Part 4 - Exercises

Object Oriented Programming

When creating own classes make sure to include the correct **namespace** so you can reference it from your Program.cs file. We'll get to namespaces later. For now whenever you create a new class use the folder name as the namespace.

You can test your own classes in the Main() if you want to, but it is not necessary. It does help you understand your code and the exercises might include some examples how the classes should work.

EXERCISE 4-1: First Account

- The exercise template comes with a ready made class named Account. The Account object represents a bank account that has balance. Account that has some amount of money in it. The accounts are used as follows.

```
Account heikkisAccount = new Account("Heikki's account", 100.00);
Account heikkisSwissAccount = new Account("Heikki's account in Switzerland", 1000000.00);

Console.WriteLine("Intial state");
Console.WriteLine(heikkisAccount);
Console.WriteLine(heikkisSwissAccount);

heikkisAccount.Withdrawal(20);
Console.WriteLine("The balance of Heikki's account is now: " + heikkisAccount.balance);
heikkisSwissAccount.Deposit(200);
Console.WriteLine("The balance of Heikki's other account is now: " + heikkisSwissAccount.balance)

Console.WriteLine("End state");
Console.WriteLine(heikkisAccount);
Console.WriteLine(heikkisSwissAccount);
```

- Write a program that creates an account with a balance of 100.0, deposits 20.0 in it and finally prints the balance.

```
120
```

NOTICE! Perform all the operations in this exact order.

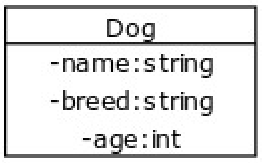
EXERCISE 4-2: First Transfer

- The Account from the previous exercise class is also available in this exercise.
- Write a program that creates an account named “Heikki’s account” with the balance 1000.0.
- And creates an account named “Personal account” with the balance 0.
- Withdraw 100.0 from Heikki’s account.
- Deposit 100.0 to Personal account.
- Print account information, using ToString, on both first Heikki’s then Personal.

```
Heikki's account balance: 900
Personal account balance: 100
```

EXERCISE 4-3: First Class

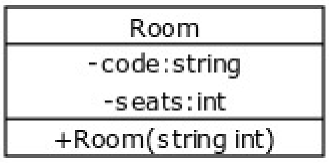
- In this exercise you’ll practice creating a class.
- **CHECK THIS MATERIAL FOR HELP**
- Name the class **Dog** and the file **Dog.cs**.
- Add the variables: private string name, private string breed and private int age.
- As a class diagram the class looks like this.



NOTICE! You have to give your Dog class **namespace exercise_95** for it to function!

EXERCISE 4-04: Classroom

- Create a class named **Room** and file **Room.cs**.
- Add the variables **private string code** and **private int seats** to the class.
- Then create a constructor **public Room(string classCode, int numberOfSeats)** through which values are assigned to the variables.



NOTICE! You have to give your room **namespace exercise_96** for it to function!

EXERCISE 4-5: Whistle

- Create a class named **Whistle**. Add the variable **private string sound** to the class.
- After that create the constructor **public Whistle(string whistleSound)**, which is used to create a new whistle that's given a sound.
- Create a method **public void Sound()** which prints out the sound (using Console.WriteLine).

```
Whistle duckWhistle = new Whistle("Kvaak");
Whistle roosterWhistle = new Whistle("Peef");

duckWhistle.Sound();
roosterWhistle.Sound();
duckWhistle.Sound();
```

```
Kvaak
Peef
Kvaak
```

EXERCISE 4-6: Product

- Create a class **Product** that represents a store product. The product should have a **price (double)**, **quantity (int)** and **name (string)**.
- The class should have the constructor *public Product(string name, double price, int quantity).*
- And a method **public void PrintProduct()** that prints product information in the following format.

```
Banana: price 1.1: 13 pcs
```

- The output above is based on the product being assigned the name banana, with a price of 1.1 and a quantity of 13.

EXERCISE 4-7: Counter

- This exercise consists of multiple sections. Each section corresponds to one exercise point.
- The exercise template comes with a partially executed class **DecreasingCounter**.

```
using System;

namespace exercise_99
{
  public class DecreasingCounter
  {
    private int value;   // a variable that remembers the value of the counter

    public DecreasingCounter(int initialValue)
    {
      this.value = initialValue;
    }

    public void PrintValue()
    {
      Console.WriteLine("value: " + this.value);
    }

    public void decrement()
    {
      // write the method implementation here
      // the aim is to decrement the value of the counter by one
    }

    // and the other methods go here
  }
}
```

- The following is an example of how the main program uses the decreasing counter.

```
public static void Main(string[] args)
{
  DecreasingCounter counter = new DecreasingCounter(10);
  counter.PrintValue();

  counter.Decrement();
  counter.PrintValue();

  counter.Decrement();
  counter.PrintValue();
}
```

```
value: 10
value: 9
value: 8
```

SECTION 1: IMPLEMENTATION OF THE DECREMENT() METHOD

- Implement the **Decrement()** method in the class body in such a way that it decrements the value variable of the object.
- The main program of the previous example should work to produce the example output.

SECTION 2: THE COUNTER'S VALUE CANNOT BE NEGATIVE

- Improve the **Decrement()** in such a way that the counter's value never becomes negative. This means that if the value of the counter is 0 it cannot be decremented. A conditional statement is useful here.

```
public static void Main(string[] args)
{

  DecreasingCounter counter = new DecreasingCounter(2);
  counter.PrintValue();

  counter.Decrement();
  counter.Decrement();
  counter.PrintValue();

  counter.Decrement();
  counter.PrintValue();
}
```

```
value: 2
value: 0
value: 0
```

SECTION 3: RESETTING THE COUNTER VALUE

- Create the method **public void Reset()** for the counter that resets the value of the counter to 0.

```
public static void Main(string[] args)
{

  DecreasingCounter counter = new DecreasingCounter(20);
  counter.PrintValue();

  counter.Reset();
```

```
    counter.PrintValue();
  }
```

```
value: 20
value: 0
```

EXERCISE 4-8: Debt

- Create the class **Debt** that has double type instance variables **balance** and **interestRate**.
- The balance and the interest rate are passed to the constructor as parameters **public Debt(double initialBalance, double initialInterestRate)**.
- In addition create the methods **public void PrintBalance()** and **public void WaitOneYear()** for the class.
- The method PrintBalance prints the current balance and the WaitOneYear method grows the debt amount.
- The debt is increased by multiplying the balance by the interest rate.

```
public static void Main(string[] args)
{

  Debt mortgage = new Debt(120000.0, 1.01);
  mortgage.PrintBalance();

  mortgage.WaitOneYear();
  mortgage.PrintBalance();

  // Wait 20 years
  int years = 0;
  while (years < 20)
  {
    mortgage.WaitOneYear();
    years = years + 1;
  }

  mortgage.PrintBalance();
}
```

- The example above illustrates the development of a mortgage with an interest rate of one percent.

```
120000
121200
147887.0328416936
```

EXERCISE 4-9: Dalmatian

- Create a class called **Dalmatian**. The dalmatian has instance variables **string name** and **int spots**.
- Both are set in the **public Dalmatian(string name, int spots)** constructor.
- Give the variables ability for get and set **Make the variables public rather than private and add { get; set; } on the declaring lines!**

```
Dalmatian spotty = new Dalmatian("Spot", 306);
Console.WriteLine(spotty.name + " is a very good dog. He has " + spotty.spots + " darker spots ir
```

```
Spot is a very good dog. He has 306 darker spots in his fur.
```

EXERCISE 4-10: Gauge

- Create the class **Gauge**. The gauge has the variable **public int value** and **a constructor without parameters**. Constructor sets the initial value of the meter variable to 0.
- The class has following three methods.
- Firstly **public void Increase()** grows the value instance variable's value by one. It does not grow the value beyond five.
- Secondly **public void Decrease()** decreases the value instance variable's value by one. It does not decrease the value to negative values.
- Thirdly **public bool Full()** returns **True** if the instance variable value has the value five. Otherwise, it returns **False**.
- Give the value ability for get and set: **Make the value public rather than private, and add { get; set; } on the declaring lines!**

```
public static void Main(string[] args)
{
  Gauge g = new Gauge();

  while (!g.Full())
  {
    Console.WriteLine("Not full! Value: " + g.value);
```

```
      g.Increase();
    }

    Console.WriteLine("Full! Value: " + g.value);
    g.Decrease();
    Console.WriteLine("Not full! Value: " + g.value);
  }
```

```
Not full! Value: 0
Not full! Value: 1
Not full! Value: 2
Not full! Value: 3
Not full! Value: 4
Full! Value: 5
Not full! Value: 4
```

EXERCISE 4-11: Agent

- The exercise template defines an **Agent** class having a first name and last name.
- The Main method tries to print the introduction for mister Bond, but with no luck.

```
public static void Main(string[] args)
{
  Agent bond = new Agent("James", "Bond");
  Console.WriteLine(bond);

  Agent bourne = new Agent("Jason", "Bourne");
  Console.WriteLine(bourne);
}
```

```
My name is Bond. James Bond.
My name is Bourne. Jason Bourne.
```

- Agent's ToString now returns an empty string. Fix it to introduce international agents in their proper form.

EXERCISE 4-12: Multiplier

- Create a class **Multiplier** as following.
- Constructor **public Multiplier(int number)**.

- Method **public int Multiply(int number)** which returns the value number passed to it multiplied by the number provided to the constructor.
- **You also need to create a instance variable in this exercise.** When you call the method Multiply store the changed value into the instance variable!

```
public static void Main(string[] args)
{
  Multiplier multiplyByThree = new Multiplier(3);

  Console.WriteLine("multiplyByThree.Multiply(2): " + multiplyByThree.Multiply(2));

  Multiplier multiplyByFour = new Multiplier(4);

  Console.WriteLine("multiplyByFour.Multiply(2): " + multiplyByFour.Multiply(2));
  Console.WriteLine("multiplyByThree.Multiply(1): " + multiplyByThree.Multiply(1));
  Console.WriteLine("multiplyByFour.Multiply(1): " + multiplyByFour.Multiply(1));
  Console.WriteLine("multiplyByFour.Multiply(3): " + multiplyByFour.Multiply(3));
}
```

```
multiplyByThree.Multiply(2): 6
multiplyByFour.Multiply(2): 8
multiplyByThree.Multiply(1): 6
multiplyByFour.Multiply(1): 8
multiplyByFour.Multiply(3): 24
```

NOTICE! The value stored in the objects is changed during the first calls! The calculations are actually.

3 * 2 = 6
4 * 2 = 8
6 * 1 = 6
8 * 1 = 8
8 * 3 = 24

EXERCISE 4-13: Statistics

- The exercise template includes class **Statistics**.

```
namespace exercise_105
{
  public class Statistics
  {
    public int count {get; set;}
    public int sum { get; set; }

    public Statistics()
    {
      // initialize the variable count here
    }

    public void AddNumber(int number) {
        // write code here
    }
  }
}
```

- The following program introduces the class' use.

```
Statistics statistics = new Statistics();
statistics.AddNumber(3);
statistics.AddNumber(5);
statistics.AddNumber(1);
statistics.AddNumber(2);
Console.WriteLine("Count: " + statistics.count);
Console.WriteLine("Sum: " + statistics.sum);
```

```
Count: 4
Sum: 11
```

- Expand the program as follows.
- When a number is added, **count** is increased by one.
- When a number is added, **sum** is increased by the number's value.

EXERCISE 4-14: Payment Card

- In this exercise series a class called **PaymentCard** is created which aims to mimic a cafeteria's payment process.
- The template includes the **Program.cs** file. You have to create the **PaymentCard.cs** yourself.
- Add a new class to the project called **PaymentCard** by creating the file mentioned above.

- Create the PaymentCard object’s constructor, which is passed the opening balance of the card and which then stores that balance in the object’s internal variable.
- Write the ToString method, which will return the card’s balance in the form **“The card has a balance of X euros”**.
- Here is the template for the PaymentCard.

```
namespace exercise_106
{
  public class PaymentCard
  {
    private double balance;

    public PaymentCard(double openingBalance)
    {
      // write code here
    }

    public override string ToString()
    {
      // write code here
    }
  }
}
```

- The following main program tests the class.

```
public static void Main(string[] args)
{
  PaymentCard card = new PaymentCard(50);
  Console.WriteLine(card);
}
```

```
The card has a balance of 50 euros
```

EXERCISE 4-15: Using Card

- Expand your answer from the exercise 106 by adding two methods.
- Method **public void EatLunch()**
- Method **public void DrinkCoffee()**

- The method **EatLunch()** should decrease the card's balance by 10.60 euros.
- The method **DrinkCoffee** should decrease the card's balance by 2.0 euros.
- The following main program tests the class.

```
public static void Main(string[] args)
{
  PaymentCard card = new PaymentCard(50);
  Console.WriteLine(card);

  card.EatLunch();
  Console.WriteLine(card);

  card.DrinkCoffee();
  Console.WriteLine(card);
}
```

```
The card has a balance of 50 euros
The card has a balance of 39.4 euros
The card has a balance of 37.4 euros
```

EXERCISE 4-16: Checking Balance

- Expand your previous answers, so that when an item is bought the balance is checked.
- If there is not enough money to buy the balance does not change.

```
public static void Main(string[] args)
{
  PaymentCard card = new PaymentCard(10);
  Console.WriteLine(card);

  card.EatLunch();
  Console.WriteLine(card);

  card.DrinkCoffee();
  Console.WriteLine(card);
}
```

```
The card has a balance of 10 euros
The card has a balance of 10 euros
The card has a balance of 8 euros
```

NOTICE! See how EatLunch() method did not change the balance when there was not enough money. DrinkCoffee() method still worked as it should.

EXERCISE 4-17: Charging Card

- Expand your previous answers, so that you can charge money on your card.

```
public void AddMoney(double amount) {
    // write code here
}
```

- The purpose of the method is to increase the card's balance by the amount of money given as a parameter.
- However the card's balance may not exceed 150 euros.
- As such, if the amount to be topped up exceeds this limit the balance should become exactly 150 euros.
- The following main program tests the class.

```
public static void Main(string[] args)
{
  PaymentCard card = new PaymentCard(100);
  Console.WriteLine(card);

  card.AddMoney(49.99);
  Console.WriteLine(card);

  card.AddMoney(10000.0);
  Console.WriteLine(card);

  card.AddMoney(-10);
  Console.WriteLine(card);
}
```

```
The card has a balance of 100 euros
The card has a balance of 149.99 euros
The card has a balance of 150 euros
The card has a balance of 150 euros
```

NOTICE! You cannot add negative money!

Objects in a List

EXERCISE 4-18: Main Class

- Implement the class **Main** described here. **Do not modify the class Item.**
- Write a program that reads names of items from the user. If the name is empty the program stops reading.
- Otherwise the given name is used to create a new item, which you will then add to the items list.
- Having read all the names print all the items by using the ToString method of the Item class.
- The implementation of the Item class keeps track of the time of creation in addition to the name of the item.

```
Name: Hammer
Name: Radio
Name: Hot Potato
Name:

Hammer (created at: 9.2.2020 13.48.16)
Radio (created at: 9.2.2020 13.48.18)
Hot Potato (created at: 9.2.2020 13.48.21)
```

NOTICE! The List has to be called "items" for the tests to work!

EXERCISE 4-19: Personal Infromation Main

- The program described here should be implemented in the class Main. Do not modify the class PersonalInformation.
- After the user has entered the last set of details, by entering an empty first name, exit the repeat statement.
- Print one empty line here for reading clarity.
- Then print the collected personal information so that each entered object is printed in the following format.
- First and last names separated by a space. You don't print the identification number.
- An example of the working program is given below.

```
First name:
> Jean
```

```
Last name:
> Bartik
Identification number:
> 271224
First name:
> Betty
Last name:
> Holberton
Identification number:
> 070317
First name:
>

Jean Bartik
Betty Holberton
```

NOTICE! You can and should ask the identification number as a string.

EXERCISE 4-20: Television Guide

- In the exercise template there is a ready made class TelevisionProgram representing a television program.
- The class has object variables name and duration, a constructor and few methods.
- Implement a program that begins by reading television programs from the user.
- When the user inputs an empty string as the name of the program the program stops reading programs.
- After this the user is queried for a maximum duration.
- Once the maximum is given the program proceeds to list all the programs whose duration is smaller or equal to the specified maximum duration.

```
Name: Rick and Morty
Duration: 25
Name: Two and a Half Men
Duration: 30
Name: Love it or list it
Duration: 60
Name: House
Duration: 60
Name:
```

```
Program's maximum duration? 30
Rick and Morty, 25 minutes
Two and a Half Men, 30 minutes
```

EXERCISE 4-21: Book Class

- This exercise is worth 2 sections.
- Write a program that first reads book information from the user.
- The details to be asked for each book include the title, the number of pages and the publication year.
- Entering an empty string as the name of the book ends the reading process.
- After this the user is asked for what is to be printed.
- If the user inputs “everything” all the details are printed: the book titles, the numbers of pages, and the publication years.
- However if the user enters the string “title” only the book titles are printed.
- If something else than “everything” or “title” is given the program should not print anything.
- Implement the class Book.
- Implement the functionality in the Main method.
- Example of how the program in Main should work.

```
Name: To Kill a Mockingbird
Pages: 281
Publication year: 1960
Name: A Brief History of Time
Pages: 256
Publication year: 1988
Name: Beautiful Code
Pages: 593
Publication year: 2007
Name: The Name of the Wind
Pages: 662
Publication year: 2007
Name:

What information will be printed? everything
To Kill a Mockingbird, 281 pages, 1960
A Brief History of Time, 256 pages, 1988
```

```
Beautiful Code, 593 pages, 2007
The Name of the Wind, 662 pages, 2007
```

```
Name: To Kill a Mockingbird
Pages: 281
Publication year: 1960
Name: A Brief History of Time
Pages: 256
Publication year: 1988
Name: Beautiful Code
Pages: 593
Publication year: 2007
Name: The Name of the Wind
Pages: 662
Publication year: 2007
Name:

What information will be printed? title
To Kill a Mockingbird
A Brief History of Time
Beautiful Code
The Name of the Wind
```

Files and Reading Data

EXERCISE 4-22: Reading Strings

- Write a program that reads strings from the user until the user inputs the string “end”.
- At that point the program should print how many strings have been read.
- The string “end” should not be included in the number strings read.
- You can find some examples below of how the program works.

```
> I
> have
> a
> feeling
> that
> I
> have
> written
> this
> wrong
> before
> end
11


> end
0
```

EXERCISE 4-23: Reading Integers

- Write a program that reads strings from the user until the user inputs the string “end”.
- As long as the input is not “end” the program should handle the input as an integer and print the cube of the number provided (number * number * number).
- Below are some sample outputs.

```
> 3
27
> -1
-1
> 11
1331
> end


end
```

NOTICE! Remember to convert to integer before calculation!

EXERCISE 4-24: Reading File

- Write a program that prints the contents of a file called “data.txt” so that each line of the file is printed on its own line.
- If the file content looks like this.

In a world
Where code is built

- Then the program should print the following.

```
In a world
Where code is built
```

EXERCISE 4-25: File Names

- Write a program that asks the user for a string and then prints the content of a file with a name matching the string provided.
- You may assume that the user provides a file name that the program can find.
- You do not have to worry about getting errors when the file does not exist.
- The exercise template contains the files “data.txt” and “song.txt”, which you may use when testing the functionality of your program.
- The output of the program can be seen below for when a user has entered the string “song.txt”.
- The content that is printed comes from the file “song.txt”.
- Naturally the program should also work with other filenames, assuming the file can be found.

```
Which file should have its contents printed?
> song.txt

No option for duality
The old is where we come
Clockspeed is fast, but we'll survive
The new will overcome
We are challengers, not followers
We take the ball to build
Easy safe services
Are here to stay

Value for society
Value for life
For you and me
Tieto is here allright!
```

EXERCISE 4-26: Guestlist Txt

- The exercise template comes ready with functionality for the guest list application.
- It checks whether names entered by the user are on the guest list.
- However the program is missing the functionality needed for reading the guest list.
- Modify the program so that the names on the guest list are read from the file.
- Name of the file: guestlist.txt

```
Enter names, an empty line quits.
> Chuck Norris
The name is not on the list.
> Jack Baluer
The name is not on the list.
> Jack Bauer
The name is on the list.
> Jack Bower
The name is on the list.
>
Thank you!
```

NOTICE! The exercise template comes with two files: names.txt and other-names.txt, which have the following contents. Do not change the contents of the files! **NOTICE2!** The exercise expects you to have a **string names** where you store the file!

names.txt:

ada
arto
leena
test
heikki

other-names.txt:

leo
jarmo
alicia
mike
potato