Object-Oriented Programming with C#

Object-Oriented Programming, Part I

By Per Laursen
24-08-2024

Introduction

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

The Object concept

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

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

State and Behavior

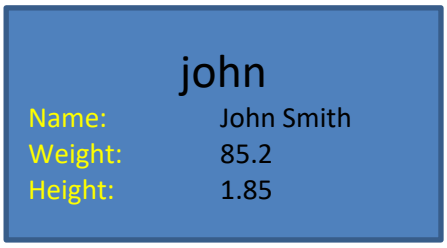
The fundamentally new idea in OO-programming is to join both data and logic into single units (objects). In the OO-world, some different terms are used:

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

It requires a bit of practice to be comfortable with these terms. As a specific example, suppose we have an object called **john** in our code. For now, we don't worry about how such an object is created – we'll learn that very soon. We already know that an object has a type – the type for the object **john** is called **Human** (very soon, we will also learn where such a type as **Human** comes from). So, the object **john** is supposed to represent a human being. More precisely, we can say that the object **john** represents a model of a human being, i.e. that model which is defined by the type **Human**.

Exactly what this model contains will be very situation-specific. In some applications, we may only need a very crude model of a human being, which only contains a little bit of state-and-behavior. Other applications may require use of a much more detailed model.

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



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

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

Public and private appearances

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

$$\text{BMI} = (\text{weight in kilograms}) / (\text{height in meters})^2$$

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

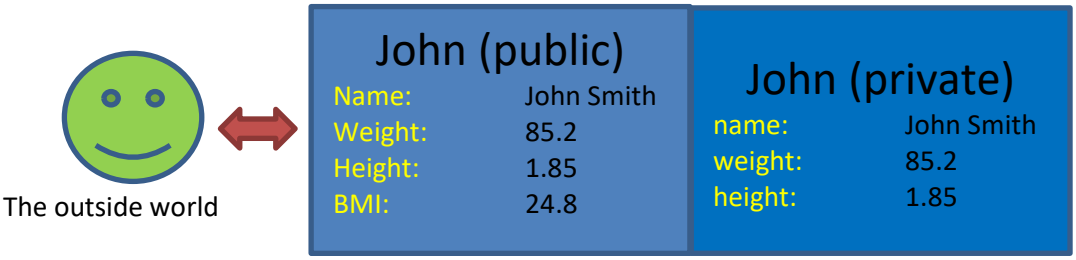
- The object would use a bit more memory
- If you update the value of either **Weight** or **Height**, you must also update the value of BMI
- You may risk that the information inside the object becomes inconsistent!

The last two points are closely related; you should definitely[1] avoid having redundant data in an object! Not only simple, duplicated data, but also data that can be calculated from other data. So, no fourth variable inside **john**!

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

[1] "Definitely" is a bit dogmatic, since there are certain cases where it might be a reasonable compromise to explicitly store data that could be calculated, e.g. if the calculation is very time-consuming, and the value is typically used many times before it needs to be recalculated.

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



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

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

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

“Hi, my name is John Smith. My height is 1.85 meters, and my weight is 85.2 kg”.

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

The Class concept

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

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

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

Once a class definition has been completed, it will be possible to create objects of this particular class. The type of such an object is thus said to be the name of the class, i.e. the type of the object **john** is **Human**. Note that we now have two categories of types in our type "universe": the "simple types" like **int** and **bool**, and the "class types" like **Human**. Instances of these types have fundamentally different behaviors, which we will investigate in more detail shortly.

It makes sense to distinguish between the creator of a class definition and a client of a class (by "client" we here mean a part of the code which needs to use objects of a particular class). The creator will of course know about all details – both public and private – of the class, while the client only needs to know about the public parts. From the client's perspective, the class is a kind of "black box"; the client can create objects of a particular class, and can interact with the objects through the public parts (often called the interface) defined in the class definition. However, the client cannot – and should not be able to – obtain information about the internal structure of the class.

This separation of the public and private side of a class also enables a certain degree of freedom, with respect to changes in code. If a client uses objects of a certain class, the client only uses the public part. So, as long as the public part remains unchanged, it is possible for the creator of the class to update the code inside the class definition, e.g. to fix errors or improve performance, without affecting the client.

Using objects of an existing class

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

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

There are various ways to try to navigate your way through a class library, and the easiest solution is often to use Google (or whatever search engine you prefer) for the job. Microsoft has recently made some effort to create a "portal" for .NET documentation[2], which is also a good starting point. The mechanics for making a class available for use in your project can vary a bit, and we take a closer look at that later. For now, we will just assume that class definitions can be made available.

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

```
Student firstStudent = new Student();
```

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

[2] https://docs.microsoft.com/da-dk/dotnet/

```
int age = 18;
```

The overall structure of these two lines is actually the same:

- A type name (remember that a class is also just a type)
- A variable name
- The assignment operator (=)
- A right-hand side

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

```
new Student();
```

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

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

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

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

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

```
string name = firstStudent.Name;
```

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

```
Student firstStudent = new Student();
Student secondStudent = new Student();

firstStudent.Name = "Allan";
secondStudent.Name = "Jane";

Console.WriteLine(firstStudent.Name);
Console.WriteLine(secondStudent.Name);
```

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

```
Student.Name = "Carl";
```

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

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

The example above illustrated how to retrieve a property from an object, i.e. a part of the state of the object. How about behaviors? Behaviors – in the form of methods – are invoked (or **called**, as we will usually say), in a very similar manner. Suppose we

have defined a method **PrintInformation**, that does just that; prints out information about the object in a human-friendly way. Calling this method will look like:

```
firstStudent.PrintInformation();
```

This is very similar to what we just saw for properties, except for a subtle difference: the parentheses following **PrintInformation**. When a method is defined, the author can choose to define that the method requires a number of **parameters**. You can think of a method parameter as a special kind of variable, which can be used to pass data to the method when the method is called. Some methods do not require parameters; the **PrintInformation** method finds the information it needs inside the object on which it is called, so the caller of the method need not provide any extra information. A method that does not need extra information – i.e. it takes zero parameters – will be called as above, using the method name followed by an empty set of parentheses.

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

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

Are the values 3 and 7 then "parameters" to the **Add** method? Actually not. We usually distinguish between method **parameters** and method **arguments**.

- The term **parameter** is used in relation to the definition of a method; we can e.g. say that the definition of the **Add** method states that **Add** takes two parameters.
- The term **argument** is used in relation to calling a method; we can e.g. say that we called the method **Add** with two arguments: 3 and 7.

An argument to a method will thus be a specific value (or an expression which is evaluated before the method is called), which is then passed to the method through the method parameters. If multiple arguments need to be specified, they are simply written one after another (separated by comma) within the parentheses.

We used two specific values in the example, but we can actually put any sort of expression into an argument list, as long as the type of the result matches the expected type of the argument! This is also a very important point. We have here assumed that the **Add** method takes two integer values as arguments. A call like the below would therefore be invalid:

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

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

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

Again; we can put any expression we can conjure up into an argument list, as long as it evaluates to a value of the expected type, i.e. the type of the corresponding parameter.

Code Quality, part II

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

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

Further on object creation

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

```
Student firstStudent = new Student();
```

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

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

The second part doesn't really hold up, if you think about it. We have not given any details about the definition of the **Student** class, but it would be reasonable to expect it to provide several properties, for instance **Name**, **Address**, **DateOfBirth**, and so on. If you create a **Student** object as defined above, what state will it then be in? What would the **Name** property return? Probably nothing (e.g. an empty string), because we have not provided any information as part of the creation process! That is hardly a realistic modeling of the real world. In a school administration system, you would probably not be able to create a new student in the system, unless some minimal amount of information is available (e.g. name, address and social security number).

Likewise, it seems reasonable to enforce a similar restriction on creation of **Student** objects. We should not be able to create a **Student** object with no information in it, since such an object is meaningless. Fortunately, the class creator can enforce such restrictions. With the knowledge we now have, we can see that the code for object creation looks similar to a method call:

```
Student firstStudent = new Student();
```

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

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

In this case, we must provide two arguments (i.e. actual values for name and year-of-birth) in order to create a **Student** object. Now it seems more plausible that a just-created **Student** object will be in a meaningful state from the moment it is created.

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

Value types and Reference types

Before we dive into how to create class definitions ourselves, we need to understand a fundamental difference between objects and variables of so-called **simple types**.

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

```
int age;
```

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

```
age = 18;
```

We can even do both in one line, like:

```
int age = 18;
```

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

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

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

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

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

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

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

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

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

```
int age1 = 18;
int age2 = 21;
```

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

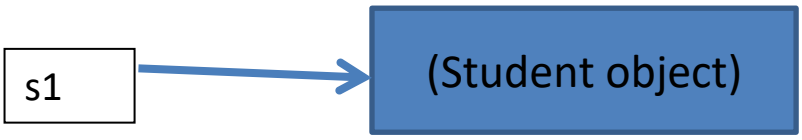
We can change the values of the two variables as we wish, even using the value of one variable to set the value of the other:

```
age1 = age2;
age2 = 23;
```

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

```
Student s1 = new Student();
```

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



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

```
Student s1 = new Student();
Student s2 = s1;
```

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

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

```
s1.Name = "John";
s2.Name = "Allan";

Console.WriteLine(s1.Name);
```

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

```
Student s2 = s1;
```

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

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

```
Student s1 = null;
```

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

```
Console.WriteLine(s1.Name);
```

s1 does not refer to any object, so there is no way to retrieve a name… If you try this, Visual Studio will respond with an error message reporting a "null reference exception". Trying to use a null reference is a very common error in programming, and is something you will need to anticipate and handle. We do not really have any tools to properly handle such a situation yet, but a slight modification of the line above will actually make it more robust w.r.t. handling **null**:

```
Console.WriteLine(s1?.Name);
```

Note the addition of the question mark just before the "dot". The question mark and the dot combined (i.e. "**?.**") is known as the **null-conditional operator**. That sounds very fancy; what it does is however fairly simple:

- If the variable just before the operator (in our example: **s1**) is **null**: do nothing.
- Otherwise, do the part after the operator (in our example: get the value of the **Name** property)

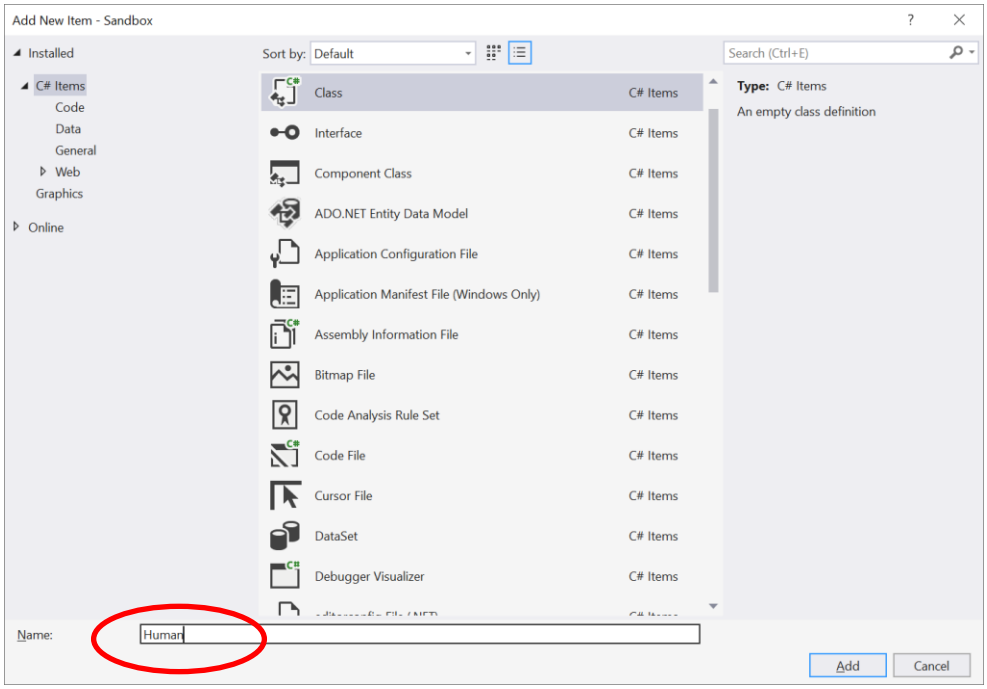
This operator makes it very simple to prevent null pointer exceptions, but it is not a silver bullet; you still need to consider what should happen if a variable is indeed **null** at some point. In some cases it might be as intended, but it may in other cases be a symptom of an underlying problem in your logic, which you should then find and solve by other means.

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

Class definition elements

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

With that in mind, we will embark on the – somewhat large – topic of creating classes from scratch. Visual Studio will help you get started: Highlight a project in the **Solution Explorer** window – this could be the **Sandbox** project we have used previously – right-click to bring up the context menu, and choose **Add | Class**. Choosing this entry brings up a dialog window, where you simply enter the name of the class you wish to create, in the text box at the bottom:



To create a class called **Human**, simply type "Human" (without the "") into the box, and hit the **Add** button.

This sets a couple of things in motion in Visual Studio. A new file called **Human.cs** is added to the project, and the file is also opened by Visual Studio in the editor area. The initial content of the file should look like this (there might also be some lines at the top of the file starting with **using**; ignore those lines for now):

```
namespace Sandbox
{
	internal class Human
	{
	}
}
```

This may look somewhat confusing, and certainly contains several elements we have not seen before. We will talk about these elements later on, but for now, please do the following:

1. Delete the entire content of the file **Human.cs**
2. Replace it with the below code:

```
public class Human
{
}
```

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

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

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

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

Instance fields

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

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

```
public class Human
{
      private string _name;
}
```

This sort of looks like what we have seen before, with a few additions. The keyword **private** indicates that this instance field can not be accessed from the outside. That is, if you create a **Human** object, and try to get hold of the value of the instance field (or try to change it), the compiler will consider that code to be in error, and therefore uncompilable. Again, you may wonder why you would create an instance field and then hide it away; the main reason is that we often want access to such an instance field to go through a so-called **property**. We have mentioned these properties in the previous chapter, and we shall see in a moment how to create a property. Also, we have prefixed the name of the instance field with an underscore (_). This has over time emerged as a standard for naming instance fields in a way that distinguishes them from plain variables. We encourage you to follow that standard as well.

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

```
public class Human
{
      private string _name;
      private double _height;
      private double _weight;
}
```

Properties

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

```
string name = firstStudent.Name;
```

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

```
public string Name
{
      get { }
      set { }
}
```

If you add this code *as-is* to the **Human** class definition, Visual Studio will inform you that the code is invalid… Don't worry; we will fix this in a moment. First, let's take a step back and see how such a property can be used by someone who has created a **Human** object.

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

```
Human firstHuman = new Human();
```

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

```
firstHuman.Name = "Adam";
Console.WriteLine(firstHuman.Name);
```

What happens in these two lines of code? It seems like the **Name** property is part of the state of any **Human** object. The type of that part of the state is **string**, and we can perform certain operations with the value of that part of the state. We can

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

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

```
public string Name
{
      get { }
      set { }
}
```

What we now need to figure out is what statements we need to write, in order to enable the two operations. The statements needed to "get" the value should be written between the **{}** in the line with the **get** keyword, and likewise for **set**. Note that this is very similar to writing a function definition, with good reason. The **get**- and **set**-part of a property definition are actually just functions, even though they look a bit different than any function definitions we have previously seen. Imagine that we had two functions named **get_Name** and **set_Name** in the **Human** class definition, defined like this:

```
string get_Name() { }
void set_Name(string value) { }
```

These definitions are not complete yet. The intention of **get_Name** seems to be to return the value of the name stored in the **Human** object on which the function is called. Where would we find this value? Well, the **_name** instance field seems to be a good candidate! Let's go with that:

```
string get_Name() { return _name; }
```

What about **set_Name**? It seems the intention is to set a new value for name in the **Human** object on which the function is called. This new value is provided through the parameter named **value**. Where should we then store this value in the object? In the **_name** instance field! Let's do that:

```
void set_Name(string value) { _name = value; }
```

Now we have a mechanism for setting a value, and retrieving it again. Is this better than just accessing the **_name** instance fields directly? Not really in this case. However, we have achieved two things:

1. Details about how the value is stored in the object are hidden from the client of the class
2. We can add any logic we wish to the basic assignment and retrieval operations

Why are these worthy achievements? W.r.t. the first point, suppose that we at some point discovered that the name value could be stored in a more efficient way (hard to imagine with a simple string, but please play along…). Since the client does not access the name value directly, we are free to change the way we store it internally, without affecting the client! This becomes a very important point when things get more complicated. W.r.t. the second point, suppose that we – somewhat arbitrarily – defined that a name must be at least two characters long. We can actually enforce this rule by adding a bit to the definition of **set_Name**:

```
void set_Name(string value)
{
    if (value.Length > 1)
        _Name = value;
}
```

We haven't quite learned enough C# to fully understand this code, but the point is that we are able to "guard" such get/set operations in whatever way we find appropriate, and thereby taking ownership for enforcing such rules. This is much better than leaving it to the client to enforce such a rule.

So, the rationale for defining properties with get/set operations that are actually functions is hopefully clear. We can then return to how this will actually look for a real property. We left off with this incomplete definition:

```
public string Name
{
      get { }
      set { }
}
```

Now, remember that **get {}** is actually just a "shorthand" (such shorthands are often called **syntactic sugar**[3]) for **string get_Name() {}**. We can then simply move the code from above into the definition:

```
      get { return _name; }
```

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

Likewise, **set {}** is a shorthand for **void set_Name(string value)**, which gives us:

```
set { _name = value; }
```

The complete definition of the **Name** property then becomes:

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

Remember that for a property, the **set** operation is always called when an assignment statement involving a property is performed, like:

```
firstHuman.Name = "Adam";
```

value is then simply set to the value of the right-hand side, be it a simple value or an expression. In this case, **value** will be set to "Adam", meaning that the **set** operation will update the value of **_name** to "Adam", and thereby changing the state of the object. For **get** and **set** operations in general, the rules of invocation are:

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

With all this in place, we could go ahead and extend the **Human** class definition further, e.g. by adding **Weight** and **Height** properties in a very similar way. What about a **BMI** property (recall that BMI was defined as BMI = weight / height^2)? How would we implement this property? First the skeleton code for a **BMI** property:

```
public double BMI
{
      get { }
      set { }
}
```

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

```
public double BMI
{
        get { return _weight / (_height * _height); }
        set { }
}
```

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

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

```
firstHuman.BMI = 20.8;
```

Yes, we can! We simply remove the **set** operation from the **BMI** property definition:

```
public double BMI
{
        get { return _weight / (_height * _height); }
}
```

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

Auto-properties

So, how often do we define properties in the "default" way like we did for **Name**, and how often do we need special handling like for **BMI**? It's hard to say anything definitive, but it's very likely that you will often just need the "default" implementation; one instance field, and "default" implementations of both **get** and **set**. Since this is so common, there is also a "shorthand" for this called **auto-properties**.

The auto-property version of the **Name** property looks like this:

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

That's it! You do not need to define a corresponding instance field yourself; this is generated automatically. Can you somehow get hold of this auto-generated instance field…? No, and you shouldn't need to. If you have a reason to access the instance field directly, then you have a case where auto-properties isn't the correct solution. Since the definition becomes so simple, it is also common practice simply to put it on a single line:

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

When seeing something like this, you might feel tempted to simply use a (public) instance field instead, since there seems to be very little gain using a property. Still, remember that using properties is generally a sound principle: we hide all details about how the value is stored and retrieved, and are thus free to change this later on, without affecting the client of the class! We might start out by implementing a property using this simple auto-property style, but we might later on switch to a more complex implementation, while retaining the "interface" to the class. Auto-properties are just an implementational convenience, nothing more.

Methods

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

```
Student aStudent = new Student();
SchoolAdminSystem adminSystem = new SchoolAdminSystem();

// aStudent is updated with relevant data (code omitted…)

adminSystem.AddStudent(aStudent);
```

The last line contains a call of a method named **AddStudent**, which hopefully does what we expect: add a new **Student** object to the collection of existing objects. In this way, we are invoking a **behavior** for the object, by calling a method.

How do we then define such methods? A method is essentially a collection of statements, that can be invoked by calling the method on a specific object. A method definition will always contain these elements:

- **An access specifier**: We have seen before that certain parts of a class definition can be declared as being **public** or **private**. This goes for methods as well. Some methods should be available for clients of the class, and should then be marked as **public**. Other methods may be "helper methods", that only exist for making the implementation of public methods easier; these methods should be marked as **private**.
- **A return type**: We saw that the **get** operation in a property definition needs to specify the value it returns. Methods may also return values to the caller, and we need to specify what type (e.g. **string**, **int** or maybe a class type) this value will have.
- **A method name**: Just as for e.g. variables, we need to give our method definition a name, so we can refer to it when we wish to call it. As mentioned before, the name should be as descriptive as possible, and be written in CamelCase with a capital first letter.
- **A list of parameters**: We have already seen that some methods may require the caller to specify a list of arguments, in order to pass actual values to the method being called. In the method definition, a corresponding parameter list must be specified, including the type for each parameter.
- **A method body**: "body" here means the C# statements inside the method definition. When the method is called, the statements will be executed.

Let us see an example of a method definition. At this point, we only know a few rather simple C# statements, so we can only create very simple methods. A - somewhat contrived – method on a **Human** class could be a method that returns whether or not this individual is higher than a given average:

```
public bool HigherThanAverage(double averageHeight)
{
    bool isHigher = _height > averageHeight;
    return isHigher;
}
```

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

- **Access specifier**: Is **public**, so an external client can call this method.
- **Return type**: Is **bool** – we can also see that the expression (in this case just a variable) after the **return** keyword indeed has the type **bool**.
- **Method name**: Is **HigherThanAverage**
- **List of parameters**: One parameter named **averageHeight**, of type **double**.
- **Method body**: The two lines of code between the { and the }.

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

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

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

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

```
public void AddAndPrintResult(int a, int b)
{
      int result = a + b;
      Console.WriteLine(result);
}
```

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

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

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

Constructors

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

```
Human firstHuman = new Human();
```

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

This “initialization code” is also written as part of the class definition, in the form of a so-called **constructor**. A constructor is also just a method, but with a special syntax and some limitations compared to ordinary methods. For our **Human** class example, the simplest possible constructor looks like this:

```
public Human()
{
}
```

This is as simple as it gets. Compared to an ordinary method, we again see an access specifier as the first part. However, there does not seem to be a return type… That is the first important difference; you do not specify a return type for a constructor. Next, we see the word “Human”, i.e. exactly the same name as the class. This is also a defining characteristic for a constructor; it always has the exact same name as the class it is defined in. Next follows the parameter list. The list is empty in this example, but we are indeed allowed to specify parameters to a constructor, in the same way as for ordinary methods. Finally follows the constructor body, on which there are no restrictions either.

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

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

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

```
public class Human
{
      public string Name { get; set; }
      public double Height { get; set; }
      public double Weight { get; set; }
}
```

then it would seem that the simple constructor – which does nothing – is not good enough. What would the initial values of the three properties be? C# does set the initial value for e.g. a **double** to a well-defined value (0.0), and for a **string** to the empty string, but that is hardly meaningful either. What then?

One solution could be to set the values to some sort of "default" values, like:

```
public Human()
{
	Name = "Adam";
	Height = 180.0;
	Weight = 80.0;
}
```

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

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

```
public Human(string name, double height, double weight)
{
	Name = name;
	Height = height;
	Weight = weight;
}
```

Note how handy it is that we defined that names for properties should always start with a capital letter, since we can then distinguish them from the constructor parameters. By adding these parameters, the previous simple statement **new Human()** becomes invalid and uncompilable. It now becomes mandatory for the caller to provide the information needed to create a meaningful **Human** object:

```
Human firstHuman = new Human("Adam", 180.0, 80.0);
```

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

Does the above then imply that the constructor should always have as many parameters as there are settable properties in the class, so we can properly initialize all of them? Not necessarily. Suppose that we extend the **Human** class definition with a property named **Level** (you could imagine that the **Human** class is part of a game project, using a typical game mechanic where you are able to "level up" a **Human** as

the game progresses). What would a proper initial value for **Level** be? If we imagine that the game allows you to customize your **Human** w.r.t. name, weight and height, but also restrict all **Humans** to start in Level 1, then it seems that **Level** should always be initialized to 1. If that is so, we don't need to provide that value as a parameter to the constructor:

```
public Human(string name, double height, double weight)
{
	Name = name;
	Height = height;
	Weight = weight;
	Level = 1;
}
```

This is a perfectly valid way to initialize **Level**, but there is in fact another way to achieve this. The **Level** property will look something like this:

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

In order to always initialize the value of **Level** to 1, you can add this to the definition:

```
public int Level { get; set; } = 1;
```

and then omit the initialization in the constructor. Which option is best? They are – except for some rather technical nuances we cannot discuss at this point – equivalent, so it's most a matter of taste. Some prefer to have all initialization done inside the constructor, which is the style we will usually adhere to in these notes.

These variations aside, the key point remains: the client of a class should only need to provide initial values for those properties that are individual for an object, like name, weight and height in the example. Fixed-value initialization is an internal matter.

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

Class collaboration, and a bit about Abstraction

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

```
public class Car
{
	public string ModelName { get; set; }

	private Engine _theEngine;
	private NavigationSystem _theNavigationSystem;
	// other instance fields would probably follow...

	public Car(string modelName)
	{
		ModelName = modelName;
		_theEngine = new Engine();
		_theNavigationSystem = new NavigationSystem();
		// (rest of constructor)
	}

	public void Start()
	{
		_theEngine.Start();
		_theNavigationSystem.Start();
		// (rest of Start method)
	}

	// other methods would probably follow...
}
```

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

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

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

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

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

Static – no object needed

Over the last pages, we have again and again insisted that you define methods in classes, and call methods on objects. This is a very clean distinction, but there are in fact situations where you don't need objects in order to call methods. Suppose you have a class called **SimpleMath**, which contains simple methods for addition, subtraction and so on. The "header" of an **Add** method will probably look like:

```
public int Add(int a, int b)
```

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

```
public static int Add(int a, int b)
{
      return (a + b);
}
```

We can now call the method like this:

```
SimpleMath.Add(2, 6);
```

We do not create a **SimpleMath** object, but simply call the method "on the class". This may seem confusing, now that we have been accustomed to calling methods on objects, but it is an important feature to know. Consider this line of code:

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

This should be quite familiar by now, but…what is **Console** actually? **Console** is the name of a class, not an object! If you investigate the .NET Class Library further, you will find that static methods are quite common. The very useful **Math** class is filled with static methods.

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

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

```
public class CardDeck
{
      public const int CardsInDeck = 52;
}
```

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

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

Exercises

Exercise	**OOP.1.1**
Project	MovieManagerV10
Purpose	Observe how to use an existing class. Implement simple use of an existing class.
Description	In this version of the movie manager application, a class named **Movie** has been added (in the file **Movie.cs**). It contains an absolute minimum of information about a specific movie. The class is put to use in **Program.cs**, where some **Movie** objects are created and used.
Steps	1. Load the project, and open **Program.cs**. You will see that some code is already present. See if you can figure out what goes on in each line of code. If you hover the mouse cursor over a specific element, you should see some useful information pop up. Make sure you understand where • Objects are created • Arguments to the constructors are specified • Properties are used • Methods are called 2. Feel free to create additional **Movie** objects, and exercise them a bit (call methods, use properties, etc.) 3. We can only change the value of **NoOfViews** by calling the **Watch** method. Consider why it would be a bad idea to allow a user of the class **Movie** to change the value of **NoOfViews** directly, e.g. like this: `movieA.NoOfViews = movieA.NoOfViews + 1;`

Exercise	**OOP.1.2**
Project	BankV05
Purpose	Implement minor additions to an existing class.
Description	The project contains a minimal **BankAccount** class. The class is put to use in **Program.cs**, where a **BankAccount** object is created and used.
Steps	1. Load the project, and take a look at the **BankAccount** class. Make sure you understand all the elements in the class definition. Then take a look at how the **BankAccount** class is used in **Program.cs**. 2. We now want to add an extra property to the **BankAccount** class: the name of the account holder. Add this feature to the class. This will probably involve: a. Adding a new instance field b. Adding a property which uses the new instance field c. Updating the constructor, such that the name of the account holder must be specified when a **BankAccount** object is created. 3. Once the class has been updated, make sure to test the new feature by updating the code in **Program.cs**. More specifically, you should test that you must now specify a name when creating a **BankAccount** object, and that you can retrieve the name by using the property added in step 2.

Exercise	**OOP.1.3**
Project	RolePlayV10
Purpose	Implement non-trivial additions to an existing class
Description	The project contains a **Warrior** class, which is initially very simple – it only contains a **Name** property. We now need to extend the class with an additional feature.
Steps	1. Start out by taking a look at the **Warrior** class. Make sure you understand all the elements in the class definition. Specifically, make sure you can identify: a. An instance field b. A property c. The constructor 2. Next, take a look at how the **Warrior** class is used in **Program.cs**. This is a very small test of the class. 3. We must now extend the **Warrior** class with a "level" feature. The requirements for this feature are: a. All warriors start at level 1. b. The level can be retrieved freely, but not changed freely. c. It must be possible to increment the level, i.e. increase the value of the level by 1. 4. Implement this feature in the **Warrior** class. You will need to consider if a. An additional instance field is needed (Hint: we need to store the current level of a **Warrior** somewhere) b. An additional property is needed (if so, do we need both the **get** and the **set** part? Hint: We only require that the value of the level can be retrieved through the property, not changed) c. The constructor should be updated (Hint: The constructor should initialize all instance fields with a well-defined value. What would that value be for level, given requirement 3a?). d. A method for incrementing the level is needed (Hint: the level should always just be increased by 1). 5. Test the updated **Warrior** class, by adding some code in **Program.cs**.

Exercise	**OOP.1.4**
Project	RolePlayV11
Purpose	Implement non-trivial additions to an existing class
Description	The project contains a **Warrior** class, including the functionality described in the previous exercise (a "level" feature).
Steps	1. We must now extend the class with a "hit points" feature. Details of this feature are: a. Hit points are set individually when a warrior is created. b. Hit points can be retrieved freely, but not changed freely. c. It must be possible to decrease hit points by a specified amount. This corresponds to the warrior being damaged by someone 2. Implement this feature in the **Warrior** class. You will need to consider if a. An additional instance field is needed (Hint: we need to store the current hit points of a **Warrior** somewhere) b. An additional property is needed (if so, do we need both the **get** and the **set** part? Hint: We only require that the value of the hit points can be retrieved, not changed) c. The constructor should be updated (Hint: The constructor should initialize all instance fields with a well-defined value. What would that value be for hit points, given requirement 1a? How can we provide this value to the constructor? Extra hint: How is the value of the warrior's name provided to the constructor?). d. A method for decreasing the hit points is needed (Hint: how can the specific amount to decrease the hit points with be provided to the method?). 3. Implement a property called **Dead**, which returns a boolean value. The property should return **true** if hit points are equal to or below zero (Hint: we only need to include the **get** part of this new property). 4. Test the updated **Warrior** class, by adding some code in **Program.cs**. 5. [Hard] A **Warrior** should not only receive damage, but also deal damage! Implement a method **DealDamage**, with these requirements: a. Returns an integer value b. Does not take any parameters c. The returned integer value should be a random number between 10 and 30 (Hint: Figure out how the **Random** class from the .NET class library works). 6. [Very hard] See if you can make two warriors battle against each other, to the death!

Exercise	**OOP.1.5**
Project	ClockV10
Purpose	Implement a class from scratch, including use of the class
Description	This project contains an empty class definition **Clock**. Your job is to implement the class **Clock**, given the below requirements: 1. The clock should be able to display (i.e. print on the screen) hours and minutes. 2. The clock should use the 24-hour system. 3. It must be possible to set the clock to a specific time. 4. It must be possible to retrieve the current time from the clock. 5. It must be possible to advance the clock by a single minute. **NB!** To avoid confusion: the class should only be able to "simulate" a clock, i.e. it should not in any way contain code that retrieves the real-world clock from the computer. A clock will only "tick" when you call a method which changes the time, as stated in requirement 5).
Steps	1. Implement requirements 1-4. This will involve figuring out what instance fields, constructor, properties and methods you need for this. Remember to include code in **Program.cs** for testing the class. 2. Implement requirement 5. In this case, it becomes quite important to choose relevant test cases! Note that you may need to rethink how you represent time in the **Clock** class (Hint: maybe a single instance field is enough…?).

Exercise	**OOP.1.6**
Project	DiceGame
Purpose	Work with a project containing collaborating classes
Description	This project contains three classes: **RandomNumberGenerator**, **Die** and **DiceCup**. • The **RandomNumberGenerator** class contains code which can generate random numbers within a given interval. The class is already completed, and we will not be concerned with the details of this class. • The **Die** class represents a 6-sided die, and is already completed. • The **DiceCup** class needs a bit of work to be complete. The **DiceCup** class uses the **Die** class.
Steps	1. Take a look at the **Die** class. It is complete, and fairly simple. Note how the **Die** class uses the **RandomNumberGenerator** class. 2. Open the **DiceCup** class. Note how the class contains two instance fields of type **Die**. Also note the constructor – what happens there? 3. The **DiceCup** class is not complete. Implement the **Shake** method and the **TotalValue** property, as specified in the comments in the code. Test that your code works as expected, by creating and using a **DiceCup** object in **Program.cs**. 4. How much would we need to change in **DiceCup** in order to have a dice cup with three dice? 5. [Hard] When we create a **DiceCup** object, we would also like to be able to specify the number of sides the dice should have. Implement the changes in **Die** and **DiceCup** needed to enable this feature. 6. [Very hard] Assuming you have solved step 5, now update the definition of **DiceCup** such that it is possible to specify a list of dice that the cup should contain. If the list is e.g. {6, 6, 10}, the cup should contain two 6-sided dice and one 10-sided die.

Exercise	**OOP.1.7**
Project	RolePlayV12
Purpose	Work with a project containing collaborating classes. Reflect over class responsibilities.
Description	The project contains two classes: **Warrior** and **Sword**. The **Sword** class is initially not used for anything.
Steps	1. Take a look at the **Warrior** class. It is very similar to what we have seen in the previous **RolePlayV…** projects. Here, we focus in particular on the method **DealDamage**. What elements in the **Warrior** class definition (instance fields, properties, etc.) are needed in order to implement **DealDamage**? 2. The **DealDamage** method contains the logic for calculating the damage a specific **Warrior** object deals. We would like to move this logic to another class; the **Sword** class. Take a look at the **Sword** class, and make sure you understand how it works. 3. The next step is to connect the two classes. Do this by adding an instance field of type **Sword** to the **Warrior** class. The instance field should be initialized by an additional parameter (of type **Sword**) to the constructor (why is this a better solution than just creating a new **Sword** object directly in the **Warrior** constructor?). 4. Now that a **Warrior** object references a **Sword** object, the **DealDamage** method should be modified. Modify the method, simply by letting it call the **DealDamage** method on the **Sword** object. 5. When the code for **DealDamage** has been modified, we can remove several instance fields from the **Warrior** class. Figure out which instance fields we don't need anymore, and remove them. 6. The code in **Program.cs** contains a small test. This code should also be modified, such that each **Warrior** object is provided a **Sword** object at creation. Run the test, to confirm that the **Warrior** objects are still capable of dealing damage. 7. Which class now contains the logic for specific damage calculation? Why is this a better division of responsibilities than before? 8. If you have time left, you could update the **Warrior** class further: a) Make it possible to use two swords. b) Make it possible to change a sword after creation. c) Add a **Warrior**-specific damage factor, such that the damage generated by the sword is multiplied with this factor. This could e.g. represent that the warrior is stronger/weaker than average.

Exercise	**OOP.1.8**
Project	StockPortfolio
Purpose	Work with a project containing collaborating classes. Reflect over class responsibilities.
Description	The project contains three classes: **Stock**, **Portfolio** and **StockMarket**. Together, they form a very simple simulation of a stock market, and how it influences a portfolio of stocks. Further details about each class is provided as comments in the code.
Steps	1. Take a look at the **Stock** class. The class is completed, and you need not add any code to it. Make sure you understand the meaning of the data stored in the **Stock** class. 2. Take a first look at the **Portfolio** class. The class is not complete yet, since the property **TotalEarningsPercentage** and the method **UpdateCurrent-Prices** are not implemented yet. Still, it should be possible (e.g. by reading the comments) to understand what data the class contains, and what is intended with the methods and properties. Make sure you also understand how the **Portfolio** class and the **Stock** class are related. 3. Take a look at the **StockMarket** class. The class is completed, and you need not add any code to it. Make sure you understand how the class is intended to be used. Make sure you also understand how the **StockMarket** class and the **Portfolio** class are related. 4. Take a look at the code in **Program.cs**, where the **StockMarket** class is used. If you run the application now, it will always report the earnings to be 0.0 %, since the **Portfolio** class is incomplete. 5. Now return to the **Portfolio** class, and implement the property **Total-EarningsPercentage** and the method **UpdateCurrentPrices**, as described in the comments. A correct implementation will typically report earnings of a few percent. 6. Once the application works as intended, you can reflect a bit over the structure of the application. See if you can answer these questions: a. What class holds information about stock prices? b. What class decides how stock prices are changed? c. What class decides what stocks we have in the portfolio? d. If we wanted more than three stocks in the portfolio, where would we need to change the code? e. Could some parts of the code become simpler, if additional methods or properties were added to some of the classes? Is so, how…?

Exercise	**OOP.1.9**
Project	StaticExamples
Purpose	Defining and using static methods and instance fields.
Description	The project contains the class **ListMethods**, which defines two methods **FindSmallestNumber** and **FindAverage**. The names should hopefully indicate what the methods do. Code that tests the class is included in **Program.cs**. The class is tested in the traditional way; create an object, and call methods on the object.
Steps	1. Change the methods in the **ListMethods** class into static methods, by adding the keyword **static** to the method definitions. 2. Modify the code in **Program.cs**, such that it uses the methods in **ListMethods** as static methods. The output of running the application should of course be as before. 3. The project also contains a simple class **Car** (see the code). We would now like to track how the class is used. More specifically, we wish to track the number of a. **Car** objects that have been created b. Uses of the property **LicensePlate** c. Uses of the property **Price** 4. Add static instance fields to the **Car** class, to enable the tracking described above. Increment the value of each variable at the appropriate places in the class. 5. Add a static method that can print out the values of the static instance fields. It could be called **PrintUsageStatistics**. 6. Test that your additions work, by including some test code in **Program.cs**. Create and use some **Car** objects, and finally call the static method created in step 5) to observe the **Car** usage statistics.