Course Overview

Course Overview

Hello, everyone. My name is Gill Cleeren, and welcome to my course, Object-oriented Programming in C#. I am the CTO of Xpirit Belgium. C# is one of the most commonly-used programming languages today. It is an object-oriented and typesafe language, and it is used to build all types of applications with, including web and mobile applications. Being an object-oriented language means that we can easily create classes and objects with C# to model the real world in our code. Understanding the concepts of object-oriented programming is vital for working with C#. Anything you create with the language will rely on the principles of OOP, short for object-oriented programming. This course is the perfect starting point to gain a good understanding of object-oriented programming using C#. You'll understand the basics about classes and objects, and you'll learn about the four pillars of object-oriented programming using C#. Some of the major topics that we'll cover include understand how C# supports object orientation and learn the pillars of OOP applied on C#, model classes and create objects, apply inheritance and polymorphism for better maintainability in code we use, use interfaces to apply code contracts, and write unit tests for the functionality of the classes. By the end of this course, you will have a good understanding of the object orientation with C#. Before beginning this course, you should be familiar with the basics of C#. I hope you will join me on this journey to learn how to create object-oriented C# code with the Object-oriented Programming In C# course here, at Pluralsight.

Understanding Object-orientation in C#

Module Introduction

Welcome to this exciting journey of learning object-oriented programming in C#. I'm Gill Cleeren, and I will be your guide throughout this course. Have you ever wondered how real-world applications are built with C#? Well, the answer is simple by writing object-oriented C# code. OOP, as we call it, is a programming approach that maps real-world models into objects in our code. Whether it's managing employees or costs, we create objects that represent these entities in our C# application. Objects are the building block of most C# applications, and that is why it is crucial to learn to think in terms of objects and understand all the options C# provides for working with objects and classes. This course is the perfect starting point for you. In just a few hours, I will equip you with everything you need to know about OOP in C#. So let's get started by exploring what is in store for us in this module. Get ready to take your C# skills to the next level. Now, while I just gave you a very short introduction on object-oriented programming, there's a lot more to learn, so I think it's a good idea that we start with understanding what OOP, short for object-oriented programming, in C# is all about. Once you know what the concept is, we are going to go over to the C# features which made this possible, so we'll see the basics of classes and objects. Throughout his course, will dive into much greater detail in the topic that we'll cover in this part, but it's useful to have these top of mind right from the start. And finally, we'll look at the principles of object orientation, often referred to as the four pillars of OO. Now, a short word of warning is definitely in place here. This module will be a bit theoretical, and yes, I think that is needed here. The C# code that we'll see will be a high-level overview of relevant topics, but which are vital to understanding OOP in C#. Don't worry, we'll have plenty of time for code and demos, but that'll only start in the next module. While this is an introduction course still, I am assuming that you are at least already a bit familiar with the basic syntax of C#. I'm not expecting you to be a C# ninja expert level. Far from it. Basically, what I cover in the C# Fundamentals course here, on Pluralsight, is what I am assuming that you already know. That being said, some things that I will be explaining and using in this module and the next ones too are touched on in the fundamental courses too, but will go into much greater detail here. I think we're good to start exploring OOP in C#. Let's go. This course was created using C# 12. I'm using Visual Studio 2022, and I'm using the free edition, namely the community edition. All code and samples will run perfectly fine if you use the same version that I'm using. However, all you will learn in this course is perfectly applicable to C# 10 and C# 11 too. What we are learning isn't impacted by the version of the language. Also the edition of Visual Studio 2022 you will use won't affect your learning experience as you are using the fundamentals of the C#

language. And if you're using an older version of the language, then too most code will still work fine. Now I do recommend that if you have the ability to work with a new version of the tools and languages like we are doing here. Now .NET and C# evolve quickly nowadays, so it's best not to stay behind.

Understanding OOP in C#

So I said in the first part of this module, I will be explaining you what OOP in C# is all about. And let's start with a short history lesson and not a boring one like the one you got in school, I promise. First, OOP stands for object-oriented programming, and I'll be using both the abbreviation and the full name interchangeably. OOP is nothing new, far from it. Well, that definitely has been an important language dates back to the 19 sixties, even over time, other object oriented languages were invented. And I think that Java has probably the main next important milestone and yes, Java is definitely also an object oriented language. Java became popular in the 19 nineties already together with the internet. You'll probably remember that you had to install the Java plugin for many websites to work correctly. Java really made OOP even more popular. And then Microsoft came up with an answer launched together with .NET in the early two thousands. The shop today C# is one of the most commonly used object oriented languages out there. Understanding how OO came about won't help you with understanding what it really is. So let us stop looking at the history and focus on understanding the concept of oo in object oriented programming, things evolve around objects. Pretty much everything is an object and these objects will perform interactions with each other. One object will talk with another object and ask it to do something or get data from it. When I say that everything is considered to be an object, I am not lying. All items that we work with are to be seen as objects, including things like variables or complete data structures. OOP is done for a reason with every increasing complexity in the applications that we built OOP will help us with code being more reusable, we can write different classes that basically build what we may have already created, thereby extending the behavior. But when doing so, we can reuse what we have already created and we don't have to change or duplicate code. This may not be obvious if you don't have too much experience with coding yet, but know that this is a huge win. It's a time saver and a code will become easier to maintain. Since we can reuse code. If a bug is found in that part, we only need to fix it once instead of in multiple places. Again, a huge win. As I said, the code bases that we build become ever more complex and they have been doing so already for many years. Ob definitely helps with solving more complex problems but no, it's not something you'll only use when you are developing large applications. No, on the contrary, you'll apply object oriented practices in all applications you'll build ob helps us to map real world objects to code as well as the interactions between these objects. This way, we're breaking down the problem into smaller ponds, each with their own responsibility. And that will definitely help us to solve more complex problems in an easier way. Now that we already have an idea of what the term means. The next question is how can we write applications in an object oriented way? Well, the development language that were used to router application will need to be an object oriented language. Not every programming language is in fact an object oriented language. In fact, we can say that languages can have a different degree of object orientation support. Python can also be considered in this category. As can Ruby another commonly used language, JavaScript, the language mainly used for interactive web applications can also be considered as an object oriented language, but it's definitely in a different category. It definitely doesn't have all the oo features that the other mentioned languages have. And of course, C# should definitely also be in this list which is of course what we are going to be focusing on in this course C# is a general purpose development language which supports object oriented programming. We'll see its features which support OOP next part of this module CP is also type safe. Which means for example, that once a variable has been assigned to type, this variable can't change its type anymore. I've explained this in detail in the CRP fundamental schools and it's a big deal. Absolutely. While you may not be able to appreciate this featuring full yet, it will be of great help. When creating applications, the compiler will add compile time, verify that the type remains the same and that we don't accidentally call the method not supported on the type errors we make. While w ratting the coat will be called more easily at compile time rather than at runtime, which is a huge time saver being an object oriented language, we will be able to reuse a lot of the code for red. I will see that later in this course, SARP is also an actively maintained language and new versions are at present being released on a yearly cadence and yes, the tools typically used to create C# applications are top notch registered, you and registered. Your code are Microsoft's tools to write and compound C# applications but also third party tools exist including jetbrains rider. All these will give the developer a great experience and finally, another powerful shall I say feature of CRP is that it has a large community of developers. You will find answers to your questions easily. We have already understood that object orientation means that we will be working with objects in code C HARP has everything on board to support object oriented development and supports the four pillars of object orientation which we will discuss soon CSAR has support for classes and objects based on these classes, we can define methods for functionality and we have support for properties to work with data. C# also comes with interfaces which can be seen as contracts in our code. All of these together make C# an object-oriented language.

Working with Classes and Objects

Classes and objects are definitely the most fundamental concept of an object-oriented language. In this part, I want to go over the foundational aspects of classes and objects in C#. If you're already familiar with C#, you will probably recognize most of these. In the coming modules, everything we touch on in this part will be explored in much more depth, so don't worry if everything isn't fully clear yet. This part will focus mostly on the programming constructs. Later in this course, you'll also start understanding the why of the code you'll see here. Classes are by far the most important concept in C#. The concept of a class in C# pretty much corresponds with how we categorize objects or concepts in the real world. We all categorize things in the real world in our heads. When you were young, you may have had a dog named Bob or a cat named Molly. Being a child, you may have thought that a lion in the zoo was also a Bob or a Molly, so also a dog or a cat. While that lion was, in fact, well, slightly more dangerous. Over time, you start categorizing cats, dogs, and lions differently. You start seeing that not every dog is the same. Probably, you have at some point understood that Bob was a poodle and that other breeds exist, and you started understanding that Molly, the cat, and Bob, the dog, have similarities, but also differences. Animal becomes a class, and dogs, cats, lions, and more are all animals since they have similarities, but definitely also their own unique properties and functionalities, so to say. A dog can bark, but a cat doesn't. So over time, while growing up, you learned in the real world how to categorize things into different classes, you build up a hierarchy of things. Classes help us in C# to bring classes, let's say type of things of the real world into a model in code. Creating these classes and using them as objects, which I'll touch on in just a minute, is what we do when we write object-oriented C# code. As said, classes are the most fundamental building block of any C# application. The double type, the string type, the datetime type are all built into C#. And of course, we can create our own types of classes to capture a concept in code. All of these classes have their own behavior, but they will also have similarities. For example, we can typically for each type, ask to get a string representation. In C#, we can thus model a hierarchy of types, a powerful concept to model in code what we have in the real world. This will help us solving complex problems in code. Classes are reference types just like interfaces and delegates. Did I say classes are important? I most definitely did, and I'll probably do it a few more times. Nearly all code in our C# applications will live inside of a class. In fact, we can say that everything is a class. However, I do think we need to make a distinction at this point between two types of classes. We have, in any C# application, a main method inside of a program class, or if we're using top-level statements in the program.cs, we just have statements. If all the code is either directly in this main method or directly in the program.cs, we are probably writing a small application which doesn't use the OOP features of C# in full. While that is possible, that is not how you will write real-world C# applications. Most of the time, you will be creating a set of classes which you will then create objects from. In other words, these classes will be instantiated and objects will be created. These objects will then interact with one another. Let me explain the latter in more detail, so you are clear on the difference between classes and objects. Think back of what I mentioned about Bob, the dog. Bob is a dog, and while I don't want to insult Bob, the dog, he's a thing, an object created as an instance of a general dog, class, or dog type. Objects are instances of classes, and so is Hillary, the dog, and violet, the dog. The latter two are my dogs. They are two objects. They are both specific instances of the dog class. Classes can be seen as the blueprint for making or instantiating objects from. We can create a dog class and that will contain information about the dogs that we probably understand. The class will define functionality as said before like bark or walk or eat. It will also define that we can store data about the specific dog instance such as the weight or the age. The class won't define values for this data, it will just define that it can store this information. The functionality defined on the class will typically work with the data. For example, eating will increase the weight. In C#, it can create a class using the class keyword. It's what I like to refer to as the class template. All classes will follow this template from very simple ones to very complex ones. As you can see, we are using the class keyword to define a class here called MyClass. Then between a set of curly braces, we have the class body, and let's look in more detail at this body part of the class. Inside the class, so in the body, we can put several other C# constructs. We can use fields, so variables on the class level. We can add methods for functionality. We can add properties to work with data, and we can add events. Let's look at this in more detail. In this course, I won't be using the same, let's say, sample all the time in the slides and that is on purpose. So you understand, not to be just one simple sample what I'm explaining. We've talked about dogs and cats already. Let's now work for a bit with employees. If you have watched my C# Fundamental course, this class will look familiar. So what I want to do is now go through this simple class here and look at the contents of its body and how I'm using these different options we just saw in the previous slide. As we saw in the class template, things start with the class declaration, class employee. That part is saying what is coming next is the definition of the class employee starting and ending with a set of curly braces. Class is a keyword, and employee is the name of the class. It's a convention that we use pascal case for the name. Pascal case means that every new word in the name starts with a capital, and here we have just employee, so that starts with a capital, therefore, too. Classes can contain fields. Fields are variables defined on the class level, and they will be used to store data when instantiating employee objects. If Mary will be a 25-year-old employee, then that object Mary will have a copy of the first name field containing Mary, a string, and age containing the value 25. It's common to make fields private and this has to do with ensuring that the class can control access to its data. Using private, we ensure that only from within the class, this first name and age, can be changed. I will talk about access modifiers of which private is one in just a minute. As said, I mean it's a good practice not to expose data, but to control access. For that, C# comes with the concept of properties which basically wrap the field we just looked at. Why this is will become clearer later in this course when we create our application from scratch. For now, remember that properties are typically used to control the changes to the underlying data field. We'll look at this syntax of properties in more detail later on too as some options exist in this area. Our dog could bark. Our employees won't bark, but they will hopefully perform work for us. Being an employee, it is the typical function that all employees will have. Functionality is wrapped in a method. Here, it is the PerformWork method. This method has a void return type, meaning that it will be returning anything to its caller when invoked. After the name of the method, there has to be a set of brackets possibly containing a list of parameters this method needs, and in between the set of curly braces, the method body will be defined. Typically, we'll have code in there that works on the fields. Now, I have highlighted public and private throughout the class. These are access modifiers, and they control the access to members in our class. They play an extremely important role in OOP as they allow us to restrict access from the outside to parts of our class. As you can probably imagine, private is restrictive and will block access while public is opening up access to the given member, I have used the term members in the previous slide, and that's a term I hadn't explained yet. In C# terminology, members refers to what classes can contain. It's let's say a group name. When I talk about the members of a class, that means I'm referring to the fields, the properties, and methods of the class, but also to other things that we haven't looked at here yet such as events, constructors, or constants, which can also be members of a class. On members, we can apply access modifiers. It's easy to get confused, so for now, just remember that members point to what we can put inside of the body of the class. Classes will always contain one or more constructors. Even if we don't define one ourselves, one will be created for us. They are similar to methods, but they are a bit of a special case. They are typically used for initializing the data of the fields of the class, so when a new object, a new instance is being created. In other words, when we new up an object, so using the new keyword, a constructor will be invoked automatically. In order for a constructor to be a constructor, it has to have the same name as the class and it can't have a return type. Optionally, there can be again a list of parameters and it's, in fact, very common that there will be parameters since this way we can pass initialization data to the constructor. We'll explore the many options around constructors later in this course. Now constructors are a perfect bridge to jump from the concept of classes to individual objects. You see that the class contains the definition of the thing we want to describe in code. It is used to define a model. It is really what I described earlier as the blueprint for objects. On the Employee class, we have defined, for example, the age field, which can then be used to store the value of the value for the age value for an employee. When you and I are talking about an employee, we understand the concept, it's a general idea, but the employee class is not really an individual. No, that is where objects come into play. When indeed we instantiate a class, we'll create an object, an instance. Object 1, John here, will be an employee, and he will have a value for the age field, here 27, and a value for the FirstName field, John. Now John is a thing, an instance, an object of the general class we have defined, the Employee class. This object is a different one than the second one here which has Mary as the first name and 31 as the age. And at another instance, another object is George here with the value of 22. Classes define the blueprint, objects are the individually-created instances. To create an object from a class, we use a new keyword and that will, if you remember from just a minute ago, automatically invoke the constructor. As a deconstructor, well I should, in fact, say a constructor because we will typically have more than one defined and will differ in the list of parameters. More on that later in the course. For now, remember that this index is really the way to go from classes to an object. Now you really have an employee to work with. You will see that throughout this course, I will use both the terms object and instance. They are really the same. And indeed, once we have that employee object, we can start using it. We can evoke the PerformWork method on it, which is what we can see on the first line here. Notice index, invoking the method on the object requires that we close it with a set of brackets. If the perform method requires a parameter, we need to pass it an argument 10, in this case, between the brackets. This value 10 will then be passed to the method which and can then be used inside of the method. In the last line here, I'm using the FirstName property and I pass it a string value, though, there is a difference here, no brackets this time to invoke a property. If these differences between methods and properties aren't fully clear, don't worry, we'll come to them in much more detail later on. We have, in this brief overview, looked at classes and they are really the most important category that we'll work with. As a category of types, indeed, in C# and in .NET, we can use different types defined in the CTS, or the common type system. Across the .NET languages, a common sense, common ground is known making it possible to write code in multiple languages and have it work together seamlessly. Although we'll write classes most of the time, there are five other known categories of types. Enumerations allow us to give names to constants. Instead of using magical numbers in our code, we can use more readable names, thus improving our code readability. We'll use a few enums and the application will build in this course, enums or value type. Closely related to classes are structs. Structs can be described as lightweight classes as they can also contain data and functionality. However, one big difference is that they are value types. In the C# Fundamentals course, I explain structs in more depth, so take a look there if you want to learn more. Another category of types in the CTS is interfaces, again similar to a class, but this time, it contains just a contract that can be implemented by classes. It typically will be completely abstract, meaning that it won't contain any implementations for its methods. The latter need to be defined by types implementing the interface. This will probably sound a bit abstract, but intended, but we will have a module dedicated to working with interfaces, so I'll keep the details for later. Delegates are types that point to a method with parameter list and return type. They are typically used to pass methods as parameters to other methods. That's a bit too advanced for this course. Courses later in the C# path will explain you all there is to know about delegates. Finally, there's records, a new type available since C# 9. Records are internally treated as classes, so although I put them here in the list of supported types by the CTS. The CTS don't really know about them as they are compiled into classes. One key difference between classes and records is that they come with support for value-based equality checks. Again, records are outside of the scope of this course. Later courses in the path will go into detail on records. In this course, we will focus mostly on classes, that should be clear from what I have touched on in this module. What I have gone over quickly here covers pretty much the basic syntax of classes. Knowing the syntax is, of course, important, but what is equally important is knowing how to design your classes, what will be a class, and how will we structure these classes? Going forward in this course, we will extend our knowledge deeply on classes and objects and the way they are used to build object-oriented applications with C#. That will be really the goal of this course, and you'll be able to apply this knowledge with pretty much every application you'll build.

The Principles of Object-oriented Design

In the final part of this module, I want to focus on the oh so important principles of object-oriented design. As I just mentioned, it is one thing to know the syntax, but it is another thing to know how to structure classes in an object-oriented way, and that will heavily rely on these principles we'll look at here. We've seen how C# supports object-oriented programming through classes and more. These are the building blocks for object-oriented programming. Creating those classes in an object-oriented way is something that developers have been doing for a long time, and good practices have been distilled from this knowledge. We have these so-called four pillars of object-oriented design that will help us in achieving clean, maintainable, extendable, object-oriented code, and yes, C# has everything on board to support us in writing our code following these principles, these pillars. Throughout his entire course, we will be creating an application that follows these principles. These four pillars are abstraction, encapsulation, inheritance, and polymorphism. We'll look at each of these in detail next, and we'll start with abstraction. When you're asked to, in real life that is, to think in an abstract way, what are you going to do? When someone asks me this question, I'd start thinking about essentials. I'd start generalizing and I'd come up with indeed, essential features, essential functionalities, not details to represent the item at hand. Abstracting is really creating a layer of abstraction around the object, the item at hand, defining it so that we expose only the necessary details in order to work with it, but, as this is important, hiding the implementation details. Remember this, we expose what is essential, what is needed, and the rest we hide to the outside world. We expose only simple handles to interact with the item. Applied on C# classes, what we'll do with our classes is expose functionalities. The user of our class can interact with without requiring that the user knows how the class works. Those are the details we abstract away, we hide them. Abstraction can be a bit abstract at start to grasp and that is totally normal. Let's try to get a better understanding of abstraction by looking at an example. This time, we will use the concept of a car. I think we'll all be able to relate with that. When driving a car, you are, in fact, working with an abstraction. You use the steering wheel, the brake pedal, and a few other interfaces exposed to you as the driver. Those interfaces are the abstraction, a simple way to interact with the car. When driving a car, you don't need to fully understand how the car's engine works or how the brakes will help you to slow down. You just interact with the exposed interface. The car's manufacturer exposed for you to drive the car only when is necessary is, in fact, exposed to you, and that is a good thing. Imagine that you need to understand how a car works internally before you can drive it. It would be really hard. Well, that concept is abstraction, and we will apply that also when creating our classes in C#. We'll make them so that the users of our class will only be able to interact with certain handles that we, as the creator of the class, define. When applying abstraction, we will definitely make it easier for the user of our class to work with it. We don't expect the user of our class to fully understand how the class works. No, instead we expose a few simple handles, a simple interface through public methods and public properties. On the other hand, we will hide inside the class it's inner workings. Similar to how the car manufacturer hides how the engine works, we will hide how our class works internally. Of course, this improves the security dramatically since we don't allow users of the class to just use everything. Only what we deem necessary is allowed to be used. The fact that we hide the details of the class is a win for its maintainability. If we need to change its inner workings, we'll typically try not to change the public interface so that we don't break how the user works with the car. Today's cars get updated frequently typically, and while that may change the inner workings of the car, it is not the case that all of a sudden the brake pedal changes how it works. That's how this improves maintainability of the code. One of the main tasks when thinking objects is creating this abstraction. Typically, we'll get some requirements for the application we'll need to write. We need to start thinking an abstraction. We'll need to identify from requirements the entities which will become the classes and the characteristics which will become the class members. Imagine you are asked to create an application that allows employees to perform work. Well, then the employee will be something to work with it. It will become entity, a class in your application. Next, we can also identify a number of characteristics of what an employee might do or might contain. An employee will have a name and age, their weight, an address, their hair color, and so on. Next, they can also perform work, they can walk, they can talk, and more. Now, not all of this is relevant to our application. So out of these, we pick the ones we need in our application, so probably the name and the ability to perform work. That will become the public interface of our employee entity. Other internal functionalities may be included internally in the class, but won't be exposed. Perhaps a calculation is included in the class and performing more work than allowed based on the age that we don't want the user of the class to use. We are abstracting away all that isn't relevant to what we are creating at a design level, and that's what the abstraction is all about. In the next module, we'll get in requirements and we'll together think of the structure of the classes we'll need to create. Next to abstraction, we need to understand encapsulation, and it's quite closely related to abstraction really. Encapsulation is the concept where we will bring together all the data in a single unit together with the functionality that works on that data, that unit is the object. We are grouping, encapsulating all this together, and we can define which access is given to the data, handling all of this together is useful. Otherwise, all the different parts would be separate, and if so, changing something in one place could break things elsewhere. Data is really important. It will store the state of the object and it's typically hidden. As said, we can determine which data and which functionality is accessible and how. One way is to define access modifiers like public and private. This way, we can control that data of the object can't just be changed from everywhere, but only through a well-defined interface, which could be properties or methods. Again, let's try to make things more tangible by looking at an example. A car will encapsulate lots of data, such as the speed it's currently driving at, it will contain an engine that based on the speed knows how much gas is needed. It has a drive functionality which will work together with that engine and other parts, it defines a speed up method, and much more. All of this is bound together in one unit. It's all encapsulated into one unit. Encapsulation will then also help us to hide what we, as the user of the class, aren't supposed to directly interact with. As the driver of the car, it probably wouldn't be a good idea to inject extra gas manually into the engine, that is, therefore, hidden from us. We also shouldn't be manually changing the temperature of the car just because we can. Encapsulation gives us the ability to restrict access to that data and the functionalities to work on that data. That doesn't mean that we can't do anything with the class, far from it. Through a public interface, we can invoke methods, for example, and they might in turn work in a controlled way with the internal data, but a lot of information isn't even visible since it's internal to the object. Here's an example of how it can apply encapsulation in C# code. As said, the first step is grouping everything together, so the data and the functionality that works on that data. The result of this process is the designed class here, the Car class. That class defines public and private members as we can see, but also more importantly, it has private fields. That's the data that is fully encapsulated within the cost and can't be accessed from the outside. Maybe that data is accessible through a property or maybe it's completely not accessible and it's just internal to the class that we, as the designer of the class, can control. You can define a property or method that allows indirectly to work with the encapsulated data. This way, it's impossible for the external user of the class to accidentally or intentionally change the data they aren't supposed to change, but we will create our classes in the next modules. We will be applying abstraction and encapsulation to do so. The third pillar of object-oriented programming is inheritance. Remember our discussion about dogs and animals from earlier in this module? I've mentioned that what we do is understanding that a dog is an animal, and a poodle is a more specific type of dog, just like a golden retriever is. We implicitly are putting these in a hierarchy going from more generic to more specific. And we understand that all dogs are animals and poodles are dogs, and thus, they will inherit features from their parent or parents. This is where inheritance comes into play. We will be able to, in our code, define this hierarchy and define the functionality on a higher level, therefore, being able to reuse code on the parent or the child, instead of having to duplicate the code. The child class will typically build on what the parent has already defined and will also typically extend this with its own functionality and data. Dogs will inherit features from the more generic animal, but add extra features on top of that. So inheritance will allow us to define common functionality on a higher level, thereby allowing inheritors to reuse that functionality and extent when needed. Let's continue as we have done before and apply inheritance now in class. We can define a car which wraps the data and functionality of the concept of a car. It may define the speed, the max speed, the color, the age as data, and functionality such as drive, brake, and park. Just like with dogs, more specific types of cars exist. A minivan is a car, but it will typically have more specific features while still being a car. Through inheritance, the minivan inherits the data and functionalities of the parent car, but might extend it with a functionality, open slide door. Similarly, a sports car with an open roof might define its roof open and closed roof are benefiting from the drive, brake, and park method which are defined on the parent car class. And an electric car class might defined a charge method. Having these functionalities on the parent class will really help with code reuse. If there's a bug in the drive method defined on the car class, we only need to change that in one place and all other classes will benefit from that. While we will look at inheritance in much detail, it has its own module even. It is useful to see how this translates into code. This here will be the parent car class which defines a max speed field and a property and a drive method. Nothing is special about this class really. Inheriting classes, such as a SportsCar, inherit all this behavior from the parent class. So a sports car can also drive, but it extends this with specific sports car functionality. such as isRoofOpen, which is now available on this class, but it's not available on the parent car class. The final pillar is polymorphism. One issue we'll get with just plain inheritance is that in the hierarchy, some classes will define the same behavior, but they'll need a different implementation. The drive method will be available on all the cars in the hierarchy, but driving a sports car is a different experience than driving a minivan. This is what polymorphism is all about. The same method must be able to execute in different forms, hence the word, polymorph, so multiple forms. Within the hierarchy, we can still define the base implementation, but inheritors can decide they'll need a different implementation. In C#, this will be possible using the virtual and override keywords. We'll explore these in the polymorphism part in the inheritance specific module in much detail. One key aspect in OOP is that we don't need to worry about how this will be handled. We can still invoke the drive method no matter if a more specific version exists. That's a problem for the language to handle. We can just work with the drive method, and the most specific version will be selected. Again already as a sneak peek here's how this will be handled in code. We can define the drive method on the base class to be virtual, thereby saying that inheriting causes can provide their own implementation. Electric car class is opting to do so by adding the override keyword and then it gives its own implementation. For consumers of this class, nothing changes. We can still invoke the drive method, but different versions, different forms, I should say, of the same method now exist.

Summary

With that, we can conclude this module, which I said was more theory, but I'll make up for that starting in the next module when writing code in C#. We have seen how C# is an object-oriented language. Its support for classes and objects will help us write easy to maintain object-oriented code. Then we have looked at the OOP design principles, abstraction, encapsulation, inheritance, and polymorphism. In the next module, we are going to look at the business requirements for the application and translate those in an object-oriented solution.

Designing an Object-oriented Solution from a Business Case

Module Introduction

Hello, and a warm welcome to this module on object-oriented programming in C#. My name is Gill Cleeren, and I will be your host for this module too. In this module, we'll delve deeper into how to apply the principles of object-oriented programming using C#. As you may already know, C# is one of the most powerful object-oriented programming languages available today, and we'll be exploring its capabilities together. Now that you have a good understanding of the four pillars of object-oriented design, it is time to put theory into practice. We'll start by discussing how to gather requirements from customers and how to translate these requirements into a set of classes. We'll then define the functionality and data for each class using the principles we've already covered such as abstraction and encapsulation. Throughout this module, I'll guide you through the process of creating classes, explaining key concepts and techniques along the way, so buckle up and let's get started by taking a closer look at the outline for this module. Now the outline is pretty simple. We'll start by getting an understanding of what the customer demands are. We will try to gather the requirements and then, based on that, we will come up with a set of classes that can help us to create our application. We'll translate the real-world model into a model in code consisting out of a number of classes with data and functionality. We'll apply for this abstraction and encapsulation. We'll need to think in terms of classes and objects leaving out what we don't need, but considering what we need to include in the public interface of the class and what will need to be hidden from the users of our class. But let us, as said, first have a chat with our customer to understand what we are asked to build.

Gathering the Requirements

Gathering the correct requirements for a new application isn't always easy. Typically, you'll need to communicate with a business person who has a certain need, a certain demand for an application that will make their lives easier. Let us start by understanding the business case. For the rest of this course, we are going to assume that we are working for Bethany's Pie Shop as a software developer. Bethany's Pie Shop bakes the most delicious pies, and these pies can be bought in her stores as well as online. She started a business with one tiny store, but the pies are so good that she has several stores and a central bakery where more pies can be made. And this is Bethany from Bethany's Pie Shop. Bethany has the need for a new application. The new application will be used to manage the inventory of ingredients in the central bakery's warehouse. There have been occasions where the stock on certain ingredients was too low or even completely depleted, causing one or more types of pies not being possible to bake. That's a pity because we know for a fact that once someone has ordered one of those delicious pies, they really look forward to having it delivered on the next day. Bethany doesn't like disappointed customers and she also doesn't like the missed income. And this is you, a member of the software development team at Bethany's Pie Shop who is tasked with the creation of this new application. Typically, when a new business request comes in, it'll start with a high-level question. It is up to the development team to gather the requirements. Sometimes this is done by an analyst who will create a full business requirements. Often, a developer will need to do this. Let's assume, in this case, that you will be responsible for this application end-to-end, and thus, you will be the one who needs to understand the requirements of the application Bethany has in mind. Mind you, getting this information isn't always easy, so let us start chatting with Bethany to understand what she's after. First things first, let us think about the name of the application, shall we? Well says Bethany, since the application will be used to manage the inventory, let's call it Bethany's Pie Shop Inventory Management, that's a mouthful, but indeed, it covers what the application will do. It's always good to have clear names for the application and its components, so I think that's a good start. Okay, and who will be using this application? Bethany explains that this application will be used by the people working in the warehouse where the ingredients are stored for the bakery. There are a few people who are responsible for that. They are the inventory managers, and they will use the application on a computer within the company's warehouse. It's important to understand how the users will work in the real world with the inventory. That way, we will be able to map things to models in code more easily. So we ask Bethany to explain it a bit how these inventory managers will work with the inventory when it will be handled from within the application. And so Bethany goes on by explaining that quite logically, when an ingredient to a product is used in the baking process, its stock needs to be updated to reflect the new situation. Now, it's not always as straightforward as we think in the first place, some products are stored and used for items such as a cake decoration kit. That applies for most, but others like flour, an often used ingredient in the pie baking business, is stored per kilo, and things like eggs are purchased per box, not per egg, of course. Okay, that's good information to know already. Bethany continues by explaining that the goal of this application, as already mentioned, is really preventing the bakery to run out of stock for one or more items. Inventory managers, therefore, must be able to see which items are low on stock perhaps when looking at the list of all products. That's a good idea. I think we'll be able to do this. Now, of course, says Bettany, the application will only shine if from within the application, the inventory managers can directly place a new order for the products they want. In one order, it should therefore be possible to have more than one product being ordered. Talking about orders, Bethany's Pie Shop places orders with suppliers in several parts of the world. They only want the very best ingredients, that's why some products are purchased in dollar, but some are bought in euro and a few even in British pound. One other thing to watch out for is that we can order too much of a certain ingredient says Bethany. The warehouse can only store so much, and for each product, we can store a certain amount, but not more. So is your hat spinning already? Well, don't fear. In the beginning, we'll often get lots of information. I think we can definitely go to the drawing board and start thinking about our application. Lots of information and I'm sure more will follow. That's how it goes. Let's try to summarize what we have already understood and see if this way we can discover the entities that we'll have in our application. What, at least, I have understood is that the application will need to work with products that we'll need to manage as part of the inventory. From the conversation, I also understand that different unit types exist for products. Bethany explained that we'll have eggs coming in in a box and flour coming in per kilogram. The application will also need to be able to create an order, and to an order, we will increase the stock of one or more products. I also remember that products so ingredients can have a price that is in a different currency, so we need to take that into account too. Looking at this list, I think we can start identifying the entities we are going to work with. We are going to abstract out different entities that will model later in code as classes. I think we'll have a product entity. The unit type will be a thing we'll need to take into account. We'll also need an entity to work with order and then the price will also have a currency. Well, great job. This can be the list of at least our initial entities that we'll need to dig into further to define what they can do and what data they will need. Oh, wait, I am getting a phone call. Bethany came back to me. Seems she forgot to mention something in our discussion. The users will also need to be able to add new products, so new ingredients to the inventory. The list of products isn't really fixed, new products need to be added. To be clear, this is an entirely new product, not a product of which we are increasing the stock. Okay, thanks Bethany. See, we haven't even started developing the application, and it seems there are already some new requirements popping up. This one is, in fact, pretty simple, I think, it's just a new functionality, but what if we would have already started and the call would have been about a change to the requirements. Well, that will happen, absolutely. But building our application based on object-oriented principles will limit the impact of these sort of changes. Using OOP, we are going to be creating more maintainable software, something that you'll see will be really needed in the real world. So I'll add this requirement to the list, add a new product. As said, this doesn't really require a new entity to be added to our list, so the impact of this forgotten functionality will be small apart from the fact that it is something extra we'll need to build.

Identifying the Classes and Their Functionalities

In the second and already last part of this module, we are going to dive deeper into the classes that we'll need to create and the functionalities we'll need to include all these classes. Based on this, we can then start coding. From the requirements discussion we've had with Bethany and the high-level list of requirements we have created, I think we can start building a list of entities we'll need. Of course, since we will be managing the inventory and the inventory consists out of products, we'll have a class product. Next, we will be working with an order. As mentioned by Bethany, we will be placing orders possibly for multiple products in one go. In other words, we'll work with an order class which will contain one or more items. Each order item will basically wrap how many items of a certain product we'll want to order. Then, remember the price that was a bit of a special case, and I think we'll need a class for that one too, that the price will contain the price value and the currency used for that price. That currency will be another type we'll need to work with. And finally, I think we'll have the unit type to be a type of work within code too. Now that we think we know which types we'll have, we need to think about what functionalities and what data we'll define on these. We're going to be applying abstraction and encapsulation here now. The real design of the classes will happen in the next module. Based on the discussion we had about products with Bethany, this is a list of data I think we'll need. A product will have at least an ID, a name, and a description. For each product, there will also be a maximum in stock amount since storage, as said, we're limited. A product will also have a price and that price will have a currency. Next, a product will also have a unit type which can be per item, per kilo, and per box. Of course, the application will need to keep track of the stock, so how many items are currently in stock? That will be a value that will be calculated by the application, and if the stock is low, we also need to set that product is low on stock. That too will be something the application determines itself. It's not something the inventory managers will set. Now, what function narratives do we need to provide on a product class then? Now for starters, it's possible to use a product from the inventory. We can have 100 eggs in stock, so we could need 30 to bake a few pies. That's what using a product entails. On the last phone call. I also understand that it must be possible to add a new product to the stock, say a new type of flour that will be used for baking pies. And also a product must be able to, let's say alert when it's too low on stock. An inventory manager must also be able to see all the products with their stock and details, and to do so, we'll create a display method that shows the details of a product both in a short and a long way. And of course, we can place an order for a product, so when that product arrives, the amount in stock of the given product will also need to be changed. Now, let's talk about the Order class. When I create an order in the application, that'll have an ID and a list of order items. Such an order item will let's say be the link between the product I'm ordering and the amount of that product I'm ordering. An order will also have a fulfilled field, so a Boolean field that keeps track of the fact if the order has already been delivered to the warehouse, and a fulfillment date is also included so that we can see when the order has arrived. The list of functionalities for the order class isn't very long. An order should be able to display itself, so its details and its order lines. I've also mentioned, I think we'll need a class for the price. A product will have a price as such a price will then include the item value, as well as the currency. Since I think this can be used also on other classes, it makes sense to abstract this into a separate class. With the currency, I will also create a separate type namely an enumeration containing the different available currencies. So to be clear, price will be a class which will use a currency and the latter will be an enumeration. Talking about enumerations, I think we'll need another one. Bethany also mentioned that a product will have a unit type, per kilo, per box, and per item that here I'm going to wrap in an enumeration, and the product will have a field of this type so that we know what it is. Otherwise, we'll end up with something like one being the kilo type, two being the per box type, and so on. Using an enumeration is much better than these magical values. It won't make sense if you need to make changes to the application in a year or so. And so here, we have the first class diagram. We have the four classes I have just discussed with the methods and data, so product order, order item, and price. The relation between product and price as we can see, this will be composition. A product has a price and that is what we see here. You can also see two enumerations, currency and unit type. Order lines also can only exist in the context of an order, so you can't really create an order line without there being an order, that too is composition, and I will talk more about that later in the course. Well then, I think we have enough to get started with coding. Since we will apply OOP, making changes down the road shouldn't be too hard. Finally, in the next module, we'll start creating the classes in code. See you there.

Creating the Classes

Module Introduction

Welcome back. In the previous module, we learned how to create our first class diagram and identify the classes we need for our application. Now it is time to dive deeper and start creating these classes in this module aptly named, Creating the Classes. We'll begin by writing the code for the product class, and along the way, we'll incorporate the principles of object-oriented programming that we have already learned such as abstraction and encapsulation. I'm still your guide, Gill Cleeren, and I will be walking you through this exciting module step-by-step. So let's get started by taking a closer look at the outline for this module. As said, I will create the classes and we'll start with the core one for our application, the product class. This will be quite a large topic as it will create the different types of members, including methods and constructors and will come to understand the different options that C# offers. Next, I will introduce you to composition where we'll use one class, in our case, the price class inside of another class. To finish this module, I have two smaller parts planned. We'll look at class-level members, and we'll close the module by looking at partial classes. This entire module will show you how to write object-oriented C# code, exactly what we're after in this course, so no time to waste.

Creating the Product Class

And as said, we will start with the creation of the product class in all its glory. We will build it up step-by-step, thereby, adhering to the abstraction and encapsulation principles. Pretty much every class that we will create will follow the class template we've looked at earlier in this course. We are using the class keyword and the name of the class and what follows is the body of the class. In there, we'll have its members. Do you remember what the term members refers to? It points to the set of fields, methods, constructors, and more which are part of the class. The class we are going to build up together is the product class, so in order to create it, we'll specify product as the name of the class. What you see here is simply the class definition of our product class. Only one class with a given name can exist within the given namespace. So the class name needs to be unique within the namespace. The class does have an access modifier and it's set to public here. To be clear, the access modifier is optional when creating a class. Access modifiers are used to determine from where the class can be used from within your application code. Indeed, it gives us a way to block using the class, which means that we, as the developer, can choose to allow or deny usage of the class. There are a few options available for access modifiers on a class. I specified that our class is public. This basically means that the class is available to use from everywhere. Internal is another access modifier. When the access modifier is set to internal, it basically means that the class can also be used from anywhere, but limited to within the current assembly. Now, what do I mean with an assembly? Well, when we compile our application project, it will result in an executable file or a DLL being generated, but when we compile an application, it'll result in an executable file over DLL being generated. That becomes the assembly, and so internal then means that from anywhere within that assembly, our class can be used. Other code which references the assembly can't, however, access the class, hence the name internal. If we omit the access modifier, internal is also used. Now to be clear, it is more restrictive than public. For regular classes, we are basically limited to public or protected. Classes can also be nested inside of another class. Although I won't use it in this course, know that the class can be inside of another class, hence the name nested class. Such a class can have a protected or private access modifier, basically, further limiting the access to the class. We'll see the use of private and protected in the context of members on the class soon. As said, it is perfectly fine to leave out the access modifier, and then the C# compiler will treat it as internal, which is the default. I do think it is good practice to include the access modifier, even if you intend to make the class internal. This shows that you have actually spent time thinking about it, and it's not that you have well, just forgotten about it. Nowadays, when you add a new class to your application, registered to remain the class internal by default. I tend to make my classes public, but it depends on the situation which is best suited for your class. Now that we have the class definition ready, we can start with the implementation of the body of the class. I will start with the data of the class. This is the list of data items. I think we'll need at least that is based on the requirements we know now. We have the ID which will be integer value, the name of type string, the description also of type string, the maximum number of this particular product we can have in stock, the price at its currency, the unit type, so for example, per kilogram, the calculated amount currently in stock and a Boolean value indicating if the stock for this product is too low. This data, I'm now going to define on my product class inside of the body in the form of fields. Fields are defined on the class level, and doing so is pretty simple. We define the type and the identifier, so int and then ID, and we now later create an object based on this class. This will then contain the data for the object, so for example, 100 for a certain product. For another product, a different variable will be created and it will contain another value, say say 205. In other words, each object or instance that will get created will get its own copy and assign its own value, that is why fields are also often referred to as instance variables. While we can leave out the access modifier, it's again good practice to include it here, I have specified this to be private, which would be in this case also the default should I have left it out. Now why did I make this field private? Well, think back of what we learned when discussing encapsulation. The class should have its data wrapped, encapsulated, and not everyone should just be able to change the data of a class. We, therefore, make it private, therefore hiding the information within the class. If data is needed, we'll create a layer around it which can then control the access to the data field. Private does still allow this field to be changed from within the class, of course, so we add a property or a method inside this very class that will still be able to change the value of this field. While most fields will be private, other access modifiers can be used on a field. Public basically does what I want to avoid typically on a field, and that is just making the value accessible from anywhere. So while that works, most of the time, that won't be a good idea to use only a field. Protected we'll look at when we introduce inheritance in our application. Internal can also be used on a field which will make this field public within the assembly. Again, probably you won't be doing that on a field since you're opening it up for other code within the project. I may be on repeat here, but it's important that you grasp how we are applying encapsulation on our class already. Our class will define data and that is part of the class. The class can work with it, but access to the data from outside should be restricted. However, some of the data may need to be available from outside of the class. And we do this, we can include code in the method or property that can validate the data before setting it, say that the name of the product should be a string that is never longer than 50 characters. If we make this name a public field, we have no way to validate this. From everywhere in code, a longer string could be set. If you make the string private and write a method of property to change the value, we can include logic that checks the new value and perhaps chose an exception if the passed-in string is too long. Not all data is even typically exposed to the outside, so fields are just internal to the class and should even never be accessible, even through a property or method from the outside. This encapsulation is wrapping of data within the class we can achieve through access modifiers which will restrict access.

Demo: Creating the Class

Let us, in our first demo, create our class and add the data fields we have specified. For the purpose of the demo that we'll use throughout the course, we're going to be working with the console application. Hence, we don't need to worry about any UI yet. We can just do everything in the console using things like console writeline and console readline to accept input. So I'm here in Visual Studio, and we are going to create a console application. We are asked to give it a name and that will be here BethanysPieShop.InventoryManagement. It's a good practice to start the name of the application with the name of the company, so that'll be the first part, and the inventory management part describes the functionality of the application. In this next screen, we'll select .NET 8, but as already mentioned, what we learn in this course is applicable to nearly all versions of C# and .NET since these are very core features that have been in the language since the beginning pretty much. I'll click on Create here. A new program.cs has been created as we can see and that uses top-level statements which are explained in the C# Fundamentals course. We won't use this program.cs yet, that'll come later. We are for now going to create a new class by right-clicking on the project note here in the Solution Explorer, and the name will be Product. As we can see, a new class is created which is marked as internal by default. That is something that has changed, well, pretty recently in Visual Studio. Previously, the template always generated a public class. However, I think it makes sense for our class to be public, so we'll change it to public here. We can see the class keyword and then the name of the class product. In between this set of curly braces, we'll put the class body. It's time to actually start doing that, and we'll start with some of the data fields we already know. It's very well possible that others might need to be added as we go along when the requirements are changing, for example. From Bethany, we got the information on the data that we'll need. Of course, the business requirements often don't reflect all the fields that we will need. In most cases, we'll need to bring in fields as developers of the class too. In any case, for now, we'll already need an ID, a name, and a description. ID is of type integer, and name and description are both strings. Notice, I have defined name to default to an empty string. Description, however, I have marked as nullable, which we can see by the question mark of the string. This basically says that it's possible that when a product is later on created that the value of this description field might be left empty, blank, null. What is definitely important is that the fields have received the private access modifier. The class keeps those for itself, not letting anyone access them from the outside directly. This is what encapsulation is about. Data should, in most cases, be kept private. For a product, we will also need to bring in a field that can be used to store the maximum amount of items we can store for that very product. Next, since some products are per item, some per kilo, and so on, we are for now going to store that as a type of unit, Now we could say something like one is per item, two is per kilo, and so on. While that would definitely work, knowing afterwards what the one or the two means is hard when you need to come back to the application in, let's say a year. Also, if a value needs to be changed, we need to do so in many places, hence bringing down the maintainability. What I'm therefore going to do is create an enumeration called unit type. So let's right-click on the project again and select Add, New Item, and well, let's select class here again. The name will be unit type. A new class is generated, but I need these to be an enumeration, so we'll change these to be an enum that basically contains the named values for these different types. So each of these enum values responds with an integer default value so that we don't need to keep track of that ourselves. We can now use these named values, instead of magical numbers. Let's go back to the product, and let's add some more fields here. I've now brought in a field of type, unit type, so that's the enumeration, and I've added two more fields which will be managed also by our product class, the amount in stock and the isBelowStockThreshold. Amount in stock will keep track of the amount of items we have in stock for the product, and the isBelowStockThreshold will be managed too by the product class to see if the product's stock is below a given threshold. The latter are definitely fields which will be managed by the class. Notice, we haven't done anything with the price yet. I could add a double for the price value and another enumeration for the currencies. However, we will come back to the price later on, so I'm going to leave that out for now.

Adding Functionalities

We now have added the data as fields. In the previous module, we have also discussed the different functionalities we need at this point. We'll add more later on in the course. As a small recap, here's the list of things we need to include now. It must be possible to use the product which will basically lower the amount of the product we have in stock. It's a method that we'll see that will work on the private data we have just added. It must also be possible to add an entirely new product and then manage the stock for that product too. If a product is too low on stock, that must be visible from within the application too. You also need to be able to see the product in the application together with information about the product, but I think we'll need to add a short and a long version of that ShowDetails method. And of course, since we don't just make the number of items in stock of a certain product, a value can just update. It must also be possible to increase the stock. Functionalities, as you probably will know, will translate into a method we can create on the class. Here's an example method, it's one version of the IncreaseStock method. This method accepts no parameters, and I've implemented it so that it'll just increase the amountInStock with one, so I used the double plus which is the increment operator. The method is public which means it can be invoked from everywhere, and that is okay if you want this method to be part of the public interface, that is. What I mean with that is that it's okay to make this public if we, as the developer of the product class, decide this can be seen as an operation that the outside world should have access to to interact with our class. I think that makes sense here. It is definitely part of the functionality to increase the stock from the outside. In the method itself, we are working with a private field and that too is okay. We have decided that this method is an entry point really to work with the private state of the class. Although this is definitely nothing new, I want to quickly show you the generic method template. A method will have a name, and behind that, a set of brackets. An access modifier can be used here as we saw and that can amongst others be public, private, protected, and internal, which are the most commonly-used ones. We will work with these on methods throughout the course. A method must also define a return type, which basically says I will be returning after completing a value of this particular type. If no value is returned, the return value will be void, and so even if no value is returned, the method needs to indicate that. In between the brackets after the method name, we can have a parameter list. I say can since it's definitely possible that the method doesn't take any parameters. Each parameter also needs to have its time defined. Options exist here for optional parameters, default values, and more, but I won't go into the details of this. Then in between the set of curly braces, we see the method body. All code parts in this body will need to return a value that corresponds with a defined return type.

Demo: Adding Methods

Alright, time for another demo. Let's start adding methods. And let's now look at the methods that we need to support the required functionalities. In order to save us some time, I've gone ahead and I added the methods already. There are definitely more interesting things than watching me type those, I suppose. You can find the code for these methods in the snippets that come with the course downloads. By the way, you will find a snippet file for all modules in these downloads too, so if you want to follow along and don't want to type, take a look there too. Let us now look at these methods in detail. And to start with this, use product method. I've marked this method as public on the Product class. The reason is that this is really part of the public interface of the product. It has to be possible for the user of the Product class to call this useProduct method. This is a good example of hiding internal logic, but offering a simple interface for the user of the class. The method also specifies it requires one single parameter, the items parameter of type int which points to how many items you will want to use. As said, there is a logic in this method that will work with the private data fields of the Product class. The product itself will keep track of the amount of items there are in stock of the given product inside the amountInStock field that'll be used here wrapped inside this public method. We're going to check if the requested amount is less than what we have in stock. If that is the case, we can take that from the amountInStock. Next, I will call another method, UpdateLowStock. Let's first look at that method. We see it over here. This method will check if the amount in stock is less than 10. That is a hardcoded value for now, and if that's the case, we're going to set another private field isBelowStock threshold to true. Notice this method is private, meaning it's to be used only by the class itself. I've created this method since I may use this logic from multiple places, and this way, we avoid that we have to duplicate the logic. Okay. Back to the usedProduct method which has now worked already with two private fields. We then also called another method Log passing in a message. The Log method is another private method which for now writes that to the console. I decided to create a method to the Log method to work with logging output. The reason is that for now, we are logging to the console, but maybe in the next version of the application, we may want to log to a file. This way, I only need to update this in one place. Again, this is a private method. This method shouldn't be called from the outside. Since then, we could perhaps mess with the log output, not what we want. Back in the useProduct method, we can see that if we don't have enough items, we just call the Log method again passing in a message using string interpolation. Through string interpolation, I can write my string in one go, and between the curly braces, I can write expressions which are then appended to the string. I use yet another method, the CreateSimpleProductRepresentation method, that is yet another private method, a utility method, let's say. With this, I mean that I can wrap some logic for easy reuse in the method, this time to create a simple representation of the product by concatenating the ID and the name. Next, we have the IncreaseStock method which is public. It should be possible to update the stock for the user of the Product class. We don't have any parameter here for now, so the default implementation will increase the current amount of stock with one using the increment operator. Next, you have the decrease method which does accept two parameters, the items. So the amount and the message being the reason for the decrease. This is a private method, well, for now, since I don't know yet if this will be called internally only or it should be possible for users of the class to call this method. Essentially, it's going to reduce the stock with the number requested or if we don't have enough, the stock is set to 0. In any case, the UpdateLowStock method and the Log method are invoked again. It was also required that it should be possible for a product to display itself. I've included for that two public methods, DisplayDetailsShort and DisplayDetailsFull. The short version returns a simple string through string interpolation, and the full version uses a string builder and it will include a message STOCK LOW, so using the private is below stock threshold, so there we go. We have already now defined the basic functionality of the Product class.

Introducing Properties

Our methods we now have in our class can work with the private data of the class, but shouldn't there also be a way to expose data on the class so that the user of the class can ask for the value of a field or maybe set the value of a field? Well, first keep in mind what we said earlier, the creator of the class decides which data should be exposed. And secondly, the data itself should typically be wrapped inside of the class. And that being said, we will probably decide that there should be public access to some data. So we could write methods, often referred to as getter and setter methods, or getters and setters, which we return a private field or allow us to specify a value for a field. See how we are controlling access to the private field? We could, in fact, for example, in the SetName, so the setter here add extra logic that checks that the name isn't longer than a certain value. Data is owned by the class, it decides who gets to change it and what are the rules to change the data. Now, when the class becomes bigger, it might contain quite some data fields. Having to write multiple methods might also become cumbersome and will, in my opinion, diminish the readability of the code. So while getters and setters work, we will rarely use them. Instead, we'll typically use properties. Let's learn what these are. Here is a first example of a property. Just like in methods, a property is a member of a class and is used to read or write a private field. It's basically wrapping a private field with a code that we otherwise write in a getter and setter method. In other words, through a property, we can expose the value and hide the implementation logic. Typically, they are public and they work, of course, in combination with a private field. Properties have themselves a get and set accessor. The get is used to return the value. They believe the value of the private field. The set is used to well set the value of the private field. Just like I explained with the setter method, inside this set, we can write logic that can work on the value, such as validation logic. It will check again that the length of the string isn't longer than 50 characters. Let us look at the syntax of our property in more detail. I see here that this is a full property. What I mean is that this is the syntax for a fully-written property as a shorthand also exist, which I will show next. A property has a name, here that would be name. Notice that there are no brackets like with methods, but just like a method, the property will have an access modifier and a return type. In the body of the property, we have get and set which are both C# keywords. The get accessory is used to return the value here of the private name field. The latter is often referred to as the back-end field, the field that sits behind the property and is invisible from the outside of the class. Next you have the set and that is used to assign a new value, either directly like we see here or possibly via custom logic. In the set, we also see another keyword, value, that is used to define the value being assigned by the set accessor. While the full syntax is often used, there is another option called auto implemented properties or automatic properties. Take another look at the full property on the left. The get and set accessor are doing nothing more than just getting and setting the value of the back-end field. Now, one could argue that this way of working so including a back-end field might be useless and that it could be simplified. Auto implemented properties do exactly that. That's what we see here on the right.

Things are more simplified. We have just a get and a set with a semicolon and no implementation. Behind the scenes, the back-end field is generated by the compiler, so we don't have to do it, but you also don't have access to it. If there's no validation, all the logic inside the accessors, you'll see that most developers default to using automatic properties. With properties, we can also add variations. This first one, for example, is a property that'll only return the value of a back-end field. See there is no set in there. It's a read-only property. The value is, in other words, not acceptable from outside of the class since it back-end field name is private. This is a good practice to return calculated values from within the class and expose them, but add security, well, encapsulate them really so that they can't be changed from the outside. Here's another variation that does the same, but with automatic properties. Here, I've defined private set. This means that, in fact, within the class itself because of the private access modifier, the value can be changed using the property. From outside of the class, the property isn't settable, it can only return the value.

Demo: Adding Properties

We have seen a lot of new things. Let's apply this in the next demo where we'll add properties, and in one go, we'll also refactor our code base to use inside of the class, also the property, instead of the field. At this point, we have private fields which I've created a few demos ago, but being private, the user of this class can't access this data. Now for some data, that's okay. If the data is useful only to the class for its internals to work, then that data shouldn't be accessible. Think back of what I said with our car. We shouldn't be able to inject extra gas manually, but on our data, we should be able to get and set, and to do so, I'm going to bring in some properties which will wrap the private fields. I'm going to add a property, a full property that is using get and set to get access to the id of the product. This is a very simple property that'll just allow us to get the value of the id or set it. In some cases, changing this value might not be allowed. So yes, there are definitely cases where we won't allow the ID to be set from the outside. We'll see one of those read-only properties soon. Next, I have added two properties to the protocols for the name and description. These are full properties too, and they really show what I've said in the slides. We prevent direct access to the private data, so the fields of the Product class since sometimes there might be a logic that needs to run when we work directly with the value. Here's a good example of that. We can see that in the name property they get is, well, just returning the value, but the setter contains extra logic that will check if the value is longer than 50 characters, and if so, it'll be truncated, so only the first 50 characters are used. This rule, well, that's a business rule, could be in place because the underlying database only accepts 50 characters, so we need to apply this logic in code. By the way, what we see here is the range operator, which basically means that we'll take all characters until we reach 50. This works on collections and the string here is seen as a collection of characters. I'm doing something similar with the description here, but there we will allow 250 characters. It's important that these full properties require that we create a back-end field manually. For the name and the description, this way of working is thus needed. However, it's not always the case that there will be rules that require checks before we write to the back-end field. If not, we can, in fact, rely on automatic properties, which will still follow the same rule, namely that there will be a public property and a private data field. However, we don't need to create the back-end field manually. The C# compiler will do that for us. We don't have direct access to this field, though, as we would have when we create it manually. So now I have added three automatic properties for the UnitType, the AmountInStock, and the Boolean calculated value, IsBelowStockThreshold. UnitType allows for the value to be retrieved and set, both exist. However, the AmountInStock uses private set, that's an important construct. The value is available for use inside the class, but not from the outside. The private set will block setting this value for the user of the Product class, and that's what I want here since it's a value that's internal to the Product class. We do expose it for read to a regular get. The same applies for the IsBelowStockThreshold. That too is a calculated value inside of the Product class. If we scroll up, we will still see our private fields, but these aren't linked with these automatic properties, so in fact, to avoid mistakes, let us remove these. We will leave the private maxItemsInStock and notice I don't even have a property for that. I think that is data that we don't require to expose, so we'll leave that as a private field, but whoops, lots of build errors all of a sudden. What happened? Well, we deleted the fields. In fact, I recommend, in most cases, to also inside the Product class to use the property, instead of the field. The reason you may ask, well, then the logic, such as we have defined in the name property, will also be applied when working with this inside the Product class. I've now updated our class's logic so that it uses the properties, instead of the fields, and things are working fine again. We have now nicely encapsulated our data, and we have added public properties that can access that data.

Creating Objects with Constructors

We have now spent a lot of time creating the class. It's now time to start creating instances, objects based on the class definition. Creating a new object of the Product class can be done using the new keyword as we can see here, but what are we, in fact, doing here? Are we using something that we haven't discussed yet? The constructor. Each time we create a new object or a new instance, a constructor will be invoked. A constructor is a member of the type as well. We can say it's a special type of method which is, as said, automatically invoked when a new instance is being created, so using the new keyword. Here, we can see a constructor I have created for the Product class. As you can see, just like a method, a constructor has a name and a set of brackets. However, the name here needs to be the same as the class name for it to become a constructor. A constructor can have an access modifier, but can't specify a return type like a regular method. That's quite logical. You can't invoke a constructor manually, it only gets called when using new. In a constructor, we'll typically add code to initialize the newly-created object as we can see here. Here, I'm setting maxAmountInStock to 100 and name to an empty string, and we did specify more fields on the Product class. In fact, how we instantiate an object, all fields are set to their default values. For example, numeric fields are set to 0 and Boolean fields will be set to false, and so far, we didn't have a constructor. Still, we would have been able to write product, product equals new product. So really without a constructor being there, yes, that is because if we don't specify a constructor ourselves, a default constructor will be used. I say created in the slide here, and yes, it is created, but not in a way that you see the code being written for you. A default constructor is created behind the scenes and will allow us to new up an object for which we haven't defined a constructor. One important thing, a default constructor will be available as long as we don't specify a constructor ourselves. As soon as we do so, the default won't be available anymore. This is that default constructor or at least what you should consider it to be without this code, as said, being generated within the class. It's a very simple to structure this default one. It doesn't contain anything. It is there so we can new up an object. As said, we get this for free as long as we don't define any other constructor. If we write this constructor ourselves, which we can, again, the default isn't generated. Constructors can, and most of the time will, have parameters. As said, a constructor is used typically to initialize a new instance. How we can use it to specify default values, it is definitely useful that the caller can specify values which are used as initialization values for the newly-created object, that's what I'm doing here. The constructor receives two parameters, id of type int and name of type string. I then use these values to specify the name for the property's id and name, thus using these for initialization.

Demo: Adding Constructors

Time for another demo where we will add constructors to the Product class. Alright, our class is getting more and more functionality. Let us now add the constructor. There is one constructor already here. Can you see it? No. Well, that's normal since it is the default constructor which is generated, but it's not visible. A default constructor allows us to create an object using the new operator. Now, we can, of course, add our own constructor too. We can start with the code that the default constructor well, looks like, contains by default, let's say, so this code here. Look at this constructor. We can see it's a constructor and not the method since it does not define a return type, and the name is the same as the class name, so product. So while it's a constructor, it is not doing an awful lot. Typically, constructor is used to initialize an object. If we use this construction, all fields would be set to their default. So integers are set to 0, Booleans are set to false, and so on. While that's okay, it might not be what we want. By creating a new product, so a new object, which we will do soon, will often want to pass in values used to initialize the fields of the new product that we are creating, in this case. It is now a real first constructor which accepts two parameters int id and string name. Inside the body of the constructor, I'm then using the properties to assign the values. The Id property is sent to the passed-in Id, and the name is sent to the passed-in name. Since we're using the properties here, we will pass by the validation logic that we have added in our name property. This really explains why we are typically using the properties throughout the class, instead of the fields directly. Now, while this is still a simple constructor, at least we have one ready now. Let's head back to the slide for more information on constructors.

Introducing Method Overloading

We have now created a few methods and a constructor. However, sometimes we'll find ourselves in a situation and we'll have two or more methods, these do pretty much the same thing, but perhaps they accept a different set of parameters. We could find different names for these different methods, but in fact, as long as the combination of parameters is unique, C# will see it as a different method and the name can be the same. This is called method overloading. We can thus create multiple methods with the same name, but a different parameter list. The latter needs to differ in number, order, or data types for the parameters. Let's look at an example. Say that I have the IncreasedStock method. I can define a version without parameters which will, as we have already seen, increase the value of the AmountInStock with one. Let's say we also want to add a method where we allow the user of the class to define with how many items you want to increase the stock. We shouldn't find a new name like IncreaseStock with amount. While that would work, in large classes, you'll need to be pretty creative to come up with new names. So instead, we'll use overloading, we use the same name, again, IncreaseStock, but now this second version has a parameter, int amount. And C# will see this as two different methods since the parameter list isn't the same. In the C# Fundamentals course, I explained the different options we can use here in the parameter list. Constructors too can be overloaded, which is good since we can't even come up with different names here. This means that we can have multiple constructors, and based on the list of arguments, we use new-ing up an instance, the correct one will be used. Here, we can see I have two constructors defined, the first one with one single parameter, and the second one with two parameters. While we could write the initialization code in each of these constructors, it will probably result in code duplication which we want to avoid. Therefore, we can use a small trick here. We can let one constructor invoke the other. That's what I'm doing with the this here. The this keyword will result in invoking the other constructor, the one with two parameters since I passed to this two arguments.

Demo: Overloading Methods

Alright. Time for another demo. This time, we'll look at method overloading and constructor overloading. Because we ended the previous demo with adding our first constructor, let's first look at overloading the constructor. As explained, it is possible to add multiple constructors as long as they are unique as its constructors can't have a different name. The only way they can differ from one another is through the use of a different parameter list. Let me bring in one first. I've now brought in this one first, so this one is even simpler. It's not accepting a value for name. Now these two are pretty similar. They're both setting the ID and the other one sets the value for the name as well, but in fact, this part is repeated. Code application, no matter how simple can and should be avoided in most cases. Therefore, I am going to refactor my constructors so that one calls the other. Notice that this simple one now has this added. This basically says call on this type a constructor with two parameters. The first one is this id, and for the name a string, we pass in an empty string, so string.Empty. The main benefit of having this constructor call the other one is that we avoid code duplication. Let's add another constructor. This is a longer version accepting more parameters. I could, in fact, add this to this constructor, but it's not really needed. We set the values of the past in arguments again using the properties, thus initializing the product. The maxItemsInStock is using the field. Remember, there is no property for that one. Also, I have some logic that will also check again if we should set that IsBelowStockThreshold value. Yes, we could also use the UpdateLowOnStock again, avoiding code application, so let us make that change. Notice that we don't have anything here yet on price either. We'll come back to the price later in this module. So at this point, we have a number of constructors. The id is that where we will create our objects, we now have a number of options to create a new product. Based on a number of arguments while passed in, a constructor will be selected. Only these options are, of course, allowed. Next, we can also overload methods. At this point, we have this simple version of IncreaseStock which accepts no parameters. I have now added an overload that does accept a parameter, and it contains some logic too. Notice that this method has the same name, same return type, but a different parameter list. In this case, this one has one parameter and the other method had no parameter, so indeed, they differ, and that's okay for overloading to work. We are checking what the total stock for this product will be with the added items, and I'm assigning this value to a local newStock variable. If that is lower or equal to the max items in stock for this product, then we're good to go and we increase the value of AmountInStock with the passed-in amount. Remember that was the property with a private set, and although it's private to set this value from within the class, we can set it. That's all good. If the value is higher, though, we set the AmountInStock to the maxItemsInStock, and we log a message that we have too many for the given product. Finally, we also need to do the opposite with the IsBelowStockThreshold. It is now higher than 10 again, we set the value to false. We could extend the update LowInStock method so that this logic moves there too. Maybe that's a good exercise for you to make already. I will add another overload, this time for the DisplayDetailsFull. Again, this method differs from the other one since it accepts a parameter, thus these two methods can coexist. Know that method overloading is a very common thing to do. In this case, the logic is very similar. We could, in fact, write the following so that the one overload calls the other. The parameter will then receive an empty string and the result will be the same. We're, again, avoiding code duplication, and that's what we should strive for at all times.

The this Keyword

When we create the class, in fact, one single copy of the class is created. From that class, one, or it could be even more than one object, are initialized. Each of these objects will get a copy of the fields defined on the class. These are in instance variables and each object can thus give a distinct value to its own variables. The class methods to save memory space aren't copied to each object since they aren't the same anyway. The same goes for properties, they are shared under class level. However, when we evoke a method, it needs to know which is the object we want to work with. This we can, again, do using the this reference, which basically is a reference to the current instance. This is really a good name here. It is saying this is the object you need to work with. We often used the this in a constructor, Say that you have the constructor you see here on the screen, again, with an ID and name parameters, but assume that in this code of the constructor, we want to initialize these fields. We'll often use this this in the constructor. Say that when I have a constructor here, as you can see on screen, again with an id and name parameter and assume that in this code of the constructor, we want to initialize the fields this time, not the properties, but wait a minute, they have the exact same name. If you say id equals id so both lowercase, nothing will happen. Since this id will be used and C# will assume that I want to assign the value of the id to this id, that won't get us very far. Luckily, we can use the this reference here. Through the syntax you see here, we are basically saying to C# use the fields of this current instance, this object as at least to the passed-in value. We are basically pointing to the reference of the object this we are working with.

Demo: Adding this

Let's see this in another demo. This is going to be a pretty short demo. Here is the updated constructor. We can use this here to indicate that we're pointing to the current instance, this object. For this case, it's not really needed as you can see since Visual Studio is saying we can omit the this, but let's make another change. Now, the ID parameter has the same name as the property. If we omit this this here, we're basically saying that this Id is equal to itself and the value of the property wouldn't be set. Therefore, this is really required here now saying that we'd want this Id, so the property of this object to be set to the passed-in value. Now, this can be used in many locations. For example, this will work here, but again, it is not needed since that is the default here.

Using Primary Constructors

Since C# 12, a new syntax has been introduced to create a constructor called a primary constructor. Using primary constructors, classes can now have a parameter list. These parameters are then available in the body of the class, mostly used for property initialization or inside a method body. Take a look here at the code snippet where I have declare definition, and next to it, I now have declared two parameters, ID and name. Note that these are lowercase that's let's say the convention since they are really parameters which would otherwise also specify as parameters on a regular constructor. I can then use their values to set the value of a property, which is what I'm doing here for the Id and Name property. This is called capturing the value and is needed to have access to their values after the constructor has finished executing. Essentially, this is the same as writing ourselves a constructor and assigning the values to properties, but yes, shorter. As said, the primary constructor is really just one constructor. It takes away some of the ceremony of creating a regular constructor for a class combined with private fields which are typically assigned from the constructor and thus can be used to make your classes well more concise. I have seen that classes often need multiple constructors, so overloaded constructors. When using primary constructors, any other constructor will need to use the this operator pointing to the primary constructor. This is required. Also, once you declare a primary constructor, there is no default constructor being generated anymore. As said, this syntax was only introduced with C# 12, so you won't come across it very often yet. Now, we won't be using it in our demos here, but remember, it's just a new shorthand for doing the same thing.

Using Composition

Our class is coming along nicely. One thing we haven't looked at yet is how we can add the price onto the product. For that, we'll use composition. Let's start with understanding this type of relation first. I just mentioned the word relation, and that is key here. So far, our class members have been properties and fields of primary types, such as name of type string and ID of type integer, but think back of how we talked about the price before. Bethany mentioned that the price is related to the product along with the currency. In fact, we could choose the easy way out and add two more fields and the corresponding properties for this price value and this currency type directly onto the Product class. But in my opinion, the price is an object itself which in turn defines two properties itself and may then also contain functionality to display the price formatted based on the currency, for example. So what I propose is that we create a price class, and yes, our Product class will have a price. That's the relation I just mentioned. The price belongs to the product. There will be a property of type price defined on a Product class. This relation is composition. Composition creates an association, a relation between two classes, in fact, I should say between two objects of two classes. In one class, we'll create a property or a field or both of another class. In the case of composition, we'll typically say that one object has an instance of another object. In our case, the let's say, containing class will be our product and the contained class will be the price class, and we use composition, the nested type, in our case, the price. We'll have no logical existence without a product, and that does make sense. Moreover, we can later on reuse the functionality of the Price class for another type in our application as well, thereby, sharing code.

Demo: Adding the Price Class

In the next demo, I will show you how composition works and we'll bring it into the Price class. One of the properties of the product that we haven't added is the price. I mentioned earlier that we'd come back to this one as I want to work with it in a special way. As said, we could on the product define a double variable for the price and an enum property for the currency. While that would definitely work, I'm going to create another class for the price since that's an entity that we may be able to reuse from other classes in the domain of our application. Before we continue, let me add some grouping here in the project. At this point, I have just a very limited set of classes, but that is going to continue to grow, so it's a good idea to group things. I'm going to create a few folders, and then I'll take you through the folder structure. Okay, so what have I changed? I've added the Domain folder. The word domain is referring to the fact that this is related to the business problem, the business domain that we're building our classes for. In there, I've added another folder, ProductManagement, and in there, we can find our Product class and the unit type and enumeration. Notice that I've also updated the namespaces to reflect the folder structure. While this isn't really needed, it's a good and commonly-used practice. Next, we can now create the Price class. Now, as I said, the price, I think we can possibly use also from other classes, so I want to place this in the ProductManagement folder. Instead, I'll create a General folder. In there, I'll now add the Price class which I'll make a public class again. The price will contain two properties as I see it now. The first one is in double the ItemPrice. Next, we also need to keep track of the currency that this price value is in. For this, I will create another enumeration, the currency enumeration. Bethany has specified that most products have a price in dollar, but also products in euro and pounds are bought. So now I have coded these into an enumeration. Next, we will extend the Price class with another property, the Currency property. Now we need to create the link, let's say, between the product and the price. A product has a price, therefore, we will use composition. Another way of thinking about composition is the following. Can the price be standalone? I think not. A price will be created as part of, in this case, a product, hence by using composition. So let's head back to the Product class, and in there, I'm going to now add a property of type Price, it's again an automatic property, and indeed, it is part of the product. The product now has a price. I'm also going to use this from the constructor. It's now the updated constructor which now includes a parameter of type price and it is used to set the local property. We're also going to use this price property from the DisplayDetailsFull methods, so from both overloads. You're going to see and have added the price in the string that is going to be returned from this method. There's one issue with this. The default behavior will use the toString method of the Price class, and it won't show the value of the price nor the currency. We'll learn more where this toString comes from, but for now, we are going to create our own representation of the price as a string. Let's go back to the Price class, and in this class, I'm going to create my own version of the toString method. This method we get by default from .NET. It requires that we use the override keyword here. Now don't worry about that just yet. We'll see that when we work with inheritance. What this essentially will do is it will override the default behavior and will give .NET a new way of representing our price as a string. In this case, we return the concatenated value of the item price and then the currency. With this, we now have added composition.

Splitting into Partial Classes

The last thing that I'll add for now, at least, to my Product class is making the class partial. As the name suggests, making class partial is nothing more than splitting up the code of the class in more than one physical file. By default, a class is contained in one file. In most cases, you follow the rule one class per file, although sometimes you'll have more than one class in a single file. This is now the opposite. We will be splitting up the class into multiple parts, so multiple files. We can split up classes, interfaces, and structs by using the partial keyword. So we'll write public partial class product. This will allow us to make multiple files with the same class declaration, and the only thing that'll happen is that the different parts are combined together at compilation time. This, of course, means that the rules for the class stay the same. You can't have multiple members with the same names in these different partial files. It is still one class and is now simply spread over multiple files. Now you may be asking yourself, what is the use of this? Well, for one, when you have generated code, so generated by a tool perhaps from a designer in Visual Studio, you can have one partial class, perhaps from a design inside Visual Studio. You may want to put this generated code in a separate file. Using a partial class, you can have one partial class for the generated part or you can have another partial class where you write your own code. The two files will be combined into a single class at compile time, but things will indeed be more readable. Also, if you happen to have a very large class or if you're working with multiple developers on one large class, it may still be useful to split it up into multiple smaller, partial classes. Making class partial is really not that hard. You can see that I've created two fast, product.cs and product2.cs, both define a class product, but the class is defined as partial. If you emit the partial modifier, the compiler would flag an error.

Demo: Partial Classes

Alright, time to add partial classes. I'll make the Product class partial, and we'll add another file that contains more code for the Product class. So far, our Product class is one class, and that is to be clear, the default. In most cases, a single class will be in a single file, but like I mentioned in the slides, there are definitely cases where it may be useful to split things up into more than one file. This is just a split at the file level. The class stays one class. At compile time, all the code will be merged into a single class. Therefore, even if you split up the class into multiple files, the same rules still apply, such as the unique names and so on. So that's when I split up our class. I'm going to bring in a second file called ProductPart. By default, Visual Studio will create a new class called ProductPart, and I'm going to remove the part here and also make it public. When I do this, Visual Studio will start complaining saying that the Product class name isn't unique within this namespace, which is of course, required. If we now add the partial modifier, and we then go to the regular product.cs and we made that partial too, things will look fine again, and we have effectively split this class in two files, nothing else. If inside this ProductPart, we add say a property called Price, we'll get an error because that, again, isn't unique within these class, so I have to remove that again. Say that we agree that another developer works on some of the private methods and that we work on the rest of the code. We could decide that we move these private methods here into the other file. So let's get these from Product.cs and paste them in ProductPart.cs. When we now compile things, everything will work as before, since indeed, these two files are merged into one single class.

Demo: Adding the Order Class

In the final part, we'll now look at how we can work with the Order class. So we're off again to Visual Studio where we'll look at this. In the final demo of this module, we will bring in a few more classes related to orders. We will allow from the application to create an order and that will contain one or more order lines or order items, if you will. For each product, I'm going to order, as part of the order, I'm going to create an order item that points to the product we're ordering, as well as the amount of this product we are ordering. These classes I'm not going to place in the ProductManagement folder. Instead, let us start with the creation of a new folder, OrderManagement. In there, I'm going to create the Order class next, and next, I'm going to create another class, the OrderItem that's based in a snippet that contains the properties for OrderItem. An OrderItem will have an ID that's going to be an internal value let's say. Next, as said, an OrderItem can be seen as a link between the product we're ordering and the amount as part of the order. So I've included two properties for the product ID and the product name. Both the ID and the name are required. They're not nullable. I also must be able to store how many of the given product we're ordering, and that's what we have the AmountOrdered property for. Just like we did with the Price class, we're also going to create a new implementation for the toString method which will return a readable version of an order item. Next, we're going to work on the Order class. Let's start by making the class public first. I have now added a few properties. Let's go over them. An order will again have an ID, a fulfillment date, which can't be set from the outside, hence private set. Also there's a property of type List in OrderItems. That's a generic list, meaning it's a list that can contain only order items. As mentioned, this order will contain an order item per product we're ordering, and those are stored in the OrderItems list. If you are unfamiliar with lists, just think of them as an intelligent array, and in this case, is only going to store order items. Finally, an order also defines a Boolean property to indicate whether the order was already fulfilled. Next, I've added the constructor. I've added some code for demo purposes. First, the Id will be set from this constructor as a random value lower than this high number here. Then I fake the fulfillment date so that it's going to happen somewhere between now and a maximum of 100 seconds. We'll see why I'm doing that here. This is indeed for demo purposes so that our orders will be randomly fulfilled when we run the application later on. Finally, I also initialized the order items to be a new list. This we'll look at in much more detail in the next module. With that, I think that the Order and the OrderItem class are ready for use. We'll come back to them later in the course.

Summary

There we go. We have reached the end of this very interesting module where we have used classes. Classes are the main building block in C# applications. Pretty much all code goes into a class. We've looked at the different types of members that we can create in a class, including fields, methods, properties, and constructors. We've also applied encapsulation. We've seen it as the grid of the class. We can decide which data is accessible and how. I've also looked at the concept of composition where we added the Price class, and we finally looked at making the class partial. We now have classes, but I haven't seen any objects yet. In the next module, we'll start creating objects, and while we're at it, we'll write tests for our code too.

Using and Testing Classes

Module Introduction

Welcome back to this course on object-oriented programming in C#. In our last module, we created the product and all the classes, and now it's time to put them to use. These shiny new classes are waiting to be instantiated, and we'll be doing just that in this module, so we'll explore the creation and use of objects using object-oriented programming principles. We'll start by creating instances of our newly-created classes, and you'll gain experience in using them effectively. We'll also dive into writing unit tests for our classes where we will again be instantiating our classes to ensure they are functioning as expected. And by doing so, you'll learn how to write effective unit tests and how to debug your code when issues arise. Now, before we get started, let's take a quick look at the outline for this module and what you can expect to learn. As said, we need to start creating objects, and that'll be the first thing we do in this module. We'll see how we can create instances and how these will all have their own data, but there's also this thing called static data and static members, so basically members defined at the class level. These have their own use, and we'll learn about them next. In the next part, I'll show you the running application, the interface which will be console-based. In there, we will be working quite a lot with objects of the classes we have created already, and as said, we'll look in this module to add writing unit tests for the classes we have created. If you're not familiar with unit tests yet, don't fear as it's not that hard to learn, and it will end up giving us more certainty about the quality of the code we have created.

Working with Objects

But finally, we will start creating objects in this first part. So far, we have created a few classes, but we haven't created any instances of these classes just yet. In fact, a class is just the abstract description of objects of a given type, how the object will be once it's instantiated. The class is a blueprint, and the plan that is created before anything well real is created. It's the recipe for the pie that Bethany will want to bake, but the recipe for a pie doesn't help me a lot when I'm hungry, only a real pie will do so. The actual pie created on the recipe will be the object. We have created a class with fields and methods, but not any objects that give their own value to these fields and are, therefore, different from one another. When objects are instantiated, memory will be allocated on the heap rather than the stack, and we basically get a reference to the memory space. While that's transparent to the C# developer, that is what is really happening. That object will then have its own values for the fields completely isolated from other objects that will have other values for the fields. So this is again, what is happening. We create a blueprint where you, for example, have a field called color on the class, and we then instantiate the class, different objects are created, and each of the objects will give their own value to the caller field. They are making each of these objects unique. As said, objects are created on the heap, while primitive types are added on the stack. If we create a variable of type integer, that variable is created on the stack, and that value is placed in that very position too. When we create an object, things will happen differently. The object itself taking up all the space it needs for its different fields will be placed on the heap, the larger one of the two. We, of course, need access to the object on the heap, and for that, a reference is created with the name here o that we specify. That is basically how I'll point you to the memory location, that's why this is sometimes also referred to as the pointer. Since a reference is created, we also say that classes are reference types. We already, in the previous module, discussed the constructor. It's the member of the class that'll get called where we instantiate an object. You can see the default line of code to create a new product. Product product is new Product. Let's analyze this a bit more. We will work with the new keyword. So as you can see here and we are saying to the new keyword which class we want to create an object from, or in other words, which class I want to instantiate. Here, I'm newing up a product since that's the type we are creating. In other words, saying new Product will create a new object of type product. Notice the brackets here at the end. Between the brackets for now, I don't have any values defined, but as you'll see soon, we'll often do that. So the part to the right of the assignment operator has now created the object. Let's now focus on the part to the left of it. Well, that will create a variable called product, so the lowercase, and the type of that variable will be product on newly-created class. And what does that variable contain then? With primitive types like integers, the variable contains the value itself, classes or reference types as said. So the variable contains a reference to the object, a pointer, if you will. Once we have created an object, we can invoke members on it, which I will show you in just a minute. Now, before we do that, let's look at how we can work with constructor which requires parameters, and you can see that I'm using the new keyword to generate a new object of the prototype again. But now to the constructor, I'm passing two arguments, the value for the ID field and for the name of the product. Of course, a constructor with this parameter list needs to be defined on the Product class for this to work. We've seen in the previous module that the constructor, just like a method, can be overloaded. One implementation needs to map to the combination we are invoking here unambiguously. Once we have the object, we can put it to work. We work with the object through the reference that's returned to us when we invoke the constructor. Here in the first line, I'm invoking the UseProduct method. Seeing that I'm not passing any value so any arguments between the brackets, we are expecting this method to exist on the Product class with this very signature. Using this method from anywhere in code also requires that the access to this method is possible. If the method would have been set to private, it wouldn't be able to invoke it on the instance. We can also set the value of a property, which is what I'm doing next. Name is of type string, so I can pass a string value after the equals sign, so the assignment operator. If a method is returning a value, we can capture that, which is what I'm doing here. Here, we capture this value returned by the UpdateStock method in the number variable. This assumes that the method will return an integer value. If the method were to return a string, for example, we would get a compile time error, that is strong typing at work in C#. It will typically create an object with the full syntax that I've just shown you. It is now, however, possible in C# to use a new shorthand, which is what you can see here. Can you spot the difference? Indeed, on the right-hand side, the type name is omitted. This basically says that if you specify the type on the left-hand side, you can omit it on the right-hand side. This does assume that you aren't using var, of course, so implicit typing.

Demo: Using Object Initialization

Alright. Time to start instantiating some objects. Let's return to Visual Studio for a first demo in this module. We have so far just graded classes, and yes, as I now have already said a few times, with just classes, we won't get there. They are the plan. Now it's time to create objects, and that is what we're going to do starting with this demo. So I'm here in the program.cs and I have, in fact, already prepared a welcome screen. As you can see here, this is some ASCII art that shows a logo and some pies, and after that, we are asked to log in. Logging in is not part of this application, but it gives you an idea on how we can do this. And after that, the screen is cleared, what comes next will be added in the coming demos, but you may be thinking those aren't objects. This is well, UI. No. Well, correct. You got me, and let's create a product and is quickly take a look at the product constructors. Since we have a constructor defined or more than one, it means that there is no default constructor which gets created automatically for us, so we can only use the ones that we have defined here. As we can see here, this one requires a few parameters in order to be used. Most are primitive types, but there's one exception though, which is the price here. This means that in order to construct a product, we'll also need to pass in a price object which is then set to this Price property on the product, but we've seen this when we discussed composition. Alright, so for now, I'm going to create a few products inside the program. Well, this is just going to be temporary code. We'll change it soon when we start using objects from the UI of the application. I said I'm going to need a price object before I can create a product object. So I'll do that first. If we look at the Price class in there. We see that we don't have a constructor in there. Oh, do we? Well, yes, we have, in fact, again the default constructor, but not one that we can pass parameters to really initialize the object. So I've added that one here. This one accepts that two parameters and sets the properties as we've seen before. Okay, so heading back to the program.cs, and now I'm going to create a price first. Here, we are creating a new price object, and I pass in two arguments, 10 being the item price and an enum value. The part on the right is the object that we're creating, I'm going to create a variable of type price that will point to the object and that is this sample price here. The object is created on the heap. The variable is created on the stack. One thing, I created the constructor with parameters, but I could have used the default one too and then passed in values through the properties. I'll come back to that in the next demo. Now, we are ready to create the product and let me paste in another snippet here. Here we now created a product P1 again using the new keyword and passing in the required arguments. On that P1 product, we can now invoke methods. For example, we can call the IncreaseStock method. When we invoke a method, we need to add a set of brackets. If the method requires no parameters, then the set of brackets will be left empty. What is happening here is that we're now invoking this method on this instance, so on this object. If we need to pass an argument here, then we do that also here in between the brackets like I'm doing here. If you want to use a property, we use this syntax here. Notice no brackets and an equal sign. This will now set the value of the description to this text here for this particular object. This value is unique for this object. We can also construct an object using implicit typing, so using var. The result is the exact same thing. Although we have used var here, there is still an object of the product type written on the stack, and we still have a reference that points to it. It is still strongly typed. If we hover over it like I'm doing here, we can see that. There is also another let's say more shorthand syntax we can use now where we leave out the type name on the right. Nothing different is happening here to be clear. The only difference is that this product here isn't needed anymore where we define it on the left-hand side, this of course, won't work in combination with var. So there we go, we have already created a few objects.

Demo: Using Object Initialization

Take a look at this class. I've defined here a constructor, which is basically the same as the default constructor. It's not really useful indeed. It's not setting any values, so the fields will be initialized to their default values. There are also some automatic properties defined. But yes, since the constructor doesn't have any parameters, we can't really use the constructor here to initialize the object we're about to create. Wouldn't it be better that we create a constructor that takes parameters for all fields? Well, it's an option, but if the class gets large and will have a lot of fields, this might be a tedious task. Maybe tomorrow a new field is added and more parameters need to be passed, which will also then require that all the code that uses this constructor is updated. Well, there is another way without defining a full constructor and that is to object initialization. That is what you see here. Take a look at the syntax being used here. We're not specifying any arguments between the brackets, that simply wouldn't work since there is no constructor that exists for that, but notice the highlighted part here. In between a set of curly braces, I'm now specifying in a key value-based syntax the properties with the values I want to initialize them with, comma separated. We can specify them in any order and also choose which ones to include. Since the only thing that, in fact, doing is specifying a value for properties of the object. Let us look at object initializers in a demo. Remember that I had changed the Price class in the previous number to include a constructor, well I've removed that one again, and I'm now going to rely on something that I often do, and it is object initializers to create my object and pass it values. If we look at this, we can indeed see that we're now using the parameterless, default constructor, but then I use another set of this time curly braces. In there, I use a key value syntax to in one go pass values for the properties. I use ItemPrice to the property, then an equal sign, and then the value 10, and a comma and the next one. The order in which I pass them isn't important here since we're using this key value-based approach, so now I don't need a specific constructor. With a syntax that is pretty similar, I can also create my object. You can see that I'm now doing pretty much the same, but for a product. The sample price we just created is passed here as the value for the Price property. One thing though, if we look back at the constructors of the Product class, we can see that only this one here accepts a value for the max amount in stock. That value is currently a private field, so it can't be set from the outside. In a real application, well, this might be a trigger to change this so that this is also accessible through a property. I'll leave things as-is for us, so it clearly shows the difference between different approaches. I've updated the other products too. Notice that I'm now using an object initializer for the price inline in the declaration of the product. I find that a pretty clean syntax and it's one I often use for my C# code.

Adding Static Members

In the next part, let's explore static members. The classes that we have defined contain methods, properties, and fields containing the data wrapped inside the objects. We have seen that each object gets a copy of these fields and can specify the value it wants, which is then isolated from other objects. But in some cases, it makes sense that all objects that are instantiated of a given class share data. This data is installed on the class rather than on the individual objects. That's where static members come into play. So really what static data is is data stored on the class, and all objects created based on that class will have access to this data. This data is effectively shared between all objects and they can change it. If one object changes it, it's changed for all other objects using the data. Here, we have the Product class again. Say there is a new requirement when we say that there's a threshold value that is used to determine when a product is low on stock. That could be a value that is the same for all products, and it doesn't make sense to store this on each object, rather this could and should, in fact, be shared between all objects, hence, this can be static data, so data stored on a class level. To make a field static, we use the static modifier. This is now available at the class level and, in fact, exists without there being an object created even. Indeed, this value is created at the class level, not the object level, and the class is created at the start of the application. We can, inside the class, also define a static method that can work with static data. A static method, that is a method that is also defined on the class, can work with static data and it can be invoked on the class rather than on an object. Indeed, remember what I just said. The static data exists without there being an object created. Therefore, we can define a static method in order to work with static data without instantiating an object. Note that an object can work with static data, of course, as well. Since static data and static methods are defined on the class, we don't access them through an object reference, instead, we go through the class. Take a look at the syntax I'm using here. I'm not creating an object product. No, I'm invoking the static members to both the field and the method through the Class product. When I now change this value, this value will change for all product objects since they share this value.

Demo: Adding Static Members

Working with static members can be a bit confusing. Therefore, it's definitely good to return to Visual Studio and see this in action, and we used the Description property and gave it value in a previous demo. We're setting the value of the actual object. Another object can have a different value for this property. Each object has its own copy of these fields that we're wrapping with properties, and thus the values are independent of one another. However, we can also declare data and methods to be on the class level. In this case, so with class data, the data is shared amongst all objects of the given class. This data, so static data, can be changed through the class, and when we do so it is changed for all objects of this class. Let us for a change, go to the ProductPart class, so the second partial layer, and in there, I'm now going to specify the value for the threshold of when to consider the stock to be low. I think so far we had hardcoded this value to 10. This is a good example of data that is shared since it is the same for all products. However, it can change, and when it's changed, all objects should also know about this new data. Here is the static field, and I have now set it to 5. We can change this value from a method too. An object can work with static data, but we can also create what is known as a static method which can only be accessed through the class, instead of through an object. Here is a static method. This method can only work with static data. See, I type here a description that won't work since that's a property or a field that is linked with an individual object, not with the class. Only a single copy of this data exists shared for all objects of the given type. If we now go back to the program.cs, we can see how we can work with the static field and static method. Here, I'm using a static method and I can access it through the Product class, not from a product instance. Now since a static field is, in fact, set as public, we can also access it through the product type. With this new value added, we can now go ahead and update our class code so that all objects will use it. In other words, when I was using the hardcoded value of 10, I'm now going to update my code so it uses in regular object code, this static field. In that direction, this will work, not the opposite way. So we need to make a change over here in the UpdateLowStock method, and also in the IncreaseStock method have this value hardcoded, so let me update that here too. There we go. Now we're using the static field.

Demo: Exploring the Interface of the Application

So now we have a better understanding of working with objects. In this vault, we're going to work with them extensively as we'll really make our application interactive, and we'll do this in a demo. I'm going to take you through the interface code where I am creating and using objects extensively. So we are going to start using the application using the UI. The UI is a simple console application like the one that was created in the C# Fundamentals course. Let's start by looking at the running application, and then I will take you through the parts of the code which are relevant for what we are learning here, and not everything is useful. Showing some statements and capturing input is always the same, so we won't spend any time there. So here is the running application with some ASCII art showing as the welcome screen. We are asked to press Enter to continue with the application. Once I hit Enter, I will get this main menu which shows a couple of options, and then we can enter our selection. Not everything is implemented yet. We'll build the missing parts throughout the rest of the course, that being 0 will also close the application. So first, I'm going to go into the inventory management, and the first thing that we see is a list of the current inventory. We see the three products that we have in our inventory, and there is no stock for any of these. From here, we now get a sort of sub menu that allows us to work with this inventory. We can type 1 to view the details of a certain product. Now, I need to enter the product ID I want to view more details of. Let's select 1 for sugar. We now see the details. If you look closely, you can see that at the top, we're using a short details output, and now we're using the full details one which includes that the product is low on stock, which is correct. So I'll type 1 again and I'll enter, I want to use 10, that will invoke the use product methods saying that we want to use 10. Of course, we don't have 10, so we'll get output saying that this isn't currently possible. Adding and cloning will be added later. We can, however, already see another overview of products low on stock, as you can see here. Let's go back to the main menu. In here, we can now go into the order management. I get a choice to see a list of all open orders, and currently there are none, but let's now place an order by typing 2. Okay. So now we're creating an order, and in one order, that can be like we saw one or more order lines, so one or more products can be ordered in one go. I'm going to say that I want to order 100 of sugar and 10 cake decorations. There we go. Now we type 0 to close the creation of the order, and so the order is now created. Now, since this is a demo application, what I've done is that an order will be fulfilled somewhere within the next 100 seconds. Maybe you remember that code with a random value that we already came across, well, that is used here. When I now go to the open order view screen, we may see that the order is still pending. Okay, so none pending, that means that in the background, the order has arrived. Let's see. If we navigate back, we indeed see that there are no items in stock. Let's now try using one of these products now, and there we go. Now, we have used a product that was on stock. If you look at the low on stock overview, again, we can see that now only strawberries are currently low on stock. Alright, let's now look at the code for all this goodness. I have added this Utilities class, and you can find it with the snippets that come with the downloads of the course. This will contain the UI of our application and some other methods that we'll look at. Now I've added this file from the snippets, and it seems that we have two errors as we can see here. Let's look at these errors first. I seem to be using a method that we don't have yet, the ShowOrderDetails method, so on the Order class. We don't have a method yet that gives me the details of an order, so I'll need to add that first. This is a public method on the Order class which uses a string builder to create a string representation of an order with its details. The letter I'm looping over in this for each, which is a loop statement which will loop over the OrderItems in this case until there are none left. For each of these OrderItems, I show the product and the amount ordered. Let's look at the other error we need to fix. The UpdatedLowStock isn't accessible due to its protection level. If we look at the method again, we indeed decided earlier on to make this private. It seems that in the implementation I have now created, I need this to be accessible. So if we, as the designer of the class, agree with opening this, we'll make it public. There we go. If we now compile again, things built, that's good. And this was actually the application I was running earlier on. What I showed you as the running app is what we are at at this point. So let's now look at some of the code. As mentioned, I have added a file called utilities. It contains all the code that shows the menu, and in fact, also works with a lot of objects as I will show you. We can see that this class contains two private fields which I've made static to inventory and orders. Inventory is the list of products, and it's going to be the list in which I load products and will basically be the list of products my application works with all the time. This is an in-memory list, and I've made it static, meaning it's created at the class level and can be used from static methods too. Also, one other advantage is that because it's static, the list is created at the startup of the application and also just a single one of these is created. That's exactly what I want. There will be just one single inventory, I then have this method, InitializedStock, also a static method, meaning I don't need to create a utilities object to invoke this method. In fact, in the program.cs, I've written a line of code that invokes this method. So at the very start of the application, this method is invoked. In there, for now, I'm creating a few products as you can see, that's what we saw already earlier while I was doing that in the program.cs. Then I add these products using the Add method into the static list, so the inventory. Now I have some products in my inventory. Next, I have this ShowMainMenu method which essentially is showing the main menu as the name cleverly gives away. Already. I have a switch statement in here that based on the input will jump to another method. In a real life application, we need to check the input, of course, but to keep things simple, I've left that out here. I want to focus on the object-oriented side of things. In the program,cs, I'm invoking this method too as you can see here.

When I enter 0 here, we will close essentially the application. Let us now look at the ShowInventoryManagement manual method next. This method is, again, just showing some output and accepting some user input. This time, I wrap this in a do while loop as you can see here. So this will show as long as the user hasn't entered 0 to go back to the main menu. I'm also calling the ShowAllProductsOverview method here. That method is pretty simple. It will use the list of products, so the inventory, and for each product, we invoke the DisplayDetailsShort method. From this screen, we can now go to the details by invoking the ShowDetailsAndUseProduct method. In this method, we ask the user to enter the ID of the product they want to see more details of. If that input isn't null, we are going to let's say search for the product. It is a bit special, and in fact, it goes a bit beyond the scope of this course. I'm using something called LINQ, which stands for Language Integrated Query. Using LINQ, we can easily work with a list of data in C# and perform queries on that like I'm doing here. I'm searching in the inventory list for the product where the ID is equal to the entered ID. If found, we ask the user what they want to do, which essentially just offers one selection, namely to use the product. We then capture this input and invoke again a method on the product object, the UseProduct. The UseProduct method requires a parameter, so that will get in the data entered by the user. Now we saw in the running demo that we couldn't use the number of items first since the initial stock is 0, that logic is handled entirely in the product class, which is good design. The functionality is entirely wrapped inside this class, and it uses private data which is accessible through this method. Again, that's what we're after. I'll let you take a look at the ShowProductsLowOnStock method yourself. It is pretty similar, after all. The ShowOrderManagementMenu is, again, very similar, and it offers two options, showing open orders or creating a new order. Showing open orders will first perform a little trick. Remember, I said that orders will be fulfilled automatically after a random amount of seconds? When we create an order, I set the fulfillment date in the near future, so I add some seconds over here in the constructor of the order. In the ShowFulfilledOrders method, I'm going to loop over all orders, if any. After which order, I'm going to check if this time has passed, and if so, we're going to add the products of the order to our inventory. I'm going to search for the product in our inventory, and then I'm invoking on that product the IncreaseStock method passing in the number contained in the order item. Then I set the order to fulfilled, which is the status value, so I know that the order has been added to the inventory already. Again, the IncreaseStock method is doing the heavy lifting here. This logic is contained inside of the Product class. It will use the maxItemsInStock value for the given product, a private field that is set using the constructor to see that we don't overstock, and it may also change the value of the IsBelowStock threshold. Adding a new order, I will leave up to you to explore. One thing I want to point out here is that I'm using the object initializer syntax here since the OrderItem class doesn't have a constructor I can use apart from the default constructor. Finally, via the show settings, we can change the stock threshold value, that was a static value on the Product class. That means that we don't access this through an object, but instead through the class. Here, I'm getting the value, and then we can change it. Since this makes a difference for the value of the IsBelowStockThreshold property for all products, we loop over all products in the inventory, and I invoke the UpdateLowStock manually. That is the one I now changed to become a public method. So there we go, we now have a real interface to work with our objects already.

Adding Support for Loading Data

I think it's time to sit together with Bethany again. I heard she has a question. Our application at this point works with hardcoded data. The initial inventory is loaded from code, but that is only for testing purposes. Bethany asks us if it's possible to be able to load in a file that contains the current inventory. Is that possible? Well, yes, I think so, Bethany, I think we can create that. Let us recap this new requirement and see how we're going to implement this. First, a file with a given format that we will know is given to us containing the current inventory. We'll create a new class which will read the file line per line. Assume that each line will contain a product with all information such as the name and the price. We will see the file soon. That class will be what's known as a repository. It'll be just a class, but it has the functionality to work with data, in our case, from the file. We're not going to add this functionality on the Product class. Instead, we will create a new class, the product repository, and we'll have that read in the file and create new product instances based on the data it reads from the file. What we are replying here is the single responsibility pattern. Each class will do one thing and one thing only. Our Product class is concerned with representing the product entity, but not with reading products from a file. That's the function of another class, and this will be that new class, the product repository.

Demo: Loading Data from a File

Time for another demo where we will look at this product repository. Making sure that classes do one thing is, in fact, a good thing, and that is referred to as the single responsibility principle. It's a principle that basically says that we shouldn't create classes that try to do everything, but instead, they focus on one thing. A focus, it's important to us, and it's important for our classes too. There you go. So we are asked now to load products for my file. If we think about this requirement for a second, is this something that the product should be doing? Well, this functionality is really read from a file and basically create or convert the contents of the file into products that we can work with in the application. If you ask me, that is not something the product is doing. Instead something is going to be creating products. In many applications, reading from a file is a functionality for which the code is placed in one class that is going to be handling this. So in order not to put this code throughout the entire code base, that will become the repository, a class that sits between the data and the code that uses the data. When you start working with databases, you will see the concept of repositories being used very often as the class that sits between the database and the application. So what we will do here is create a new class and that'll be the product repository since it will be used to load products from a file. I'll paste in the code and take you through what I'm doing here. Again, there's a snippet for that. Alright, let's look at the code. I have, for the sake of simplicity, to find the fixed file name and directory. When you run this on your machine, you might need to change this. The directory and the file are being created, but of course, if you don't have a D drive, things won't work. Alright, there's one public method here, so let's look at that one first. As we can see, it'll return a list of products. That's what we were using already in our utilities, so that looks good already. Here's a sample file that we will work with, and you can find those again with the downloads. Each line contains a string representation of a product with its data. All data fields are separated with a semicolon. In the code, I am first going to check if the file exists, and if not, I create an empty file. For that, I'm going to create the directory if it doesn't exist, and then I'll create the file. I do the latter using file.create passing in the path and that'll return a file stream object, which we, in fact, can close, and that I do by adding the using keyword. Let us now assume that we have found the file I just showed you where there is some data already in there. I'm going to have to read out the contents line-by-line, and that I do using File.ReadAllLines, again, passing in the path. We'll now parse each line, so that'll be the loop we enter here. That'll return an array of strings. Each element of the array will be a line of text. Next, I'm going to split the line on the delimiter and that was the semicolon. The first part is the product ID, so I try to parse this into an int using TryParse. The out int syntax here creates the product ID variable and sets the value if the parsing went okay, that'll return the Boolean success, and if that is false, I set the ID to 0. We then split all other data, parsing it into the correct type. For example, when we pass the currency, I'm doing another TryParse and I parse into a currency enum. Finally, we create a product using the constructor, so this syntax we already know, and there we add it to the local products list which I had created over here. All of this can go wrong. We're working with files, so I have a few catch blocks defined which will cause our application to fail gracefully if the parsing goes wrong somewhere. Finally, we return the list of projects to the caller. This is exactly what the repository is responsible for, reading the data and returning it back to let's say the rest of the application. I have now updated the initialized stock method as you can see, so it uses the product repository. I'm creating a new instance of the product repository. I haven't created this as a static method, so I need to have an instance of that product repository, and one could argue that it would be better to make this static too. We don't really make any use of the instance, so it would definitely be an option, but I haven't done so, so we need to create an object of this type first. I've used the shorthand syntax as you can see here. Then, we set the local list of products, so the inventory, to the result of invoking the LoadProductsFromFile method. Let us run the application now again. We now get some extra output that products have been loaded from a file. That's good. If we now go to the Inventory overview, we see that now four products have been loaded, which was the data from the file that I have now successfully loaded in.

Writing Tests for the Class

In the final part of this module, we are going to look at writing tests for our classes, more specifically, unit tests. You may be asking, why do we jump from creating objects to testing classes? Well, great question. When writing tests, we're going to also need objects that will invoke methods on, so it makes sense to talk about this now. Now, before we look at using unit tests, let's first introduce them, so if you've never heard about unit tests, sit back and relax, it'll all become clear. As the name implies, a unit test is a type of test we can write, so in code to test other code, it's indeed code to test our regular code. We will be using unit tests, test small parts of the classes we create. In theory, this will typically be a method that we test since that we can invoke. We can't ask C# to invoke part of a method. In general, using unit tests, we can test small isolated parts of our code. Now, what do I mean with isolated? Well, at this point in your C# journey, the code in all the methods is still very simple, but when you write larger, real-world applications, those methods will grow and they will call other methods, which will in turn call other methods and so on. When you're using a unit test, you really want to test just the method, not all the other parts it invokes. That's what isolated means, and there are techniques to do so, including writing mocks. Now, that would take us too far. We are going to focus on small unit tests for now. What we will then end up with is code as said that we can use to test other code. What we'll do is invoke the method on an object, and the test will validate the result of that invocation. I will show you unit tests in just a minute. Now let me first explain why you should be writing more code just for the sake of testing other code. Well for one, to find bugs. We're not perfect, and an error is easy to make. When we are writing test code that will run the regular code and validate if the result is what we'd expect, we know we have done a good job, and over time, you'll have to come back to your application code and you'll need to make changes because the requirement has changed. If you change code, you're at risk of breaking existing functionality, introducing bugs in code that was working fine before. If you've written unit tests for that code, you can run those tests again and validate that the original functionality hasn't been broken. In the end, you'll end up with better quality code, that's a guarantee, less bugs and more maintainability, that's what we want. And like I said, we'll use our code, our objects like they will be used. In other words, this code is unit testing code is documentation for the regular code and can help you or another developer to understand better how the objects are supposedly used. Unit tests are a set code which will test our regular code. They will follow a certain pattern, often referred to as the AAA pattern or the AAA pattern. These three A's stand for arrange, act, and assert. Here is a small test. First, I'm going to set up things for the test. This will be the Arrange part. Here, I am now creating a new product. Then, we are in the Act part going to invoke the code we need to test. Here, we're going to test the IncreaseStock method, so that's the one we're going to invoke. Finally, we can check if the code inside the IncreaseStock method worked as expected. In the Assert part, we are going to verify that using Assert.Equal. I'm going to check that 100 is now effectively the value of the amount in stock, which it should be if the IncreaseStock method worked fine.

Demo: Adding Unit Tests

Alright, time for a demo where we'll look at writing a few unit tests for the methods in the Product class. We are going to write a few unit tests, so code to test the functionality of our order code or regular code so to say. This unit test code will ensure that our code does what it's supposed to do and that it keeps on doing so when we make changes. By running this test code, we can verify this behavior. Alright, where do we place that code? Well, good question. We won't place it in the regular project. We'll typically place it in another project. Visual Studio comes with support for different types of unit tests. In fact, I should say that there are a few frameworks, so helpers don't write unit tests, and Visual Studio has a few templates for these different frameworks. Today, most commonly in the xUnit unit testing framework is used so that's the one I'll use here too. So this time, I'll right-click on the solution, and I'll select Add, New project. In the template window, let us now search for xUnit, and we'll select this xUnit Test Project template. We'll name the project BethanysPieShopInventoryManagement.Tests, and of course, we'll select .NET 8. Again, the version won't make a lot of difference here. First things first, we'll create a reference from the test project to our normal project. Okay. Let's now look at the test project. A new class was created, UnitTest1. We are going to create a few unit tests for our Product class. And when writing tests, it's important to have a good naming strategy in place. You'll often end up with many tests. Since we are testing, we're going to test the Product class, we'll create a class called ProductTests. So I'll go ahead and I'll rename the file to product tests, and Visual Studio will also ask us if we want to rename the class. As you can see, there is a method in there already with something above it, that something is an attribute, and it is used by xUnit, so the testing framework, to understand that the method that comes next is, in fact, a unit testing method. Fact is used to indicate this. Alright, what are we going to test? Asset or Product class? In fact, we can test methods, but we can also test constructors or properties. When we want to test the method, we're going to invoke the method and verify that the outcome of the method is what we expected it to be. A test constructor will often test that the created object has all the correct initializations done, but we will look at methods now. So I'm again here in the Product class, and I want to write a test first for the UseProduct method. If we look at this method again, what does it do exactly? There are, in fact, a few parts that we can follow in this method based on the result of checking if the requested amount is less than the amount in stock or not. If true, then we can deduct the items from the amount in stock and update the low stock value if necessary. If not, then we don't change the amount in stock since there we go through this else here. So in the first test, I'm going to test this happy part, so that's the one where we have enough items in stock. I'm going to test that if we have a product with enough items in stock and invoke to use product on that, that we actually see the lower amount in stock. Alright, let's write the first test. A test is just a public method, and I'll rename this one here. Because tests also serve as a way of documenting our code, it's vital that we give clear names to our tests. In this case, I want to test that when I invoke the UseProduct method that it reduces the amount in stock. I will, therefore, name my test method UseProduct reduces amount in stock. Now inside the test, we are going to use objects again and invoke a method on that object. Remember what I said about the tests. They will typically follow the AAA pattern, arrange, act, assert. In the first part, arrange, we're going to set things up. What do I need to test? An object. So I'll create an object of that product first. This is a hardcoded product, but that's okay, that's in fact what we do most of the time when writing tests since we need to know exactly what we are going to run the method on so that we can verify the results. Since the constructor doesn't set an initial value for the stock of the product, I'm going to invoke the IncreaseStock method passing in 100. You are expecting for this test that this method works correctly though. In fact, we'll need to verify that in another test. This is still part of the setup, so the Arrange block. Now it's time to invoke the method we actually want to test, so that's the Act block. We're going to invoke the UseProduct method passing in a value 20. Finally, we can check if our method did its job, so in the Assert part, we're going to verify assumptions about its correct wording. In this case, the initial stock is set to 100 and then we use 20, so if the method did its job, we should have 80. I'm going to verify this using assert.Equal passing in this again, known value of 80 and then the product's amount in stock property which wraps the field that was changed in the UseProduct method. It was the same, and the test will show us that this code works as expected. We can now go to Test, run all tests, and this will open the Test Explorer in Visual Studio, and woohoo, we're getting a green checkmark. Our code works as expected. Now, I should test the other path. If we have less than what is requested, the amount shouldn't be touched. Here's a code I've written for that and take a look at the name of the method again, it really describes what this test is doing. I've set up the product, so there are 10 items in stock. I now want to use 100 and our code is not allowing this. It's not going to deplete the stock to 0 or even create a negative stock. Instead, the stock remains untouched. In the assert, I am verifying this by checking at the value of AmountInStock is still 10. Here's another test. I also want to be sure that if we use an amount that brings the stock below the threshold, that the IsBelow threshold becomes true for the given product. What I'm going to do here is set a value here 100 for the initial stock. Then I call UseProduct with just one less, so meaning only one single item remains in stock, that is definitely less than threshold, so now we can make another assertion, namely using Assert.True checking an owner-created product instance that IsBelowStockThreshold is now true. Let's run our test again and all seems to be well, we have three tests we have passed. In the code downloads, I have added more unit tests for all the methods of the Product class. I suggest you take a look at these at your own pace.

Summary

And we have arrived at the end of this module which had as its main goal giving you the need and understanding of objects. We've deeply covered the relationship between classes and objects, and we've seen how classes are reference types. These objects are created on the heap. You've then used the knowledge on objects while creating unit tests. You've seen that through the use of unit tests. We can validate the correct working of our class. We have now done a deep dive in working with classes and objects, and we looked at the four pillars earlier in this course. We have seen that inheritance is another one. and we'll look at this in the next module.

Working with Class Hierarchies

Module Introduction

Welcome back to this Pluralsight course on object-oriented programming in C#. In the previous modules, we have covered two of the four pillars of object-oriented programming, abstraction and encapsulation, where we have learned how to design our classes to encapsulate data while providing simple public interfaces for the users of our classes. In this module, we'll dive into the remaining two pillars of object orientation, inheritance, and polymorphism. Through hands-on examples, you'll learn how to use these concepts together to write code with increased reuse, making replications more maintainable and efficient. As always, we'll begin by taking a close look at what we will be covering in this module, so let's do that now. We will set things in motion by looking at inheritance. I will take you from understanding the concept of inheritance to seeing the options available to us from C#. Once we are clear on inheritance, we'll jump to polymorphism, the fourth pillar. We'll look at a few other interesting options offered by C# here as well. We'll look at sealed glasses which is actually pretty simple, and we'll look at abstract classes too, which we'll come across quite often. We'll finish the module by looking at extension methods, a clever feature of C# if you ask me.

Adding Inheritance

So let us kick things off by exploring inheritance, but before we do that, let's jump in a video call with Bethany again. She had something important to tell us. Hey, team, I've been thinking a bit more about the application. The thing is we have a few types of products that I can identify. In fact, there are a few product types which do behave rather similar, but yes, and depending on let's say that type, there will be differences. Is it helpful to know for the application? Is this something we can build in? Well, I think Bethany's call could not have come at a better time, so it's really useful that we are getting this information now. Let's try to understand in a bit more detail what Bethany was saying here. So far, we have worked with the Product class, that is a general way of looking at things, and we have described all the behavior of this class through its methods. Now, Bethany is saying that we should think of the fact that there are, in fact, types of products which have differences, but also many similarities. What we see here is that there is going to be a structure, a relation between different classes. We could say that we have a boxed product which is still a product, but it will probably behave a bit differently. Probably we will have to use the entire box when we use just one product, but ordering such a product will be the same as for a regular product, for example. And perhaps the fresh boxed product also exists, which means that these products will need to be stored in the cool part of the warehouse. Many characteristics from the fresh boxed product are the same as the ones of the boxed product and probably a few are different. So what are we doing here? We are organizing the different classes in a structure since we see similarities and differences between the different classes. By organizing our classes in a structure, more specifically a hierarchy, we'll come to understand which behavior the one class inherits from the other. Uh huh, I have said the word inherit indeed, inheritance is what we are seeing here. The third pillar of object-oriented programming. So what is this inheritance thing then really? The concept of inheritance is something that we are familiar with in our daily lives as well. Using inheritance, we apply what we know about a general set of types to a more specific set of types. For a second, not thinking about C# and about biology instead, inheritance is what we know that causes our eyes to be blue since it is caused by inheriting genes, but inheritance also allows us to create or adopt a behavior based on what we have already experienced. We did use a new behavior based on an existing one. If we apply this to programming in C#, inheritance points to a relation between classes, one class will inherit behavior from another class, therefore, introducing the concept of a parent and a child class. The child class, since it is based on the parent, can reuse the functionalities defined on the parent. The word reuse is really important here. Automatically, that existing, that already-tested and known behavior, is now available on the new class too. Because of this, a relation is created, often referred to as the is-A relation between these two classes. A sports car is a car, a dog is an animal, a poodle is a dog. The parent defines the behavior, and the child receives this now as well. And the parent class is still in control what a child will inherit. Through the use of access modifiers, the parent can decide which members become available for reuse. I've mentioned it already a few times that reuse is really key here. Through inheritance, we can define a behavior in one place and reuse it in many other classes. If we need to change that parent behavior, we only need to do so in one place, instead of many. This avoids code duplication, a practice that you should, in fact, avoid at all times. I have used intentionally some different wording around inheritance. Of course, let's make sure that we're all on the same page since you'll see different terminology being used around inheritance, and it can be a bit confusing. Here's a slide we saw earlier with product, boxed product, and fresh boxed product. Boxed product inherits its behavior from the Product class. In this structure, product becomes the parent or the base class. Alternatively, sometimes you'll also hear the word superclass being used. They mean essentially the same thing. Boxed product inherits and thus will be the child class. Other names include the derived class, the extended class, which I don't often use, or subclass. Again, they all mean the same thing really. Our inheritance can be multiple levels deep. In this case, the boxed product becomes parent for the fresh boxed product. One important note is that the class can only have one direct parent, other languages support that classes have multiple parent classes known as multiple inheritance. Now, while we can have multiple layers in C#, we can have only one direct parent. Now that you have a good understanding of the concept of inheritance already, let's look at it in code. The base class here is a regular class. No special syntax is required to make a class become a parent class. Next, we see the inheriting class called DerivedClass. To indicate, the base class will be its sole parent, we use colon and then the name of the parent class. Now the relation is established. Let's look at this with real classes. On the left, we have the Product class and it defines, as we have already seen, the UseProduct method. On the right, we have the boxed product class, which specifies that it inherits from the Product class. Now, boxed product is a product, and therefore now knows the UseProduct method too. The method is defined as public on the parent which means it's accessible from anywhere, including from a child class. That's how we use a child class. We can now create such a boxed product as we would do with any other class. We new it up, nothing special. But although we haven't defined the UseProduct method on this for because we can still use it. Members defined on the base class become available for the child class. And that however depends on the used access modifier. The parent class is in control to decide which data and which methods are available to the child class, if any. So access modifiers again will play an important role, and I'm talking here about the access modifiers as used on the members of the parent class. Typically, we'll use public, private, or protected. Public still means that the members are accessible from anywhere, so as said, including the child class, of course. Private is the opposite that hides the member of the parent for the child class. The child simply can't access the member in a parent class. The third one is protected, and we have briefly touched on this already, but this one really makes sense in combination with inheritance. Protected makes the member accessible for the child class, but for other classes, so outside of the inheritance tree, the member is not accessible. In other words, protected behaves like public between parent and child, but like private for all other classes. Here, we can see this in action. In the Product class, I now have the string name defined as public, and the DisplayDetails method, also defined as public. On the inheriting boxed product class, we can in the DisplayBoxedProductDetails method use, for example, a name field. Any public member defined on the parent is, of course, accessible from the child class. If you make the field private, which we should do for interpolation as we have already seen, the code you see here simply won't compile anymore. Although the class inherits from the Product class, the parent has decided that this is private, and therefore, is not accessible within the child class. If you want to block access to the name field from all other classes, but still make it available for the child classes, we'll need to look at using the protected modifier. As said, between inheriting classes, this member is now accessible too. Creating a child type that can only do what the parent can do would be pretty useless. Of course, the child can, and in most cases, should define extra members, just fields and methods. Here, I have added on the boxed product class a field that's indeed specific to that class AmountInBox and a new method, UseBoxedProduct. This is the common way to do things. We extend the base class and we get the base class functionality, but that we add what is needed as extra on the child class. One more thing, I have mentioned that inheritance can and will be multiple levels deep. Inheritance in C# is what is said as transitive. This means that a member defined on a parent will be accessible not only to the direct child, but also to its child. Again, the access modifier can be used to restrict this. Some more input from Bethany. Before we start coding again, Bettany has said that there are different groups of products, including products we store per item, some others come boxed, and other are bulk products. Fresh products also need special care as they will expire quickly. Thanks Bethany. Very useful information.

Demo: Adding Inheritance

With this information, we can go ahead and start coding on this. We'll add inheritance and we'll create the different classes as specified by Bethany. Alright, we are going to create a class which inherits behavior and data from the Product class, which we have already created, that contains a lot of behavior that we can reuse in other classes. I'm going to start with the creation of the boxed product, so it is a product, and therefore, we can reuse the functionality. Let's add a class first to the ProductManagement folder, so right-click, select Add, New class, and let's call it BoxedProduct. I'm going to make this class public again, and now we are going to let this class inherit from the base Product class. I can do this by typing colon and then the name of the base class, so Product. Now we have a new class, and we can create objects of this class that will have the same behavior and data as the parent class, that is depending however on the access modifier. However, the first thing I see here is this red squiggly here, and if we hover over it, we see that Visual Studio indicates that it's missing a constructor that takes 0 arguments. We'll see in the next part why we're getting this error. For now, I'm going to use Alt Enter to let Visual Studio fix the error. We see that it's suggesting a few options to generate the constructor. I'm going to take this last one here. Now we get in this child class a constructor that accepts the same parameters as in the base class, but notice this part here. Using base, we invoke the base constructor passing in these arguments, and we create a boxed product using the constructor. In fact, a constructor, the one with the parameter list, will be invoked on the Product class, but more on that soon. With this class created, I can now, in fact, create a boxed product object. So let's head over to the Utilities class, and just to play around with this class, I'm going to create a new instance. On this new instance, I can now use the methods and data which are public or protected on the base class, so on the Product class. I can, therefore, invoke the UseProduct and IncreaseStock methods just like we did on a regular product instance. I can't, however, use the private methods or data from anywhere, except the class that contains the private member. If you look at the Product class, we can see that we have the DecreaseStock method declared as private. It is, therefore, not possible to invoke this method from the Utilities class. If we make this method protected, so in the parent class, it is still private for the outside. It's only accessible for the inherited classes now, so we still can't access it. From here, from the Utilities class, but it is now accessible in the BoxedProducts class. I'll show you this very soon. So this shows how we can determine what is accessible for inheriting classes and what isn't. I have the ability as the creator of the class to decide what's hidden and what's available. At this point, apart from the name, our new class is not really different from the original Product class. What's the use then? Well, we aren't limited to only what the parent can do. The child class can extend the parent behavior or data, so let's make a change in the BoxedProduct class. Thinking about the concept we're modeling here, a product that comes boxed, there's data and behavior that is specific to this new class, so we're going to extend this class. A BoxedProduct comes in a box, in fact, a number of items will be in a box. I'm going to capture that in a new field, AmountPerBox, and I'm going to expose this through a property. Now, our new class can already do a bit more than its parent. We can also, in fact, change the constructor so that the value for this field can be passed in when creating a new BoxedProduct instance. Notice I've made another change, I've removed the unit type parameter. Why? Well, since it is a boxed product, I can specify this, let's say, fixed, and it is what I'm doing here. I'm passing this when I invoke the parent constructor. Now we build will get an exception since we have changed the constructor. Let's update the use of BoxedProduct over here. There we go. Now it includes the amount per box, but the unit type is gone. Of course, I also still need to remove this call to the DecreaseStock method. Now it's happy again when we do a build. We can also add new behavior to our class. I've now created a new method DisplayBoxedProductDetails which adds this line here. And I'm also bringing in a new version, in fact, a new method for now to use a boxed product. Here's the code for that new method. What I'm going to do here is calculate how many products really are going to be used. Boxed products will always be used per box. In other words, when we don't use the full box, the entire box is considered used. That's what I'm calculating here, the real amount that'll be used. And then I'm using the UseProduct method passing in the total amount. Wait a minute. What am I doing exactly here? From this inheriting class, I invoke the used product on the parent class on the Product class, which is declared public, and therefore, is also accessible from this class. The rest of the behavior is really what's already defined on the parent. I don't want to rewrite this code here. That's exactly what inheritance is all about, reusing code and not having to write the same code multiple times. Now, while we're talking about this, we could from here now call the DecreaseStock method. That method was now defined with the protected access modifier, and since we are now in an inheriting class, we can use protected members. It shows you the difference between public, private, and protected. I'm going to add two more classes, and I'll show you the result. Here is the bulk Product class, which at this point is still very simple, it just contains a constructor, and here is the FreshProduct class, which also inherits from the Product class again. It's again a simple class, but this one does add two properties, the expiry date time and the storage instructions. Since both of these new classes inherit from the base product, they do have access to the public and protected members of the product class. The inheritance can also be multiple levels deep. Let's add another class, that FreshBoxedProduct. This now inherits from the BoxedProduct as you can see. I've added a new method here UseFreshBoxedProduct, and since this one inherits from the BoxedProduct class, it has access to the UseBoxedProduct method, which was defined on the BoxedProduct, so we can use that here too. It should already give you a good understanding of creating inheriting classes.

Inheriting from System.Object

When we create an object of a derived type, we use a constructor of that derived type as we can see here, but remember the oh so important sentence that relation between the two classes, there is a relation. Every BoxedProduct is a product, this means that the base product lies underneath, and thus, it will need to be created too. In fact, when we instantiate the child class, first, a parent constructor will run and only then will the child constructor run. If you think about it, this is quite logical. The child constructor may work with inherited fields which are initialized by the parent constructor, hence, it'll need to run first. Now we can somewhat influence this. Take a look at the code you see here on the slide. This is the constructor of the BoxedProduct as we saw it in the previous demo. I promised I'd come back to why this constructor is actually needed. We now know that the parent constructor will run as part of invoking the child constructor, and since the parent doesn't define a parameterless constructor, we need to create a constructor on the child class which invokes the parent constructor passing in the required arguments. No child BoxedProduct can exist without first executing the parent product and giving that the correct values for initialization. We are, in fact, using a new keyword here, base. This base, which is similar to what we already saw with the this and overloaded constructors, will invoke the base constructor passing in these arguments. This way, the product instance will be correctly initialized. When we create a BoxedProduct, we would now then assume that two constructors have executed. In fact, three constructors have executed. Let me explain. And I'll start with this one. Everything is an object. There is that is-a relation once again. You may say, well, yes, I already know that when we instantiated the class, we'll get an object. Yes, but there's more. Every type, so every class derives in turn from the base system object type, that is the parent type to all types in .NET. Everything is an object. And since all other types, including custom ones we create and built-in ones, does all derive from a base class, that class's constructor will also execute for every object we create of just any type. It, therefore, comes with a parameterless constructor and it defines a few other members that all types will get for free because of this built-in inheritance. The ToString method is probably the best example. Every type can use this method defined on the base system object. So in fact, we should update our diagram again here. To see the entire inheritance string, we need to add one extra level here, system object. Our custom class that we create first has no other parent class that we have defined, so this means that this class will get System.Object as its parent. BoxedProduct inherits from product and thus already defines a parent. However, indirectly through through product, it also indirectly inherits again from the system object and thus gets, for example, access to the ToString method which is defined on System.Object.

Demo: Inheriting from System.Object

Alright, let's look at how we can see that everything inherits from System.Object. If we head back to the Product class, we can see indeed that it doesn't define the base class. Now, while it's not shown, this class does, in fact, inherit from the base System.Object class. I can write colon and then System.Object or just object. As you can see, Visual Studio face this immediately saying that this isn't needed. It's not an error, but explicitly adding this is also not necessary. Nothing changes when we do this, this is implicit. If you go to the System.Object class, we can see that this is a class that comes with .NET and it defines some basic functionality available on all classes. This includes ToString, Equals, GetHashCode, Finalize, and MemorizeClone. The ToString method is the one you used most often. It will be done for every type, a string representation of the type. So if we return, for example, to the Utilities class, and we call on our BoxedProduct the ToString method, this will work. Why? Since a BoxedProduct is a product and a product is a System.Object, and therefore, it can access this method which Microsoft added on this built-in base type. So without really knowing, we have used types, every type, in fact, which we're already inheriting from the base object class..

Working with Polymorphism

We have now thoroughly covered three of the four pillars of object orientation. It's now time to tackle the fourth and last one, polymorphism. Let us first understand it, and then we'll see how we can use it from C#. What you have seen so far is that through inheritance, we can write the base behavior on the base class, and in the inheriting types, we'll get in this behavior, and therefore, we can reuse code by routing it once on the base class. While that is all nice, what if you wanted to have a different behavior for the functionality on the derived type? Should we then go back to writing all the code in separate classes? But that again takes away what we have just one, and like namely having a common interface or different types in the hierarchy. There must be a better way. Introducing polymorphism, the final pillar in our object-oriented journey. If we analyze the words, we can distinguish two paths, poly and morph. These are Greek for many-shaped, pointing to the fact that we can have multiple forms of the same method, so with the same name. In fact, there are multiple aspects to polymorphism, so let us dig into this principle. Through polymorphism, and in fact, supported, of course, through inheritance, it is possible in C# to add runtime, treat an object of a derived type, so in our example, BoxedProduct, through a base product type reference. We'll see this is an example, but what can this be useful for is the following. We could create an array of objects of derived types, so child classes of product. While they are all objects of specific types, we can, in fact, treat them as base type objects. Therefore, enabling us to invoke the methods defined on the base class on all elements in the array, that is pretty cool. However, we then are invoking, by default, always the same method, being the one defined on the base class. Through polymorphism, we can add new implementations in the derived type, basically, overriding the behavior defined of the base class. This means that we can create a new implementation for a method which is defined in the base class in one or more of the inheriting types, and to do so and thus enable polymorphic behavior between parent and child class, we use two C# keywords, virtual and override. We'll see this in action in just a minute. We are then effectively creating methods with the same name and signature on the base and child type. We have already seen something similar, namely method overloading. If you remember, method overloading points to the fact that within a single class, we can have multiple methods with the same name, but a different signature. This is often also referred to as a form of polymorphism, namely compile time polymorphism. Now we won't be looking at this one here though. Now that you already have a basic understanding of polymorphism, let's build it up step-by-step and see all the options since it's a pretty neat concept that you'll often use in your C# development activities. Let's start with this slide with something we, in fact, already saw. Because of inheritance, a BoxedProduct is a product. This means that we can, in fact, treat each BoxedProduct as a product. There are products, and thus we can use a base type reference, sub product to point to an instance of a derived type like BoxedProduct or FreshProduct. That's important. I'm using a base type reference to point to an object of a more derived type, and that is possible thanks to polymorphism. We cannot work with this instance and evoke the UseProduct method, defined on the base product class, and then I'll have the same result at least for now, and this is, in fact, what happens. The object on the heap is really a more specific type, but because of inheritance and there is a relation, we can use a base reference to point to an instance of a more derived type. Say that we have a method which accepts a product instance as parameter. In the body of this method, we can then work with the product, that's nothing new. Now, look at the highlighted line here. I am now calling this method, but now I'm passing in a BoxedProduct instance. Does that work? Will that compile? Well, the answer is yes. Since every BoxedProduct is a product, what is happening here is that C# will make what is known as an implicit reference conversion from the derived class to the base class so product. Because of inheritance, we know that this method exists and is accessible, and therefore, C# will know it's possible to make this conversion for us. We don't need to do this manually. Now pay attention, the object is passed as a base type reference, so again, only the members exposed on the base type are usable. Now let's assume I want to create what I already mentioned, an array of product objects. In this array, I add a product, but in fact, I can also add a BoxedProduct. A BoxedProduct is a product, therefore C# will do the implicit conversion and this will work fine. The array will contain, not only real products, but also objects of derived types. Here's the array now with in-app products, but I also have added BoxedProducts and FreshProducts. I can now loop over the array and call the UseProduct method since it is defined all the base product class. That is super handy, isn't it? Well, yes, as long as all the derived types can live with the base implementation of the used product methods, but if they need a different form, we'll need to create a new version of the method, multiple shapes of the same method polymorphism. Making this possible in code is done in two steps. First, the base method needs to be declared as a virtual method, so using the virtual keyword. Now this code is saying it is possible for deriving types to define a new version of this method. If they want, they don't have to, they can use this default implementation, but they can include a new one. If the derived type needs a different implementation, we can do so using the override keyword. Note that that method has the same name and signature, this is required for this to work. Once we add a new implementation on the derived class, this custom implementation will be used when the UseProduct method is invoked or an object of the BoxedProduct type. The base implementation will be used anymore when working with an object over a derive type that contains an override. Now you may be thinking, and do I need to indicate this somehow? Well, great question, but you don't need to do anything yourself. C# will always take the most specific implementation based on the type of the object we are working with. Let us return to our earlier example with the array of products and auto-derived classes. Notice I have the UseProduct defined as a virtual method on the base class and the BoxedProduct has its own implementation, so a method with the override keyword is included. The FreshProduct class, on the other hand, doesn't have a custom implementation, so it relies on the default method. If we now loop over the array, first, we'll encounter a product. So quite logically, this method here on the base class itself will be used. Next, we encounter a BoxedProduct. Since this class now defines an override, this version here will be used. Again, we don't need to do anything for this to work. Automatically, C# will take the most specific version. Next, we will encounter a FreshProduct object as-is, that doesn't have a specific implementation, the default defined on the Product class will be used.

Demo: Using Polymorphism

Let's head back to Visual Studio and see polymorphism in action. Take a look at the current version of the BoxedProduct class. It has the DisplayBoxedProductDetails and use BoxedProduct methods, but it also has the inherited DisplayDetailsFull and UseProduct methods. Now this is a bit confusing for the user of these courses. Are these methods doing the same thing, but with a different name? In fact, the code that I've written here is basically the specific version of these methods for the BoxedProduct class, in other words, these are new implementations for methods defined on the base class, just like with method overloading where we have multiple methods with the same name, but different parameters, we can have different methods in parent and child classes with the same name and the same parameters. In this case, we're going to override it in an inheriting class, the behavior defined in a parent class inside the inheriting class. We are going to create multiple forms on the same method, so we're going to apply polymorphism. We'll see that C# is going to use always the most specific version for us, and to enable this behavior, we, of course, need to work with inheriting classes, and then we need to make a change on the methods in the primary class where we'll want to allow this, so this overriding. The parent class can decide if a method can be overridden in a child class. So let's start with going to the parent product class, and there, I'm going to open up a few methods for overriding by adding the virtual keyword, adding virtual means it's possible for an inheriting type to create a new implementation in a polymorphic way. Now it is not required that the inheriting class overrides this base behavior. If no override is created, the base behavior will be used. We'll see that in action when we run the application. So what I'll do is I'll add a virtual here. So I have now added virtual on pretty much all the methods, and by default, this isn't changing any of the behavior as it stands, but it thus open up these methods to be overridden with a new version in inheriting classes. Now let us go back to the BoxedProduct class. I'm going to comment out the UseBoxedProduct since that specified in fact the same behavior, but with a different name. Instead, I'm going to create an override version. In there, I can add the code that we have in the other method. One important difference is that I now need to specify here at the end based off UseProduct, so first, we execute this custom logic and then we explicitly call the base version of use product. Now the BoxedProduct has a different implementation, a different override of the UseProduct method. I mean, I'll work with the BoxedProduct later on. This method will be used, instead of the one defined on the base Product class. Also for the increased stock method, this class needs a different implementation since these are BoxedProducts. Adding a box basically means that we're adding the amount per box in one go. That's what we see here. Now, it seems that we're getting another red squiggly. Let's see what Visual Studio is saying here. AmountInStock can be set from here it seems. Let's take a look at the parent class again. Indeed, we defined this not to be settable from the outside. This was indeed a good decision since this is a value that is calculated from within the class and shouldn't be set, so I made it private, but now, we're inheriting from this class, and thus this property should be accessible to set from within inheriting classes too since they might give a different implementation which will require access to this property, so we're going to make this into a protected set so that inheritors can access it, and I'll do the same for the IsBelowStock threshold as you can see here. We may need to access this property too from other inheriting classes. There is another implementation of the increased stock that we're going to override where we can specify the number, again, as a parameter, thus this implementation went up adding the amount of boxes times the amount per box, more red squigglies. It seems that we have more private members on the parent class we need to access from the inheriting classes, maxItemsInStock should become protected, and also in the Log can they CreateSimpleProductRepresentation aren't accessible, so I'll make this protected too. I'm now working on this, technically. I need to build this check into the other IncreaseStock as well. Maybe that's a good exercise for you to try. In fact, you can do this by letting this method invoke this one with one being the value, thus reusing the code, but I leave it up to you to make an improvement. We also still had the DisplayBoxedProduct method. If we look at the parents in DisplayDetailsFull method, it is slightly different, so we can indeed create yet another override. This added this Boxed Product string, which isn't in the same place in the string output as what the virtual method with the parameter did, so we are indeed good with adding another override here. Let's do a build now. No, it seems that we're getting an error. Indeed on the FreshBoxedProduct class, you were still working with that specific UseBoxedProduct method. I'll remove this since this was just to show you that inheritance can be multiple levels deep. Of course, other classes can contain overrides as well. Here is the FreshProduct class. I'm going to add another override here since I want to show the storage instructions and expiration date in the output, so I've added these in the override. Alright, now we have defined our classes so that they have different versions, different overrides if needed in the inheriting classes. The child classes now define new behavior where it's necessary to do so. In other cases, the default implementations will still be used. Let's now see this polymorphic behavior in action. I'll also show you how we can treat a more specific type through a parent reference. Again, something we get through polymorphism. And for this, we are going to work on the a method to create a new product from utilities class. We don't have that yet as we can see here, but I'll uncomment this. Of course, we need to write this method, and for this, we are going to rely on polymorphism. Let me paste in some code, and I'll take you through this. I've added three methods. The show all unit types and show all currencies are the simple ones, so let's look at them first. In fact, they loop over the enumeration and they are used from the console UI to show all values of the enumeration for the user to choose from. Now let's look at the more important

ShowCreateNewProduct. The bulk of this first part is pretty straightforward as we'll ask the user to enter the values that you request. Notice I'm also asking for a product type which I also capture. Then also important is that I create a product reference, so a variable of type product, but it's set to null. So the reference is created, but the object on the heap is not. We'll come to the latter in just a second, but remember for now that we already have a reference of the parent type ready, so the parent type, remember that. This line here is again using LINQ syntax to find the next available id. In real world applications, this could be, for example, determined by a database. Alright, now the polymorphic part which happens in this switch here. I'm going to, based on the select productType, initialize a new product instance or one of the inheriting types. If the user selected regular product, then we are going to create well for now at least a product instance using the product constructor. Now notice that if the user selects 2, then a BulkProduct is created, but the object is referenced again using the new product reference. This is polymorphism at work by using a base reference to point to an instance of a more specific type, but since the BulkProduct is a product, this works fine. In the third case, we are going to create a new FreshProduct, which is pretty much the same. For the BoxedProduct, we need the number in box which we ask first and then we call the BoxedProduct constructor. Finally, we add the product which may be of the product type or one of the inheriting types to the inventory. That is a list of products, but since all these inheriting classes are products, we can treat them as such. Now, at this point, the data in the inventory list that's already there is loaded from the repository. Let's look at what changes we need to make there currently so in the code that we already have, we create just a product instance, so that means that the items loaded by reading in the file are just products, not the inheriting types. We'll need to change that. I'm going to show you the outdated code next. I have now changed the code so that we expect to read another value from the file, so on each line, this one here, that will be the product type. In a similar way, I'm going to create either a product or one of the inheriting types. The convention used here is that one in the file means that it's a BoxedProduct, it doesn't have to be the same as the order we had in the UI. If that is what we found, we then need to pass another value the amount per box, and that's what I'm doing here. Then I create a new BoxedProduct through the constructor again, and then I assign that to the base product reference. The other ones are the same and a 4 means that we're creating a product. Then the product is added to the list of products which is then returned. Neither product is loaded. When we start the application, we'll be either of type product or one of the inheriting types, but they treat them all to a product reference though. Now, finally, let's look at how the overriding methods will be used. For that, we'll run the application. If we now select inventory management, and let us say that I want to show the details of product ID 3, which is strawberries and they're a FreshProduct, we see that we effectively arrive in the display the first override of the FreshProduct class. If I select 1 here, we will arrive in the default implementation in the Product class, so what we see here is that C# will always look at the real type of the object even though we reference it through a base type reference and it will then use the most specific implementation that two is available to us through polymorphism.

More on Polymorphism

If the BoxedProduct extends the product with functionality, so by adding another member, we can only use that member through a reference of the child type. Say, let us for example, the BoxedProduct class has a new method, UseBoxedProduct. Here we are again using a base reference, so product, to point to instances of BoxedProduct. That's all good, but you will get an error though, if we try to invoke a child method, so the method defined in the child class through a base reference. That's quite normal. This base class doesn't know anything about his member, and therefore, can't use it. When we now pass a product reference to a method, we can indeed, by default, only use the methods defined on the Product class, but can we, in fact, now say go back to the derived type inside this method? Well, yes, but we're not really sure what we're getting in. We could be getting in a real product instance or we could be getting in an instance of a derived type, but converted to the base type. If we want to, inside this method, try to use one of the methods defined on one of the derived types, we need to do an explicit cast. Of course, it would be dangerous to just do this cast. If the cast to the type conversion fails, the application will crash. However, C# comes with the is keyword which we can use as we see here. In this snippet, I'm again still getting in an instance of the base product type, but then if the instance is, in fact, a BoxedProduct, I want to invoke a specific method only known on the BoxedProduct class. I'm now using this highlighted line here to ask C# to take a look at the real object. I'm asking C# to take a look at the real object if that is a BoxedProduct, but then cast it and then we can invoke a specific method, for example. We are going to do an explicit cast here. This way, we are safe, and we know that the application won't crash. Next to the is keyword, C# also has the as keyword for this, which in some cases, is simpler to perform the exact same thing as we have just seen. Using as the type conversion will be tried, and if it succeeds, the instance is assigned to the variable, but if the cast doesn't work, the result will be null. I try to cast using as, and if it works, we'll get a reference to the derived type. If it fails, we don't and we'll get back now, but again, we've done an explicit cast here. We have seen it every type inherits from system object, which is sometimes referred to as the root class. C# goes with the object keyword, which is really an alias for the system object type, just like int is an alias for Int32. Now since everything is an object, we could rewrite the method now accepting a parameter of type object. So then passing through the method an object reference, inside the method, unless we perform again, an explicit cast. We're now limited to using the methods defined on the object type. These include ToString, Equals, GetHashCode, and GetType. The one you use most of the time is what I'm using here, the ToString method. What this will do in its base implementation is returning the name of the type as a string. Again, this ToString behavior we can override if we want. Now I'm showing here we have, in fact, already seen in the previous demo, but I want to call it out again, say that we are faced with the following. Again, a derived type has a more specific need for the UseProduct method, but this time, instead of being a complete new version, we basically want to execute logic and then invoke what exists on the base class, so instead of duplicating this code entirely, which we should avoid at all costs, we can call the code in the base class. The base keyword, we've therefore, already seen for calling the base constructor can also be used here. This will effectively invoke the base implementation when we ask for it. Keep in mind that method overriding won't do this by default, unlike what happens with base constructors. Now, what happens if we omit the override keyword and create a method with the same name in the derived type like we see here? We are doing what is known as shadowing or hiding. We are indeed hiding the base implementation and we will be creating an entirely new version of the method as the one defined in the base class. If you would write this code, you'd get a compiler warning stating that indeed, you are hiding a base member. The warning is, in fact, triggered to warn you that this may not be what you want. You, therefore, should either specify that you override or that the hiding is intentional. If hiding is intentional, the best practice is to be explicit about this, this we can do by adding the new keyword in the method declaration as we see here. Now, this method is a new one called the derived type. Now you may be thinking, why would this be useful? We'll take a look at the code here. If we shadow the method, so using new, invoking the UseProduct method now on the derived type directly will still have the exact same effect, the new method, so the one on the BoxedProduct is used, but if we invoke the method through a reference of the base Product class, we no longer invoke the derived class method.

Instead, the base implementation will be used. In general, using shadowing isn't a great idea, but there are some use cases for it.

Exploring Abstract and Sealed Classes

We have now seen an abstraction in object-oriented programming is about hiding the internal details and showing only the functionality to work with the type. C# has support for what's known as abstract classes where we can omit the implementation of a method. Also, it's possible to create what is known as sealed classes. Let's look at both these options we have with classes in the next part. We have now used the different types which inherit from the product base class, and we have also used the product class to define common functionality, but if you think about it, aren't the inheriting classes the ones we should work with all the time and is, therefore, the concept of the product, not something abstract, something that exists in code, but doesn't exist in real life? And if so, should we then perhaps block that it's possible to create a product, but is that only allowed inheriting causes to be created? The latter is possible through the use of abstract classes. They are classes that cannot be instantiated. We can't new them up to create new objects. You may be thinking what's the point then? Well, great question. We can inherit from them, that still works. Using abstract classes, we can thus define an abstract concept in code so that checked functionality is still defined in one location for the child classes, but we can, on the other hand, block that the type is instantiated, and we would do this when the concept of the abstract class well, isn't real, it's not something we'd create. A product might be too generic to instantiate. That's what abstract classes are useful for. Being abstract, it might be so that you want to define the functionality, but you may also not be able to even define the base functionality. That two is possible with abstract classes. We can define one or more methods to be abstract inside of an abstract class. They don't have an implementation in this very class. Inheriting classes will need to provide an implementation though, unless again, they are too abstract. It's possible to have again, multiple layers of abstract classes. Creating abstract classes and abstract methods can be done using the abstract keyword. Now this may be a bit abstract, pun intended. Let's see abstract classes in action. Through the use of the abstract keywords, a class becomes abstract. It can define members as usual, but more on that in just a second. As said, maybe the concept of a product isn't something we should be using. We maybe shouldn't be creating object of this class since it's too much of an abstract id. Perhaps all products have specific shared behavior and does the

Product classes sole purpose in life is that we can use it to define that common functionality which can then be inherited by other classes. Because we define the classes as being abstract, the first result is that we cannot instantiate a product anymore. If then, BoxedProduct inherits from the abstract base product class, this will still work. I'm instantiating a new BoxedProduct, but because of polymorphism, we can use a reference of the base type to point to an instance of the inheriting type, that hasn't changed. This assumes that the BoxedProduct is a regular non-abstract class. Inheriting from an abstract class is the same as inheriting from any other base class. Not being able to instantiate the class is just one consequence of making class abstract. As said, sometimes we may want to define a functionality on the shared base class, but defining an implementation isn't possible since the id is, well, too abstract. In that case, we can define one or more methods to be abstract. Notice here that we have a method with a semicolon, no set of curly braces, and the abstract keyword. This is an abstract method. I define it so that because of inheritance, I know that all inheriting types will define the method, and that through polymorphism, I can invoke the method through a base class reference. Inheriting types now need to provide an implementation for all abstract methods, or alternatively, they need to be defined as an abstract method, and therefore, abstract class too. An abstract class can also define regular methods and virtual methods. Neither of these require that they receive an override in the inheriting type.

Demo: Converting to an Abstract Class

Still too abstract? I can relate, but I need to stop making these stupid jokes. Anyway, time for a demo where we'll make the Product class abstract, and we'll see what the consequences of this will be. In the previous demo, we ended up with this code here in the repository. When 4 was found in the line of text, we ended up creating a product. Whereas, in all other cases, we were creating one of the inheriting classes, maybe the concept of product is too broad, too high level, and we only want to work with inheriting classes. We, of course, leave the Product class as-is in terms of data and functionality it defines. It's definitely a good idea to group reusable members on that parent so that all inheriting classes can benefit from code reuse, but if we decide not to allow to work with products directly, we can go ahead and prevent the creation of product instances by making the Product class abstract. Doing this means, like I said, that the concept of product isn't something we want to work with. We use it as a place to group common functionality. All the code that is in here can still be in here. All our methods now have an implementation and that's okay. These methods can still be virtual and can still be overridden in inheriting classes. Let's do a compile. We're getting a few errors. Indeed, we've written code both in our application as well as in our test project, how we instantiate an object of type product as if the class is now abstract, that's not allowed anymore. What I'll do as a fix is I'll create another class. I'll call it the regular Product class, which basically will just inherit from Product class like I've done before. Here's the class. I'm not adding any extra functionality on it. It's basically inheriting all from the parent Product class. What I can now do is replace the usage of the product instantiation with regular product instantiations, and then we should be okay again. Let me quickly do that. So here's the result. Here in a repository, instead of newing up a product, I am now newing up a regular product. The constructor uses the same set of parameters, so no extra change was needed here. The same goes for the utilities class. As we can see here, and also in the test project, I replaced the use of product with regular products. If we now compile again, things will build fine as before. Now apart from not being able to instantiate a product anymore, making the class abstract also will allow us to create methods without implementations, I should say method declarations. If we say that it makes no sense to even come up with the definition of the base functionality or if we know that the functionality will be different for all and everything types, it is not useful to basically create the base implementation, and thus we can leave it out, but we do need to define the behavior of the Product class, else, we lose polymorphic behavior. Therefore, it is possible inside abstract classes to define an abstract method which uses the abstract keyword, but doesn't require an implementation in the base class. The fact that we can define methods without implementation explains why we can't instantiate abstract classes. C# wouldn't know what to use. Okay. So assume for our application, and it makes no sense to create a default implementation for the IncreaseStock method, so this one, the one without parameters. I'm going to comment it out, and I'm going to add an abstract method instead. Notice that this method has no implementation, so there's no method body, but it does add in a semicolon. Making this abstract now requires, again for polymorphic behavior, that all inheriting types need to provide an implementation for this method. Otherwise, they become abstract classes too. If we do a build, we'll see that a few of the inheriting types don't define one yet, so I'm going to add these now. I've added the code for the method in all types that didn't have one yet. It's the same code, but that's of course, because we're creating a demo, not a real-world application. Writing a method that implements an abstract method is nothing different from one that overrides a virtual one, as we can see. If we now build the application again, things will be fine again, and the application works as before.

Demo: Using Sealed

When you are creating classes and you wrap them inside of a library, you may want to block that other users inherit from your class, thereby, creating a new type that basically gets all the functionality of your class for free. If you want to block this behavior, you can make the class sealed. Now, it becomes impossible to create a class that inhabits from your original class. If you make the Product class sealed, which would be weird, but anyway, the code here on the slide wouldn't compile since we can't inherit from it anymore. Now let's return to Visual Studio and work with a sealed class. We can end up in a situation where we don't want another developer to extend a certain class any further. We can prevent inheriting from an existing class by adding sealed to the class definition so that we decide, not sure for what reason, but anyway, that the class BoxedProduct shouldn't be inherited from, so I add sealed to the class definition. When we build the application now, we'll get an error again, in fact, we'll get two, but they're both related to what I have done, namely making the class sealed. Indeed, we had earlier on created the FreshBoxedProduct and that was inheriting from the BoxedProduct. However, the latter is now sealed, and thus, it is not allowed to inherit from. Sealed can be useful if you want to distribute your code, maybe as a library, for example. By making your classes sealed, you prevent others from inheriting, and thus, repackaging your code.

Using Extension Methods

In the final part of this module, we'll take a look at using extension methods, a cool feature that I often use when writing my applications. We have now looked in detail at the different options of inheritance. Through inheritance, we not only get the ability to reuse code, we can also, of course, like we have seen, define extra functionality on inheriting classes without having to duplicate code since that is defined on the base class, so inheritance is definitely one way of extending an existing class. Of course, another way to extend the functionality of an existing class is simply adding new functionality to the original. This, of course, changes the original, and we may break existing functionality, so our unit test will come in handy here, but what if we don't have access to the original class? Perhaps since the type you want to change is created by someone else or is part of .NET or if it's sealed, then as we have seen, we also can't do anything with it. Well, there's still a way out and that is using extension methods. Through an extension method, we can extend an existing type without changing the original type, so we can use them, even if we don't have access to the original class or, for example, if it's sealed. I say extend, but it's not really adding or changing the original class, of course, since that wouldn't be possible. However, from a syntax and user point of view, it feels as if the method we are adding is part of the type. Say that our Product class is sealed and perhaps it's in a library, so it cannot be modified. Well, then through an extension method, we can still add a method to it. Extension methods are static methods defined in a static class. Although they are defined in a static method, they can be used as if they were instance methods on the type we were extending, so using the method won't be any different. Here, we see indeed that I have defined the static ProductExtensions class, and I have defined the static RemoveProduct method. As the first parameter, we need to then specify which type we are extending, and this requires that we include the this keyword here again, so the this Product product. Now C# will bind to remove product method onto the Product class, even though it is a sealed class which can be extended through inheritance. As said, we can now use this method in a regular way. Take a look at the snippet here. We, indeed see that I'm instantiating a new product, but now I call the RemoveProduct onto the instance. So although this method is a static method, we use directly on the instance, rather than the type like we normally do with static methods.

Demo: Using Extension Methods

In the final demo, we will look at creating and using an extension method. In the final demo of this module, let's look at extending a class through extension methods. Extension methods are a way to extend classes without changing the original class. There are a bit of a trick that .NET allows us to use, and using extension methods, we can even add functionality to a class that is sealed or that's inside a library that we use, so one we don't have access to the source code. An extension method, as we'll see, will behave like a regular method when used on an object. Indeed, they will result in an instance method, so one that we can invoke on an object, not on a class like a regular static method. Let us add an extension method to the Product class. Now to be clear, we could add this method directly onto the Product class since we have access to the source code. Now assume for this demo that we don't have access to it or aren't allowed to make any changes to it. Using extension method, it still seems as if we have added a new method onto the Product class. I'm going to start by adding a new class called ProductExtensions. Remember that Bethany explained to us that they have products that have a pricing euro, dollar, or pound? Well, what we'll do is creating a new method to convert the price from the current currency to a target currency. I'll create this functionality as an extension method to the Product class. Indeed, what I'm going to do is I'm going to write functionality for the Product Class, even though I'm in another file, that is very weird. First, in order to be able to do that, the class that contains these extension methods needs to be static first, then I'm going to create the method. Hey, that's a bit weird too, no? True. Let me explain. Extension methods are always static themselves. I've given the method a name and a return type. Nothing special there. Then the first parameter is the type we are extending. Here we are adding this functionality to the Product class, so we add this as the first parameter prefixed with the this keyword. The second, and optionally, other parameters are regular parameters we add to this extension method. Keep in mind that the first parameter is always the type we're extending. It can't be anywhere else in this list. Let me add the rest of the functionality. I've added some static double values being the currency conversion rate between the different currencies we support in the application. Then inside the body of the method, we look at the product's price object. That's the one we have through composition and it has a currency. The rest of the logic is pretty simple as it will look at what we have and what is the target currency. In the end, the converted price is returned to the caller. Now with this extension method in place, we can start using it. For the sake of simplicity, I will create a new regular product manually In the Utilities class. I can now simply call the ConvertProductPrice method passing in the target currency. I'm evoking this method just like any other method we define on the Product Class, even though this one was defined in a separate file and not part of the Product class. That is the power of extension methods.

Summary

Inheritance is a super important topic in object-orientation. I think the length and depth of this module has proven that. We have seen that inheritance allows us to move common members to a base type and that inheriting types inherit this functionality and data and can make use of it. To do access modifiers on the numbers, the creator of the class can indicate what is accessible and what should be hidden from the outside. Then we moved onto using polymorphism allowing us through the use of the virtual and override keywords to create different implementations of a method on the different inheriting types. You've seen abstract classes and extension methods as two other features in this area. One more module to go, and there, we'll explore interfaces.

Reusing Code through Interfaces

Module Introduction

Congratulations on making it this far in our Object-oriented Programming in C# course. You've already gained an in-depth understanding of how C# supports the different pillars of object-oriented programming, including the powerful concept of inheritance. Through careful consideration of these principles, we've been able to create maintainable and efficient code that is easy to test and reuse. Now, in this final module, we'll continue our journey of exploring the different types in C# by looking at interfaces. You'll learn how to use interfaces to create more flexible robust code and is capable of adapting to different situations. By the end of this module, you'll understand how to create and implement interfaces in your C# code, giving you even greater control and flexibility over your applications. I'm still Gill Cleeren, and I'll be your guide for this module, walking you through each step of the process and providing you with practical examples, so let's dive into the world of interfaces and learn how they can enhance your C# development skills. We will start the module by getting an understanding of interfaces first, what they are and why we should use them, and our applications will be the questions we try to solve first. Once we get the hang of interfaces, we'll start working with them, so we'll create and implement an interface in our application. In the final part, I want to discuss how also interfaces have a polymorphic behavior. Ready? Alright. Let's do this.

Understanding Interfaces

So as said, let's look at interfaces first so we're all on the same page. We looked in the previous module at abstract classes. I'm sure you remember their characteristics. By adding the abstract keyword to a class, the class becomes abstract and can't thus be instantiated. The idea behind an abstract class is really that it's too much of an abstract given that shouldn't be possible to instantiate. Only inhibitors can be instantiated. Abstract classes will typically contain abstract methods which don't have an implementation. The idea is the same. Since we're working with something that is too abstract, it doesn't make sense to create a default implementation even, so we emit doing so. However, we do include a method step, so a bit of a placeholder with the abstract keyword added, and that's an indication that inheriting classes must provide an implementation for the method. If they omit doing so, which they can, they will become abstract too. By adding the method in the abstract class, we know for a fact that the method is there, and through polymorphic behavior, we can evoke the method on references of the base abstract type. Perfect. One more thing, abstract classes can and typically will contain regular and virtual methods as we have seen for functionalities where the base implementation makes sense. In a way, what we create is a contract. The inheriting classes will need to provide an implementation that we know. Interfaces are similar to this behavior will take the abstraction, well perhaps a level further. An interface is a contract and will define a set of functionalities that the class needs to implement. The latter class is set to implement the interface, and it needs to provide an implementation for all members defined in the interface. Interfaces are often used as a mechanism to define a set of related functionalities that classes that implement the interface must then provide an implementation for. This is indeed similar to what we had with abstract sources and abstract methods. A class can also implement multiple interfaces which allows us therefore to include multiple behaviors in one class. This makes up for C# on being able to inherit from multiple classes. Instead, a class can implement multiple interfaces. It shouldn't come as a surprise that creating an interface is done using the interface keyword. So where we normally write class, we will now write interface. As said, interfaces are contracts and contain method definition, but no implementations. That's the main difference between interfaces and abstract classes. The latter can and will still contain implementations. Now, I do need to make a side note here. Since C# 8, interfaces can also contain method implementations, Now while this works, this is a rarely-used feature, so we won't use it here either. We'll stick to just method definitions which together define the contract that needs to be implemented by another class that implements the interface because that is really what an interface is, a contract that a class will provide an implementation for the members defined in the contract, so in the interface. Just like abstract classes, it is not possible to create an instance of an interface type. I think the reason for that is pretty clear. Indeed, it doesn't have any implementations, so how could we create an instance since no functionality exists apart from the placeholders. Interfaces are often used throughout the .NET based class library too. We'll look at quite a few commonly-used ones later. What we'll see is that their name typically will start with an I. Now while it's not required to do so, it's a convention that makes it clear that the type is an interface. Members in an interface are public by default. Here is our first interface. Say that you want to create a behavior that you want to be sure different classes in your application provide an implementation for. In other words, I want to create a contract, have a class, agrees to sign this contract. I know it'll provide an implementation. Here, the interface is called ISaveable, and it contains the method definition ConvertToStringForSaving. The interface keyword is used to create the interface, and the method definition is similar to what we saw with abstract classes, although we don't even need to include the abstract keyword here. New implementation is provided, just a semicolon. What the contract just states is that each class that influences the interface must provide an implementation for the ConvertToStringForSaving method. Implementing an interface is simple. Using a syntax that is the same as we used with inheritance, we can specify that the Order class here implements ISaveable, choosing colon and then ISaveable. Now it becomes required for this calls to include an implementation for everything that is included in the contract. We then know that this type has this method and it enables polymorphic behavior, but we'll see that later. The method implementations must, of course, match what it says in the contract, so the name and the signature need to be the same as in the interface. As said, creating an instance of an interface type won't fly. Since it's definitely an abstract type, you can't new up an ISaveable.

Demo: Creating an Interface

That is in the first demo of this module, create a simple interface and implement it already. We are going to kick off this first demo by creating an interface, a simple one that is. I will implement that interface on a few classes. As said, interfaces are contracts, and therefore, I suggest we create a folder Contracts under the Domain folder. In there, we're going to create our very own interface. What I want to work towards in this part is that it becomes possible to also save the data to a file. At this point, we can only read from a file. It wouldn't be very useful if we can't write to a file. And so what I want to do is create a contract that classes can sign saying that they will provide an implementation to be saveable. It's not really words, but it means that the class provides a way that it returns itself so that it can be saved to a file. So if you want classes to subscribe to a contract to be saveable, I'll start with the creation of the ISaveable interface, that'll be our contracts. So in the Contracts folder, I'll go ahead and I'll make a new class. Yes, that's still a class. Didn't you get an interface? True. There is in fact a template, but I always use this one, and I replace the word class here with interface. Note the name. I am following the convention used most often in C# and .NET, and that is to have the name start with an I. It is a saveable convention. Things will work fine if you don't use an I as the first letter, but it does help when looking at code to easily see what is an interface. Okay, so now in here, I'm going to define a method that classes must then implement that this classes that implement this interface. I want them to return a string representation that I can save. Here's the method, it's again just the method declaration and there's no body. Just like with abstract methods in abstract classes, there's a semicolon at the end. Okay. So that's the interface done. Now we can go ahead and start implementing this. Let's go to our already well-known class, the BoxedProduct, and I'll add that I want to implement the ISaveable interface. This class already extends the product class, but it is still possible to also implement an interface. We just include the comma and then the name of the interface ISaveable. We're again getting a red squiggly because we don't have an implementation for the contract yet. Indeed, we'll need to create an implementation for the ConvertToStringForSaving method. Let's do Alt+Enter, implement interface. Visual Studio has now generated this method, which means that the contract is satisfied. And yes, we'll get an error when this method would execute, so we'll add our own implementation. This code returns the BoxedProduct with all that's needed to be able to save this to align in the file. The hardcoded 1 that you see here is the type that we used already in the repository. I'll add the implementation to the other classes and show you the result. Okay. So I've now implemented the interface on the BulkProduct class, on the FreshProduct class, and on the regular Product class. All these classes now sign the contract that we know and will be able to use this guarantee when we look at using polymorphism in combination with interfaces later in this module.

Implementing and Using Interfaces

We now have the basics in our head around interfaces. Let's learn some more and look at some built-in interfaces. Our interface, so far, had just one method. Of course, real-world interfaces will typically contain more. Although empty interfaces also exist. In general, we can put inside of an interface methods, properties, events, and indexes. Here's a more elaborate version of the ISaveable interface. It now contains a method definition, a property, and an event. When a class implements an interface, it needs to provide an implementation for each of the members defined in the interface. I've already mentioned that, but it's important enough to come back to it, and we'll see that in the next demo again as well. A class can also implement multiple interfaces. This means then, quite logically, an implementation needs to be provided for all members of all interfaces the class implements. If naming collisions occur, we need to use explicit naming which I'll show you in just a second. A class can also combine inheritance with interface. It is possible that the cause inherits from another class, but still implements at the same time one or more interfaces. Interfaces themselves can also use inheritance. It is possible to create a hierarchy of interfaces. Classes that can implement an interface part of this hierarchy must provide an implementation for all members of the interface and its parents. Implementing multiple interfaces isn't rocket science. Using a comma-separated list, we can list out the different interfaces we want the class to implement. An implementation needs to be provided for all members of all interfaces, and we can also choose to implement interface explicitly as we see in this variation here. I'm now indicating which interface the method belongs to, so using the interface name then a dot and then the name of the method. This is not required unless the interfaces have the same method names. In that case, though using explicit naming will allow us to distinguish which method we're implementing. As mentioned already, since a few versions of C#, adding implementations is not possible in an interface too, which makes interfaces even more closely related to abstract classes. Now it is possible, but I've rarely seen this being used in real life, so I will not discuss this feature here.

Demo: Implementing Multiple Interfaces

Time to explore interfaces somewhat more. Let's look in this demo at how we can implement multiple interfaces. I have now added a second interface, ILoggable. It's, again, a simple interface which defines a Log method that accepts a single parameter. In C#, we don't have the ability to inherit for more than one class, but it is possible to implement multiple interfaces, or as we have already seen, inherit from a class and implement an interface, so that means that we can own, for example, the BoxedProduct, add that we want to implement the ILoggable interface too. There we go. Now our BoxedProduct implements both the ISaveable and ILoggable interface, and it's also still inheriting from the Product class. Again, this will be interesting when we look at the influence this has on the polymorphic behavior in combination with interfaces.

Exploring Built-in Interfaces

The .NET base class library comes with many interfaces baked in, some of these are used internally, but quite a view are available for us to use. By implementing a built-in interface on our own types, you subscribe to a certain contract and that then may open support for other behavior. Let me show you. Here are a few commonly-used interfaces. The IEnumerable is used to indicate that we can enumerate over items and enables the use of for each over a type. IEnumerable is also an interface which is very commonly used in combination with link. The IDisposable interface is used where we need to release in our class, typically, unmanaged resources such as a file. Also, when using a using block, IDisposable is important. The ICloneable allows us to provide an implementation for an cloning object, so a new instance with the same values as an already existing instance. The IComparable interface is another commonly-used one. It is used to provide a method for comparison, typically used when ordering or sorting instances. IList is typically used for collections where we want to access an object in the list based on the index. ICollection, on the other hand, defines methods to work with generic collections. Finally, ISerializable is used to provide a method used when serializing and deserializing an object. Serialization basically means converting an object into a string, and deserialization does the opposite. This is just a small list of interfaces that come with the .NET base class library. There are many many more, but by implementing these interfaces, so by providing an implementation for its members, our objects get extra functionality like being soldable or cloneable. Most of these interfaces are pretty simple. Here's for example, the ICloneable. This interface just defines a clone method. If we want our own objects to be cloneable, what we need to do is implementing this interface on our own type. Here's how we should do that. We implement the interface in the exact same way as we did with our own interface, so we need to provide an implementation for the clone method in this case. In there, we'll need to write code that creates a copy of the current instance. It's up to us to decide how we do that though. Microsoft simply provides this interface and by implementing it our objects know how to clone themselves and return that cloned version.

Demo: Implementing ICloneable

Let's return to our application and see how we can add support for ICloneable. In the running application, you may have already seen that there's an option that we haven't used yet, and that is this one here Clone product, and so I'll implement this using one of the built-in interfaces in .NET, the ICloneable interface. This time, I've done things slightly differently to be able to show you the difference. Here's the Product class again, and it is still an abstract class. I'm going to specify here, so on the base class that this influence the ICloneable interface. ICloneable is an interface which is part of .NET. If I use F12 to go to the definition of this interface, we can see that this indeed is part of .NET. It's again just like ours, a simple interface which just defines a method named clone and that will return an object. In other words, what it is used for is to allow us to define an implementation to create a copy of an object. In the system object, there's also the MemberwiseClone, but that just provides a basic implementation. Through this interface, we can specify our own version of the Clone method, and we decide how the cloning of the object at hand should happen. Of course, Visual Studio will say that we need to implement the interface, so let's do Alt+Enter and then the Clone method is generated. Now defining a base implementation doesn't really make sense since different subclasses have different properties. Hey, that sounds familiar, that smells like an abstract method, so although this is a method that we need to implement, we can define it here on the abstract base class as abstract and then all classes that inherit from products will again need to define their own implementation. If we now build things, we'll get errors since the inheriting types don't define an implementation yet. Let's head over to the BoxedProduct class first. If we let Visual Studio implement the abstract Product class, again, it'll add the Clone method. I will need to provide my own implementation. What this code will do is, as said, create a copy of the current object, so it will create a new BoxedProduct object using the constructor and passing it the values of the current object. We need to do the same thing for the other classes as well, so here is the implementation on BulkProduct, here we have it for FreshProduct, and here's the implementation for the regular product. There we go. Things are compiling again. So now we can go ahead and implement the clone functionality from the UI as well, so I'll uncomment this, and we'll add the implementation for the ShowCloneExistingProduct method. What I've added here is the ability to clone an existing product. We ask the user to enter the ID of the product they want to clone. If found, we're asked to enter the ID, and then we can rely on the Clone method. ICloneable was implemented on the Product class, that's what the contract requires. The base class defined this as an abstract method, therefore, we know that all inheriting classes have to provide an implementation. That's what I'm relying on here. We have the product, and we call the clone method onto it. Clone will return an object, so I'm casting it here using the as operator into a product, then that new product is inserted in the inventory, so the list of all products. Let's run the application and test the clone. Okay. So now I can choose to clone. On the tree here, and the ID I want to clone, so let's try to clone sugar. I'll say that the new ID is 9, and indeed, we now see that the list has been extended. A new copy has been created through clone, and it has been added to the list.

Interfaces and Polymorphism

In the final part of this module, I want to discuss interfaces and a link with polymorphism. Interfaces enable polymorphic behavior too. We have seen before that we simply can't instantiate an interface for obvious reasons. However, just like with base classes, we can use an interface reference to point to an object of a class that implements that very interface. On that reference, we can then invoke the methods defined on the interface, thus enabling polymorphic behavior. Of course, again, like with inheritance, only the members defined on the interface can be invoked through that reference. In this last line here, what I'm trying to do won't work assuming that the UseProduct method isn't defined on the ISaveable interface, but perhaps on the Product class itself, so really what interfaces enable is the is-a relation. If the product implements the ISaveable interface, a product is an ISaveable and can be addressed as such. Interestingly, if a class implements multiple interfaces, such as ILoggable and ISaveable, it is addressable using both an ILoggable and an ISaveable reference. Just like with interfaces, we could create an array of interface references and loop over that array invoking, in this case, a Save method on each element in the array. There's no base implementation, so for each element, the implementation on the specific type will be used.

Demo: Using Polymorphism with Interfaces

In the final demo, let's look at using polymorphism with interfaces. In the very final demo of this course, we're going to rely on the ISaveable interface to add the functionality to be able to save products to the file, so we can continue where we left off. We already know that interacting with the file is something we should do from the repository, so we'll start there with bringing in a new method

SaveToFile. What I'm doing here is the following. I'm going to basically create a string to write all products to a file. For that, I'm going to rely on the ConvertToStringForSaving method. Notice that we are passing in a list of saveables, ISaveables, I should say, not products this time. Indeed, I didn't implement the ISaveable interface on the base Product class, but on the inheriting classes. By adding and implementing an interface, I can add a functionality on different classes and treat them in the same way. In that sense, implementing an interface will result in the is-a relation again. So since we're going to get in a list of objects of types that all implement the ISaveable interface, we know for a fact that they have an implementation for the ConvertToStringForSaving method, that I'm going to rely on here to build up the string, so we are relying entirely again on the polymorphic behavior. We treat the different types as an ISaveable, and that's how we can invoke the method defined on the interface. That string is then written to the file here. Now we have this method on the repository, and we're going to use it from the Utilities class. Let's uncomment the SaveAllData, and let's add an implementation for the method. In the code here, I'm again going to use the product repository. Now, the SaveToFile method on the repository actually required a list of ISaveable objects. We are ready with our inventory, which is a list of products or inheriting types. Now since product doesn't implement the ISaveable interface, we'll need to do some extra work. What I'm going to do here is creating a new list of ISavables, so interface types, these are just references. I'm not instantiating ISaveable objects. Instead, this is going to contain references to objects that implement the ISaveable interface. Our inheriting classes like BoxedProduct do implement the ISaveable interface and are just addressable as ISaveable. In this for each loop, I'm going to loop over all objects inside the inventory, so list of products. I'm then going to ask C# to check if that object, so the real object on the heap, whatever the type implements the ISaveable interface, and for that, I'm using the is here, that means indeed that there can be several types as long as they implement the interface, we can work with them here. In the next line, since we know that the object is an ISaveable, we are going to cast it as such using the as keyword, that is going to perform the cast, and then the reference to the object which implements the ISaveable interface is added to the list. In the end, we end up with a list of ISaveable references. Again, these objects can be of different types, but as long as we know they implement the ISavable interface, we know that they have the method we need and that is enough. We pass that list to the SaveToFile method which will then save the file to disk containing the string representation of the different products. That file we can then use again in the Load method and restart the application again.

Course Closing

With this, we have reached the end of this module on interfaces. Through interfaces, we create abstraction too. We define a contract, but typically, no implementations. Interfaces enable polymorphic behavior, and we've also seen that .NET comes with quite a few built-in interfaces. I think we have some good news for Bethany. The application is ready and the inventory managers can start using it. Bethany was really excited with the application we have created. Through the application, she said, we can make sure we no longer run out of stock and can keep on delivering our pies. And with that, we have reached the end of this course. I want to congratulate you on finishing it. Thanks for watching, and bye for now.