How To…

Repetition material for Object-Oriented Programming with C#

By Per Laursen
24-08-2025

Introduction

This document contains several exercises intended as repetition material for *Object-Oriented Programming with C#*. The document does not contain detailed discussions of the topics as such, see the other chapters for such discussions.

The exercises fall into these categories:

- The **Person.XX** exercises: Focusing mainly on the basics of object-orientation, by starting from an empty class definition and slowly building up a single, moderately complex class.
- The **Player.XX** exercises: Also focusing on building up a single class definition, but at a slightly faster pace. Some of the later exercises will also illustrate class collaboration and control statements.
- The **Collection.XX** exercises: Focusing on using the collection classses **List** and **Dictionary** for managing values of simple types and of class types. This will also involve using control statements for processing collections.
- The **Inheritance.XX** exercises: Focusing on using various aspects of inheritance and interfaces, and seeing polymorphic behavior in practice.
- The **Generics.XX** exercises: Focusing on using Generics to avoid code duplication, and seeing the effect of type parameter constraints.

The exercises revolve around a few central classes like **Person** and **Player**, and will contain a lot of variants of these classes. In order to be able to keep the source code for all exercises in a single Visual Studio project – and in order to avoid using explicit namespaces – these variants are given names like **Person01**, **Person02**, etc. So, when the exercise text just refers to, say, a "person" class, you should perceive this as the "person" class as a concept, not a specific C# class. The specific classes change from exercise to exercise, as indicated in the exercise texts.

Exercises

Exercise	**Person.01**
Project	HowTo
Folder	Person/01
Purpose	Try to use an existing – but empty – class.
Description	An empty class definition of the class **Person01** is given, in the folder **Person/01**.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/01** folder – delete any content that might already be in the file. 2. In **Program.cs**, now define a variable of type **Person01**, and set it to refer to a new object of type **Person01**. Do all of this in a single line of code. 3. Make sure that the project can compile (right-click on the project, choose ***Build*** from the menu). If it can, your code will be free of syntax errors. **4. You are done** 😊
Hints	A variable definition will look like this: `Person01 aPerson;` Creating a new object (of type **Person01**) is done like this: `new Person01();`

Exercise	**Person.02**
Project	HowTo
Folder	Person/02
Purpose	Add a single property to a class.
Description	An empty class definition of the class **Person02** is given, in the folder **Person/02**.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/02** folder – delete any content that might already be in the file. 2. In **Person02.cs**, now add a single property to the class definition. a. The property should be named **Name**. b. The property should be of the type **string**. c. The property should have a **get**, and a **set**. d. The property should be **public** (what does "public" mean?) e. The property should be an **auto-property**, i.e. we do not need to add any instance fields to the class. 3. In **Program.cs**, now define a variable of type **Person02**, and set it to refer to a new object of type **Person02**. Do all of this in a single line of code. 4. In the next line (still in **Program.cs**), set the value of the **Name** property on the object you created in step 3. Set it to *"Peter"*. 5. In the next line (still in **Program.cs**), print out the value of the **Name** property on the object you just created (use the "." to access the property). Print it on the screen using the **Console.WriteLine** statement. 6. **You are done** 😊. Don't worry if there is a green curly line under **Name** in the **Person02** class definition, we will fix this later on.
Hints	A definition of a public **auto-property** looks like this: `public string Name { get; set; }` Accessing a property on an object looks like this (assuming we have defined a variable named **aPerson** which refers to a **Person02** object): `aPerson.Name = "Peter"; // set the value of the property` `string name = aPerson.Name; // get the value of the property`

Exercise	**Person.03**
Project	HowTo
Folder	Person/03
Purpose	Add a constructor to a class definition. Change an existing property to *read-only*.
Description	A class definition of the class **Person03** is given, in the folder **Person/03**. It corresponds to the completed definition of **Person02** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/03** folder – delete any content that might already be in the file. 2. We have now made the following decisions about how a Person class should work: a. When creating a new object, the value for the **Name** property should be set as part of the creation process. b. It should not be possible to change the value of **Name** after the object has been created. 3. These changes will require us to implement two changes to the definition of the **Person03** class: a. It must now have a constructor, that sets the value for the **Name** property. b. The **Name** property should now only have a **get**, i.e. not a **set**. 4. In **Person03.cs**, now add a constructor to the **Person03** class definition: a. A constructor must have exactly the same name as the class. b. A constructor must be **public**. c. A constructor is a method, so it must have **()** after its name, and it must have a code block, starting with **{** and ending with **}**. d. The constructor must be defined inside the class definition. e. Since the constructor should set the value of the **Name** property, add code inside the constructors code block to do that. Set the value of **Name** to "Peter". 5. In **Person03.cs**, now update the definition of the **Person03** property. a. Delete the **set**, (including the **;** just after it), so only the **get** remains. 6. In **Program.cs**, now define a variable of type **Person03**, and set it to refer to a new object of type **Person03**. Do all of this in a single line of code. 7. In the next line (still in **Program.cs**), try to set the value of the **Name** property on the object you created in step 3. Try to set it to *"Peter"*. Is it possible? It shouldn't be, since you have removed the **set** from **Name**. 8. In the next line (still in **Program.cs**), print out the value of the **Name** property on the object you just created (use the "." to access the property). Print it on the screen using the **Console.WriteLine** statement.

	9. **You are done** 😊. But does this feel like a good implementation of how to handle the **Name** property…? If no, why not…?
Hints	A constructor looks like this: `public Person03()` `{` `    // This is the code block of the constructor` `}` A *read-only* property – i.e., a property which only has a **get** – looks like this: `public string Name { get; }`

Exercise	**Person.04**
Project	HowTo
Folder	Person/04
Purpose	Add a parameter to the constructor of a class definition.
Description	A class definition of the class **Person04** is given, in the folder **Person/04**. It corresponds to the completed definition of **Person03** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e., not in the **Person/04** folder – delete any content that might already be in the file. 2. It is very inflexible that we can now only create objects which all have the same value for the **Name** property. We therefore decide to update the class definition: a. When creating a new object, a value for the **Name** property must now be supplied by the creator of the object. 3. These changes will require us to implement two changes to the definition of the **Person04** constructor: a. The constructor must now take a parameter, which is the initial value for the **Name** property. b. In the constructor code block, we must use the parameter to set the value of the **Name** property. 4. In **Person04.cs**, now update the constructor definition of the **Person04** class: a. The constructor should now take one parameter. This parameter is specified between the **(** and the **).** b. The type of the parameter must be **string**. c. The name of the parameter is up to you, but you could choose simply to call it **name**. d. In the constructor code block, use the parameter to set the value of the **Name** property. 5. In **Program.cs**, now define a variable of type **Person04**, and set it to refer to a new object of type **Person04**. Do all of this in a single line of code. Note that it is now not enough to simply write `new Person04()`. Why is that? Because of the change we just made, the creator of the object must now supply a specific name here! Do this, if you have not already done so. Remember that the name must be specified between the **(** and the **).** 6. In the next line (still in **Program.cs**), print out the value of the **Name** property on the object you just created (use the "." to access the property). Print it on the screen using the **Console.WriteLine** statement. 7. In the next couple of lines line (still in **Program.cs**), repeat the steps 5 and 6, so you create a new variable and object (still of type **Person04**), but this time supply a different name, when you create the **Person04** object.

	8. **You are done 😊.** Note that we have now created two **Person04** objects. These objects have the same type, but they have different states. The state of an object is defined as the set of values the properties have at a certain time. In this case, we only have one property, so the value of that property defines the entire state of the object.
Hints	A constructor with one parameter looks like this: `public Person04(string name)` `{` `    // use the parameter to initialize a property value here` `}` Creating a new object will then look like this: `new Person04("Peter"); // Supply a specific value here`

Exercise	**Person.05**
Project	HowTo
Folder	Person/05
Purpose	Add two additional properties to a class definition.
Description	A class definition of the class **Person05** is given, in the folder **Person/05**. It corresponds to the completed definition of **Person04** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/05** folder – delete any content that might already be in the file. 2. We now wish to add more content to the person class, specifically: a. A new property named **Height**, which represents the height of the person in meters. An initial value for height must be supplied at object creation, but it must also be possible to change its value later. b. A new property named **Weight**, which represents the weight of the person in kilograms. An initial value for weight must be supplied at object creation, but it must also be possible to change its value later. 3. These additions will require us to implement several changes to the **Person05** class definition: a. Two new properties **Height** and **Weight** must be added to the class definition. They will both have the type **double**, and must have both a **get** and a **set** (why do they need both…?) b. The constructor must now take two additional parameters, so the object creator can supply initial values for height and weight. c. In the constructor code block, we must use these new parameters to set the values for the **Height** and **Weight** properties. 4. In **Person05.cs**, now update the **Person05** class definition: a. Add two new properties **Height** and **Weight**, as defined in step 3a. b. Update the constructor parameter list – which is defined between the **(** and the **)** – to take two additional parameters. The parameters should both be of type **double**, and you can name them **height** and **weight**. Remember that parameters are separated by a ",". See the ***Hints*** section, if you are in doubt. c. Update the constructor code block, such that you use the two new parameters to initialize the value of the **Height** and **Weight** properties. 5. In **Program.cs**, now define a variable of type **Person05**, and set it to refer to a new object of type **Person05**. Do all of this in a single line of code. Note that you must now supply three values instead of just one.

	6. In the next three lines (still in **Program.cs**), print out the value of the **Name** property, then the value of the **Height** property, then the value of the **Weight** property. 7. In the next two lines (still in **Program.cs**), set the **Height** and **Weight** properties to some new values. If you cannot do this, then you have probably left out the **set** in the definition of the properties. 8. Repeat step 6, and confirm that the **Height** and **Weight** properties indeed have been updated to the new values. 9. **You are done 😊.** The state of a person is now defined by the values of three properties instead of just one.
Hints	A constructor with three parameter looks like this: `public Person05(string name, double height, double weight)` `{` `    // use the parameters to initialize property values here` `}` Creating a new object will then look like this: `new Person05("Peter", 1.8, 75); // Supply specific values here`

Exercise	**Person.06**
Project	HowTo
Folder	Person/06
Purpose	Add a calculated property to the class definition.
Description	A class definition of the class **Person06** is given, in the folder **Person/06**. It corresponds to the completed definition of **Person05** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/06** folder – delete any content that might already be in the file. 2. We now wish to add one more property to a person: the **BMI** (Body-Mass Index). The BMI is calculated as: ***BMI = weight / (height * height)***, with weight measured in kilograms and height measured in meters. It could be tempting to just add yet another property, with a **get/set** and so on. However, since we can calculate the **BMI** from the values of two existing properties, we decide to do this instead. 3. In **Person06.cs**, now update the **Person06** class definition, by adding a single new property **BMI** of type **double**, with only a **get**. However, write the **get** like this: `get { return Weight / (Height * Height); }`. Remember that the property as such should still look like `public double BMI { … }`, it is only the part inside the **{ }** we are replacing. 4. In **Program.cs**, now define a variable of type **Person06**, and set it to refer to a new object of type **Person06**. Do all of this in a single line of code. Note that this is done exactly as in the previous exercise. 5. In the next four lines (still in **Program.cs**), print out the value of the **Name** property, then the value of the **Height** property, then the value of the **Weight** property, then the value of the **BMI** property. 6. In the next two lines (still in **Program.cs**), set the **Height** and **Weight** properties to some new values. 7. Repeat step 6, and confirm that the **Height** and **Weight** properties indeed have been updated to the new values, and that the value of **BMI** has also changed according to the new values for **Height** and **Weight**. 8. **You are done 😊.** Why is this a better solution than just using a standard **BMI** property with a **get** and a **set**…?
Hints	The full calculated property looks like this: `public double BMI` `{` `  get { return Weight / (Height * Height); }` `}`

Exercise	**Person.07**
Project	HowTo
Folder	Person/07
Purpose	Add another calculated property to the class definition.
Description	A class definition of the class **Person07** is given, in the folder **Person/07**. It corresponds to the completed definition of **Person06** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/07** folder – delete any content that might already be in the file. 2. We now wish to be able to determine if a person is "underweight", defined as having a BMI of less than 18.5. 3. In **Person07.cs**, now update the **Person07** class definition, by adding a single new property **Underweight** of type **bool**, with only a **get**. However, write the **get** in this way: `get { return BMI < 18.5; }`. Remember that the property as such should still look like `public bool Underweight { … }`, it is only the part inside the **{ }** we are replacing. 4. In **Program.cs**, now define a variable of type **Person07**, and set it to refer to a new object of type **Person07**. Do all of this in a single line of code. Note that this is done exactly as in the previous exercise. 5. In the next five lines (still in **Program.cs**), print out the value of the **Name** property, then the value of the **Height** property, then the value of the **Weight** property, then the value of the **BMI** property, the value of the **Underweight** property. 6. In the next two lines (still in **Program.cs**), set the **Height** and **Weight** properties to some new values (preferably some values that will result in a different value for **Underweight**). 7. Repeat step 6, and confirm that the **Height** and **Weight** properties indeed have been updated to the new values, and that the value of **BMI** and **Underweight** have also changed according to the new values for **Height** and **Weight**. 8. **You are done 😊.**
Hints	None

Exercise	**Person.08**
Project	HowTo
Folder	Person/08
Purpose	Add a method to the class definition.
Description	A class definition of the class **Person08** is given, in the folder **Person/08**. It corresponds to the completed definition of **Person07** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/08** folder – delete any content that might already be in the file. 2. We now still wish to be able to determine if a person is "underweight", but we wish have a more flexible definition of being underweight. More specifically, it should be the client – i.e. that part of the code which uses a person object – which supplies the specific BMI value used for deciding if a person is underweight. 3. Consider if we could do this by adding one more property – say, **BMILimit** – to the person class? It's probably possible, but should such a limit value be part of the state of a person object…? It doesn't feel right. Instead, the client should provide this value, whenever the client wants to determine if a person is underweight. This can be done using a **method**. 4. In **Person08.cs**, now update the **Person08** class definition, by adding a method named **IsUnderweight**. The method should be implemented like this (if in doubt, see the ***Hints*** section): a. It must be **public.** b. It must have a return type of **bool**. c. It must take one parameter of type **double**, named **bmiLimit**. d. In the code block, it must compare the given **bmiLimit** value to the value of the **BMI** property. If **BMI** is smaller than **bmiLimit**, the method must return **true**, otherwise **false**. 5. In **Program.cs**, now define a variable of type **Person08**, and set it to refer to a new object of type **Person08**. Do all of this in a single line of code. Note that this is done exactly as in the previous exercise. 6. In next the line, call the new method **IsUnderweight**. Remember that you must supply a specific value, when you call the method. Also remember that you probably need a variable to hold the result of calling the method (if in doubt, see the ***Hints*** section). 7. Print out the value returned by the method, and verify that it has the expected value. If not, check your implementation of **IsUnderweight**. 8. Repeat steps 6 and 7, using some other values as arguments for **IsUnderweight**. 9. Compare the property **Underweight** to the method **IsUnderweight**. What are the similarities? What are the differences?

	10. **You are done 😊.**
Hints	The definition of a method taking one parameter looks like this: `public bool IsUnderweight(double bmiLimit)` `{` `    // Compare BMI and bmiLimit` `}` Calling a method looks like this: `bool result = aPerson.IsUnderweight(19.2);`

Exercise	**Person.09**
Project	HowTo
Folder	Person/09
Purpose	Add another method to the class definition.
Description	A class definition of the class **Person09** is given, in the folder **Person/09**. It corresponds to the completed definition of **Person08** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/09** folder – delete any content that might already be in the file. 2. We now want to be able to print out all data on a person in a simple way. An obvious solution could be simply to add a new method **PrintPersonData** to the class definition. 3. In **Person09.cs**, now update the **Person09** class definition, by adding a method named **PrintPersonData**. The method should be implemented like this (if in doubt, see the ***Hints*** section): a. It must be **public.** b. It must have a return type of **void** (what does "void" mean…?). c. It does not take any parameters. d. In the code block, it must print out the values of all five properties, using **Console.WriteLine**. 4. In **Program.cs**, now define a variable of type **Person09**, and set it to refer to a new object of type **Person09**. Do all of this in a single line of code. Note that this is done exactly as in the previous exercise. 5. In next the line, call the new method **PrintPersonData** (if in doubt, see the ***Hints*** section). If you did not see the expected result, check your implementation, fix the error, and try again. 6. **You are done 😊. Remember**: printing something on the screen is not the same as "returning a value to the caller". A method that returns a value to the caller will always: a. Have a return type that is not **void**. b. Have the keyword **return** at least once in the code block.
Hints	The definition of a method taking zero parameters and not returning any value looks like this: `public void PrintPersonData()` `{` `    // Print out property values, using Console.WriteLine.` `}`

Exercise	**Person.10**
Project	HowTo
Folder	Person/10
Purpose	Add a way to return the object state as a string to the class definition.
Description	A class definition of the class **Person10** is given, in the folder **Person/10**. It corresponds to the completed definition of **Person09** in the previous exercise.
Steps	1. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Person/10** folder – delete any content that might already be in the file. 2. We now want to be able to retrieve the object state of a person object in the form of a string. That is, the client of a person object can then "transform" the object into a string, and then perhaps print out that string. 3. In **Person10.cs**, now update the **Person10** class definition, by adding a method named **PersonDataAsString**. The method should be implemented like this (if in doubt, see the ***Hints*** section): a. It must be **public.** b. It must have a return type of **string.** c. It does not take any parameters. d. In the code block, it must construct a string that contains all of the values of the properties, and return it. The specific way to construct the string is up to you (if in doubt, see the ***Hints*** section). 4. In **Program.cs**, now define a variable of type **Person10**, and set it to refer to a new object of type **Person10**. Do all of this in a single line of code. 5. In next the line, call the new method **PersonDataAsString.** Use the returned value as an argument to **Console.WriteLine**, and see if you get what you expected (if in doubt, see the ***Hints*** section). 6. **You are done 😊.** But consider if we could have used a property instead of a method…? How would you define such a property…?
Hints	If the person class only contained the **Name** property, this would be enough: `public string PersonDataAsString()` `{` `    return $"{Name}";` `}` We can then use the method like this: `Console.WriteLine(aPerson.PersonDataAsString());`

Exercise	**Person.11**
Project	HowTo
Folder	Person/11
Purpose	Consider the difference between **PrintPersonData** and **PersonDataAsString**
Description	A class definition of the class **Person11** is given, in the folder **Person/11**. It corresponds to the completed definition of **Person10** in the previous exercise.
Steps	1. This is a very short exercise. The only thing to do is to consider the difference between the methods **PrintPersonData** and **PersonDataAsString**. They serve similar purposes. Why might one be a better approach than the other…?
Hints	What method ties the class most tightly to being used in a console application?

Exercise	**Player.01**
Project	HowTo
Folder	Player/01
Purpose	Implement the first parts of a class from scratch.
Description	An empty class definition of the class **Player01** is given, in the folder **Player/01**.
Steps	1. We will now try to implement a class from scratch. The class is a "player" class, intended to model a player in a role-playing application. 2. The initial requirements for the class are as follows: a. The class must contain a property called **Name**, of type **string**. b. The **Name** property should be initialized in the constructor. c. The initial value of **Name** should be provided as a parameter – of type **string** – to the constructor. d. The value of **Name** can always be retrieved from a player object, but cannot be changed after the object has been created. 3. Add these features to the **Player01** class. You will need to add two things to the class definition (if in doubt, you can go back and see how the first Person classes were implemented in the earlier exercises): a. A property called **Name**. b. A constructor. 4. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/01** folder – delete any content that might already be in the file. 5. In **Program.cs**, now define a variable of type **Player01**, and set it to refer to a new object of type **Player01**. Do all of this in a single line of code. 6. In the next line, print out the value of the **Name** property for the object you created (use the "." on the variable which refers to the object). 7. In the next line, try to set the value of the **Name** property on the object you created in step 5, to a different value. Is it possible? It shouldn't be, since **Name** should only contain the **get** (why not the **set**…?) 8. **You are done 😊.**
Hints	No hints here, but check some of the first Person classes if in doubt about any of the steps.

Exercise	**Player.02**
Project	HowTo
Folder	Player/02
Purpose	Add a property to the Player class
Description	A class definition of the class **Player02** is given, in the folder **Player/02**. It corresponds to the completed definition of **Player01** in the previous exercise.
Steps	1. We will now add a new property **LifePoints** to the player class. Life points model the health of a player; the more life points, the healthier the player is (more features relating to life points will be added in later exercises). 2. In order to implement this new property, you must add (or update) some elements of the **Player02** class definition a. Add a new property named **LifePoints**, of type **int**. It should at all times be possible to retrieve and change the value of the property. b. In the code block of the existing constructor, set the value of **LifePoints** to 100. c. Add a new constructor (remember that it is perfectly fine to define more than one constructor for a class), which takes a name and an initial life point value as parameters. Use the parameters to initialize **Name** and **LifePoints** in the constructor code block. 3. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/02** folder – delete any content that might already be in the file. 4. In **Program.cs**, now define a variable of type **Player02**, and set it to refer to a new object of type **Player02**. Do all of this in a single line of code. You are free to use either one of the constructors (in in doubt, see the **Hints** section). 5. In the next lines, experiment a bit with retrieving and setting the values of **Name** and **LifePoints**, and printing out their values. Can you change the value of **LifePoints** after the object has been created? If not, you might have forgotten to include a **set** for **LifePoints**. 6. **You are done 😊.**
Hints	You should now be able to create a player in both these ways: `Player02 playerA = new Player02("Anne");` `Player02 playerB = new Player02("Benny", 85);`

Exercise	**Player.03**
Project	HowTo
Folder	Player/03
Purpose	Add two methods to the Player class
Description	A class definition of the class **Player03** is given, in the folder **Player/03**. It corresponds to the completed definition of **Player02** in the previous exercise.
Steps	1. In a typical role-playing game, a player will – over time – lose and gain life points. The player may lose life points if being attacked, and may gain life points by e.g. being healed, taking medicine or perhaps just resting. In any case, we would now like to add two methods that can either increase or decrease the life points value. This may at first seem unnecessary, since we can just set the value of the **LifePoints** property to whatever we want… Still, there are good reasons to create such methods as well, as we will see later. For now, we will just add these two new methods. 2. We will therefore need to add/update the class definition as follows (if in doubt about defining a method, see the **Hints** section): a. Add a new method named **RaiseLifePoints**. The method should take one parameter named **points**, of type **int**. The method should not return anything. In the method code block, **LifePoints** should be increased with **points**. b. Add a new method named **LowerLifePoints**. The method should take one parameter named **points**, of type **int**. The method should not return anything. In the method code block, **LifePoints** should be decreased with **points**. a. Update the **LifePoints** property such that it is no longer possible for a client to directly change its value. By "directly", we mean that it should no longer be possible to update the value of **LifePoints** like this: `player.LifePoints = 55.` Note, however, that it must still be possible for the two new methods to update the value of **LifePoints**, so we cannot just remove the **set** completely… but we could change how "accessible" it is (This is a bit tricky. If in doubt, see the **Hints** section). 3. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/03** folder – delete any content that might already be in the file. 4. In **Program.cs**, now define a variable of type **Player03**, and set it to refer to a new object of type **Player03**. Do all of this in a single line of code. 5. In the next lines, experiment a bit with raising and lowering the **LifePoints** value by using both methods (if in doubt about calling a method, see the **Hints** section), and print out their values to confirm that the methods work as expected.

	6. Can you still change the value of **LifePoints** directly <u>after</u> the object has been created (as shown in Step 2c))? If so, you might have forgotten to change the **set** for **LifePoints** to be private. 7. **You are done 😊.**
Hints	A method which takes one parameter and does not return anything looks like this: `public void LowerLifePoints(int points)` `{` `    // Use ”points” for something...` `}` A property which allows updating of its value by the methods in the class, but <u>not</u> by a client (i.e. code that uses objects of this class), looks like this: `public int LifePoints { get; private set; }` Calling a method on an object looks like this: `player.RaiseLifePoints(25);`

Exercise	**Player.04**
Project	HowTo
Folder	Player/04
Purpose	Update methods with parameter validation
Description	A class definition of the class **Player04** is given, in the folder **Player/04**. It corresponds to the completed definition of **Player03** in the previous exercise.
Steps	1. The names of the two new methods (**RaiseLifePoints** and **LowerLifePoints**) should help to make their intentions clear. Still, what if you e.g. called **RaiseLifePoints** like this: `player.RaiseLifePoints(-20)`? That would be in contradiction with the intention. It should ideally not be possible to call these methods with negative values. That is unfortunately not possible, but we can at least make it so that nothing happens if the methods are called with negative values. 2. In order to add this feature, you now need to update the code blocks for both methods, such that they only change the value of **LifePoints** if **points** has a non-negative value. If **points** is negative, nothing should happen. If in doubt, see the **Hints** section. 3. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/04** folder – delete any content that might already be in the file. 4. In **Program.cs**, now define a variable of type **Player04**, and set it to refer to a new object of type **Player04**. Do all of this in a single line of code. 5. In the next lines, experiment a bit with raising and lowering the **LifePoints** value by using both methods, and print out their values to confirm that the methods work as expected, including that the values must not change, if the methods are called with negative values. 6. **You are done 😊.**
Hints	We should probably use an **if**-statement here, like this: `public void RaiseLifePoints(int points)` `{` `  if (// write the condition here)` `  {` `  // Whatever that needs to happen if` `  // the condition is true.` `  }` `}`

Exercise	**Player.05**
Project	HowTo
Folder	Player/05
Purpose	Add a calculated property to the class definition
Description	A class definition of the class **Player05** is given, in the folder **Player/05**. It corresponds to the completed definition of **Player04** in the previous exercise.
Steps	1. We now wish to add one more property **IsDead** to a player. The definition of being dead is as such simple: if your life points are zero or less, you are dead. In other words: the value of **IsDead** can be calculated from the value of **LifePoints**. 2. We therefore need to add a new property **IsDead** of type **bool** to the **Player05** class definition. However, write the **get** like this: `get { return LifePoints <= 0; }`. Remember that the property as such should still look like `public bool IsDead { … }`, it is only the part inside the **{}** we are replacing. 3. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/05** folder – delete any content that might already be in the file. 4. In **Program.cs**, now define a variable of type **Player05**, and set it to refer to a new object of type **Player05**. Do all of this in a single line of code. 5. In the next lines, experiment a bit with raising and lowering the **LifePoints** value, and also retrieve the value of **IsDead**. Print out their values to confirm that the relation between the properties is as defined in Step 1. 6. Why is this approach better than defining **IsDead** as a "standard" property with a **get** and a **set**…? 7. **You are done 😊.**
Hints	No hints here.

Exercise	**Player.06**
Project	HowTo
Folder	Player/06
Purpose	Add another method to the Player class
Description	A class definition of the class **Player06** is given, in the folder **Player/06**. It corresponds to the completed definition of **Player05** in the previous exercise.
Steps	1. Another typical feature of a role-playing game is the ability for a player to "deal damage" to another player. Dealing damage will thus lower the life points of the player being damaged. Until now, our player class can indeed "receive" damage being dealt, through the **LowerLifePoints** method, but a player cannot deal damage. 2. We will therefore now add a new method **DealDamage** to the player class. The intention is that a call of the method will return a number, which represents – measured in life points – the damage dealt. This number can then be used in a call of **LowerLifePoints** on another player object (we will not use the number for that purpose in this exercise). 3. Now implement the new method **DealDamage** like this: a. It has return type **int**. b. It does not take any parameters. c. It always returns the number 15. 4. In **Program.cs** – which is found in the "root" of the project, i.e., not in the **Player/06** folder – delete any content that might already be in the file. 5. In **Program.cs**, now define a variable of type **Player06**, and set it to refer to a new object of type **Player06**. Do all of this in a single line of code. 6. In the next lines, call the new method a couple of times on the player object, and print out the value returned by the method. 7. Not surprisingly, the method always returns 15… which is just as specified, but probably not particularly useful. We will improve this in the next exercise, but for now… 8. **You are done 😊.**
Hints	No hints here

Exercise	**Player.07**
Project	HowTo
Folder	Player/07
Purpose	Call one method from another method
Description	A class definition of the class **Player07** is given, in the folder **Player/07**. It corresponds to the completed definition of **Player06** in the previous exercise, plus a bit of extra code.
Steps	1. It is not very useful that the method **DealDamage** always return 15. There will often be an element of randomness in deciding the damage dealt by a player, and we will now introduce as such element, by changing the way **DealDamage** works. 2. Look in the **Player07** class definition. You will see that it now contains a new method named **GetRandomNumber**. Don't worry about how the method is implemented, only worry about how you call the method. 3. Now update the code block of **DealDamage**, such that it uses the method **GetRandomNumber** to return a random integer number between 10 and 50. Remember that since this is just a method that calls another method in the same class definition, we call the method just by writing its name and provide specific values as arguments (if in doubt, see the **Hints** section) 4. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/07** folder – delete any content that might already be in the file. 5. In **Program.cs**, now define a variable of type **Player07**, and set it to refer to a new object of type **Player07**. Do all of this in a single line of code. 6. In the next lines, call the updated **DealDamage** a few (at least five) times on the player object, and print out the values returned by the method. Now the method – hopefully – returns a random number in the specified interval. 7. **You are done 😊.**
Hints	The method **GetRandomNumber** should be called like this: `GetRandomNumber(10,50);`

Exercise	**Player.08**
Project	HowTo
Folder	Player/08
Purpose	Using a **for**-loop
Description	A class definition of the class **Player08** is given, in the folder **Player/08**. It corresponds to the completed definition of **Player07** in the previous exercise. **NB**: this exercise does not involve any updates to the player class!
Steps	1. The values returned by **DealDamage** should be in the interval 10 to 50, both values included. Also, the values should be evenly distributed in this interval. This should imply that the average damage dealt is (10 + 50)/2 = 30. How can we verify this? We can e.g. call **DealDamage** a lot of times – say, 1000 times – and find the average value of all the returned values. Not difficult… but how do we call **DealDamage** 1000 times? We do this with a **for**-loop. 2. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/08** folder – delete any content that might already be in the file. 3. In **Program.cs**, now define a variable of type **Player08**, and set it to refer to a new object of type **Player08**. Do all of this in a single line of code. 4. In the next lines, write a **for**-loop, which will iterate 1000 times. If in doubt about how to write this, see the **Hints** section. 5. Inside the **for**-loop code block, do a call of **DealDamage**, and print the returned value (just as in the previous exercise). Run the program… we probably see the result of 1000 calls, but it is not very useful for calculating an average, unless you want to manually write down all those values… 6. Let's do better: a. In a line of code just before the **for**-loop, define a variable named **sum** of type **int**, and initialize it to 0 (zero). b. Inside the **for**-loop code block, remove the call of **Console.WriteLine**, but keep the call of **DealDamage**. c. Also inside the **for**-loop code block, add the value returned by **DealDamage** to **sum** (if in doubt, see the **Hints** section). d. In a line of code just after the **for**-loop, print out the value of **sum**. 7. What value should we expect **sum** to have? If the average damage is 30, and we call **DealDamage** 1000 times, **sum** ought to be close to 30 x 1000 = 30000. Hopefully, you see that as well. Try to run the program a couple of times. You will probably see that the sum varies a little from run to run, but is always relatively close to 30000. You could also try to run the program with 1000000 iterations. 8. **You are done 😊.**

Hints	A **for**-loop which iterates 1000 times looks like this: `for (int i = 0; i < 1000; i++)` `{` `    // Do something` `}` Adding a value to a variable looks like this: `sum = sum + damage;` It can also be written like this: `sum += damage;`

Exercise	**Player.09**
Project	HowTo
Folder	Player/09
Purpose	See two objects collaborate
Description	A class definition of the class **Player09** is given, in the folder **Player/09**. It corresponds to the completed definition of **Player08** in the previous exercise, with some small modifications (we only have one constructor now). **NB**: In this exercise, we also use the class **Sword**, found in the **Utilities** folder.
Steps	1. In a role-playing game, the ability to deal damage is often dependent on several factors, one typical factor being the weapon – say, a sword or a club – a player is using. To model this in our example, we have now defined a **Sword** class. Start out by examining that class. Consider these questions: a. How many properties does the class contain? b. How many constructors does the class contain? c. How many methods does the class contain? How many of these methods can an external client of the class call? d. What does *static* mean? 2. We now want to model a scenario where a player uses a sword. In order to do this, we need to make these changes to the **Player09** class: a. Add a **private** instance field named **_sword** to the class. The type of the instance field should be **Sword**. If in doubt about how to define an instance field, see the **Hints** section. b. Initialize the instance field in the constructor. The instance field should refer to a new **Sword** object. This is just like initializing a variable, as you have now done several times in **Program.cs**. c. Change the implementation of **DealDamage**, such that it uses the **Sword** object – which **_sword** now refers to – for calculating the damage dealt. d. Remove the helper method **GetRandomNumber** and the static instance field **_random**, since they are not needed any more. 3. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/09** folder – delete any content that might already be in the file. 4. In **Program.cs**, now define a variable of type **Player09**, and set it to refer to a new object of type **Player09**. Do all of this in a single line of code. 5. In the next lines, call **DealDamage** a few times on the player object, and print out the values returned by the method (you can also use a **for**-loop like in the previous exercise, if you feel comfortable with it). You should see values between 5 and 25, since this is the damage interval for the

	sword, if you have constructed it using the default constructor (remember that **Sword** has <u>two</u> constructors). 6. In the **Player09** constructor, now change the initialization of **_sword**, to use the **Sword** constructor which takes two arguments instead (set them to 20 and 100). 7. Re-run the program, without any changes to **Program.cs**. You should now see that the damage is indeed between 20 and 100. 8. Consider this question: What sort of <u>relation</u> do the classes **Player** and **Sword** have? Is it a **composition** or an **aggregation**? 9. **You are done 😊.**
Hints	An instance field definition looks like this: `private Sword _sword;`

Exercise	**Player.10**
Project	HowTo
Folder	Player/10
Purpose	Change class relation from composition to aggregation.
Description	A class definition of the class **Player10** is given, in the folder **Player/10**. It corresponds to the completed definition of **Player09** in the previous exercise. **NB**: In this exercise, we also use the class **Sword**, found in the **Utilities** folder.
Steps	1. Letting a player use a sword is an improvement, but it is also somewhat inflexible that the specific choice of sword is "hard-coded" into the player class (it is the player class which creates a sword object). It would be more flexible if a specific sword was supplied to the player object when the player object itself is created. We will now update the **Player10** class definition to make this possible. 2. You must therefore make the following changes to the **Player10** class: a. Add a third parameter to the constructor. The new parameter should be named **sword**, and have the type **Sword**. b. In the constructor, use **sword** to initialize the **_sword** instance field, such that **Player10** does not create a **Sword** object itself. 3. In **Program.cs** – which is found in the "root" of the project, i.e. not in the **Player/10** folder – delete any content that might already be in the file. 4. In **Program.cs**, now define a variable of type **Sword**, and set it to refer to a new object of type **Sword**. Do all of this in a single line of code. 5. In the next line, now define a variable of type **Player10**, and set it to refer to a new object of type **Player10**. Do all of this in a single line of code. **NB:** now you need to supply the **Sword** object you just created as an argument to the **Player10** constructor! If in doubt, see the **Hints** section. 6. In the next lines, call **DealDamage** a few times on the player object, and print out the values returned by the method. Check that they are consistent with the sword object you created. 7. In **Program.cs**, now change the initialization of the **Sword** object by changing the min- and max-damage values. 8. Re-run the program, and check that the damage values have changed accordingly. 9. Finally consider why this solution is an improvement over the previous solution. Has the coupling between Player and Sword become more loose or more tight? 10. **You are done 😊.**
Hints	The object creation code in **Program.cs** will look something like this:

	`` Sword sword = new Sword(15, 80); Player10 player = new Player10("Per", 200, sword); ```

Exercise	**Player.11**
Project	HowTo
Folder	Player/11
Purpose	Implement a more complex logic, using control statements. Use an interface to achieve even looser coupling.
Description	A class definition of the class **Player11** is given, in the folder **Player/11**. It corresponds to the completed definition of **Player10** in the previous exercise. **NB**: In this exercise, we also use the class **Sword**, found in the **Utilities** folder. **NB**: We also use the incomplete class **DamageTester**, found in the **Utilities** folder. **NB**: We also use the interface **IPlayer**, found in the **Utilities** folder.
Steps	1. Testing if the damage dealt by a player object is as expected is a bit tedious. We will therefore create a class which can run a test of a player object w.r.t. damage dealt. An incomplete version of such a class – named **DamageTester** – is found in the **Utilities** folder. 2. Your first job is to implement the **TestPlayerDamage** method in the **DamageTester** class, according to the specification written in the class definition. This is a fairly complex task… one approach could be first to think about how you would do this manually. How would you keep track of the various values, etc.? 3. Next, you should – in **Program.cs** – create the objects needed to run a test. You will need to create a **Sword** object, a **Player11** object, and a **DamageTester** object. Once these objects are created, you should be able to call **TestPlayerDamage** on the **DamageTester** object. 4. Experiment a bit with testing, by varying the min/max-damage for the sword, and the number of runs. 5. **DamageTester** is a very useful class… but in this form, in only works for testing **Player11** objects (try creating a **Player10** object, and try to call **TestPlayerDamage** with that object. It won't work…). We can improve this by using the **IPlayer** interface (what is an "interface"…?)! 6. Update **TestPlayerDamage** such that the first parameter has the type **IPlayer**. Also, let **Player11** implement that interface (if in doubt, see the **Hints** section). Re-run the program; the code you wrote in **Program.cs** should still work, but using a **Player10** object still doesn't work. 7. Now update **Player10** such that it also implements **IPlayer**. See if you can now test a **Player10** object as well. 8. See how far back you can go in the set of **Player…** classes, w.r.t. letting them implement **IPlayer**. It should be all classes from **Player06** and beyond. Now you should be able to test player objects of all these types!

	9. If you feel really adventurous: maybe we can also use the interface concept w.r.t. weapons, so a player is not restricted to having a sword as a weapon…? Try to define such an interface, change the player class accordingly, and create some alternative weapon classes. This is a difficult and large task, but see how far you can push it. 10. **You are done 😊.**
Hints	If a class must implement an interface, it will look like this: `public class Player11 : IPlayer`

Exercise	**Collection.01**
Project	HowTo
Folder	No folder, we only write code in **Program.cs** in this exercise.
Purpose	Create a **List** object, insert and retrieve elements.
Description	In **Program.cs**, we will work a bit with a **List** object. Remember that the **List** class is a class that enables us to work with a collection of values. These values may have a simple type (**int**, **string**, etc.) or a class type (or even an interface type). The type of values we can hold in a specific **List** is specified as a type parameter to the **List** definition, like e.g. **List<int>** for a **List** which can hold **int** values, **List<Person>** for a **List** which can hold references to **Person** objects, etc..
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **myList** of type **List<int>** and set it to refer to a new object of type **List<int>**. Do all of this in a single line of code. If in doubt, see the **Hints** section. 3. Now insert three values (say, 17, 42 and 30) into the list, using the **Add** method. Remember that you must call the **Add** method on the variable referring to the **List<int>** object, by using the ".". This is just as in the previous exercises. 4. Now retrieve the values, using the []-syntax, and print them on the screen. If in doubt, see the **Hints** section. The value we specify inside the [] is the index of the value we wish to retrieve. 5. What happens if you try to retrieve a value with index 3? Why do you suppose this happens? Hint: what is the index of the first value in a **List**? 6. **You are done 😊.**
Hints	A new **List<int>** object is created like this: `List<int> myList = new List<int>();` A value is retrieved from the **List** like this: `int value = myList[2];`

Exercise	**Collection.02**
Project	HowTo
Folder	No folder, we only write code in **Program.cs** in this exercise.
Purpose	Create a **List** object, insert and retrieve elements. Use a loop-statement.
Description	In **Program.cs**, we will work a bit with a **List** object. Since a **List** will often contain several values, it is very convenient to use a **loop-statement** (e.g., a **for**-loop or **foreach**-loop) to do something with each value, like e.g. printing it on the screen.
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **myList** of type **List<int>** and set it to refer to a new object of type **List<int>**. Do all of this in a single line of code. 3. Now insert three values (say, 17, 42 and 30) into the list, using the **Add** method. This is done exactly like in the previous exercise. 4. Now retrieve the values, using the []-syntax, and print them on the screen. However, this time do it with a **for**-loop. If in doubt, see the **Hints** section. 5. Now try to insert a few additional values into the **List** (**NB**: Do this before the **for**-loop) and run your program. It should still work, even without changing anything in the **for**-loop. How does the **for**-loop know how many values the **List** contains? 6. **You are done 😊.**
Hints	A **for**-loop which iterates through a **List** can look like this (assuming the variable **myList** refers to the **List**): `for (int index = 0; index < myList.Count; index++)` `{` `    Console.WriteLine(myList[index]);` `}`

Exercise	**Collection.03**
Project	HowTo
Folder	No folder, we only write code in **Program.cs** in this exercise.
Purpose	Create a **List** object, insert, retrieve and delete elements. Use a loop-statement.
Description	In **Program.cs**, we will work a bit with a **List** object. When a value is added to a **List** using the **Add** method, the value will be added to the end of the **List**. This means that all values already in the **List** will remain in the same place and keep their index. However, if we delete a value somewhere in the **List**, the index of a value may change!
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **myList** of type **List<int>** and set it to refer to a new object of type **List<int>**. Do all of this in a single line of code. 3. Now insert three values (say, 17, 42 and 30) into the list, using the **Add** method. This is done exactly like in the previous exercise. 4. Now retrieve the values, using the []-syntax, and print them on the screen. Do this with a **for**-loop, and the printing statement should print both the index and the value. If in doubt, see the **Hints** section. 5. Now try to remove a value from the **List**, by calling the method **RemoveAt** with index 1. If in doubt, see the **Hints** section (**NB**: Call **RemoveAt** after the **for**-loop). 6. After calling **RemoveAt**, print out the values again (you can simply *copy-paste* the entire **for**-loop). Notice how the index for the values after the value you removed have now changed! 7. **You are done 😊.**
Hints	A **for**-loop which iterates through a **List** – and prints the value and the index for each value – can look like this (assuming the variable **myList** refers to the **List**): `for (int index = 0; index < myList.Count; index++)` `{` `    Console.WriteLine($"Index {index} -> {myList[index]}");` `}` Removing a value with the **RemoveAt** method looks like this: `myList.RemoveAt(1);`

Exercise	**Collection.04**
Project	HowTo
Folder	No folder, we only write code in **Program.cs** in this exercise.
Purpose	Create a **List** object, insert and retrieve elements. Use a **foreach**-statement.
Description	In **Program.cs**, we will work a bit with a **List** object. The **foreach**-statement is specifically tailored to fit with collections, like **List**. The syntax is simpler, since you don't need to keep track of the index value.
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **myList** of type **List<int>** and set it to refer to a new object of type **List<int>**. Do all of this in a single line of code. 3. Now insert three values (say, 17, 42 and 30) into the list, using the **Add** method. This is done exactly like in the previous exercise. 4. Now retrieve the values (and print them), using a **foreach**-loop. If in doubt, see the **Hints** section. 5. If you haven't done this already, now write the **foreach**-loop exactly like in the **Hints** section, and consider these issues: a. Try changing **int** to **var**, and run the program. Does it make any difference? Can you remember what **var** means? b. Try changing **value** to **val**, but only in the first line. What is the consequence? Why did this break the program? c. Now also change **value** to **val** in the **Console.WriteLine** statement. Now your program should work again. Why is it important that we use the same variable name in both places? 6. **You are done 😊.**
Hints	A **foreach**-loop which iterates through a **List** and prints the values looks like this: `foreach (int value in myList)` `{` `    Console.WriteLine(value);` `}` The **int value** part in the first line is the definition of a **loop variable**, which will – as we iterate through the **List** – be set equal to the first value in the **List**, then the second value, then the third value, etc.

Exercise	**Collection.05**
Project	HowTo
Folder	No folder, we only write code in **Program.cs** in this exercise.
Purpose	Create a **List** object, insert and retrieve elements of a class type. Recall the purpose of the **ToString** method.
Description	A **List** object can contain values of simple types, but also of class types. **NB**: In this exercise, we also use the class **Company**, found in the **Utilities** folder.
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **companies** of type **List<Company>** and set it to refer to a new object of type **List<Company>**. Do all of this in a single line of code. 3. Now create three **Company** objects, and insert them into the list, using the **Add** method. 4. Now retrieve the values (and print them), using a **foreach**-loop. If in doubt, take the **foreach**-loop from the **Hints** section in the previous exercise, and adapt it to this case. **NB**: In the **foreach**-loop code block, just call **Console.WriteLine** with **Company** objects as-is, i.e. without explicitly referring to the individual properties. 5. Run your program. It should – hopefully – print out information about the companies. How come that the information is so detailed, even though we just call **Console.WriteLine** with **Company** objects (hint: what purpose does the **ToString** method in the **Company** class have?). 6. Remove the **ToString** method and run the program again… Put it back in, but now just remove the keyword **override** from the method definition. What happens then…? And why…? 7. **You are done 😊.**
Hints	None given.

Exercise	**Collection.06**
Project	HowTo
Folder	Collection/06
Purpose	Perform a search through a **List**. Return one value or **null**.
Description	A class definition of the class **CollectionMethods06** is given, in the folder **Collection/06**.
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **companies** of type **List<Company>** and set it to refer to a new object of type **List<Company>**. Do all of this in a single line of code. 3. Now create three **Company** objects, and insert them into the list, using the **Add** method. 4. Also create a variable **collMethods** of type **CollectionMethods06** and set it to refer to a new object of type **CollectionMethods06**. 5. Now open the **CollectionMethods06** class definition. Here you must implement the **FindCompanyFromCVR** method as specified in the comment. You will probably need a **foreach**-loop for this, and probably also an **if**-statement in the **foreach**-loop code block. 6. After implementing **FindCompanyFromCVR**, go back to **Program.cs**, and call **FindCompanyFromCVR** two times; once with a cvrNo that matches a company, and once with a cvrNo that doesn't. If in doubt, see the **Hints** section. 7. Print out the returned value for each call. Does it match what you expect? If not, check your code for **FindCompanyFromCVR**, but also check if you are using correct CVR numbers in each case (one that matches a company, and one that doesn't). 8. **You are done 😊.**
Hints	Calling **FindCompanyFromCVR** twice could look like this: `Company? company4518 = collMethods.FindCompanyFromCVR(4518, companies);` `Company? company4581 = collMethods.FindCompanyFromCVR(4581, companies);` **NB**: Your call will probably not look exactly like this, since you might have chosen different CVR numbers for your companies.

Exercise	**Collection.07**
Project	HowTo
Folder	Collection/07
Purpose	Perform a search through a **List**. Return a **List** of values.
Description	A class definition of the class **CollectionMethods07** is given, in the folder **Collection/07**. It corresponds to the completed definition of **CollectionMethods06** in the previous exercise.
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. In **Program.cs**, now define a variable **companies** of type **List<Company>** and set it to refer to a new object of type **List<Company>**. Do all of this in a single line of code. 3. Now create three **Company** objects, and insert them into the list, using the **Add** method. 4. Also create a variable **collMethods** of type **CollectionMethods07**, and set it to refer to a new object of type **CollectionMethods07**. 5. Now open the **CollectionMethods07** class definition. Here you must implement the **FindCompaniesByEmployees** method as specified in the comment. You will probably also need a **foreach**-loop for this, and probably also an **if**-statement in the **foreach**-loop code block. 6. After implementing **FindCompaniesByEmployees**, go back to **Program.cs**, and call **FindCompaniesByEmployees** a couple of times. Use a **foreach**-loop to print out the returned lists. If in doubt, see the **Hints** section. 7. Do the lists match what you expect? If not, check your code for **FindCompaniesByEmployees**. 8. **You are done ☺.**
Hints	Calling **FindCompaniesByEmployees** and printing the result could look like this: `List<Company> result = collMethods.FindCompaniesByEmployees(11, companies);` `foreach (Company company in result)` `{` `    Console.WriteLine(company);` `}`

Exercise	**Collection.08**
Project	HowTo
Folder	Collection/08
Purpose	Use a **Dictionary** to store keys and values. Reimplement methods from previous exercises.
Description	A class definition of the class **CollectionMethods08** is given, in the folder **Collection/08**. The final implementation of **CollectionMethods07** is also included for reference, but is commented out.
Steps	1. In **Program.cs**, delete any content that might already be in the file. 2. We will now use a **Dictionary** for storing **Company** objects. The main difference between a **List** and a **Dictionary** is that a **List** only contains values, while a **Dictionary** contains keys and corresponding values. A key will thus refer to one specific value, and all keys must be unique. In this case, the **CVRNo** is a good candidate for being a key for a **Company** object. 3. In **Program.cs**, now define a variable **companies** of type **Dictionary<int, Company>** and set it to refer to a new object of type **Dictionary<int, Company>**. Do all of this in a single line of code. If in doubt, see the **Hints** section. Why do we need to specify both **int** and **Company** as type parameters? 4. Now create three **Company** objects, and insert them into the dictionary, using the **Add** method. Note that **Add** now takes two arguments. If in doubt, see the **Hints** section. 5. Also create a variable **collMethods** of type **CollectionMethods08**, and set it to refer to a new object of type **CollectionMethods08**. 6. Now open the **CollectionMethods08** class definition. This class contains the two **Find…** methods we have implemented in the previous exercises, but now they take a **Dictionary** as the second parameter, so they must be re-implemented. 7. First implement the **FindCompanyFromCVR** method. Remember that you – given a key – can look up a value using the []-syntax (If in doubt, see the **Hints** section). 8. After implementing **FindCompanyFromCVR**, go back to **Program.cs** and call it a couple of times. Also try to call it with a CVR that does not match any company…. What happened…? And why…? 9. Looking up a value from its key needs to be "safe-guarded" by a call to **ContainsKey**. Update the **FindCompanyFromCVR** method to include this check (If in doubt, see the **Hints** section). 10. Go back to **Program.cs** and do some calls of **FindCompanyFromCVR** again. You should hopefully not experience any program crashes now. How does the final implementation of **FindCompanyFromCVR** compare to the **List**-based version, in terms of lines of code?

	11. Now implement the **FindCompaniesByEmployees** method. This is actually fairly easy; you can *copy-paste* the **foreach**-loop from the old **List**-based version of the method (available in the bottom of the class definition) and do a single change: change `in companies` to `in companies.Values`. Try it! 12. Go back to **Program.cs** and do some calls of **FindCompaniesByEmployees**. Hopefully it works as expected. What data do we obtain from a **Dictionary** in the **Values** property? There is also a **Keys** property, what data does that property contain? 13. **You are done 😊.**
Hints	Defining and initializing a **Dictionary** could look like this: `Dictionary<int, Company> companies = new Dictionary<int, Company>();` Adding keys and values to a **Dictionary** with the **Add** method could look like this: `Company c1 = new Company(2304, "Auto-Bots", 37);` `Company c2 = new Company(8912, "Beer Jam", 12);` `Company c3 = new Company(4518, "Carl's Woodshop", 2);` `companies.Add(c1.CVRNo, c1);` `companies.Add(c2.CVRNo, c2);` `companies.Add(c3.CVRNo, c3);` Given a key – say, as a variable **cvrNo** of type **int** – the corresponding value can then be looked up like this: `Company c = companies[cvrNo];` Safe-guarding with a call to **ContainsKey** can look like this: `Company? c = companies.ContainsKey(cvrNo) ? companies[cvrNo] : null;`

Exercise	**Collection.09**
Project	HowTo
Folder	Collection/09
Purpose	Use collection classes for a more complex task. Recall the **enum** type. **NB: This exercise is considerably harder than the previous exercises!**
Description	An incomplete class definition of the class **Purse** is given, in the folder **Collection/09**. The folder also contains a definition of the **enum** type **CoinType**.
Steps	1. Open **CoinType.cs**. It contains the definition of the type **CoinType**, which is an **enum** type. What is an **enum** type in general…? 2. Open **Purse.cs**. It contains the outline of an implementation of a purse. A purse can hold coins of various types, and various amounts of each coin type, just as a real-life purse. Notice the helper method **ValueOfCoinType**; this method is complete, and you need not change it. 3. Your job is now to implement the methods **AddCoins** and **GetNoOfCoins** and the property **ValueInKr** correctly. The comments in the code should explain the intent of the methods/properties. A key issue will be to store the coin amounts in a proper way, so give that some consideration first. 4. Once you have completed the implementation, make sure to test it from **Program.cs**, by adding some coin amounts to a **Purse** object and subsequently retrieve the purse value. 5. **You are done ☺.** 6. …or if you are really ambitious: Does the **Purse** class need to be tied to a specific coin type definition? Could we also turn the coin type definition into a parameter to the class…(Hint: Generics)?
Hints	None.

Exercise	**Inheritance.01**
Project	HowTo
Folder	Inheritance/01
Purpose	Work with a simple example of inheritance.
Description	A class **Character** is given in the folder **Inheritance/01**. The folder also contains an incomplete definition of the class **NPC**.
Steps	1. First, study the **Character** class. It is intended to be a simple model for a character in a role-playing game (see the comments for the individual properties). Take note of the constructor, which takes three parameters. 2. We now want to model a so-called **NPC** (Non-Player Character). The game designers define an **NPC** as "a Character which can talk, and when it talks, it just chooses a line (a "line" just being a string) randomly from a set of lines given to it when it is created". 3. Now study the (incomplete) class **NPC**. The instance field **_lines** should be used to store the lines given to the NPC when it is created (what method is called when an object is created?). Also, the method **Talk** chooses a line randomly (note that the method doesn't work yet). 4. Now finish the implementation of the **NPC** class by letting it inherit from the **Character** class (if in doubt, see the **Hints** section). It then becomes necessary to implement a constructor for **NPC** that calls the base class constructor (if in doubt, see the **Hints** section). Once this is in place, you can uncomment the **Talk** method. 5. In **Program.cs**, you can now create an **NPC** object (you will also need to create a **List** of strings to use as an argument in the call of the **NPC** constructor). Use a **for**-loop to call **Talk** a few times on the object, to see if you get random lines returned. 6. **You are done** 😊**.**
Hints	A class **B** inherits from a class **A** by defining it like this: `public class B : A { /* content of class */ }` Calling a base class constructor can look like this: `public NPC(string name, int lifePoints, int damageLimit, List<string> lines) : base(name, lifePoints, damageLimit) { _lines = lines; }`

Exercise	**Inheritance.02**
Project	HowTo
Folder	Inheritance/02
Purpose	Work with a simple example of inheritance. Override a method.
Description	A class **NPC02** is given in the folder **Inheritance/02**. It corresponds to the completed definition of **NPC** in the previous exercise. The folder also contains an incomplete definition of the class **PassiveNPC**.
Steps	1. We now wish to create a more specialized NPC called a "passive NPC". A "passive NPC" is an NPC which a. Only returns an actual line 50 % of the time it "talks" (i.e. the **Talk** method is called), and b. Always deals 0 (zero) damage (i.e. when the **DealDamage** method is called). 2. Implement the **PassiveNPC** class, according to this specification. This will involve three steps: a. Let **PassiveNPC** inherit from **NPC02** b. Implement a constructor for **PassiveNPC**, which calls the base class constructor. Does the **PassiveNPC** constructor need a **damageLimit** parameter, given the requirements? If not, what value should we then use when calling the base class constructor? c. Override the **Talk** method, to implement the "passive" behavior (your can probably make use of the **GetRandomNumber** method here…). Remember that you can call the base class implementation of **Talk** by using the syntax **base.Talk()**. If in doubt, see the **Hints** section. 3. In **Program.cs**, you can now create a **PassiveNPC** object (you will also need to create a **List** of strings to use as an argument in the call of the **PassiveNPC** constructor). Use a **for**-loop to call **Talk** and **DealDamage** a few times on the object, to see if it behaves as expected. 4. **You are done 😊.**
Hints	Overriding a method looks like this: `public override string Talk()` `{` `  // Your implementation goes here` `}`

Exercise	**Inheritance.03**
Project	HowTo
Folder	Inheritance/03
Purpose	See polymorphic behavior.
Description	The classes **Character03**, **NPC03** and **PassiveNPC03** are given in the folder **Inheritance/03**. **Character03** and **NPC03** are almost identical to the corresponding classes in the previous exercise, with the small – but important – change that **Character03** now contains an abstract method **Talk**. **PassiveNPC03** corresponds to the completed definition of **PassiveNPC** in the previous exercise. **NB**: You do not need to add anything to these classes in this exercise.
Steps	1. **Character03** now contains an abstract method **Talk**. What is an "abstract" method? What could be the reasons to define a method as "abstract"? Can we now create a **Character03** object (try to do so in **Program.cs**)? Can we define a variable of type **Character03**? And if so, can we then set this variable to refer to an object of type **NPC03**? What about an object of type **PassiveNPC03**? Why is this possible? 2. If you have not already done so, define a variable of type **Character03** in **Program.cs**, and let it refer to a new object of type **NPC03** (if in doubt, see the **Hints** section). Use a **for**-loop to call **Talk** and **DealDamage** a few times on the object, to see how it behaves. It should hopefully behave as an **NPC03** object. Now let the same variable refer to a **PassiveNPC03** object instead. How does the object behave now (hopefully as a **PassiveNPC03** object)? How is it possible that we see different behaviors, even though the variable has the base class type in both cases…? 3. **You are done 😊.**
Hints	Step 2 will look like this: `List<string> lines =` `  new List<string> { /* add some strings here*/ };` `Character03 c03 = new NPC03("Bo", 50, 15, lines);`

Exercise	**Inheritance.04**
Project	HowTo
Folder	Inheritance/04
Purpose	See polymorphic behavior. See an interface definition.
Description	The classes **Character04**, **NPC04** and **PassiveNPC04** are given in the folder **Inheritance/04**. They are all almost identical to the corresponding classes in the previous exercise, with the small – but important – change that **Character04** now implements the interface **ICharacter** (also given in the folder). **NB**: You do not need to add anything to these classes in this exercise.
Steps	1. **Character04** now implements the interface **ICharacter**. What is an "interface"? How is it related to "abstract methods"? Can we define a variable of type **ICharacter**? And if so, can we then set this variable to refer to an object of type **NPC04**? What about an object of type **PassiveNPC04**? Why is this possible? What about an object of type **NPC03**? Why is this not possible, even though **NPC03** and **NPC04** contain the same methods and properties? 2. Do the same steps as in Step 2 of the previous exercise, but this time with a variable of type **ICharacter**. 3. **You are done 😊.**
Hints	None

Exercise	**Inheritance.05**
Project	HowTo
Folder	Inheritance/05
Purpose	See polymorphic behavior. See decoupling through use of an interface.
Description	The folder contains the class **Zombie**, which implements the **ICharacter** interface, but does not inherit from **Character04**. The folder also contains the incomplete class **GameSetup**.
Steps	1. The game developers now decide that the game should contain **zombies**. All zombies have the below features: a. Are named "Nameless Zombie" b. When attacking, alternates between dealing 10 and 0 (zero) damage c. Have 0 (zero) life points d. Cannot have life points raised or lowered e. Are not considered to be dead. f. When talking, always say "Brraaaaiiinnnsss....." 2. A zombie is clearly very different from other characters, so the developers decide that the new **Zombie** class should not inherit from **Character04**. Is that the right decision? Could we still get away with using **Character04** as a base class, or is a zombie indeed too different? 3. Event though **Zombie** doesn't inherit from **Character04**, it should still be possible to integrate zombies into the game. This is achieved by letting **Zombie** implement the **ICharacter** interface. In **Program.cs**, try to declare a variable of type **ICharacter**, and let it refer to a **Zombie** object. Is it possible (it should be)? Do some of the same steps as in the previous exercises, to try out the zombie behavior. 4. The class **GameSetup** is not complete. In the **GameSetup** constructor, you should now add a number of character objects (zombies, npcs, etc.) to the list **_characters**. What is the single thing all these objects must have in common, in order to be able to be added to the list? 5. In the **AllTalk** method, use a **foreach**-loop to call the **Talk** method on all the objects in the **_characters** list. Would this be possible to achieve without the **ICharacter** interface? Does the **AllTalk** method know anything about the actual objects on which it calls **Talk**?. 6. **You are done ☺.**
Hints	None

Exercise	**Generics.01**
Project	HowTo
Folder	Generics/01
Purpose	Get motivated to use Generics to eliminate code duplication.
Description	In this and the subsequent exercises, we work with an imaginary data structure called a **ChainedCollection**. A **ChainedCollection** is very similar to a **LinkedList**. We imagine that a chained collection consists of "chain link" objects, which each hold a data value, and a reference to the next "chain link" object in the collection. You can thus traverse a chained collection by starting at the first link, and then follow the reference to the next link, etc., until you encounter a null reference, which means you have reached the end of the collection. The folder contains the classes **ChainLinkInt** and **ChainLinkString**.
Steps	1. Study the **ChainLinkInt** class. Make sure you understand how you can use **ChainLinkInt** objects to form a "chained collection" of **int** values. Note that the class contains two constructors (why?). 2. In **Program.cs**, create a couple of **ChainLinkInt** objects which will form a chained collection. If in doubt, see the **Hints** section. 3. Write some code that can traverse the chained collection, and print out the values contained in the collection. If in doubt, see the **Hints** section. 4. Repeat Steps 1 through 3 for the class **ChainLinkString**. 5. **You are done** 😊**.**
Hints	Three **ChainLinkInt** objects which form a chained collection could look like this: `ChainLinkInt linkC = new ChainLinkInt(12);` `ChainLinkInt linkB = new ChainLinkInt(7, linkC);` `ChainLinkInt linkA = new ChainLinkInt(29, linkB);` Code that can traverse this chain could look like this: `ChainLinkInt? link = linkA;` `while (link != null)` `{` `  Console.WriteLine(link.Value);` `  link = link.Next;` `}`

Exercise	**Generics.02**
Project	HowTo
Folder	Generics/02
Purpose	Get further motivated to use Generics to eliminate code duplication.
Description	The folder contains the incomplete class **ChainLinkDouble**.
Steps	1. Complete the implementation of the **ChainLinkDouble** class, such that we can use it for creating chained collections of **double** values. 2. In **Program.cs**, create a couple of **ChainLinkDouble** objects which will form a chained collection. 3. Write some code that can traverse the chained collection, and print out the values contained in the collection. 4. Compare the classes **ChainLinkInt**, **ChainLinkstring** and **ChainLinkDouble**. What makes them different? 5. Consider carefully if this is the life you want; creating endless copies of the same code over and over, just for being able to manage data of different types in exactly the same manner… 6. **You are done ☹.** 7. There might still be hope… proceed to the next exercise.
Hints	

Exercise	**Generics.03**
Project	HowTo
Folder	Generics/03
Purpose	Use Generics to replace several almost-identical classes with a single class.
Description	The folder contains the type-parameterized class **ChainLink<T>**.
Steps	1. Study the class **ChainLink**. What is the significance of the **<T>** added after the name of the class? Is there a class **T** somewhere in the code…? What types of data can a **ChainLink** object contain? 2. In **Program.cs**, use the **ChainLink** class to create chained collections of **int**, **strings**, **double** or perhaps even of **Zombie** objects. If in doubt, see the **Hints** section. 3. Write some code that can traverse the chained collections, and print out the values contained in the collections. 4. [Slightly tricky] Could we even write the *traverse-and-print* code from Step 3 as a single method that can be used with any kind of chained collection? If in doubt, see the **Hints** section. 5. Breathe a sigh of relief about not having to endure pointless code duplication even again… hopefully. 6. **You are done 😥.**
Hints	Using **ChainLink** for creating a chained collection of **int** values will look like this: `ChainLink<int> linkC = new ChainLink<int>(12);` `ChainLink<int> linkB = new ChainLink<int>(7, linkC);` `ChainLink<int> linkA = new ChainLink<int>(29, linkB);` A general-purpose method for printing the values in a chained collection could look like this (notice the type parameter **T**): `void PrintChainedCollection<T>(ChainLink<T> start)` `{` `  ChainLink<T>? link = start;` `  while (link != null)` `  {` `  Console.WriteLine(link.Value);` `  link = link.Next;` `  }` `}`

Exercise	**Generics.04**
Project	HowTo
Folder	Generics/04
Purpose	See a slightly atypical example of how to use generics
Description	The folder contains the interface **IHasId**, the (type-parameterized) base class **IdBase**, and the derived classes **Item** and **Person**. **NB**: You do not need to add anything to these classes in this exercise.
Steps	1. Study the interface and the classes. Make sure you fully understand their relationships. 2. The base class **IdBase** is type-parameterized. Why? What is the type parameter **T** actually used for? 3. In **Program.cs**, try to create a couple of **Item** and **Person** objects, and print them with **Console.WriteLine()**. Just use the objects as parameters to **Console.WriteLine()**. If **IdBase** didn't have a type parameter, how should we then implement **ToString** in the derived classes **Item** and **Person**? 4. **You are done 😊.**
Hints	None

Exercise	**Generics.05**
Project	HowTo
Folder	Generics/05
Purpose	Implement a collection class based on the chained collection concept.
Description	The folder contains the interface **IChainedCollection**, and the unfinished class **ChainedCollection**.
Steps	1. Study the **IChainedCollection** interface. This is a definition of a somewhat rudimentary collection concept. Note the type-parameterization, and also the type parameter constraint **where T : IHasId**. What is the consequence of such a constraint? Can you store **Person** objects in a class which implements this interface? **Item** objects? **Zombie** objects? **int** values? 2. Now study the class **ChainedCollection**. It is intended to implement the **IChainedCollection** interface, but it is incomplete (as indicated by all of the **// TODO** comments). Your job is to finish the implementation of the class, by implementing the incomplete public properties and methods. Note that two completed properties **Start** and **End** are available, and also a few private helper methods, which might be useful. **NB**: This is a rather large task, so focus on one thing at a time, and make sure to write some code in **Program.cs** to test your implementation. 3. **You are done 😊 (whew).**
Hints	You can find a completed version of **ChainedCollection** in the folder **Generics/06**, where it has been renamed to **ChainedCollectionDone**.

Exercise	**Generics.06**
Project	HowTo
Folder	Generics/06
Purpose	See what it takes to make the **ChainedCollection** class "compatible" with **foreach**-loops.
Description	The folder contains the class **ChainedCollectionDone**, which is the completed version of the class **ChainedCollection** from the previous exercise. It also contains the class **ChainedCollectionEx**. **NB**: The topic for this exercise is not part of the curriculum, it is just a little bit of "show and tell".
Steps	1. In order to test your implementation of **ChainedCollection** in the previous exercise, you might have retrieved all the objects by calling **GetAll**, and then used the returned list in a **foreach**-loop. What if we could simply use the **ChainedCollection** object itself in a **foreach**-loop, just like we use e.g. a **List** object? What does it take to make that possible? In this case, not that much. 2. Study the class **ChainedCollectionEx**. It "extends" the **ChainedCollection** class (here named **ChainedCollectionDone**) by also implementing the **IEnumerable<T>** interface (this is the interface that all **.NET** collection classes implement). The fact that **GetAll** returns a **List** of all the objects stored in the collection makes it very easy to implement the interface. Feel free to find more information about the **IEnumerable<T>** interface online. 3. [Quite tough, and requires use of the **yield return** langage feature, which you may need to look up online] Try to implement **GetEnumeratorInternal** without relying on the **GetAll** method. 4. **You are done 😊.**
Hints	None