Object-Oriented Programming with C#

Programming – Part I

By Per Laursen
07-08-2022

Introduction

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

Concerning processing of data, we can detail this activity further:

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

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

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

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

- Numeric data (numbers)
- Text data
- Logical data (we will explain what this means very shortly)
- Special-purpose data (pictures, music, etc..)

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

Data Types

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

The first common level of bit-grouping is to define a group of 8 bits as being a **byte**. We very rarely work at a finer level than the byte-level. As you may know, a modern PC will usually have between 8 and 32 *Giga*bytes of working memory (RAM). Each byte in the computer memory can be specified by its **address**. The address is just a counter starting from zero, up to the number of bytes in memory. When we place some data in the working memory of the computer, we are essentially just writing a sequence of bytes into a specific area of the memory, starting at some given address.

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

00000000
00000001
00000010
00000011
00000100
00000101
…and so on, until
11111111

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

00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
00000100 = 4
00000101 = 5
…and so on, until
11111111 = 255

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

This is very nice, but what if we need to store larger numbers? We could then choose to use four bytes instead of just one byte to represent a number. That would give us a more impressive range, from –2147483648 to 2147483647. You could even use 8 bytes, giving you an even large range. A natural thought could then be *"well, let's be absolutely sure we have a large enough range! Let's use 64 bytes for each integer number!"*. For most applications, this would probably also work fine in this day and age, where even tiny devices have multi-gigabyte memories. Still, there was a time where memory was a more scarce resource, and there are indeed still many applications today where you have to be quite careful w.r.t. memory consumption. A Machine Learning algorithm may need to represent billions of numbers, making it a highly relevant matter if each number takes up 4 or 8 bytes.

The general point, however, is this: Even though the computer stores everything as sequences of bits, we can fortunately make use of more human-friendly data types, that are available in C#. The compiler – and underlying code – will handle the details of the translation to bits for us. A lot of these so-called **primitive data types** are available; we only present a few of them here, but feel free to look for additional information about other such types elsewhere. The names of the types are sometimes a bit obscure, which is often for historical reasons.

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

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

Variables

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

```
// Reserve space in memory for an int,
// refer to the address by the name "age"
int age;

// Store the value 24 in the address
// referred to by "age"
age = 24;
```

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

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

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

```
age = 28;
```

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

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

```
// Reserve space in memory for an int,
// refer to the address by the name “age”,
// and initialize the value to 24.
int age = 24;
```

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

Arithmetic

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

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

```
int age;
age = 24;
```

Be aware that the second line means *“change the value of **age** to 24”*, and NOT *“compare the value of **age** to 24”*. Below is an example of very simple arithmetic:

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

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

```
age = age + 10;
```

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

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

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

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

Doing addition, subtraction and multiplication with integer numbers is fairly straight-forward (even though integer overflow[1] is a pitfall). Division can be slightly more tricky. Consider the below code:

```
int a = 7;
int b = 4;
int c = a / b; // a divided by b
```

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

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

```
int a = 7 % 4;
```

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

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

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

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

[1] https://en.wikipedia.org/wiki/Integer_overflow

Code Quality, part I

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

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

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

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

Writing code that is "as easy as possible to maintain and extend" is somewhat subjective. What does this mean in practice? This is quite hard to pin down, and there are diverging opinions about it. Historically, it has to some extent been assumed that as long as you are careful about specifying and designing your application, the resulting code will automatically be of high quality. This has proven to be an illusion; in the real world, requirements may change rapidly, and we cannot up-front anticipate how

the optimal end design will be. We must therefore do the best we can on the basis of the available information, but also be prepared to spend time on making quality improvements to our code, that do not add extra functionality. Over the years, some agreement has been reached concerning what such improvements might specifically be; the famous (in the software development community) book **Refactoring**[2] by Martin Fowler gave a first comprehensive presentation of a large number of so-called "refactorings" that can be applied to code, with the sole purpose of improving code structure, while keeping the functionality intact. These refactorings range from the very simple – as we shall see in a moment – to the quite sophisticated.

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

Consider the code below:

```
double x = 25.00;
double y = 6.00;
double z = 0.08;
double t = x * (1.00 + z) + y;
```

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

```
double netPrice = 25.00;
double shipping = 6.00;
double tax = 0.08;
double totalPrice = netPrice * (1.00 + tax) + shipping;
```

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

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

or

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

[2] https://martinfowler.com/books/refactoring.html

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

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

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

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

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

- Make code that is **compilable**
- Make code that **conform to requirements**
- Make code that is of **high quality**

Screen output and type conversions

We should now be capable of writing small pieces of code…but we cannot really present the results of e.g. an arithmetic calculation anywhere. It would be nice to be able to print it on the screen, in a fashion similar to the ***"Hello, World!"*** example.

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

```
Console.WriteLine("Hello, World!");
```

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

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

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

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

So, maybe we can print the value of a variable like this:

```
int age = 24;
Console.WriteLine("age");
```

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

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

```
int age = 24;
Console.WriteLine(age);
```

Indeed we can! We can now print the value of the **age** variable, and thus print the value of any variable we wish. The **Console.WriteLine** method (we will talk much more about methods pretty soon) is quite flexible with regards to what it can print on the screen. It will print almost everything you put inside the parentheses, or rather; it will try to print the best possible string representation of it. When the value of **age**

was printed, it was in fact the string "24" that got printed. For us, that distinction is a bit academic, since the conversion from the number 24 to the string "24" is trivial. We will see later that such conversions can be more complicated.

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

```
string message = "The value of age is " + age;
Console.WriteLine(message);
```

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

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

```
int age = 24;
double someNumber = 1.3;
Console.WriteLine("Dividing age by 1.3 is " + age/someNumber);
```

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

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

Using string concatenation is one way of printing a longer message – consisting of a mix of strings and variable values – on the screen, using **Console.WriteLine**. However, there is another way to do this which might be more convenient, by using so-called string interpolation. Suppose we have two variables like:

```
string name = "James";
int age = 23;
```

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

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

The first thing to notice is the **$** (dollar) sign in front of the string. This signals to the compiler that this string is used for string interpolation. If omitted, the actual content of the string – including the brackets – would just be printed as-is.

Also notice the use of the curly brackets. This should be understood as follows: For each pair of brackets, calculate the value of the expression inside the brackets (in this case simply the value of a variable) , and replace **{…}** with that value. In this example, **{name}** is replaced with "James", and **{age}** is replaced with "23", in total producing the string above. You can use this principle for as many expressions as you wish.

Logic

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

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

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

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

```
int firstNumber = 12;
int secondNumber = 14;
bool areTheyEqual = (firstNumber == secondNumber);
Console.WriteLine($"The numbers are equal : {areTheyEqual}")
```

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

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

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

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

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

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

```
int age = 8;
bool isSmaller = (age < 10);
```

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

- Smaller than 20, and
- Larger than 12

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

```
int age = 14;
bool isTeenager = (age < 20) && (age > 12);
```

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

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

- Larger than 19, or
- Smaller than 13

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

```
int age = 14;
bool isNotTeenager = (age > 19) || (age < 13);
```

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

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

```
int age = 14;
bool isTeenager = (age < 20) && (age > 12);
bool isNotTeenager = !isTeenager;
```

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

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

A	B	A && B	A || B	!A
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Functions

We now have some basic tools available that allow us to write non-trivial pieces of code. As an example, consider the calculation of the so-called **Body Mass Index**[3], or just **BMI**. According to the definition, we can calculate the BMI like this (we assume that **weight** holds the weight of a person in kilograms, and **height** holds the height of a person in metres):

```
double bmi = weight / (height * height);
```

This is a fairly simple formula. Still, if we must write this formula several places in our code, it becomes a bit tedious to write it again and again. This becomes even more problematic if the logic is more complex. Instead of this, we would like to define the logic in one place, and then refer to that logic instead of writing it over and over. We can do this by defining a so-called **function**. A function is a fundamental concept in programming. The syntax may vary a bit from language to language, but you (almost) always need to

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

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

```
double CalculateBMI(double weight, double height)
{
      double bmi = weight / (height * height);
      return bmi;
}
```

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

1. The name of the function is **CalculateBMI**
2. The function takes the two doubles **weight** and **height** as input
3. The logic is to calculate the BMI as per the definition
4. The output is the BMI value just calculated

[3] https://en.wikipedia.org/wiki/Body_mass_index

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

We can now use the function **CalculateBMI** elsewhere in the code, whenever we need to calculate a BMI value. We can **invoke** or **call** (we will usually use the term **call**) the function by writing code like this:

```
double bmiJohn = CalculateBMI(weightJohn, heightJohn);
```

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

```
double bmiJohnPlus10 = CalculateBMI(weightJohn, heightJohn) + 10.0;
```

This is perfectly valid C# code; the compiler will look at the function and think *"Well, the output of calling that function will produce a double value, so I can just add 10.0 to that value, no problem!"*. As long as the type of the value produced by the function can be used in the expression the function is part of, everything is fine.

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

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

Pre-OO programming

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

What we have learned so far, does indeed relieve us of many considerations that earlier software developers had to handle themselves:

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

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

Exercises

Exercise	**Pro.1.1**
Project	MovieManagerV05
Purpose	Discuss variables with regards to types and naming
Description	We imagine this project to be the very first steps in creating an application for movie management. The application could be used to keep track of relevant information for movies, e.g. a private collection of movies on DVD/Blu-ray (yes, some people still watch movies on physical media ☺).

Steps	1. Think about what specific information it could be relevant to store for each movie.
	2. For each specific piece of information, think about how you can represent this information. Think about the nature of the information; is it text, numeric, or something else.
	3. In **Program.cs**, define a variable for each piece of information. You should
	  a. Choose a proper type for the variable
	  b. Find a descriptive name for the variable
	4. Once you are done, pair up with another student.
	5. Switch computer with your partner
	6. Review the work of your partner. For each variable in the partner's project, think about if
	  a. The purpose of the variable is easy to understand
	  b. The type seems properly chosen
	7. Discuss your findings with your partner.
	8. Were there any types of information that were particularly hard to find a good representation for?
	9. Suppose you need to store information about a lot of movies. How would you need to change your code in order to make that possible?

<table>
<tr><td>Exercise</td><td>Pro.1.2</td></tr>
<tr><td>Project</td><td>WebShopV05</td></tr>
<tr><td>Purpose</td><td>Get some practice in using arithmetic operators</td></tr>
<tr><td>Description</td><td>Part of the business logic in a web shop involves calculating the total cost of an order. The logic for calculating the total cost is as follows:
An item has a net price
You pay a 10 % tax on top of the net price
Shipping costs 49 kr., no matter the number of items
There is a credit card fee of 2 % on top of the entire cost, including tax and shipping.</td></tr>
<tr><td>Steps</td><td>Load and open the project, and open Program.cs. You will see that some variables for the net prices and number of items in an order have already been included. Also, the order details are printed on the screen.
The variable totalPrice is supposed to contain the total price for the order. You must add the calculations needed to do this, given the logic in the description.
Test your solution by varying the number of books, DVDs and games in the order (you do this by assigning new values to the noOf… variables, and running the application again)</td></tr>
<tr><td>Extra info</td><td>Some test examples you can use to verify your solution.<table><tr><th>Books</th><th>DVDs</th><th>Games</th><th>Total price</th></tr><tr><td>8</td><td>3</td><td>2</td><td>711,96 kr.</td></tr><tr><td>0</td><td>12</td><td>4</td><td>1171,98 kr.</td></tr><tr><td>23</td><td>16</td><td>7</td><td>2507,16 kr.</td></tr></table></td></tr>
</table>

<table>
<tr><td>Exercise</td><td>Pro.1.3</td></tr>
<tr><td>Project</td><td>WebShopV06</td></tr>
<tr><td>Purpose</td><td>Get some practice in using logical operators</td></tr>
<tr><td>Description</td><td>Another part of the business logic in a web shop involves deciding if a customer qualifies for certain special offers, based on the order. The shop has four special offers. The logic for qualifying for each offer is:
The net total price (no taxes, etc.) is more than 1.000 kr.
You have ordered more books than games
You have ordered at least 10 items of one kind
You have ordered between 10 and 20 (incl.) DVDs, or at least 5 games</td></tr>
<tr><td>Steps</td><td>Load and open the project, and open Program.cs. Again, some variables are already present. Note the boolean variables receiveSpecialOffer…
For each of these variables, you must specify a logical expression, corresponding to the logic given in the description.
Test your solution by varying the number of books, DVDs and games in your order.
The web shop decides to offer an extra special offer. You qualify for the extra offer, if you qualify for <u>exactly two</u> of the previous offers. Update your code to include this extra offer.</td></tr>
<tr><td>Extra info</td><td>Some test examples you can use to verify your solution (SO#1 means “special offer 1”, and so on):

<table>
<tr><th>Books</th><th>DVDs</th><th>Games</th><th>SO#1</th><th>SO#2</th><th>SO#3</th><th>SO#4</th></tr>
<tr><td>8</td><td>3</td><td>2</td><td>false</td><td>true</td><td>false</td><td>false</td></tr>
<tr><td>0</td><td>12</td><td>4</td><td>false</td><td>false</td><td>true</td><td>true</td></tr>
<tr><td>23</td><td>16</td><td>7</td><td>true</td><td>true</td><td>true</td><td>true</td></tr>
<tr><td>3</td><td>5</td><td>4</td><td>false</td><td>false</td><td>false</td><td>false</td></tr>
</table></td></tr>
</table>

Exercise	**Pro.1.4**
Project	FunctionExample
Purpose	Define and use a simple function
Description	You can define a rectangle by two points (x1, y1) and (x2, y2). The area of the rectangle is then: the absolute value of (x1 – x2)*(y1 – y2) The absolute value just means that if the value is negative (e.g. -4), the absolute value is the corresponding positive value (e.g. 4 in this example). If the value is already positive, it just says positive.
Steps	1. Review the code in **Program.cs**. It calculates the area of two rectangles, and prints out the values. Make sure you understand the calculation, including the use of **Math.Abs**. 2. An (incomplete) method **AreaOfRectangle** is also included in **Program.cs**. See if you can implement it correctly (Hint: try to move the calculation logic from the existing code into the method, and rename the variable names to match the parameter names used in **AreaOfRectangle**). 3. Once you have implemented the method, use it to perform the area calculations in the code above the method. Check that you get the same results as before. 4. Why is it a good idea to replace the area calculations with calls to the method **AreaOfRectangle**?