Object-Oriented Programming with C#

Programming, Part II

By Per Laursen
26-07-2025

INTRODUCTION **3**

CONDITIONAL STATEMENTS **4**

The if-statement 4

The if-else statement 6

The multi if-else statement 9

The switch statement 11

Nullable value types (a slight detour) 13

REPETITION STATEMENTS **16**

The while-loop 16

The for-loop 20

DEBUGGING **22**

INTRODUCTION TO DATA STRUCTURES **25**

Arrays (and why you should be careful using them…) 25

The List class 30

The Dictionary class 34

The HashSet class 38

Enumerations 40

CODE QUALITY, PART III (KEEPING YOUR CODE DRY) **42**

DRY and values 42

DRY and instance fields 44

DRY and methods 44

DRY and classes (and a brief introduction to Inheritance) 47

When does code become DRY? 50

EXERCISES **52**

Pro.2.1 52

Pro.2.2 53

Pro.2.3 54
Pro.2.4 55
Pro.2.5 56
Pro.2.6 57
Pro.2.7 58
Pro.2.8 59
Pro.2.9 60
Pro.2.10 61
Pro.2.11 62
Pro.2.12 63
Pro.2.13 64
Pro.2.14 65
Pro.2.15 66
Pro.2.16 67

Introduction

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

The two main categories of control statements are

- Conditional statements
- Repetition statements

Conditional Statements

In the category of conditional statements, we find

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

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

The if-statement

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

```
if (condition)
{

}
```

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

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

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

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

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

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

```
int age = 17;

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

if (age < 18)
{
      Console.WriteLine("You are still a child...");
}

Console.WriteLine("Finished checking age");
```

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

```
Starting to check age...
You are still a child...
Finished checking age
```

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

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

```
Starting to check age...
Finished checking age
```

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

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

```
int age = 17;

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

if (age < 18)
{
      Console.WriteLine("You are still a child...");
}

if (age > 18)
{
      Console.WriteLine("You are an adult!");
}

Console.WriteLine("Finished checking age");
```

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

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

The if-else statement

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

```
if (condition)
{

}
else
{

}
```

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

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

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

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

```
if (age < 18)
      Console.WriteLine("You are still a child...");
else
      Console.WriteLine("You are an adult!");
```

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

```
if (age < 18);
      Console.WriteLine("You are still a child...");
```

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

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

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

```
if (age > 12)
{
      if (age < 20)
      {
            Console.WriteLine("You are a teenager");
      }
}
```

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

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

```
if ((age < 20) && (age > 12))
{
      Console.WriteLine("You are a teenager");
}
```

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

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

```
int age = 15;
string message = "You are ";

if (age < 18)
{
      message = message + "a child.";
}
else
{
      message = message + "an adult.";
}
```

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

```
condition ? expression1 : expression2
```

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

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

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

The multi if-else statement

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

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

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

```
if (score >= 90)
{
      Console.WriteLine("Mark is: A");
}
else
{
      if (score >= 70)
      {
            Console.WriteLine("Mark is: B");
      }
      else
      {
            if (score >= 40)
            {
                  Console.WriteLine("Mark is: C");
            }
            else
            {
                  Console.WriteLine("Mark is: D");
            }
      }
}
```

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

```
if (score >= 90)
{
      Console.WriteLine("Mark is: A");
}
else if (score >= 70)
{
      Console.WriteLine("Mark is: B");
}
else if (score >= 40)
{
      Console.WriteLine("Mark is: C");
}
else
{
      Console.WriteLine("Mark is: D");
}
```

The code does exactly the same as the previous example, but is easier to read (at least the author thinks so ☺). If you compare this statement with the simple **if-else** statement, we have actually just added a number of **else-if** blocks between the **if**-part and the **else**-part. An important detail to remember is that a multi-**if-else** statement must end with an **else**-part.

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

The switch statement

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

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

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

```
switch (noOfChildren)
{
      case 0:
            childSupport = 0;
            break;
      case 1:
            childSupport = 1200;
            break;
      case 2:
            childSupport = 2000;
            break;
      case 3:
            childSupport = 2600;
            break;
      default:
            childSupport = 3000;
            break;
}
```

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

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

The above description applies to the classic **switch**-statement, and it may seem to be quite restrictive, since we are only able to create cases based on exact matching of values. However, the **switch**-statement has recently been "revitalized" somewhat, by introduction of use of **pattern matching**. This is a somewhat large topic on its own, but the simple version is that we can you create cases not just by exact matching of values, but by matching of conditions! Recall the example from above where we needed to map a test score between 0 and 100 to a mark (A, B, C or D). This would be very tedious to write using the classic **switch**-statement, but with pattern matching, we can write it like this:

```
switch (score)
{
      case (< 40):
            mark = "D";
            break;
      case (< 70):
            mark = "C";
            break;
      case (< 90):
            mark = "B";
            break;
      default:
            mark = "A";
            break;
}
```

This makes the **switch**-statement a much more viable alternative to multi-**if-else**-statements.

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

Nullable value types (a slight detour)

At the very beginning of these notes, we discussed so-called **primitive data types** like e.g. **int** and **bool**. These are also known as **value types**, which are fundamentally different from **references types** w.r.t. how they are stored in memory. Variables of a class type are of the reference type, i.e. they refer to where the data itself is stored, while variables of value type hold the data itself. A consequence of this is that a variable of reference type may refer to nothing at all, which led to the introduction the keyword **null**.

We may from time to time be in a situation where a simple type is the obvious choice for representing some sort of simple data – e.g. a **int** for representing a test score – but we also need to be able to represent the situation where no specific value is available at the time the variable is created. As an example, consider a very simplistic class **Student**, defined like this:

```
public class Student
{
    public string Name { get; set; }
    public int TestScore { get; set; }

    public Student(string name)
    {
        Name = name;
        TestScore = ...; // What should we choose...?
    }
}
```

The setup is as follows: a student can be enrolled in a school with just a name, since the school will then at some point after enrollment conduct a test for the student, which will then produce a test score. We thus need to be able to create a **Student** object without having a test score available. The big question is then: what value should we initially assign to **TestScore**…?

If the test results is, say, a percentage score between 0 and 100, we could perhaps just assign -1 as the initial value, and subsequently interpret -1 as meaning "no value". This is feasible, but tends to be somewhat fragile… What if we at some point decide to employ a different scale, say from -1000 to 1000? Could we then run into problems? It would be nice if we could truly assign "no value" to such a property. This becomes possible using **nullable value types**.

A nullable value type works just as an ordinary value type, except that we can actually assign the value **null** to a variable of that type. In the class definition above, we would need to make a very small change:

```
public class Student
{
    public string Name { get; set; }
    public int? TestScore { get; set; }

    public Student(string name)
    {
        Name = name;
        TestScore = null;
    }
}
```

The "?" symbol added after the type name signifies that this variable is "nullable", and the assignment in the constructor now becomes valid. So far, so good…

This new flexibility does come at a price. Consider the below code, which would have worked – or rather, at least be compilable – using ordinary simple types:

```
Student ann = new Student("Ann");
Student beth = new Student("Beth");

int avgScore = (ann.TestScore + beth.TestScore) / 2;
```

Visual Studio will now report errors, like *"cannot implicitly convert int? to int"*, and *"nullable value type may be null"*. Visual Studio is essentially saying *"this will not work if any of these values are null"*, which does make sense.

Then what…? Well, now that we know about conditional statements, we actually have ways to handle this (this is also why the introduction of nullable value types was postponed to this point). It is – of course – possible to check if a nullable variable has a value assigned to it or not. Two properties **Value** and **HasValue** are available on a nullable value type, making it possible to write code as this:

```
int avgScore = (ann.TestScore.HasValue && beth.TestScore.HasValue) ?
               (ann.TestScore.Value + beth.TestScore.Value) / 2 :
               0;  // ... or what?
```

We use the ternary operator introduced earlier in this chapter, so the code reads *"If both variables actually have a value assigned to them, use those values to calculate the average, otherwise return zero"*. It can be debated if the handling of the "otherwise" case is sensible, but that's not the main point here. The point is just to make you aware that these **nullable value types** are available to you, if they seem to be the best fit for your problem at hand. Still, using them often opens up a number of issues w.r.t. how to handle cases where no value is assigned. Use them carefully…

We will not be using nullable value types to any great extent in the remainder of these notes.

Repetition statements

The various conditional statements allow us some control of the flow of execution, but mostly in the sense that we may or may not execute certain sections of code. A very useful feature would be the ability to repeat execution of a code block, as long as some condition is true. The so-called **repetition statements** enable us to do just that. Just as for conditional statements, there are a few variants of repetition statements to choose from, but they are essentially all variations over the same theme: repeat execution of a block of code, as long as a given condition is true.

The two most common types of repetition statements are:

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

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

The while-loop

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

```
while (condition)
{
}
```

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

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

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

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

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

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

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

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

```
int number = 1;
while (number < 5)
{
      Console.WriteLine($"The value of the number is {number}");
      number = number + 2;
}
```

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

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

```
int number = 1;
while (number < 5)
{
      Console.WriteLine($"The value of the number is {number}");
}
```

It looks fairly innocent, but if we try to run it, we will be stuck in the dreaded infinite loop. We took out the statement that increased the value of **number**, so the condition will now remain true forever…

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

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

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

```
Reader myReader = new Reader();
string keyPressed = "";

while (condition)
{
      keyPressed = myReader.ReadFromKeyboard();
      Console.WriteLine($"You pressed {keyPressed}");
}
```

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

```
Reader myReader = new Reader();
string keyPressed = "";

while (keyPressed != "q")
{
      keyPressed = myReader.ReadFromKeyboard();
      Console.WriteLine($"You pressed {keyPressed}");
}
```

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

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

- **Initialization**: Before the loop itself is entered, we usually – but not always – initialize some variable that is also used as part of the condition.
- **Condition**: The logical condition itself, which is evaluated before performing another iteration of the loop.
- **Change**: Since the condition itself is fixed, at least one of the values of the variables in the condition must have the chance to change during an iteration. Otherwise, we have an infinite loop. Formally put, the probability that something changes must be larger than 0 (zero).
- **Action**: The set of statements that is executed during an iteration, i.e. those statements that perform the actual "action" we want to happen during each iteration. This could e.g. be to print a line on the screen.

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

The for-loop

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

```
int number = 1;  // Initialization
while (number < 5)  // Condition
{
      Console.WriteLine(number);  // Action
      number = number + 1; // Change
}
```

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

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

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

```
for (initialization ; condition ; change)
{
      action
}
```

A **for**-loop and a **while**-loop are logically equivalent; anything you can do with a **for**-loop you can also do with a **while**-loop. So why choose a **for**-loop over a **while**-loop? It is mostly a matter of taste and habit. The structure of the **for**-loop does perhaps make it harder to forget one of the elements, since the **for**-loop looks a bit "odd" if you remove one of the elements. Still, this is a subjective criterion.

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

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

```
for (int number = 1; number < 5; number = number + 1)
{
      // Action
}
```

Having a "counter variable" that is increased by one after each iteration is very common, and you will often see the above written as:

```
for (int number = 1; number < 5; number++)
{
      // Action
}
```

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

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

```
for (;;)
{
      // Action
}
```

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

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

Debugging

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

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

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

A first – and quite obvious – technique is simply to read the code closely once again. In many cases, the bug is quite simple (like e.g. using a minus instead of a plus somewhere), and surprisingly many bugs can be caught this way. Even better is to try to explain the code to someone else (even just to a teddy bear ☺); the process of trying to verbalise the intention of the code will often reveal bugs.

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

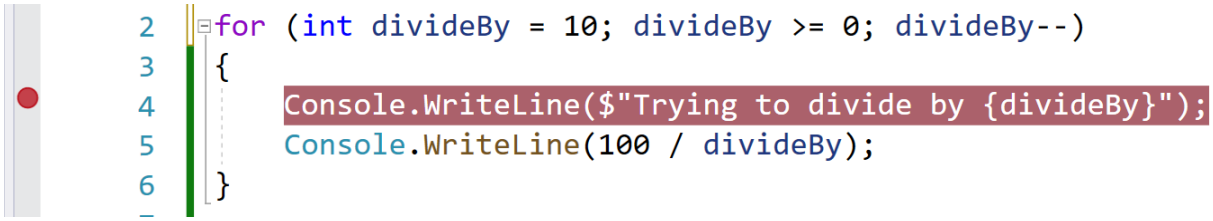
```
for (int divideBy = 10; divideBy >= 0; divideBy--)
{
      Console.WriteLine($"Trying to divide by {divideBy}");
      Console.WriteLine(100 / divideBy);
}
```

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

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

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

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



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

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

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

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

- *Step Into*
- *Step Over*
- *Step Out*

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

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

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

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

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

Introduction to Data structures

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

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

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

Arrays (and why you should be careful using them…)

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

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

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

Note that the "line of boxes" mentioned above is only the shaded part of the illustration, in the row labeled *value*. So, we have a total of 7 integer values in this particular array. What is the **index** then? The **index** is the "address" of a specific element

in the array. When we put a value into the array, we have to specify which "address" it should have. The **index** is exactly that. We can then use the index again, when we wish to retrieve an element from the array. The index itself is just an integer number; the only peculiarity is the fact that an array index always starts from 0 (zero). This also means that in this particular array, the last element has index 6 (not 7)!

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

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

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

```
int[] myFirstArray;
```

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

However, just declaring the variable does not create the array itself. Creating the array looks like:

```
myFirstArray = new int[7];
```

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

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

```
myFirstArray[3] = 23;
```

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

```
int singleValue = myFirstArray[5];
```

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

```
myFirstArray[4]++;
```

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

In the above example, we created an array of **int** values, with 7 elements. We did not really specify the content of the array. Is the array then empty initially? No, since the concept "empty" doesn't quite make sense for an array. Once the array is created, it immediately has the specified size, and the content of each element in the array is then interpreted as an **int** value. In C#, the elements in a array of numerical values are set to 0 (zero) by default, so that will be the initial content of this array.

If we already know the initial values when we want to create the array, a couple of neat ways exist to put those elements into the array in a single statement:

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

or even shorter

```
int[] myFirstArray = [ 34, -233, 9801, 67, 2, -9582, 770 ];
```

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

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

```
for (int index = 0; index < 7; index++)
{
      Console.WriteLine(myFirstArray[index]);
}
```

That's it! The variable **index** starts at 0, and keeps increasing until it reaches 7, where the condition becomes false. Also, this code (almost) doesn't change if we have an array with 10000 elements instead of 7. The only small problem with this code is the

explicit use of the size of the array. If we change the size of the array, we must also change the condition in the loop. Can we avoid that? Yes, since you can retrieve the length of the array from the array object itself:

```
for (int index = 0; index < myFirstArray.Length; index++)
{
      Console.WriteLine(myFirstArray[index]);
}
```

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

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

```
foreach (var item in myFirstArray)
{
      Console.WriteLine(item);
}
```

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

```
foreach (var variableName in arrayName)
{
      // Whatever you want to do with the value
}
```

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

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

The size of an array is fixed: When you create an array, you must specify its size. This means that already at creation time, you must anticipate how many elements you may need to store in the array. That is often impossible, or at least very uncertain.

What if the array is used in a school administration system? How many students must it be possible to handle? 100? 10000? Who knows… You could then argue that you should just create a "sufficiently large" array, maybe with one million elements. Is that a problem? It could be, since you will then use a fixed-size chunk of memory for this array, even though it may often be next-to-empty. Even though RAM is cheap these days, we should still be wary of excessive memory consumption.

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

```
myFirstArray[-2]++;
```

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

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

There is no help with common tasks: Since the array is not really a class, there are no methods available[1] for various common tasks, like e.g. sorting the elements. It should be mentioned that C# does contain an **Array** class, but this is more a collection of various static methods that can be used on arrays, not part of the array as such.

[1] Even though this is formally correct, there are actually several such methods available in the form of so-called extension methods, specifically in the **LINQ** library. This is a more advanced topic, so we will not elaborate further on this right now.

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

The List class

The .NET Class Library contains several classes for handling collections of values, and classes of this nature are often called **collection classes**. The **List** class is probably the class that has most in common with the classic array data structure. However, the **List** class improves on some of the drawbacks described above. The purpose is similar to arrays: insert and retrieve values (or object references) of a specific type. The syntax for creating a **List** object is:

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

There are a couple of things to notice here:

- The statement creates a **List** object, that can hold a collection of **int** values. The type specification goes between the **pointy brackets <>**, that follow right after the **List** class name. The **List** class is a so-called **generic** class; we will discuss generic classes later, but for now we just note that this is the syntax we must use for specifying the type of the items in the list.
- The **List** class is as mentioned part of the .NET Class Library, and is located in the namespace ***System.Collections.Generic***. This part of the library is automatically included in the exercise projects. In older versions of Visual Studio – and for certain other project types – you need to manually add a line at the top of the file where the **List** class is used, looking like this:

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

  Again; this is not necessary to do in the exercise projects, but still a useful thing to know.
- We do not need to specify an initial size of the list! Upon creation, the list is indeed empty. Once we start adding items to the list, the size is increased accordingly, and also decreased if items are deleted.

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

```
myFirstList.Add(982);
```

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

```
myFirstList.Insert(2, 980);
```

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

index	0	1	2	3
value	34	-233	9801	67

After the insertion, the list will look like:

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

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

```
myFirstList.RemoveAt(1);
```

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

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

to

index	0	1	2	3
value	34	980	9801	67

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

Just as for arrays, there is also a convenient syntax for initializing a **List** with a set of known values:

```
List<int> myFirstList = [ 34, -233, 9801, 67, 2, -9582, 770 ];
```

You can achieve the same with multiple calls of **Add**, but this is definitely easier.

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

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

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

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

```
int aValue = myFirstList[3];
```

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

```
myFirstList[3] = 111;
```

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

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

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

```
foreach (var item in myFirstList)
{
      Console.WriteLine(item);
}
```

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

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

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

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

The Dictionary class

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

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

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

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

```
int index = 0;
Student theStudent = null;

while ((theStudent == null) && (index < allStudents.Count))
{
      if (allStudents[index].SSN == givenSSN)
      {
            theStudent = allStudents[index];
      }
}
```

In plain language: Examine the items one by one, until a student with the given SSN is found, or all items have been examined.

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

would boil down to:

```
Student theStudent = allStudents[givenSSN];
```

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

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

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

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

```
allStudents.Add(aStudent.SSN, aStudent);
```

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

```
allStudents[aStudent.SSN] = aStudent;
```

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

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

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

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

```
allStudents.Remove(givenSSN);
```

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

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

```
aStudent = allStudents[givenSSN];
```

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

```
if (allStudents.ContainsKey(givenSSN))
{
      aStudent = allStudents[givenSSN];
}
```

This is in fact a perfect opportunity to use the **ternary operator**:

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

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

How do we initialize a **Dictionary** with a set of known values? Just as for the **List** class, we can either make multiple calls of **Add**, or use a more convenient syntax. The first approach would look like this:

```
Dictionary<string, int> scores = new Dictionary<string, int>();
Scores.Add("Peter", 45);
Scores.Add("Annie", 78);
Scores.Add("Frank", 83);
Scores.Add("Linda", 70);
```

The second approach would look like this:

```
Dictionary<string, int> scores = new Dictionary<string, int>()
{
	{ "Peter", 45 },
	{ "Annie", 78 },
	{ "Frank", 83 },
	{ "Linda", 70 }
}
```

The second approach is a bit shorter, but it's debatable if it is much more convenient than calling **Add**. Use the approach that feels most natural for you.

Finally, the **Dictionary** contains two very useful properties: **Keys** and **Values**. The **Keys** property returns the entire set of keys for the items in the dictionary, while **Values** return the entire set of values. If you e.g. wish to iterate through all of the values in a dictionary, it can be done like this:

```
foreach (Student s in allStudents.Values)
{
	Console.WriteLine(s);
}
```

What would happen if we just use a **Dictionary** object directly in the **foreach**-loop? For our example, it would look like this:

```
foreach (KeyValuePair<string, Student> kvp in allStudents)
{
	Console.WriteLine(kvp);
}
```

Notice the somewhat cryptic type for the **kvp** variable. In a **Dictionary**, each element is a **key-value pair**, so in this example, such a pair has the type **KeyValuePair<string, Student>**. However, just as we can pick out the keys and values for an entire **Dictionary**, properties also exist for key-value pairs for a similar purpose: **Key** returns the key part, and **Value** returns the value part. If we e.g. want to print out all keys, we can change the above **foreach**-statement to this:

```
foreach (KeyValuePair<string, Student> kvp in allStudents)
{
	Console.WriteLine(kvp.Key);
}
```

The HashSet class

The **Dictionary** class is thus our preferred choice whenever we have data with an obvious key/value relation. We may however encounter situations where data has key-like properties, but does not refer to any other data. With "key-like" properties, we mean:

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

Suppose we want to maintain a collection of Social Security Numbers (SSN), and want to be able to quickly check if a given SSN is in this collection. We can use a **HashSet** for this purpose. Our declaration would look like this:

```
HashSet<string> ssnColl = new HashSet<string>();
```

We can then add elements to this **HashSet** like this:

```
bool added = ssnColl.Add("02011987-1235");
```

The **Add** method returns **true** if the element was successfully added, or **false** if the element is already present in the collection. This is indeed different than for a **List**, where there is no requirement for uniqueness. Checking if a value is present in the collection looks very similar:

```
bool isPresent = ssnColl.Contains("02011987-1235");
```

These insertions and lookups are just as efficient as for a **Dictionary**, i.e. the time is almost independent of the number of elements in the collection. Very nice… but why is it called a **HashSet**? The "hash" part of the **HashSet** class name relates to the internal representation of the data. A so-called **hash table** is used to store data, which is exactly what makes it possible to lookup data in "almost" constant time. By "almost" is meant that even though it is theoretically possible for the lookup to take relatively long time, it turns out that we in practice can look up elements in constant time, at the expense of using a bit more memory than by using e.g. a **List**. A **Dictionary** also uses a hash table for internal storage. If you are interested in more knowledge about hash tables, there are several sources online.

A **HashSet** can also be conveniently initialized with known values like this:

```
HashSet<string> names = ["Peter", "Annie", "Frank", "Linda"];
```

A **HashSet** can also be used with ease in a **foreach**-statement, like a **List** is used:

```
foreach (string ssn in ssnColl)
{
      Console.WriteLine(ssn);
}
```

An additional **HashSet** feature is the ability to perform more advanced so-called **set-oriented operations**. A **set** is the mathematical term for a collection of elements, and certain operations are well-defined for sets, like:

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

If you need to store data which has such key-like properties, and/or need to perform set-oriented operations on the data, the **HashSet** class offers support for this. See the class documentation for further details.

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

Choose a **List** if:

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

Choose a **Dictionary** if:

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

Choose a **HashSet** if:

- The data has key-like properties (each element must be unique).
- The elements themselves do not refer to something.
- You often need to check if a given value is already stored in the collection.

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

Enumerations

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

The primitive types available in C# are often not exactly what we want. Suppose we wish to make an application that deals with fruit, say these five kinds of fruit:

Apple, *Pear*, *Cherry*, *Banana* and *Kiwi*.

Which of the existing types should we then use for representing a fruit? Let's examine some possible choices:

- **bool**: That is obviously not a good choice, since we only have two possible values for a **bool**
- **int**: Possible (say, use the values 1 to 5, where 1 = *Apple*, and so on), but not very convenient, since
  - An **int** can obviously have a value smaller than 1 or larger than 5, so we might assign a value that does not correspond to a fruit
  - The code can become hard to understand
- **string**: Possible, but with problems similar to **int**

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

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

An enumerated type can be part of a class definition, but can also be defined as a stand-alone type. If we now want to declare a variable of type **Fruit** elsewhere in the program, and assign a value to it, it will look like:

```
FruitType aFruit = FruitType.Apple;
```

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

A typical rule-of-thumb is to use enumerations for types where there are from 3 to around 10 legal values. This doesn't mean that it is forbidden to have an enumeration with 25 valid values, but your code will inevitably become more complex when the number of valid values increases.

Code Quality, Part III (Keeping your code DRY)

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

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

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

> **D**on't
> **R**epeat
> **Y**ourself

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

- Values
- Instance fields
- Methods
- Classes

DRY and values

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

```
for (int x = 0; x < 10; x++)
{
	for (int y = 0; y < 10; y++)
	{
		// Do something for each cell
	}
}
```

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

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

```
public static class WorldSettings
{
	public const int DimensionX = 100;
	public const int DimensionY = 100;
}
```

Using the constants will then look like:

```
for (int x = 0; x < WorldSettings.DimensionX; x++)
{
	for (int y = 0; y < WorldSettings.DimensionY; y++)
	{
		// Do something for each cell
	}
}
```

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

DRY and instance fields

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

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

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

DRY and methods

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

```
public void PrintNameNicely()
{
	string nameLine = "Name is : " +_name;
	nameLine = nameLine.PadLeft(20);
	nameLine = nameLine.PadRight(40);
	nameLine = "|" + nameLine + "|";
	Console.WriteLine("----------------------------------------");
	Console.WriteLine(nameLine);
	Console.WriteLine("----------------------------------------");
}
```

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

```
public void PrintHeightNicely()
{
	string nameLine = "Height is : " + _height;
	nameLine = nameLine.PadLeft(20);
	nameLine = nameLine.PadRight(40);
	nameLine = "|" + nameLine + "|";
	Console.WriteLine("----------------------------------------");
	Console.WriteLine(nameLine);
	Console.WriteLine("----------------------------------------");
}
```

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

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

```
private void PrintNicely(string data, string leadText)
{
	string dataLine = leadText + data;
	dataLine = dataLine.PadLeft(20);
	dataLine = dataLine.PadRight(40);
	dataLine = "|" + dataLine + "|";
	Console.WriteLine("----------------------------------------");
	Console.WriteLine(dataLine);
	Console.WriteLine("----------------------------------------");
}
```

This method does not contain references to specific fields, but relies on two parameters instead. This method can now be called from the original methods:

```
public void PrintNameNicely()
{
	PrintNicely(_name, "Name is :");
}

public void PrintHeightNicely()
{
	PrintNicely(_height.ToString(), "Height is :");
}
```

We do introduce a very small complication, since the caller must now convert the data to a string. Converting an **int** to a **string** is however a very simple operation. The gain is absolutely worth the price, since we now only have to do the hyphen-to-star update in one single place! Our code is now much DRYer ☺.

Why is the new method **private** – why not make it **public**, and simply discard the old methods? Remember that public methods are used by the outside world, so deleting them may cause problems for others. Even changing them by e.g. changing the name and/or the required parameters can cause problems.

Are we done? Well, take a closer look at the **PrintNicely** method. Does it contain any repetition? The statements that print the top and bottom line are indeed identical. We could create yet another method:

```
private void PrintSeparator()
{
	Console.WriteLine("----------------------------------------");
}
```

and update the **PrintNicely** method accordingly:

```
private void PrintNicely(string data, string leadText)
{
	string dataLine = leadText + data;
	dataLine = dataLine.PadLeft(20);
	dataLine = dataLine.PadRight(40);
	dataLine = "|" + dataLine + "|";
	PrintSeparator();
	Console.WriteLine(dataLine);
	PrintSeparator();
}
```

Here the gain seems a bit more marginal, but we have in fact isolated the use of hyphens even further.

Could we go even further? There are still some spots of repetition left, for instance the use of the vertical separator line (**|**). Instead of writing it explicitly twice in the code, we could perhaps add a constant **verticalSeparator** to the class:

```
private const string verticalSeparator = "|";
```

So once again, we have isolated the specification of the vertical separator to one single place in the code. If you feel up to the challenge, you can probably find even more ways to improve the method.

A valid question here is of course: Is it worth the effort? Giving a definitive answer to that question is impossible. It will always be a matter of risk versus reward, and will depend on specific circumstances. If you don't care about (or don't want to pay for) such improvements to your code, you can in principle use that saved effort for something else. However, if you anticipate that the code will indeed need to be modified later on, that modification may require more effort, due to the less-than-optimal structure of the code.

Code modifications of this kind are another example of code **refactoring**. We saw a simpler refactoring much earlier in these notes (simple renaming of variables), and we are now stepping up to some not-so-trivial refactorings here. Remember that any refactoring is supposed to:

- Improve the structure of the code, making it easier to maintain and extend.
- Keep the functionality of the code unchanged.

With regards to when it is "worth it" to do such a refactoring, Martin Fowler suggests a "rule of three strikes" in his **Refactoring** book: When you write the same code the third time, you should definitely refactor. This could be a useful rule-of-thumb to start with.

DRY and classes (and a brief introduction to Inheritance)

We have now seen examples of how to eliminate code duplication inside a class, both with respect to instance fields and methods. Code duplication may however also occur across several classes. Suppose we are creating a banking system, where we need to model different types of bank accounts. We might have a standard bank account, with the below features:

1. You can deposit money into the account
2. You can withdraw money freely from the account, as long as the resulting balance is positive
3. You get an interest rate assigned once a year

We might also have a savings bank account, with the below features:

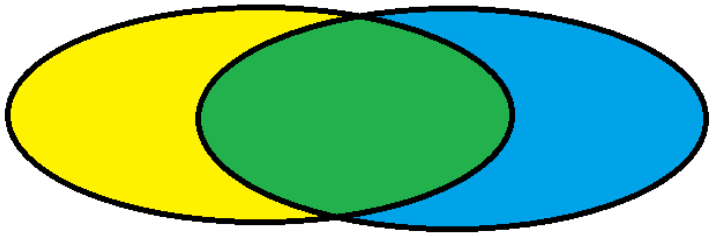
1. You can deposit money into the account
2. You can withdraw money from the account three times per month, as long as the resulting balance is positive
3. You get an improved interest rate assigned once a month

If we just go straight ahead and create code for these two types of accounts, we will probably create two classes **StandardAccount** and **SavingsAccount**. What will these classes look like, i.e. what instance fields and methods might they have? A reasonable suggestion could be that:

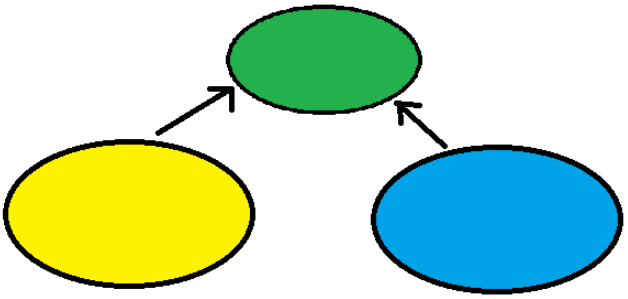
- Both classes have an instance field **_balance**, and a read-only property **Balance**
- Both classes have a **Deposit** method, and the implementation will be identical
- Both classes have a **Withdraw** method, but the implementation will be different, since different business rules apply
- Both classes have an **AssignInterest** method, but the implementation will be different, since different business rules apply

So, even though the individual classes might not contain any duplicate code, we will still have code duplication, in the sense that e.g. the **Deposit** method is implemented identically in two classes. Can we eliminate such cross-class code duplication as well? Indeed we can, by using a mechanism called **inheritance**. The concept of inheritance is one of the pillars of Object-Oriented programming. The ability to "share" common features between classes is extremely useful, and is a very powerful tool for solving the cross-class code duplication problem.

Consider again the two classes **StandardAccount** and **SavingsAccount**. We saw above that they have certain features in common, and certain features that are individual. We could illustrate this relationship like so:



Each ellipsis represent the code for one class: the yellow and blue areas will thus be code specific for one particular class, while the green area will be the code common to both classes. We would like to lift the green area out of the individual classes, and make the individual classes "refer" to the green area:



The way to express this in terms of object-orientation is as follows:

- We create a **base class** named **Account**. This class is supposed to contain those elements (instance fields, properties, methods, etc.) that are common to all bank accounts. This is the green area.
- We create two **derived classes** named **StandardAccount** and **SavingsAccount**, that only contains those elements that are specific for that particular bank account type. This is the yellow and blue area, respectively.
- We let **StandardAccount** and **SavingsAccount** inherit from **Account**. The inheritance mechanism has the effect that e.g. a **SavingsAccount** object will now contain both the common elements defined in **Account** and the specific elements defined in **SavingsAccount**.

We chose the phrase "a **SavingsAccount** object will now contain…" deliberately. Inheritance does not mean that the source code from the base class is somehow sucked into the derived class before compilation. The source code for **Account** will only exist in the **Account** class definition, which was the whole point. The effect – in terms of functionality – is however just as if we had duplicated the code. So once again, we improved the structure of the code, without changing the functionality.

How do you then specify inheritance in C#? In the example above, we would not be able to see any signs of inheritance, if we just looked at the **Account** class. That class would look like any other class, with some instance fields, methods, etc.. For the derived classes, we explicitly specify inheritance like this:

```
public class StandardAccount : Account
{
    // Rest of class definition follows here...
```

A colon, followed by the name of the base class. That's it. We have now specified that **StandardAccount** inherits from **Account**.

There is quite a lot more to know about inheritance than what we described here, but we will return to this shortly. For now, we just note that inheritance enables us to eliminate cross-class code duplication, and is yet another tool to DRY up our code ☺.

When does code become DRY?

As you are hopefully aware, these notes are about **Object-Oriented Programming**. Another discipline in Object-Oriented Software Development is **Object-Oriented Design**. These two disciplines are obviously related, but exactly how they are related and what activities they involve – and how these activities may overlap – is a matter of debate. The traditional standpoint could – very simplified – be stated like:

If we put a lot of effort into developing a sound and detailed Object-Oriented Design, the subsequent programming is almost trivial.

In other words: If you think hard and long about the domain you want to model with your software, you can probably figure out what classes you need, how they are related in terms of base classes and derived classes, what instance fields and methods you need, and so on and so forth…without writing a single line of code! Once you are done with this activity – which is traditionally known as Object-Oriented design – you will get the code right the first time!

Is that how it works in the real world? Rarely… In real life, you will typically create a design that looks "reasonable", start to create some code, discover some flaws in the design, rethink the design, rework and extend the code, and so on; a much more iterative approach, where the line between design and programming is blurred.

This state-of-affairs has been embraced by the so-called **Agile movement**. Their standpoint – again very simplified – is:

It does not make sense to separate design and programming – design happens and evolves hand-in-hand with code. We need to focus on structured ways to change the design of existing code.

Hopefully, you can see that an activity like refactoring is exactly that; a way to change the design of existing code. Refactoring is indeed considered to be one of the pillars of agile software development.

So, why all this high-level talk at this point? Think about the previous example, where we used inheritance to eliminate code duplication. If you subscribe to the traditional standpoint on design-vs-code, you would argue that code duplication should never have happened in the first place! You shouldn't market inheritance as a remedy for code duplication, since it is a tool for design purposes! We don't really want to take sides in this discussion, but either way, you may indeed find yourself in a situation one day, where you can eliminate code duplication by introducing inheritance. Also, the specific "mechanics" of inheritance are definitely in the realm of programming, and should be mastered by any object-oriented programmer, no matter if inheritance emerges as a result of up-front design or of code reorganization.

Exercises

Exercise	**Pro.2.1**
Project	BankV10
Purpose	Use simple **if**-statements in an existing class.
Description	This project contains a minimal **BankAccount** class, that sort of works. However, it has some problems…
Steps	1. Test the **BankAccount** class, by adding code in **Program.cs**. Specifically, make some tests that make the balance go negative. 2. Now change the code in the **Withdraw** method, such that a withdrawal is only done if the balance is larger than or equal to the given amount. Remember to test that the change works as expected. 3. This makes the **BankAccount** class more realistic, but there are still problems – you can call both **Withdraw** and **Deposit** with negative amounts (try it), which does not make much sense. Make changes to both methods, such that they only perform a withdrawal/deposit if the given amount is positive. Remember that for the **Withdraw** method, the change made in part 2 must still work! 4. Test that all your changes work as expected. 5. If we call **Withdraw** or **Deposit**, and detect an error situation, we don't do anything… What should we do?

Exercise	**Pro.2.2**
Project	WTF
Purpose	Use **if-else**-statements. Use method calls creatively.
Description	This project contains a class called **MysticNumbers**, with a single method **ThreeNumbers**. All that is known about **ThreeNumbers** is that it takes three integers as input, and returns one of them
Steps	1. By reading the code for **ThreeNumbers**, try to figure out what it does. Write some test code in **Program.cs** to see if you are right. 2. Write and test a new method **TwoNumbers**, that does the same thing as **ThreeNumbers**, but now only for two numbers. 3. Write and test a new method **FourNumbers**, that does the same thing as **ThreeNumbers**, but now for four numbers (hint: you can probably use the method **TwoNumbers** to make the code fairly short and easy). 4. Rewrite **ThreeNumbers** to use the **TwoNumbers** method. What code do you like best – the original code or the new code? 5. All the **…Numbers** methods only perform operations on the method parameters (also note that the **MysticNumbers** class does not contain any instance fields or properties). How can we then change the **…Numbers** methods, such that they are a bit easier to use (hint: we should add a single extra keyword to the definition of each method)?

<table>
<tr><td>Exercise</td><td>Pro.2.3</td></tr>
<tr><td>Project</td><td>WeatherStationV10</td></tr>
<tr><td>Purpose</td><td>Use multi-if-else-statements.</td></tr>
<tr><td>Description</td><td>This project contains a class called Barometer, containing two properties Pressure and WeatherDescription. The latter property gives an old-fashioned description of the weather, as a function of the pressure, according to this table:

<table>
<tr><th>Pressure (measured in hPa)</th><th>WeatherDescription</th></tr>
<tr><td>Below 980</td><td>Stormy</td></tr>
<tr><td>980-1000</td><td>Rainy</td></tr>
<tr><td>1000-1020</td><td>Changing</td></tr>
<tr><td>1020-1040</td><td>Fair</td></tr>
<tr><td>Above 1040</td><td>Very dry</td></tr>
</table>
</td></tr>
<tr><td>Steps</td><td>1. Implement the property WeatherDescription according to the table in the description, by using the multi-if-else statement. Write some test code in Program.cs to test it.
2. Try to implement WeatherDescription by using the switch statement with condition matching instead.</td></tr>
</table>

Exercise	**Pro.2.4**
Project	WhileLoopsBaseCamp
Purpose	Get some experience with **while**-loops
Description	The project contains some counter-controlled **while**-loops, and some number sequences that should be generated using **while**-loops.
Steps	1. In **Program.cs**, four **while**-loops (Case 1-4) are given. Try to figure out what the output from each loop will be. When ready, uncomment the line in each loop that prints the current value of the counter variable, and see if you were right. 2. Next follows Case 5-8. Here you must implement a **while**-loop yourself, to produce the number sequence given in the comment for each case.

Exercise	**Pro.2.5**
Project	CorrectChangeAutomat
Purpose	Use **while**-loops for a more complicated problem
Description	This exercise is about calculating the correct change when a customer pays a due amount with too much cash (yes, some people still pay with cash…). Example: A customer has to pay 266 kr., and pays with a 500 kr. bill. The customer must then receive 234 kr. in change. The tricky part is to figure out how to pay this amount using ordinary bills and coins, and paying back as few bills and coins as possible. In this example, the way to pay back correct change would be: • One 200-kr bill • One 20-kr coin • One 10-kr coin • Two 2-kr coins
Steps	1. Implement code to calculate and print out the correct change. To keep things simple, we assume that you only use 100-kr bills, 10-kr coins and 1-kr coins. Remember to test your code with a couple of different values for change. You can just add the code in **Program.cs**. 2. Once the above problem is solved, include some more bills and coins, like 50-kr bills, 5-kr coins, etc.. 3. If you used while-loops for solving the problem: Try to solve the problem without using loops.

Exercise	**Pro.2.6**
Project	RolePlayV20
Purpose	Get further experience with use of **while**-loops. Work with a project involving several classes.
Description	The project is supposed to model a very simple game, where a hero must battle against a beast, until either beast or hero is dead! The project contains four classes, which are described in general terms below – see the code for more details: • The **NumberGenerator** class, with the method **Next**. This is a helper class for generating random numbers. • The **BattleLog** class, where individual strings can be "saved", and later on printed out on the screen. • The **Hero** class, which models a game character. It is a very simple model, since it just has a number of hit points. • The **Beast** class, which also models a game character, in a way similar to the **Hero** class. Even though this is a very simple setup, it does include fundamental game mechanics from many popular role-playing games.
Steps	1. Study the classes in details, so you are sure of what they can do and how they work. Note how the **Hero** and **Beast** classes make use of the **Number-Generator** and **BattleLog** classes. 2. See if you can figure out how to code a battle between a **Hero** and a **Beast** (until the death!). A bit of code is present in **Program.cs**, but it obviously needs to be extended. 3. Once you can make the two game characters battle each other, there are a number of things to consider afterwards: a. It seems like the **Hero** wins most of the time (depending of course on how you coded the battle…). Why is that? How could we make the battle more fair? b. The damage dealt by the **Hero** is always between 10 to 30 points. How could we change that? Could we even let the creator of the **Hero** object decide this interval? Could this also be done for the number of initial hit points? c. Do we really need separate classes for **Hero** and **Beast**?

Exercise	**Pro.2.7**
Project	DrawShapes
Purpose	Get some experience with **for**-loops
Description	This exercise is about drawing some simple shapes on the screen, using **for**-loops to get the job done. A very simple class **DrawingTool** is provided to help with this.
Steps	1. Study the class **DrawingTool**. As you can see, it is very simple. Why are the methods static? 2. Using **for**-loops and the **DrawingTool** class, see if you can create code to draw the shapes A to E, as defined in the comments in **Program.cs**. NOTE: The shapes get increasingly hard to draw…

Exercise	**Pro.2.8**
Project	ListBaseCamp
Purpose	Get some experience with methods in the **List** class.
Description	This exercise is about predicting the result of applying some methods in the **List** class to a **List** object, and also about writing some code to use a **List** object
Steps	1. In **Program.cs**, a **List** object is created, and some elements are added and removed. At four points in the code (Case 1-4), you must predict the outcome of the **WriteLine** statement. When ready, you can uncomment the **WriteLine** statement, and see if your prediction was correct. 2. Following the cases above, four more cases are given (Case 5-8), where you must write code that uses the **List** object, to retrieve various information about the elements in the list. Details for each case are given as comments in the code.

Exercise	**Pro.2.9**
Project	RolePlayV21
Purpose	Get some experience with the **List** class. Work with a relatively complex setup of collaborating classes.
Description	In a previous exercise, we designed a battle between a Hero and a Beast. In this exercise, a Hero must now fight against an "army" of Beasts, represented by a new **BeastArmy** class. An "army" is essentially a **List** of **Beast** objects. In order to keep the original battle logic intact, we will try to implement properties and methods in the **BeastArmy** class, that correspond to properties and methods in the **Beast** class. However, the implementation does become more complex when dealing with several beasts… The code in this exercise will reach a level of complexity where it can become useful to use the Visual Studio debugger. Try to use it where you find it appropriate, and – once your code works – also try to use it to step through an entire battle.
Steps	1. Review the classes **Hero** and **Beast**. They are fairly similar to the **Hero** and **Beast** classes from the previous exercise, but do contain a few improvements, mostly in the form of more parameters to the constructors. 2. Review the battle logic code found in **Program.cs**. This code manages a 1-on-1 battle. Make sure you understand the logic of the code. 3. Implement the methods **DealDamage** and **ReceiveDamage**, plus the properties **Dead** and **BeastsAlive**, in the class **BeastArmy**. The specification of the methods and properties are found as comments in the code. 4. Change the code in **Program.cs**, to do a battle between a single Hero and an army of Beasts. It may take a bit of balancing (tuning the number of Beasts, their hit points, etc.), to make the battle reasonably fair. 5. Implement some sort of statistics functionality, to be able to measure if the battle setup is fair. You should e.g. be able to run 100 battles, and print out the percentage of battles won by the Hero and the Beast army, respectively.

Exercise	**Pro.2.10**
Project	LibraryV10
Purpose	Use the **List** class. Implement linear search.
Description	This exercise illustrates the concept of a repository. A repository is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the repository. The project contains the simple domain class **Book** (we consider the isbn number to be a "key" for **Book**, i.e. no two **Book** objects can have the same isbn number). Also, it contains the (incomplete) repository class **BookRepository**. The three public methods in **BookRepository** allow the user to add, look up and delete **Book** objects in a simple way (see the comments in the code for more details about each method). In this version of the exercise, the **BookRepository** class uses a **List** to store **Book** objects internally. The solution also contains a Unit Test project, which tests the public methods in the **BookRepository** class. Also remember that you can use the Visual Studio debugger to find and fix bugs in your code during development.
Steps	1. Complete the three methods in the **BookRepository** class, using the comments for each method as a guideline for implementation. 2. Build the application, run the Unit tests, and see if the Unit Test is "all green" (if not, you will have to review your code, and maybe debug it…). 3. Is there anything in your code that prevents a user from adding two **Book** objects with the same isbn value to the repository? 4. How could you prevent that **Book** objects with the same isbn value are added to the repository?

Exercise	**Pro.2.11**
Project	LibraryV11
Purpose	Use the **Dictionary** class.
Description	***NOTE: This exercise is intentionally almost identical to Pro.2.10*** This exercise illustrates the concept of a repository. A repository is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the repository. The project contains the simple domain class **Book** (we consider the isbn number to be a "key" for **Book**, i.e. no two **Book** objects can have the same isbn number). Also, it contains the (incomplete) repository class **BookRepository**. The three public methods in **BookRepository** allow the user to add, look up and delete **Book** objects in a simple way (see the comments in the code for more details about each method). In this version of the exercise, the **BookRepository** class uses a **Dictionary** to store **Book** objects internally. The solution also contains a Unit Test project, which tests the public methods in the **BookRepository** class. Also remember that you can use the Visual Studio debugger to find and fix bugs in your code during development.
Steps	1. Complete the three methods in the **BookRepository** class, using the comments for each method as a guideline for implementation. 2. Build the application, run the Unit tests, and see if the Unit Test is "all green" (if not, you will have to review your code, and maybe debug it…). 3. Is there anything in your code that prevents a user from adding two **Book** objects with the same isbn value to the repository? 4. How could you prevent that **Book** objects with the same isbn value are added to the repository?

Exercise	**Pro.2.12**
Project	SchoolAdministrationV10
Purpose	Use the **Dictionary** class. Work with an application containing several classes.
Description	The project contains the class **Student**. This is a simple representation of a student, with three properties; id, name and test scores. The first two are simple, but the "test scores" field is a **Dictionary**, holding key-value pairs of course names (string) and scores (int). The project also contains the class **StudentRepository**, which is supposed to be able to retrieve various information about the students; for this purpose, an instance field **_students** of type **Dictionary** is used to hold key-value pairs consisting of ids and **Student** objects (since a student is uniquely identified by an id) The solution also contains a Unit Test project, which tests the public methods in the **StudentRepository** class. Also remember that you can use the Visual Studio debugger to find and fix bugs in your code during development.
Steps	1. The class **Student** is complete, and you need not change anything in it. However, take a good look at the **Student** class anyway, and make sure you understand how the methods and properties work. Pay particular attention to the property **ScoreAverage**. 2. Look in the class definition of **StudentRepository**. It contains four methods (**AddStudent**, **GetStudent**, **GetAverageForStudent**, **GetTotalAverage**) that are not completed. Add code to complete these methods, according to the specification given in the comments in the code. 3. Build the application, run the Unit tests, and see if the Unit Test is "all green" (if not, you will have to review your code, and maybe debug it…).

Exercise	**Pro.2.13**
Project	HashSetBaseCamp
Purpose	Use the **HashSet** class. Work with set-oriented methods.
Description	This exercise is about using some of the set-oriented methods available on the **HashSet** class, specifically methods for calculating • Union • Intersection • Complement • Subset • Superset
Steps	1. In **Program.cs**, three **HashSet** objects **setA**, **setB** and **setC** are declared and populated with a few integer values. The helper method **PrintCollection** is also defined in **Program.cs**, and is used to print out the content of the three **HashSet** objects. 2. Your job is to write code to perform the set operations described in the **HashSet** chapter, i.e. the operations *Union*, *Intersection*, *Complement*, *Subset* and *Superset* (see the table in the chapter for further details). All operations can be done using methods from the **HashSet** class, but the methods do in some cases have names that are somewhat different from the name of the operation… Part of the exercise is therefore also to find the correct method to call. You can – for some operations – use the **PrintCollection** method to verify your results.

Exercise	**Pro.2.14**
Project	Flinter
Purpose	Use enumerations
Description	**Flinter** is supposed to be the start of a new dating app. You can create profiles for those you are interested in meeting. In the project, the class **Profile** has been included. The class contains instance fields for gender, eye color, hair color, and height. You can thus create a **Profile** object by specifying values for each of these four fields in the class constructor. Furthermore, you can get a text description of a **Profile** object by using the property **GetDescription**.
Steps	1. Code that tests the **Profile** class is included in **Program.cs**. Examine the code in the **Profile** class definition, and see if you can predict the outcome of running the test. 2. Reflect a bit on the types chosen for the instance fields, and also run the test. Several problems can be identified: a. In two test cases, we specified hair color where we should have specified eye color, and vice versa (unless you really want a partner with white eyes and blue hair…) b. In one test case, we specified a height category that doesn't exist. c. Using a boolean for gender specification seems very restrictive and indeed "binary". 3. Change the **Profile** class definition by adding enumerated types for gender, eye color, hair color and height category. Use these new types for the four instance fields. The constructor needs some changes as well. Also consider if you still need the properties **GenderDescription** and **HeightDescription**. 4. Change the code in **Program.cs**, so it is compatible with the redesigned **Profile** class. Observe how it is now only possible to specify legal values for each type. 5. Reflect a bit on the changes. Is there anything in the new code that is more complicated than it was in the original code? Was it always relevant to use an enumerated type?

Exercise	**Pro.2.15**
Project	CalculationSimulation
Purpose	Improve code structure by replacing values with constants, instance fields and parameters
Description	The project contains a simple simulation of a calculation. The intention is to simulate a calculation that takes about half a second. The calculation takes two values x and y, and returns an integer value. In order to speed up the calculation, a "cache" class is also provided. The idea is that once a calculation has been done, the result can be stored in the cache, from which it can be retrieved very quickly. See the code for further details.
Steps	1. The code is set up to do calculations in a 5x5 table (that is, x and y can be numbers between 0 and 4, both included). How many places in the project would you have to change something, if you want to do calculations in a 10x10 table instead? 2. Change the code, such that you get rid of all the instances of the number 5 in the methods. This could be done by using constants, instance fields and parameters. 3. Are there other values that are candidates for being replaced with constants or parameters? If so, make the necessary updates to the code.

Exercise	**Pro.2.16**
Project	WebShopV10
Purpose	Improve code structure by creating new methods
Description	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 found in the code in the project. In the project, the **Order** class contains an item list. For simplicity, the item list just contains the net price for each item in the order. The class also contains a property **TotalOrderPrice** for calculating the total price for the order. The solution also contains a Unit Test project, which tests the property **TotalOrderPrice** in the **Order** class.
Steps	1. The implementation of the **TotalOrderPrice** property is less than optimal. Rewrite it, with the intent of: a. Removing duplicate code b. Making the method easier to understand 2. Make sure to regularly build the application, run the Unit tests, and confirm that the Unit Test is still "all green". If not, you will know that your most recent change caused the tests to fail.