be undefined (‘null’). This is used as the exit condition for the while-loop (lines 12/13). As long as lines can be read, they are printed to the console in line 14. The Application Output shows that this includes empty lines; empty lines return an empty string “” which is different from `null`.

If the file contains numbers, the input strings have to be processed further, e.g., converted to int or double (e.g., `Convert.ToInt32(line);`). It may also be necessary to split the line into pieces, if they contain more than one number or data item (e.g., `line.Split(...)`).

Finally, note that if the file can not be found the code in the example terminates with a nasty and almost unreadable error message. Likewise, if a string cannot be converted to double or int the code breaks with an error message. These situations are usually dealt with in good code by methods called “exception handling”. This is out of the scope of the present document, but good practice in code writing.