EASJ Notes

C# Programming Exercises

(used in conjunction with Object-Oriented Programming With C#)

By Per Laursen
26-04-2017

Content

How to use this exercise set

This set of exercises is intended to be used in conjunction with the note ***Object-Oriented Programming with C#***. However, they are as such self-contained.

The formulation of each exercise follows a standard pattern:

- **Exercise**: Identifier for the exercise. The first part of the identifier is an acronymed reference to the corresponding chapter in the notes.
- **Project**: A C# project used in the exercise. The specific details of how the project is made available (.zip file, GitHub repository, etc.) may vary from course to course. The projects are self-contained.
- **Purpose**: What aspect of the learning process does this exercise concern.
- **Description**: The "setup" for the exercise, typically some sort of simplified domain-specific context.
- **Steps**: Specific steps in the exercise. The steps often become increasingly difficult. Some steps are marked in red. These steps are considered quite difficult.

To get around some technicalities with C# projects that are irrelevant for the beginner, all projects contain an extra C# file called **InsertCodeHere.cs**. In that file, an area is delimited by two comments

```
// The FIRST line of code should be BELOW this line

 (sandbox area)

// The LAST line of code should be ABOVE this line
```

This area is referred to as the "sandbox area" in several exercises. If you are required to put some code in the "sandbox area", this is the place.

Exercise	ProFun.0
Project	Sandbox
Purpose	Reality check – Visual Studio up and running
Description	The **Sandbox** project is as simple as it gets – we will just use it to verify that your installation of Visual Studio is up and running
Steps	1. Load, compile and run the project. 2. Verify that the message ***Hello world!*** Is printed on the screen.

Exercise	ProFun.1
Project	MovieManagerV05
Purpose	Discuss variables with regards to types and naming
Description	We imagine this project to be the very first steps in creating an application for movie management. The application could be used to keep track of relevant information for movies, e.g. a private collection of movies on DVD/Blu-ray (yes, some people still watch movies on physical media ☺).
Steps	1. Think about what specific information it could be relevant to store for each movie. 2. For each specific piece of information, think about how you can represent this information. Think about the nature of the information; is it text, numeric, or something else. 3. In the sandbox area in the project, define a variable for each piece of information. You should a. Choose a proper <u>type</u> for the variable b. Find a descriptive <u>name</u> for the variable 4. Once you are done, pair up with another student. 5. Switch computer with your partner 6. Review the work of your partner. For each variable in the partner's project, think about if a. The purpose of the variable is easy to understand b. The type seems properly chosen 7. Discuss your findings with your partner. 8. Was there any types of information that were particularly hard to find a good representation for?

Exercise	ProFun.2
Project	WebShopV05
Purpose	Get some practice in using arithmetic operators
Description	Part of the business logic in a web shop involves calculating the total cost of an order. The logic for calculating the total cost is as follows: 1. An item has a net price 2. You pay a 10 % tax on top of the net price 3. Shipping costs 49 kr., no matter the number of items 4. There is a credit card fee of 2 % on top of the entire cost, including tax and shipping.
Steps	1. Load and open the project – you will see that some variables for the net prices and number of items in order have already been included. Also, the order details are printed on the screen. 2. The variable **totalPrice** is supposed to contain the total price for the order. You must add the calculations needed to do this, given the logic in the description. 3. Test your solution by varying the number of books, DVDs and games in the order (you do this by assigning new values to the **noOf…** variables) 4. The web shop decides to offer a discount, based on the total number of items you buy: • If you buy at least 15 items, you get a 5 % discount • If you buy at least 30 items, you get a 10 % discount The discount is applied to the total price – update the code to include this discount.
Extra info	Some test examples you can use to verify your solution:

Books	DVDs	Games	Total price
8	3	2	711,96 kr.
0	12	4	1171,98 kr.
23	16	7	2507,16 kr.

Exercise	ProFun.3
Project	WebShopV06
Purpose	Get some practice in using logical operators
Description	Another part of the business logic in a web shop involves deciding if a customer qualifies for certain special offers, based on the order. The shop has four special offers. The logic for qualifying for each offer is: 1. The net total price (no taxes, etc.) is more than 1.000 kr. 2. You have ordered more books than games 3. You have ordered at least 10 items of one kind 4. You have ordered between 10 and 20 DVDs, or at least 5 games
Steps	1. Load and open the project – again, some variables are already present. Note the boolean variables **receiveSpecialOffer…** 2. For each of these variables, you must specify a logical expression, corresponding to the logic given in the description. 3. Test your solution by varying the number of books, DVDs and games in your order. 4. The web shop decides to offer an extra special offer. You qualify for the extra offer, if you qualify for exactly two of the previous offers. Update your code to include this extra offer.
Extra info	Some test examples you can use to verify your solution (SO#1 means "special offer 1", and so on):

Books	DVDs	Games	SO#1	SO#2	SO#3	SO#4
8	3	2	false	true	false	false
0	12	4	false	false	true	true
23	16	7	true	true	true	true
3	5	4	false	false	false	false

Exercise	OOPFun.1
Project	MovieManagerV10
Purpose	Observe how to use an existing class. Implement simple use of an existing class.
Description	In this version of the movie manager, a class called **Movie** has been added (in the file **Movie.cs**). It contains an absolute minimum of information about a specific movie. The class is put to use in the sandbox area, where some **Movie** objects are created and used.
Steps	1. Load the project, and go directly to the sandbox area. You will see that some code is already present. See if you can figure out what goes on in each line of code. If you hover the mouse cursor over a specific element, you should see some useful information pop up. Make sure you understand where • Objects are created • Parameters to the constructors are specified • Properties are used • Methods are called • Return values are used 2. Feel free to create additional **Movie** objects, and exercise them a bit (call methods, use properties, etc.)

Exercise	OOPFun.2
Project	BankV05
Purpose	Implement minor additions to existing class.
Description	The project contains a minimal **BankAccount** class. The class is put to use in the sandbox area, where a **BankAccount** objects is created and used.
Steps	1. Load the project, and take a look at the **BankAccount** class. Make sure you understand the elements it contains. Then take a look at how the class is used in the sandbox area. 2. We now want to add an extra property to the **BankAccount** class: the name of the account holder. Add this feature to the class. This will probably involve: a. Adding an instance field b. Adding a property c. Updating the constructor 3. Once the class has been updated, make sure to test the new feature by updating the code in the sandbox area.

Exercise	OOPFun.3
Project	RolePlayV10
Purpose	Implement significant additions to an existing class.
Description	The project contains a **Warrior** class, which is extremely simple – it only contains a name property. We need to extend the class with a few additional features.
Steps	1. Start out by taking a look at the **Warrior** class. Make sure you understand the elements it contains. Then take a look at how the class is used in the sandbox area. 2. We must now extend the class with a “level” feature. Details of this feature are: a. All warriors start at level 1. b. The level can be retrieved freely, but not changed freely. c. It must be possible to increase the level by one. 3. Implement this feature in the **Warrior** class. You will need to consider if a. An extra instance field is needed b. An additional property is needed (if so, do we need both the **get** and the **set**?) c. The constructor needs to be updated. d. A method for increasing the level is needed. 4. We must now extend the class with a “hit points” feature. Details of this feature are: a. Hit points are set individually when a warrior is created b. Hit points can be retrieved freely, but not changed freely. c. It must be possible to decrease hit points by a specified amount. 5. Implement this feature in the **Warrior** class, going through the same considerations as for the level feature 6. Implement a property called **Dead**, which returns a boolean value. The property should return true if hit points are below zero.

Exercise	OOPFun.4
Project	ClockV10
Purpose	Implement a class from scratch, including use of the class
Description	This project contains an empty class definition **Clock**. Your job is to implement the class, given the below requirements: 1. The clock should keep track of hours and minutes. 2. The clock should use the 24-hour system. 3. It must be possible to set the clock to a specific time. 4. It must be possible to retrieve the current time from the clock. 5. It must be possible to advance the clock by a single minute.
Steps	1. Implement requirements 1-4. This will involve figuring out what instance fields, constructor, properties and methods you need for this. Remember to include code for testing the class. 2. Implement requirement 5. In this case, it becomes quite important to choose relevant test cases.

Exercise	OOPFun.5
Project	DiceGame
Purpose	Work with a project containing collaborating classes
Description	This project contains two classes: **Die** and **DiceCup**. The **Die** class represents a 6-sided die, and is completed. The **DiceCup** class needs a bit of work to be complete. The **DiceCup** class uses the **Die** class.
Steps	1. Take a look at the **Die** class. It is complete, and fairly simple. Note that we use another class in the **Die** class, called **Random**. This class is from the .NET class library. 2. Open the **DiceCup** class. Note how the class contains two instance fields of type **Die**. Also note the constructor – what happens there? 3. The **DiceCup** class is not complete. Implement the **Shake** method and the **TotalValue** property, as specified in the comments in the code. Test that your code works as expected, by creating and using a **DiceCup** object in the sandbox area. 4. How much would we need to change in order to have a dice cup with three dice? 5. When we create a **DiceCup** object, we would also like to be able to specify the number of sides the dice should have. Implement the necessary changes in **Die** and **DiceCup** needed to enable this feature.

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

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

Exercise	ProNex.2
Project	WTF
Purpose	Use **if-else**-statements. Use method calls creatively.
Description	This project contains a class called **MysticNumbers**, with a single method **ThreeNumbers**. All that is known about **ThreeNumbers** is that it takes three integers as input, and returns one of them
Steps	1. By reading the code for **ThreeNumbers**, try to figure out what it does. Write some test code to see if you are right. 2. Write and test a new method **TwoNumbers**, that does the same thing as **ThreeNumbers**, but now only for two numbers. 3. Write and test a new method **FourNumbers**, that does the same thing as **ThreeNumbers**, but now for four numbers (tip – you can probably use the method **TwoNumbers** to make the code fairly short and easy). 4. Rewrite **ThreeNumbers** to use the **TwoNumbers** method. What code do you like best – the original code or the new code?

Exercise	ProNex.3
Project	WeatherStationV10
Purpose	Use multi-**if-else**-statements.
Description	This project contains a class called **Barometer**, containing two properties **Pressure** and **WeatherDescription**. The latter property gives an old-fashioned description of the weather, as a function of the pressure, according to this table:
Steps	1. Implement the property **WeatherDescription**, according to the table in the description. 2. Test your code.

Pressure	**WeatherDescription**
Below 980	Stormy
980-1000	Rainy
1000-1020	Changing
1020-1040	Fair
Above 1040	Very dry

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

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

Exercise	ProNex.6
Project	RolePlayV20
Purpose	Get further experience with **while**-loops. Work with a project involving several classes.
Description	The project is supposed to model a very simple game, where a hero can battle against a beast, until either beast or hero is dead! The project contains four classes, which are described in general terms here – see the code for more details: • The **NumberGenerator** class, with the method **Next**. This is a helper class for generating random numbers. • The **BattleLog** class, where individual strings can be “saved”, and later on printed out on the screen. • The **Hero** class, which models a game character. It is a very simple model, since it just has a number of hit points. • The **Beast** class, which also models a game character, in a way similar to the **Hero** class. Even though this is a very simple setup, it does include fundamental game mechanics from many popular role-playing games.
Steps	1. Study the classes in details, so you are sure of what they can do and how they work. Note how the **Hero** and **Beast** classes make use of the **NumberGenerator** and **BattleLog** classes. 2. See if you can figure out how to code a battle between a **Hero** and a **Beast** (until the death!). A bit of code is present in the sandbox area, but it obviously needs to be extended. 3. When you can make the two objects battle each other, there are a number of things to consider afterwards: a. It seems like the **Hero** wins most of the time (depending of course on how you coded the battle…). Why is that? How could we make the battle more fair? b. The damage dealt by the **Hero** is always between 10 to 30 points. How could we change that? Could we even let the creator of the **Hero** object decide this interval? Could this also be done for the number of initial hit points? c. Do we really need separate classes for **Hero** and **Beast**?

Exercise	ProNex.7
Project	DrawShapes
Purpose	Get some experience with **for**-loops
Description	This exercise is about trying to draw some simple shapes on the screen, using **for**-loops to get the job done. A very simple class **DrawingTool** is provided to help with this.
Steps	1. Study the class **DrawingTool**. As you can see, it is very simple. Why is the class (and the methods) static? 2. Using **for**-loops and the **DrawingTool** class, see if you can create code to draw the shapes A to E, as defined in the comments in the sandbox area. NOTE: The shapes get increasingly hard to draw…

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

Exercise	ProNex.8a
Project	RolePlayV21
Purpose	Get some experience with the **List** class.
Description	This exercise picks up where ProNex.6 let off. Now the Hero must face a greater challenge! (or maybe he's just farming..). The project RolePlayV21 is identical to the solution to ProNex.6. The Hero can do a battle against a Beast, and both classes now take several parameters in their constructor.
Steps	1. Change the code in InsertCodeHere.cs, to do a battle between a single Hero and an army of Beasts. (Hint: You will need a **List** and a couple of repetition statements). You might need to adjust the strength of Beasts, to give the Hero a chance…

Exercise	ProNex.9
Project	LibraryV10
Purpose	Use the **List** class. Implement linear search.
Description	This exercise illustrates the concept of a catalog. A catalog is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the catalog. The project contains the simple domain class **Book** (we consider the isbn number to be a "key" for **Book**, i.e. no two **Book** objects can have the same isbn number). Also, it contains the (incomplete) catalog class **BookCatalog**. The three public methods in **BookCatalog** allow the user to store and use **Book** objects in a simple way (see the comments in the code for more details about each method).
Steps	1. Study the test written in the sandbox area, and figure out what you expect the test to output. 2. Complete the three methods in the **BookCatalog** class. 3. Run the application, and see if the output of the test matches your expectations (if not, you will have to examine the test and your code once again…). 4. Is there anything in the code that prevents a user from adding two **Book** objects with the same isbn value? 5. How could you prevent that **Book** objects with the same isbn value are added to the catalog?

Exercise	ProNex.10
Project	LibraryV11
Purpose	Use the **Dictionary** class.
Description	***NOTE: This exercise is intentionally almost identical to ProNex.9*** This exercise illustrates the concept of a catalog. A catalog is a class that can store and use data of a certain type, without revealing the specific representation of data to the user of the catalog. The project contains the simple domain class **Book** (we consider the isbn number to be a "key" for **Book**, i.e. no two **Book** objects can have the same isbn number). Also, it contains the (incomplete) catalog class **BookCatalog**. The three public methods in **BookCatalog** allow the user to store and use **Book** objects in a simple way (see the comments in the code for more details about each method).
Steps	1. Study the test written in the sandbox area, and figure out what you expect the test to output. 2. Complete the three methods in the **BookCatalog** class. 3. Run the application, and see if the output of the test matches your expectations (if not, you will have to examine the test and your code once again…). 4. Is there anything in the code that prevents a user from adding two **Book** objects with the same isbn value? 5. How could you prevent that **Book** objects with the same isbn value are added to the catalog?

Exercise	ProNex.11
Project	SchoolAdministrationV10
Purpose	Use the **Dictionary** class. Work with an application containing several classes.
Description	The project contains the class **Student**. This is a simple representation of a student, with three instance fields; id, name and test scores. The first two are simple, but the “test scores” field is a **Dictionary**, holding key-value pairs of course names (string) and scores (int). The project also contains the class **StudentCatalog**. This class is supposed to be able to retrieve various information about the students; for this purpose, an instance field **_students** of type **Dictionary** is used to hold key-value pairs consisting of ids and **Student** objects (since a student is uniquely identified by an id)
Steps	1. The class **Student** is complete, and you need not change anything in it. However, take a good look at the **Student** class anyway, and make sure you understand how the methods work. Pay particular attention to the property **ScoreAverage**. 2. Look in the class definition of **StudentCatalog**. It contains five properties/methods (**Count**, **AddStudent**, **GetStudent**, **GetAverageForStudent**, **GetTotalAverage**) that are not completed. Add code to complete these methods, according to the specification given in the comments in the code. 3. Code that tests the **StudentCatalog** class has been added in the sandbox area,. Run the application, and check that the **Student-Catalog** class behaves as expected.

Exercise	ProNex.12
Project	Flinter
Purpose	Use enumerations
Description	**Flinter** is supposed to be the start of a new dating app. You can create profiles for those you are interested in meeting. In the project, the class **Profile** has been included. The class contains instance fields for gender, eye color, hair color, and height. You can thus create a **Profile** object by specifying values for each of these four fields in the class constructor. Furthermore, you can get a text description of a **Profile** object by using the property **GetDescription**.
Steps	1. Code that tests the **Profile** class is as always included in sandbox area. Examine the code in the **Profile** class definition, and see if you can predict the outcome of running the test. 2. Running the test reveals some problems. In two cases, we have specified hair color where we should have specified eye color, and vice versa (unless you really want a partner with white eyes and blue hair…), and in one case, we have specified a height category that doesn't exist. Change the **Profile** class definition by adding enumerated types for gender, eye color, hair color and height category. Use these new types for the four instance variables. The constructor needs some changes as well. Also consider if you still need the properties **GenderDescription** and **HeightDescription**. 3. Change the code in the sandbox area , so it is compatible with the redesigned **Profile** class. Observe how it is now only possible to specify legal values for each type. 4. Reflect a bit on the changes. Is there anything in the new code that is more complicated than it was in the original code? Was it always relevant to use an enumerated type?

Exercise	DRY.1
Project	CalculationSimulation
Purpose	Improve code structure by replacing values with constants, instance fields and parameters
Description	The project contains a simple simulation of a calculation. The intention is to simulate a calculation that takes about half a second. The calculation takes two values x and y, and returns an integer value. In order to speed up the calculation, a "cache" class is also provided. The idea is that once a calculation has been done, the result can be stored in the cache, from which it can be retrieved very quickly. See the code for further details.
Steps	1. The code is set up to do calculations in a 5x5 table (that is, x and y can be numbers between 0 and 4, both included). How many places in the project would you have to change something, if you want to do calculations in a 10x10 table instead? 2. Change the code, such that you get rid of all the instances of the number 5 in the methods. This could be done by using constants, instance fields and parameters. 3. It seems like -1 means "no value". Change the code, such that the value -1 does not occur in the methods. 4. Are there other values that are candidates for being replaced with constants or parameters? If so, make the necessary updates to the code.

Exercise	DRY.2
Project	WebShopV10
Purpose	Improve code structure by creating new methods
Description	Part of the business logic in a web shop involves calculating the total cost of an order. The logic for calculating the total cost is found in the code in the project. In the project, the **Order** class contains an item list. For simplicity, the item list just contains the net price for each item in the order. The class also contains a property **TotalOrderPrice** for calculating the total price for the order.
Steps	1. The implementation of the **TotalOrderPrice** property is less than optimal. Rewrite it, with the intent of: a. Removing duplicate code b. Making the method easier to understand

Exercise	OOPNex.1
Project	EmployeeV10
Purpose	See inheritance in action. Reorganise existing code to use inheritance. Call base class constructors.
Description	The project contains two existing classes **Teacher** and **ITSupporter**. They have quite a lot in common, so there is a lot of code duplication to get rid of.
Steps	Reorganise the code using inheritance 1. Create a new class **Employee**, that contains the common parts from **Teacher** and **ITSupporter**. 2. Let **Teacher** and **ITSupporter** inherit from **Employee**. The code in InsertCodeHere.cs should work as before. Remember that the derived classes will need to call the base class constructor.

Exercise	OOPNex.2
Project	RolePlayV23
Purpose	Override existing methods in derived class
Description	The project contains a working role-play system. Any character in the game is represented by an object of the class **Character**.
Steps	1. Get an overview of the application. The most interesting class is the **Character** class, which implements a generic game character. Also note the code in InsertCodeHere.cs, where two teams with two members are set up for battle. 2. Create a class **Defender**, which derives from **Character**. A **Defender** has a 50 % chance of having the received damage reduced by 40 %. This means that the **ReceiveDamage** method must be overrided. Once you have created the class, update the code in InsertCodeHere.cs to include a **Defender** on each team. 3. Create a class **Damager**, which derives from **Character**. A **Damager** has a 40 % chance of dealing double damage. This means that the **DealDamage** method must be overrided. Once you have created the class, update the code in InsertCodeHere.cs to include a **Damager** on each team. 4. Can we organise the code better, in order to e.g. make calls to the base class methods for dealing and receiving damage? (Maybe the calculation and the logging should be separated into separate methods).

Exercise	OOPNex.3
Project	SimpleGeometry
Purpose	Override abstract methods. See polymorphic behavior in action.
Description	The project contains the (abstract) class **Shape**, with an abstract property **Area**. The class also contains a static method **FindTotalArea**, that should calculate the total area of a list of shapes
Steps	1. Create two classes **Circle** and **Rectangle**. Both classes should inherit from **Shape**, and therefore implement the abstract property **Area**. You also need to figure out what instance fields, etc. the two classes need (if you need the value of π (pi), you can get it by writing **Math.PI**). 2. Implement the **FindTotalArea** method properly, such that it finds the total area of a list of shapes. 3. In the sandbox area, fill in some shapes in the given list, and see if your implementation works as expected

Exercise	OOPNex.4
Project	FilteringV10
Purpose	Use interfaces to generalise code
Description	The project contains a class **Filter**, with a **FilterValues** method. The method filters out values higher than 10 from a list of integers. The project also contains an interface ***IFilterCondition***.
Steps	1. Figure out how you can use the interface ***IFilterCondition*** to change the **FilterValues** method, into a method that can filter a list of integers according to any condition. That is, the condition itself has to somehow become a parameter to the method. Try out your solution with a couple of conditions. 2. Figure out how you can apply several filter conditions to a list in a single method call. 3. Filtering is a very generic operation. Maybe some of the .NET collection classes already support filtering…?

Exercise	OOPNex.5
Project	CarDealershipV05
Purpose	Override methods from **Object** class.
Description	The project contains a simple class **Car**, which contains a few proper-ties. In the sandbox area, we attempt to print out **Car** objects, and perform some comparisons between **Car** objects.
Steps	1. Run the program as-is, and observe the result. Can you figure out when the comparisons return **true**? 2. In the **Car** class, uncomment the **Equals** method (only that method), and run the program again. What has changed? 3. Uncomment the rest of the code in the **Car** class, and run the program again. What has changed? 4. The printing of **Car** objects is still not very satisfying. In the **Car** class, override the **ToString** method, so that it returns a string giving a reasonable description of the **Car** object. Run the program again, and see what difference it makes.

Exercise	OOPNex.6
Project	CompanyV10
Purpose	Use inheritance when creating several classes
Description	The project contains very little from the start. There is an **Employee** class with a **Name** property, and some abstract properties. Concerning salary calculation, only some very general rules exist: • Part of the salary is a fixed amount • Part of the salary is a bonus amount • The bonus amount is paid if a certain condition is met Specific definitions of the rules should be defined in classes that inherit from **Employee**
Steps	1. Create a **Worker** class. The class should inherit from **Employee**. For a worker, the below rules apply: • A worker works a fixed amount of hours per month • A worker is paid a fixed amount per hour • A worker does not receive any sort of bonus 2. Create a **Manager** class, also inheriting from **Employee**. The rules for salary calculation are more vague for a manager: • A manager has a fixed monthly base salary • A manager has a fixed monthly bonus The condition for when the bonus is paid out may vary, depending on the specific type of manager (NB: This implies that **Manager** also becomes an abstract class). 3. Create a **JuniorManager** class, that inherits from **Manager**. A junior manager will have the bonus paid out if (s)he has worked at least 180 hours during the month. 4. Create a **SeniorManager** class, that inherits from **Manager**. A senior manager will have the bonus paid out if (s)he has a performance evaluation of at least 6 during the month

Exercise	DBI.0
Project	ExamAdmV10
Purpose	Create data bindings between GUI controls
Description	The project contains a simple GUI for an exam administration system. In this version, you can just type in a name, a subject and a test score for an exam. The data is entered through two text boxes and a slider control.
Steps	1. Open the project, and open the MainPage.xaml file. Even though the file contains quite a bit of XAML, we only need to focus on the three named controls, with the names **student-Name**, **subject** and **score** (two **TextBox** controls and a **Slider** control). Make sure you can find these three controls in the XAML code. 2. We want to bind three **TextBlock** controls to the value of the three named controls. The three **TextBlock** controls are all part of the top line of the GUI, which consists of a total of six **TextBlock** controls. For each of the three relevant **TextBlock** controls, figure out which specific named control to bind to. 3. Now create the actual bindings, using the syntax described in the notes (for a **Slider**, you bind to the **Value** property; for a **TextBox**, you bind to the **Text** property). 4. Test that your bindings work as expected

Exercise	DBI.1
Project	ExamAdmV11
Purpose	Create data bindings between GUI controls and a domain object
Description	The project is identical to the project from ExamAdmV10, except that a class **Student** has been added. Right now, the constructor in **Student** just sets the properties to some fixed values.
Steps	1. Open the MainPage.xaml file, and add a data context to the **Page** control, specifying **Student** as the data context. See the notes for the syntax for adding a data context. 2. Bind the three relevant **TextBlock** controls (the same as in the previous exercise) to the corresponding properties on the **Student** class. Again, see the notes if you cannot remember the syntax for this. 3. Also create bindings for the three named controls, such that each control – or more precisely; the relevant property in each control – is bound to the corresponding **Student** property. 4. Run the application, and check that the bindings work as expected. Try to change the values as well. Are the changes reflected in the text line at the top?

Exercise	DBI.2
Project	ExamAdmV12
Purpose	Create two-way data bindings between GUI controls and a domain object
Description	The project starts off where the previous exercise left off. The project does contain data bindings, but changes in the values are still not reflected in the rest of the GUI.
Steps	1. Open and run the application. Confirm that changes in the values are not reflected in the top text line. 2. Open the **Student** class. All three properties now have a **set**-part as well. Now let **Student** inherit from the **INotifyProperty-Changed** interface, and implement the **OnPropertyChanged** method (if ReSharper if installed, Visual Studio can generate the code for you. If not, you can simply copy-paste the code from the notes). 3. Run the application again – are value changes now reflected in the text line? 4. For each property in **Student**, add a call to **OnPropertyChanged** to the **set**-part of the property, after the value has been set. 5. Run the application again – are value changes now reflected in the text line? 6. For each of the three bindings for the named controls (not the **TextBlock** control), update the binding mode to **TwoWay**. 7. Run the application again – are value changes now reflected in the text line? 8. Why don't we need to update the three **TextBlock** bindings to being **TwoWay**?

Exercise	DBI.3
Project	ExamAdmV13
Purpose	Create data bindings between collection-oriented GUI controls and domain object collections
Description	We now introduce a **StudentCollection** into the application. For now, it contains a **Student** list (with one entry), and a subjects list (initia-lised with five entries). The class also has a property **SelectedStudent**, that for now just returns the single entry in the **Student** list.
Steps	1. Open the **StudentCollection** class, and make sure you under-stand the instance fields and properties it contains (except the **NewSubject** property) 2. Open the MainPage.xaml file. The bindings are now a bit more complex, since the data context is now **StudentCollection**. Most properties are now bound to the corresponding property on the **SelectedStudent** property (i.e. **Student** object). Make sure you understand the new bindings. 3. Run the application (ignoring the two extra lines beneath "Score"). The application does work, since updates to **name**, **score** and **subject** are reflected in the top text line (try it!). 4. In the "New subject" line, the intention is that when a new subject is entered, it should show up in the Subject combo-box. Confirm that this is <u>not</u> the case right now (remember that you must leave the text box, before the update is triggered). 5. The "No. of subjects" field tells how many entries the **_subjects** list in **StudentCollection** contains. Right now, the number stays at 5. Figure out how to create a binding for the text box next to the "New subject" text, such that a new entry is indeed added to **_subjects** (Hint: Take a look at the **NewSubject** property in **StudentCollection**). 6. Once this binding works, the number should increase every time a new subject is added. Still, the new subjects do not show up in the combo-box. Figure out why this is the case, and fix it. (Hint: Are we using the correct collection class for **_subjects**?)

Exercise	DBI.4
Project	ExamAdmV14
Purpose	Create a data template for presenting objects in a **ListView**
Description	The application main view contains a **ListView** control, where the **ItemsSource** property is bound to the **Students** property on the **StudentCollection** class. The list contains five students. However, the presentation of the students in the list view is not optimal.
Steps	1. Try to improve the presentation of **Student** objects in the list view, by providing an implementation of **ToString** in the **Student** class. 2. Improve the presentation further by defining a data template for the **Student** class (Tip: you can probably use the properties in the **Student** class for this purpose).

Exercise	DBI.5
Project	ExamAdmV15
Purpose	Create a Master/Details view
Description	The application main view again contains a **ListView** control, where the **ItemsSource** property is bound to the **Students** property on the **StudentCollection** class. The list contains five students, and a reason-able data template has been provided for presentation. The **Student** class has however been extended with several additional properties.
Steps	1. Create the Details part of a Master/Details view (the **ListView** is the Master part), such that all details of a given **Student** object are shown in the Details part. The Details part should show the details of the **Student** which is currently selected in the list view. (Tip: use the example in the notes for inspiration). 2. Use the styles **TextBlockStyle** and **TextBoxStyle** to specify the appearance of the Details part.

Exercise	DBI.6
Project	ExamAdmV16
Purpose	Add deletion functionality to a Master/Details view
Description	The application contains a working Master/Details view for the **Student** class. We now wish to add functionality to delete a student
Steps	The steps needed to create deletion functionality are very similar to the steps described in the notes. Almost all changes are done in the **StudentCollection** class. It can be assumed that the **Name** property can be used as a key for **Student** objects. In the **StudentCollection** class: 1. Change the **_students** instance field, such that it uses **ObservableCollection** instead of **List** 2. Add a **DoDelete** method, similar to the **DoDelete** method in the notes (it should call the existing **Delete** method) 3. Add a **DoDeleteRelay** method, similar to the **DoDeleteRelay** method in the notes 4. Add a **StudentIsSelected** method, similar to the **CarIsSelected** method in the notes 5. Add a **_deleteCommand** instance field, of type **RelayCommand** 6. Initialise the **_deleteCommand** instance field in the constructor, using **DoDeleteRelay** and **StudentIsSelected** as parameters 7. Add a **DeletionCommand** property, similar to the **DeletionCommand** property in the notes 8. Update the **set** part of the **SelectedStudent** property, such that it calls **_deleteCommand.RaiseCanExecuteChanged**() In the **MainPage.xaml** file: 9. Add a **Delete** button in a proper place in the view, and bind its **Command** property to **DeletionCommand** 10. Check that you can now delete students from the view!

Exercise	DBI.7
Project	ExamAdmV17
Purpose	Consider how insertion/editing functionality can be added to the application
Description	The application contains a working Master/Details view for the **Student** class, with deletion functionality. A natural extension of the application could be to add functionality for editing existing students, and adding new students
Steps	1. Consider what it would take in order to add insertion and editing functionality to the view. Consider for instance: • How can we enable editing of specific fields? • Should all fields be editable? • Should editable fields be editable all the time? • How can we manage the "editability" of fields in the Details view? • What are the detailed steps needed in order to create a new student? • What sort of validation will be needed when creating a new student (remember we assume that student names are unique)? 2. If you are up to the challenge, feel free to start on the actual implementation of the functionality 3. Finally, consider if the **Student** and **StudentCollection** classes are appropriate classes for containing all this functionality. Can you envision a better distribution of the functionality?

Exercise	DBI.8
Project	ExamAdmV18
Purpose	Save the day at StudentSoft A/S (see memo below)
Description	**MEMO**: Finish the Show/Hide details feature in the Student view **From**: Maurice Fischer (StudentSoft A/S CTO) **TO**: EASJ Intern (can't remember the name…) Hi, Unfortunately, our main developer on the Exam Administration application died yesterday, due to an unfortunate incident involving a hamster, three small oranges and a large piece of brown cardboard. We would therefore like you to finish up the Show/Hide Details feature he was working on in the Students view. I think it was something about being able to toggle the visibility of parts of the Details view on and off, using a ToggleSwitch or something… Anyway, you can probably figure it out by looking in the C# project, as he said he was "almost done" with it, and he always puts…uhh, used to put comments in the code. I would like a demo of it later today, as we are shipping a new version of the application tomorrow. Regards, M. Fischer
Steps	Do as you're told…

Exercise	MVVM.0
Project	ExamAdmV20
Purpose	Change the given application from a Model-View (MV) architecture to a Model-View-ViewModel (MVVM) architecture.
Description	The project initially contains a class **Student**, which acts both as a domain class and a "provider" to the main view (via data bindings in MainPage.xaml).

Steps	Add a new class **StudentViewModel** to the project, which will acts a the ViewModel in an MVVM architecture. This involves:
	1. Create the class **StudentViewModel**.
	2. Add an instance field **_domainObject** of type **Student** to **StudentView-Model**, and initialise it to refer to a new **Student** object in the constructor.
	3. Let **StudentViewModel** inherit from **INotifyPropertyChanged**, and generate the code needed (Tip: click the lightbulb ☺). If the includes are not generated automatically, add to the top of the file:
	`using System.ComponentModel;`
	`using System.Runtime.CompilerServices;`
	4. Add properties **Name**, **Subject** and **Score** to **StudentViewModel**, in the style described in the notes.
	5. Clean up the **Student** class, such that it no longer inherits from **INotifyPropertyChanged**
	6. Change the data context in **MainPage.xaml**, and check that the new bindings work as expected.
	7. Now create a new property **TopLineText** in **StudentViewModel**, of type **string**. The intention is that this property should provide enough information to enable you to delete the six **TextBlocks** in the top line of the GUI, and replace them with a single **TextBlock**, that binds to **TopLineText**.
	8. Delete the six **TextBlocks**, replace them with a single **TextBlock**, and bind the new **TextBlock** to **TopLineText**. Are changes to the data reflected in the top text line?
	9. Add extra calls of **OnPropertyChanged** to the **Name**, **Subject** and **Score** properties, in the style described in the notes. Check that changes are now reflected in the top text line

Exercise	MVVM.1
Project	ExamAdmV21
Purpose	Add and use view model classes in an application
Description	The given application provides simple read-only functionality for a collection of students (only the Master part of a Master/Details view). For now, the class **StudentItemViewModel** is not used.
Steps	1. Add a new class **StudentMasterViewModel** to the project. The intention is that MainPage.xaml should use this class as its new data context. 2. Add two instance fields to the new class: • **_studentCollection** of type **StudentCollection** • **_studentItemViewModelCollection** of type **ObservableCollection<StudentItemViewModel>** 3. Initialise the two instance fields in the constructor, by setting them to refer to a new object of each type. 4. Still in the constructor, add **StudentItemViewModel** objects to **_studentItemViewModelCollection**, by looping through the list of **Student** objects in the collection (Hint: use the **Students** property in the **StudentCollection** class), and create a new **StudentItemViewModel** object for each **Student** object. 5. Add a property **StudentItemViewModelCollection** to **StudentMaster-ViewModel**. Only the **get**-part of the property is needed; it should just return **_studentItemViewModelCollection**. 6. Change the data context in MainPage.xaml to use **StudentMaster-ViewModel** instead of **StudentCollection**, and change the binding of the **ListView** property **ItemsSource** from **Students** to **StudentItemViewModelCollection**. 7. Rebuild the application, and check that the data is still shown properly when running the application. 8. Do these changes enable you to clean out any properties from **Student** and **StudentCollection**, that were only there to supply GUI-specific data?

Exercise	MVVM.2
Project	ExamAdmV22
Purpose	Add deletion functionality to a working read-only Master view.
Description	The application contains a functional read-only Master view, where students can be viewed. The application uses the MVVM architecture. We now want to add deletion functionality to the application
Steps	In the **StudentMasterDetailsViewModel** class: 1. Add a new property **DeletionCommand** (only the **get**-part is needed) of type **ICommand**, that returns **_deleteCommand**. 2. Implement a **CanDelete** method, that returns a **bool**. The method should return **true** when it is meaningful to execute the deletion functionality (Hint: A student should probably be selected in the view). 3. Implement a **DoDelete** method, that deletes the selected student, using the name of the student as key (Hint: Use the already implemented **Delete** method). 4. With **CanDelete** and **DoDelete** implemented, now make a proper initialisation of the **_deleteCommand** instance field (Hint: Use the **RelayCommand** class, using **DoDelete** and **CanDelete** as parameters). In MainPage.xaml 5. Add a ***Delete*** button just after the **ListView** control, and bind its **Command** property to **DeletionCommand** 6. Build and run the application. Does the ***Delete*** button work as it should (probably not…) Back in the **StudentMasterDetailsViewModel** class: 7. In the **set**-part of the **StudentItemViewModelSelected** property, add a call of **RaiseCanExecuteChanged** on the _**deleteCommand** instance field, just before the call of **OnPropertyChanged**. 8. Build and run the application again. Does the ***Delete*** button now work as it should (Hopefully it does ☺)

Exercise	MVVM.3
Project	ExamAdmV23
Purpose	Rewrite domain-specific classes to use provided base classes
Description	The project contains a working Master view with delete functionality. All domain-specific classes are located in the folder **DomainClasses**. A number of base classes are available in the folder **BaseClasses**, but are not used yet.

Steps	1. Let the **Student** class inherit from **DomainClassBase<string>**. This will require that you override the **get**-part of the property **Key**. The property should just return **_name** (name acts as key for a **Student** object).
	2. Let the **StudentModel** class inherit from **ModelBase<Student, string>**. You can then delete everything else from the **StudentModel** class except the constructor. In the constructor, change the calls of **_students.Add** to use the base class method **Add**.
	3. Let the **StudentItemViewModel** class inherit from **ItemViewModel-Base<Student>**. The constructor must then call the base class constructor with **obj** as parameter. Also delete the instance field **_domainObject**, and replace the use of **_domainObject** with **DomainObject** in the properties.
	4. Let the **StudentMasterViewModel** class inherit from **MasterViewModel-Base<Student, string>**. You can then delete the method **GetStudentItem-ViewModelCollection** from the class.
	5. Open the **StudentViewModelFactory.cs** file, and uncomment the class **StudentViewModelFactory**. (tip: select all of the code, and press Ctrl+K+U)
	6. Let the **StudentMasterDetailsViewModel** class inherit from **MasterDetailsViewModelBase<Student, string>**. Then delete everything (yes, everything) from the class…
	7. Implement the constructor for **StudentMasterDetailsViewModel** like this:
	`public StudentMasterDetailsViewModel()`
	`: base(new StudentViewModelFactory(), new StudentModel())`
	`{}`
	8. In **MainPage.xaml**, change the binding of **ItemsSource** to **ItemView-ModelCollection**, and the binding of **SelectedItem** to **ItemViewModel-Selected**
	9. Make sure all files are saved, then build and run the application

Exercise	Files.1
Project	NoteBookV10
Purpose	Add Load- and Save-functionality to an MVVM application
Description	The given application contains a very simple system for creating notes. A note consists of a title and some content. It is not allowed to have two notes with the same title. However, the application does not support saving and loading of notes yet.
Steps	In the **NoteMasterDetailsViewModel** class: 1. Add a new instance field **_loadCommand**, of type **RelayCommand** 2. Add a new property **LoadCommand**, of type **ICommand**. It should just return the instance field **_loadCommand**, in the same style as e.g. the **AddCommand** property. 3. Add a new method **Load**, in the same style as in the notes. That is, it should call **Load** on the **_model** instance field. 4. In the constructor, initialise **_loadCommand** in the same style as in the notes. 5. In the method **NotifyCommands**, add a call of **RaiseCanExecuteChanged** on the **_loadCommand** instance field 6. Repeat steps 1-5 for the Save functionality In the **MainPage.xaml** file: 7. Add two new buttons **Load** and **Save** to the view, and bind them to the **LoadCommand** and **SaveCommand** property, respectively. 8. Rebuild the application, and see if you can now load and save notes. Create some notes, click **Save**, close the application, start it again, click **Load**, and see if the saved notes reappear.

Exercise	Excep.1
Project	NoteBookV20
Purpose	Use exceptions for error-handling in an MVVM application
Description	The given application checks that notes cannot have the same title (try it!), but the implementation is quite a mess… The handling is all done in the **set**-part of the **Title** property in **NoteDetailsViewModel**, with several calls to the model and the master-details view model
Steps	Our aim is to clean up the error handling. This involves using exceptions for error signaling and handling, and also to distribute various responsibilities to the proper classes. An exception class **TitleExistsException** is included in the project. 1. In the **NoteModel** class, add checks to the methods **Add** and **UpdateTitle**, such that a **TitleExistsException** is thrown if the new title exists 2. In the **NoteMasterDetailsViewModel** class, uncomment the method **UpdateTitle**. See if you understand why the method is structured in this particular way. 3. In the **NoteDetailsViewModel** class, go to the **set**-part of the **Title** property. Remove ALL the code in the **set**-part, and replace it with a single line of code: `_masterDetailsViewModel.UpdateTitle(value);` 4. Clean up the **NoteDetailsViewModel** class a bit, since it no longer needs a reference to the model (remove the instance field, and remove the parameter from the constructor) 5. Rebuild the application and run it. See if the validation of titles still works as before. 6. See if you can answer the below questions: a. Which class **detects and signals** the error? b. Which class **assumes responsibility** for handling the error? c. Which class **reports** the error to the user? 7. If you have more time, see if you can update the application such that the **Title** and **Content** fields in the view are only enabled if the user has selected a Note in the list view.

Exercise	MVVMStarter.1
Project	MVVMStarterStudent
Purpose	Integrate a domain class (**Student**) into the **MVVMStarter** framework
Description	A domain class **Student** has been added to the **MVVMStarter** frame-work application – now the subsequent steps described in the guide need to be performed. About the **Student** class: • Some sample images have been added to the folder **Assets/Domain/Student**, named ***"01"*** to ***"05"***. • The property **PhotoID** is included in the **Student** class, to make it easy to set the photo used for a student. If you create a **TextBox** in the Details view that is bound to **PhotoID**, the user can type in a short string like ***"04"*** • The aggregated property **ImageSource** will then contain the full path to the image file, and can therefore be used in the **Image-Source** property in the **ItemViewModel** class. <u>Example</u>: if the value of **PhotoID** is set to ***"04"***, the value of **ImageSource** will become: **"..\\..\\..\\Assets\\Domain\\Student\04.jpg"**. This is the relative path to the images in the **Assets/Domain/Student** folder
Steps	1. Use the guide *"MVVMStarter Guide"* on the **MVVMStarter** website to integrate the **Student** class into the framework. Remember to follow the steps carefully! It is always a good test to try to build the project, after having completed a step.